Structuring Communication Software for Quality-of-Service Guarantees

Ashish Mehra Atri Indiresan
Kang G. Shin

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109–2122
{*ashish,atri,kgshin*}@*eecs.umich.edu*

ABSTRACT

A growing number of real-time applications require quality-of-service (QoS) guarantees from the underlying communication subsystem. The communication subsystem (host and network) must support real-time communication services to provide the required QoS of these applications. In this paper, we propose architectural mechanisms for structuring *host* communication software to provide QoS guarantees. In particular, we present and evaluate a *QoS-sensitive* communication subsystem architecture for end hosts that provides real-time communication support for generic network hardware. This architecture provides services for managing communication resources for guaranteed-QoS (real-time) connections, such as admission control, traffic enforcement, buffer management, and CPU & link scheduling. The design of the architecture is based on three key goals: maintenance of QoS-guarantees on a *per-connection* basis, overload protection between established connections, and fairness in delivered performance to best-effort traffic.

Using this architecture we implement *real-time channels*, a paradigm for real-time communication services in packet-switched networks. The proposed architecture features a *process-per-channel* model that associates a channel handler with each established channel. The model employed for handler execution is one of "cooperative" preemption, where an executing handler yields the CPU to a waiting higher-priority handler at well-defined preemption points. The architecture provides several configurable policies for protocol processing and overload protection. We present extensions to the admission control procedure for real-time channels to account for cooperative preemption and overlap between protocol processing and link transmission at a sending host. We evaluate the implementation to demonstrate the efficacy with which the architecture maintains QoS guarantees on outgoing traffic while adhering to the stated design goals. The evaluation also demonstrates the need for specific features and policies provided in the architecture. In subsequent work, we have refined this architecture and used it to realize a full-fledged guaranteed-QoS communication service that performs QoS-sensitive resource management for outgoing as well as incoming traffic.

Key Words — Real-time communication, traffic enforcement, QoS-sensitive resource management, CPU and link scheduling

The work reported in this paper was supported in part by the National Science Foundation under grant MIP–9203895 and the Office of Naval Research under grants N00014–94–1–0229. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF or ONR.

1 Introduction

The advent of high-speed networks has generated an increasing demand for a new class of distributed applications that require quality-of-service (QoS) guarantees from the underlying network. QoS guarantees may be specified in terms of parameters such as the end-to-end delay, delay jitter, and bandwidth delivered on each connection; additional requirements regarding packet loss and in-order delivery can also be specified. Examples of such applications include distributed multimedia applications (e.g., video conferencing, video-on-demand, digital libraries) and distributed real-time command/control systems. To support these applications, the communication subsystem in end hosts and the network must be designed to provide per-connection QoS guarantees. Assuming that the network provides appropriate support to establish and maintain guaranteed-QoS connections, we focus on the design of the *host* communication subsystem to maintain QoS guarantees.

Consider the problem of servicing several guaranteed-QoS and best-effort connections engaged in network input/output at a host. The data to be transmitted over each connection resides either in an input device (such as a frame-grabber) or in host memory; the computation subsystem prepares outgoing data in a QoS-sensitive fashion before handing it to the communication subsystem. Each guaranteed-QoS connection has traffic-flow semantics of unidirectional data flow, in-order message delivery, and unreliable data transfer. Generally speaking, these connection semantics are applicable to a large class of multimedia and real-time command/control applications. Best-effort traffic does not require in-order delivery, but may require retransmissions to ensure loss-free data transfer.

Protocol processing for large data transfers, common in multimedia applications, can be quite expensive. Resource management policies geared towards statistical fairness and/or time-sharing can introduce excessive interference between different connections, thus degrading the delivered QoS on individual connections. Since the local delay bound at a node may be fairly tight, the unpredictability and excessive delays due to interference between different connections may even result in QoS violations. This performance degradation can be eliminated by designing the communication subsystem to provide: (i) maintenance of QoS guarantees, (ii) overload protection via per-connection traffic enforcement, and (iii) fairness to best-effort traffic. These requirements together ensure that per-connection QoS guarantees are maintained as the number of connections or per-connection traffic load increases.

In this paper, we propose and evaluate a QoS-sensitive communication subsystem architecture for guaranteed-QoS connections. Our focus is on the architectural mechanisms used within the communication subsystem to satisfy the QoS requirements of all connections, without undue degradation in performance of best-effort traffic (with no QoS guarantees). While the proposed architecture is applicable to other proposals for guaranteed-QoS connections [1, 2], we focus on *real-time channels*, a paradigm for guaranteed-QoS communication services in packet-switched networks [3, 4]; for this work we use the model proposed and analyzed in [4].

The proposed architecture features a *process-per-channel* model for protocol processing on each channel, coordinated by a unique channel handler created on successful establishment of the channel.

While the service *within* a channel is FIFO, QoS guarantees on multiple channels are provided via appropriate CPU scheduling of channel handlers and link scheduling of packet transmissions. Traffic isolation between channels is facilitated via per-channel traffic enforcement and interaction between the CPU and link schedulers. Channels violating their traffic specification are prevented from consuming processing and link capacity either by blocking the execution or lowering the priority of the corresponding handlers. Protocol processing can be work-conserving or non-work-conserving, with best-effort traffic given processing and transmission priority over "work ahead" real-time traffic. The architectural framework adopted utilizes an abstraction of the underlying communication subsystem in terms of various processing costs, overheads, and policies, which are used for admission control [5] and run-time resource management.

We have implemented the proposed architecture using a communication executive derived from x-kernel 3.1 [6] that exercises complete control over a Motorola 68040 CPU. This configuration avoids any interference from computation or other operating system activities on the host, allowing us to focus on the communication subsystem. We evaluate the proposed architecture under varying degrees of traffic load, and demonstrate the efficacy with which it maintains QoS guarantees on real-time channels and provides fair performance for best-effort traffic, even in the presence of ill-behaved real-time channels.

While the proposed architecture is designed to handle both incoming and outgoing traffic, this paper focuses primarily on run-time communication resource management for outgoing traffic. The issues involved in QoS-sensitive handling of incoming traffic and the associated admission control extensions are presented in [7], but are beyond the scope of this paper. We note, however, that the proposed architecture facilitates provision of per-connection QoS guarantees while preventing receive livelock [8]. Based on the proposed architecture and admission control extensions, we have designed, implemented, and evaluated a full-fledged guaranteed-QoS communication service on Open Software Foundation's Mach MK/x-kernel framework in the context of the ARMADA project [9]. Details of resource management for outgoing and incoming traffic in this service are provided in [10].

For end-to-end guarantees, resource management within the communication subsystem must be integrated with that for applications. The architecture proposed and analyzed in this paper is directly applicable if a portion of the host processing capacity can be reserved for communication-related activities [11, 12]. The proposed architectural extensions can be realized as a server with appropriate capacity reserves and/or execution priority. Our implementation is indeed such a server executing in a stand alone configuration. More importantly, our approach decouples protocol processing priority from that of the application. We believe that the protocol processing priority of a connection must be derived from the QoS requirements, traffic characteristics, and run-time communication behavior of the application on that connection.

We note that the proposed architecture is also applicable to application-level framing [13] and user-level protocol processing architectures explored in recent efforts [14–16] to improve data transfer throughput in high-speed networks. In our design approach we have not made any specific assumptions about the location of the protocol stack, which could reside in the kernel or in user space. Architec-

tural features such as CPU scheduling of channel handlers and cooperative preemption can be utilized irrespective of the address space and protection domain. With user-level protocol processing, however, allocation of communication resources is coupled directly with allocation of computation resources, and as such the admission control and run-time resource management must be more comprehensive. Regardless of the location of the protocol stack, realization of QoS guarantees in practice necessitates architectural mechanisms and extensions similar in nature to the ones considered in this paper. The issues and mechanisms for integrating the proposed architecture with QoS-sensitive application scheduling [17–19] and/or processor capacity reserves will be addressed in a forthcoming paper.

The architectural framework and methodology adopted in this paper can be applied to other host platforms and networking technologies. It is important to note that provision of QoS guarantees is platform specific; specific instances of this architecture depend on the CPU and network capacities of a platform. Implementing this architecture requires that the host communication subsystem be parameterized accurately to capture overheads and processing costs that comprise the abstraction of the underlying communication subsystem. The admission control extensions and run-time management support can then be re-targeted for a given host platform and/or networking technology. We are addressing the issues of portability and accurate parameterization of QoS-sensitive communication subsystems separately [20]. While we have implemented the architecture on an x-kernel platform, the architecture and the implementation do not utilize any features specific to this platform. The underlying resource management policies can be supported on any operating system platform.

The rest of the paper is organized as follows. Section 2 discusses architectural requirements for guaranteed-QoS communication and provides a brief description of real-time channels. Section 3 presents a QoS-sensitive communication subsystem architecture realizing the requirements outlined in Section 2, and Section 4 describes its implementation. Section 5 experimentally evaluates the efficacy of the proposed architecture. Section 6 discusses related work and Section 7 concludes the paper.

2 Architectural Requirements for Guaranteed-QoS Communication

For guaranteed-QoS communication, we consider unidirectional data transfer, from source to sink via intermediate nodes, with data being delivered at the sink in the order in which it is generated at the source. Corrupted, delayed, or lost data is of little value; with a continuous flow of time-sensitive data, there is insufficient time for error recovery. Thus, we consider data transfer with unreliable-datagram semantics with no acknowledgements and retransmissions. To provide per-connection QoS guarantees, host communication resources must be managed in a QoS-sensitive fashion, i.e., according to the relative importance of the connections requesting service. Host communication resources include CPU bandwidth for protocol processing, link bandwidth for packet transmissions, and buffer space.

There are two related aspects to provision of guaranteed-QoS communication [1]: traffic specification and resource management. When requesting this service, an application must specify its traffic characteristics and QoS requirements. Since network resources (buffers, processing capacity, link bandwidth)

are finite, the communication subsystem and the network must perform admission control to provide QoS guarantees. As part of admission control tests, the resources required to satisfy the application's request are computed based on the specified worst-case traffic, and the request accepted if sufficient resources can be reserved for it. Once the application's request has been accepted, the communication subsystem and the network maintain QoS guarantees via resource management and traffic enforcement policies. When the application no longer needs guaranteed-QoS service, all resources allocated to it are released by the network and the communication subsystems at the source and destination hosts.

Figure 1(a) illustrates a generic software architecture for guaranteed-QoS communication services at the host. The components constituting this architecture are briefly discussed below.

Application programming interface (API): The API must export routines that can be used to set up and teardown guaranteed-QoS connections, and perform data transfer on these connections.

Signalling and admission control: A signalling protocol is required to establish/tear down guaranteed-QoS connections across the communicating hosts, possibly via multiple network nodes. The communication subsystem must keep track of communication resources, perform admission control on new connection requests, and establish *connection state* to store connection specific information.

Network data transport: Protocols are needed for unidirectional reliable and unreliable data transfers, including fragmentation (reassembly) of application data into smaller units (i.e., packets) for network transmission (reception).

Traffic enforcement: Traffic enforcement forces an application to conform to its traffic specification and provide overload protection between established connections. This is required at the session level, and may be required at the link level depending on the nature of the traffic violation; link level traffic enforcement may be required at receiving hosts.

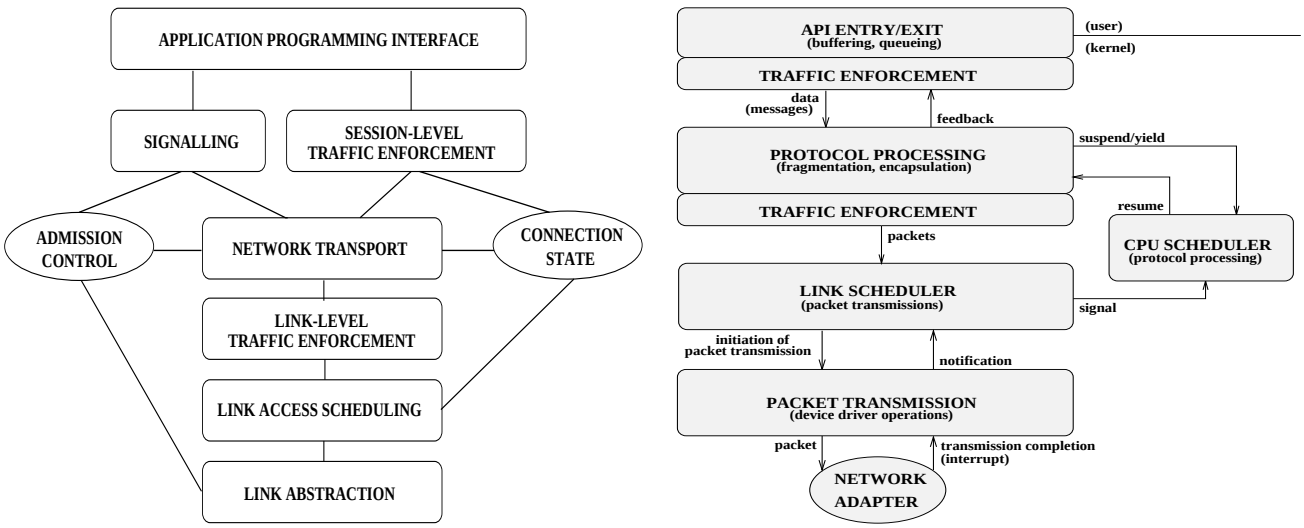
Link access scheduling and link abstraction: Link bandwidth must be managed such that all active connections receive their promised QoS. This necessitates abstracting the link in terms of transmission delay and bandwidth, and scheduling all outgoing packets for network access. The minimum requirement for provision of QoS guarantees is that packet transmission time on the link be bounded and predictable.

Assuming support for signalling, our primary focus is on the components involved in data transfer, namely, traffic enforcement, protocol processing and link transmission. In particular, we study architectural mechanisms for structuring host communication software to provide QoS guarantees.

2.1 Software Structure for QoS-Sensitive Data Transport

In Figure 1(b), an application presents the API with data (*messages*) to be transported on a guaranteed-QoS connection. The API must allocate buffers for this data and queue it appropriately. Data that is conformant (as per the traffic specification) is forwarded for protocol processing and transmission.

Maintenance of per-connection QoS guarantees: Protocol processing involves, at the very least, fragmentation of application messages, including transport and network layer encapsulation, into *packets*



(a) Overall architecture (b) Protocol processing and transmission

Figure 1: Desired software architecture.

with length smaller than a certain maximum (typically the MTU of the attached network). Additional computationally intensive services such as coding, compression, or checksums may also be performed during protocol processing. QoS-sensitive allocation of processing bandwidth necessitates multiplexing of the CPU amongst active connections under control of the CPU scheduler, which must provide deadline-based or priority-based policies for scheduling protocol processing on individual connections.

Non-preemptive protocol processing on a connection implies that the CPU can be reallocated to another connection only after processing an entire message, resulting in a coarser temporal grain of multiplexing and making admission control less effective. More importantly, admission control must consider the largest possible message size (maximum number of bytes presented by the application in one request) across *all* connections, including best-effort traffic. While maximum message size for guaranteed-QoS connections can be derived from application-level attributes such as frame size for multimedia applications, the same for best-effort traffic may not be known *a priori*. Accordingly, mechanisms to suspend and resume protocol processing on a connection are needed. Protocol processing on a connection may also need to be suspended if there are no packet buffers available for that connection.

The packets generated via protocol processing cannot be directly transmitted on the link as that would result in FIFO (i.e., QoS-insensitive) consumption of link bandwidth. Instead, they are forwarded to the link scheduler, which must provide QoS-sensitive policies for scheduling packet transmissions. The link scheduler selects a packet and initiates packet transmission on the network adapter. Notification of packet transmission completion is relayed to the link scheduler so that another packet can be transmitted. The link scheduler must signal the CPU scheduler to resume protocol processing on a connection that was suspended earlier due to shortage of packet buffers.

Overload protection via per-connection traffic enforcement: As mentioned earlier, only conformant data is forwarded for protocol processing and transmission. This is necessary since QoS guarantees

are based on a connection's traffic specification; a connection violating its traffic specification should not be allowed to consume communication resources over and above those reserved for it. Traffic specification violations on one connection should not affect QoS guarantees on other connections and the performance delivered to best-effort traffic. Accordingly, the communication subsystem must police per-connection traffic; in general, each parameter constituting the traffic specification (e.g., rate, burst length) must be policed individually. An important issue is the handling of non-conformant traffic, which could be buffered (shaped) until it is conformant, provided with degraded QoS, treated as best-effort traffic, or dropped altogether. Under certain situations, such as buffer overflows, it may be necessary to block the application until buffer space becomes available, although this may interfere with the timing behavior of the application. The most appropriate policy, therefore, is application-dependent.

Buffering non-conformant traffic till it becomes conformant makes protocol processing *non-work-conserving* since the CPU idles even when there is work available; the above discussion corresponds to this option. Alternately, protocol processing can be *work-conserving*, with CPU scheduling mechanisms ensuring QoS-sensitive allocation of CPU bandwidth to connections. Work-conserving protocol processing can potentially improve CPU utilization, since the CPU does not idle when there is work available. While the unused capacity can be utilized to execute other best-effort activities (such as background computations), one can also utilize this CPU bandwidth by processing non-conformant traffic, if any, assuming there is no pending best-effort traffic. This can free up CPU processing capacity for subsequent messages. In the absence of best-effort traffic, work-conserving protocol processing can also improve the average QoS delivered to individual connections, especially if link scheduling is work-conserving.

Fairness to best-effort traffic: Best-effort traffic includes data transported by conventional protocols such as TCP and UDP, and signalling for guaranteed-QoS connections. Best-effort traffic should not be unduly penalized by non-conformant real-time traffic, especially under work-conserving processing.

2.2 Real-Time Channels: A Model for Guaranteed-QoS Communication

Several models have been proposed for guaranteed-QoS communication in packet-switched networks [1]. While the architectural mechanisms proposed in this paper are applicable to most of the proposed models, we focus on *real-time channels* [4]. A real-time channel is a simplex, fixed-route, virtual connection between a source and destination host, with sequenced messages and associated performance guarantees on message delivery. It therefore conforms to the connection semantics mentioned earlier.

Traffic and QoS Specification: Traffic generation on real-time channels is based on a *linear bounded arrival process* [21,22] characterized by three parameters: maximum message size (M_{max} bytes), maximum message rate (R_{max} messages/second), and maximum burst size (B_{max} messages). The notion of *logical arrival time* is used to enforce a minimum separation $I_{min} = \frac{1}{R_{max}}$ between messages on the same real-time channel. This ensures that a channel does not use more resources than it reserved at the expense of other channels. The QoS on a real-time channel is specified as the desired deterministic, worst-case bound on the end-to-end delay experienced by a message. See [4] for more details.

Resource Management: Admission control for real-time channels is provided by Algorithm `D_order` [4], which uses fixed-priority scheduling for computing the worst-case delay experienced by a channel at a link. When a channel is to be established at a link, the worst-case response time for a message (when the message completes transmission on the link) on this channel is estimated based on non-preemptive fixed-priority scheduling of packet transmissions. The total response time, which is the sum of the response times over all the links on the route of the channel, is checked against the maximum permissible message delay and the channel established only if the latter is greater. A local delay bound is derived from the worse-case response time and the specified end-to-end delay bound. Run-time link scheduling, on the other hand, is governed by a multi-class variation of the earliest-deadline-first (EDF) policy [23].

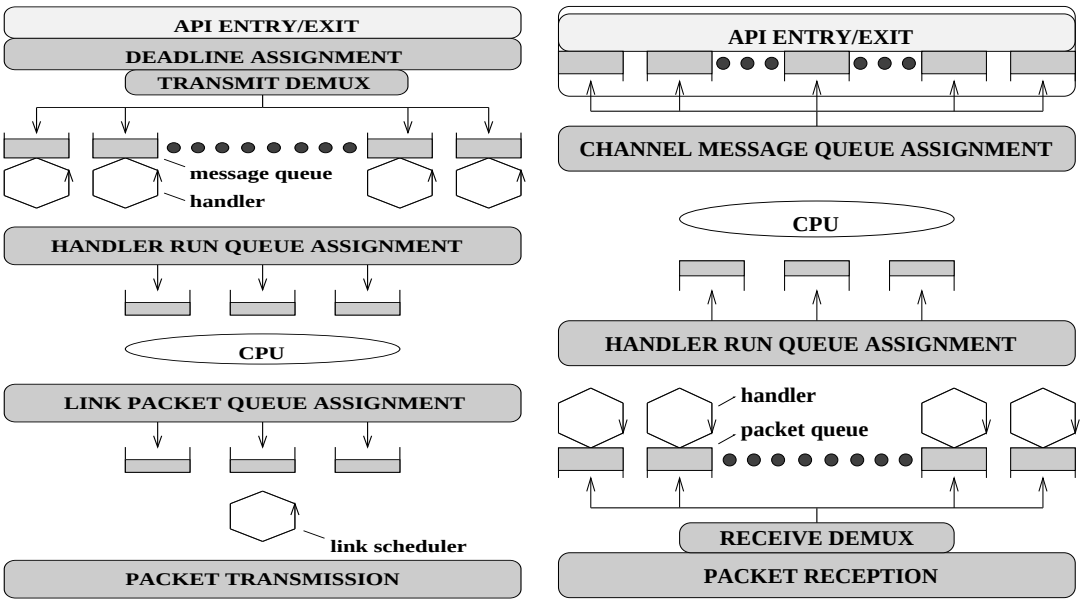
2.3 Performance Related Considerations

To provide deterministic QoS guarantees on communication, all processing costs and overheads involved in managing and using resources must be accounted for. Processing costs include the time required to process and transmit a message, while the overheads include preemption costs such as context switches and cache misses, costs of accessing ordered data structures, and handling of network interrupts. It is important to keep the overheads low and predictable (low variability) so that reasonable worst-case estimates can be obtained. An important performance metric is scalability, i.e., the number of guaranteed-QoS connections that can be serviced at the host. Resource management policies must maximize the number of connections accepted for service. In addition to processing costs and implementation overheads, factors that affect admissibility include the relative bandwidths of the CPU and link and any coupling between CPU and link bandwidth allocation. In a recent paper [5], we have studied the extent to which these factors affect admissibility in the context of real-time channels.

3 A QoS-Sensitive Communication Subsystem Architecture

In the *process-per-message* model [24], a process or thread shepherds a message through the protocol stack. Besides eliminating extraneous context switches encountered in the *process-per-protocol* model [24], it also facilitates protocol processing to be scheduled according to a variety of policies, as opposed to the software-interrupt level processing in BSD Unix. However, the process-per-message model introduces additional complexity for supporting per-channel QoS guarantees.

Creating a distinct thread to handle each message makes the number of active threads a function of the number of messages awaiting protocol processing on each channel. Not only does this consume kernel resources (such as process control blocks and kernel stacks), but it also increases scheduling overheads which are typically a function of the number of runnable threads in dynamic scheduling environments. More importantly, with a process-per-message model, it is relatively harder to maintain channel semantics, provide QoS guarantees, and perform per-channel traffic policing. For example, bursts on a channel get translated into "bursts" of processes in the scheduling queues, making it harder to police ill-behaved channels and ensure fairness to best-effort traffic. Further, scheduling overhead



(a) Source host (b) Destination host

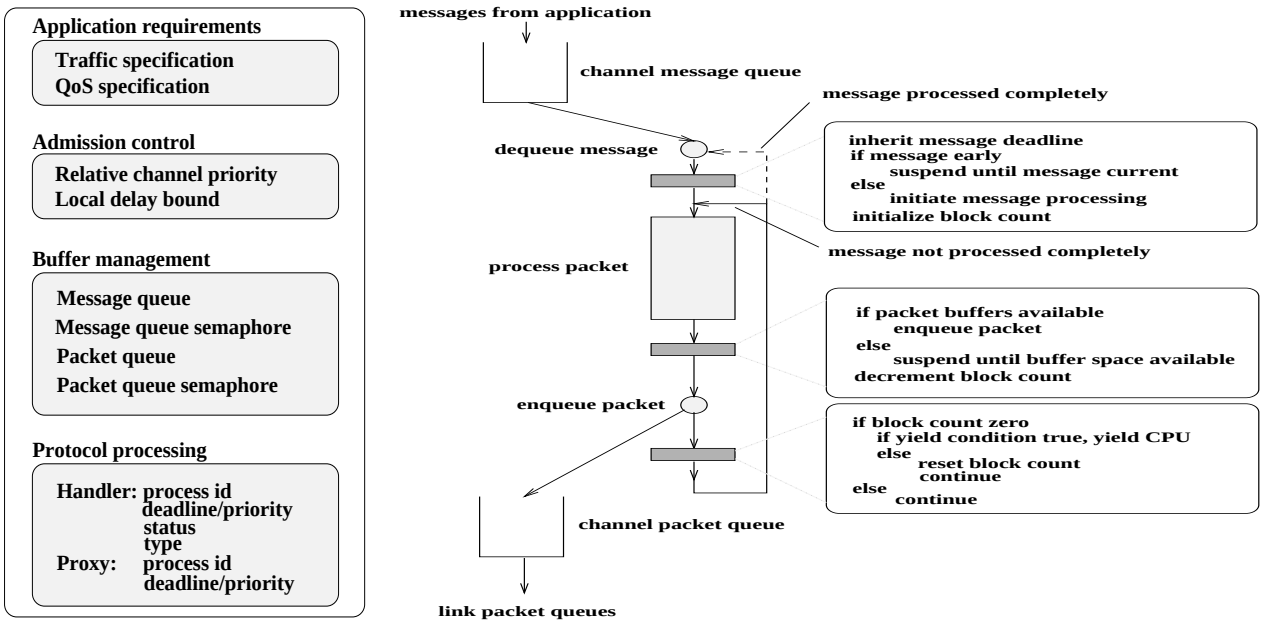
Figure 2: Proposed communication subsystem architecture.

becomes unpredictable, making worst-case estimates either overly conservative or impossible to provide.

Since QoS guarantees are specified on a per-channel basis, it suffices to have a single thread coordinate access to resources for all messages on a given channel. We employ a *process-per-channel* model, which is a QoS-sensitive extension of the *process-per-connection* model [24]. In the process-per-channel model, protocol processing on each channel is coordinated by a unique *channel handler*, a lightweight thread created on successful establishment of the channel. With unique per-channel handlers, CPU scheduling overhead is only a function of the number of *active* channels, those with messages waiting to be transported. Since the number of established channels, and hence the number of active channels, varies much more slowly compared to the number of messages outstanding on all active channels, CPU scheduling overhead is significantly more predictable. As we discuss later, a process-per-channel model also facilitates per-channel traffic enforcement. Further, since it reduces context switches and scheduling overheads, this model is likely to provide good performance to connection-oriented best-effort traffic.

Figure 2 depicts the key components of the proposed architecture at the source (transmitting) and destination (receiving) hosts; only the components involved in data transfer are illustrated. Associated with each channel is a *message queue*, a FIFO queue of messages to be processed by the channel handler (at the source host) or to be received by the application (at the destination host). Each channel also has associated with it a *packet queue*, a FIFO queue of packets waiting to be transmitted by the link scheduler (at the source host) or to be reassembled by the channel handler (at the destination host).

Transmission-Side Processing: In Figure 2(a), invocation of message transmission transfers control to the API. After traffic enforcement (traffic shaping and deadline assignment), the message is enqueued onto the corresponding channel's message queue for subsequent processing by the channel handler.



(a) Channel state at host (b) Handler execution profile

Figure 3: Channel state and handler execution profile.

Based on channel type, the channel handler is assigned to one of three CPU run queues for execution (described in Section 3.1). It executes in an infinite loop, dequeueing messages from the message queue and performing protocol processing (including fragmentation). The packets thus generated are inserted into the channel packet queue and into one of three (outbound) *link packet queues* for the corresponding link, based on channel type and traffic generation, to be transmitted by the link scheduler.

Reception-Side Processing: In Figure 2(b), a received packet is demultiplexed to the corresponding channel's packet queue, for subsequent processing and reassembly. As in transmission-side processing, channel handlers are assigned to one of three CPU run queues for execution, and execute in an infinite loop, waiting for packets to arrive in the channel packet queue. Packets in the packet queue are processed and transferred to the channel's reassembly queue. Once the last packet of a message arrives, the channel handler completes message reassembly and inserts the message into the corresponding message queue. The application retrieves the message from the message queue by invoking the API's receive routine.

At intermediate nodes, the link scheduler relays arriving packets to the next node along the route. In the following discussion, we focus on transmission-side processing at the sending host. Much of this discussion is also applicable to reception-side processing. The issues involved in QoS-sensitive handling of incoming traffic are highlighted in [7], and implementation of the receive-side architecture described in [10].

3.1 Salient Features

Figure 3(a) illustrates a portion of the state associated with a channel at the host upon successful establishment. In addition to application requirements, channel state includes parameters associated with admission control, data structures associated with buffer management, and attributes associated with protocol processing. Each channel is assigned a priority relative to other channels, as determined by the admission control procedure. The local delay bound computed during admission control at the host is used to compute deadlines of individual messages. Each handler is associated with a type, and execution deadline or priority, and execution status (runnable, blocked, etc.). In addition, two semaphores are allocated to each channel handler, one to synchronize with message insertions into the channel's message queue (the *message queue semaphore*), and the other to synchronize with availability of buffer space in the channel's packet queue (the *packet queue semaphore*).

Channel handlers are broadly classified into two types, best-effort and real-time. A best-effort handler is one that processes messages on a best-effort channel. Real-time handlers are further classified as *current* real-time and *early* real-time. A current real-time handler is one that processes *on-time* messages (obeying the channel's rate specification), while an early real-time handler is one that processes *early* messages (violating the channel's rate specification).

Figure 3(b) shows the execution profile of a channel handler at the source host. As long as messages are available, the handler executes in an infinite loop processing messages one at a time. When initialized, it simply waits for messages to process from the message queue. Once a message becomes available, the handler dequeues the message and inherits its deadline. If the message is early, the handler computes the time until the message will become current and suspends execution for that duration. If the message is current, the handler initiates protocol processing of the message. After creating each packet, the handler checks for space in the packet queue (via the packet queue semaphore); it is automatically blocked if space is not available and woken up when space becomes available.

The packets created by the handler are enqueued onto the channel's packet queue, and if the queue was previously empty, the link packet queues are also updated to reflect that this channel has packets to transmit. That is, only the head packet from the channel's packet queue resides in the ordered link packet queues at any given time. When a packet from this channel completes transmission, another packet is transferred from the channel packet queue to the link packet queues. This design incurs a worst-case packet enqueueing overhead proportional to the number of active channels, instead of the total number of packets outstanding on all active channels. The overhead of managing per-channel packet queues is, therefore, minimal, assuming that the device driver can classify a packet (i.e., identify the corresponding channel packet queue) without any extra overhead. This is true in our architecture since only one packet is kept outstanding on the network adapter. Note that our approach is similar in flavor to Rotating-Priority-Queues [25], with the service priority of each channel queue determined dynamically from the deadline of the head packet of the corresponding channel. The preemption model employed for handler execution is one of *cooperative preemption*; the currently-executing handler relinquishes the CPU to a waiting higher-priority handler after processing a block of packets, as explained below.

While the above suffices for non-work-conserving protocol processing, a mechanism is needed to continue handler execution in the case of work-conserving protocol processing. Accordingly, in addition to blocking the handler as before, a *channel proxy* is created on behalf of the handler. A channel proxy is a thread that simply signals the (blocked) channel handler to resume execution. It competes for CPU access with other channel proxies in the order of logical arrival time, and exits immediately if the handler has already woken up. This mechanism ensures that the handler is made runnable if the channel proxy obtains access to the CPU before the handler becomes current. Note that an early handler must still relinquish the CPU to a waiting handler that is already current.

Maintenance of QoS guarantees: Per-channel QoS guarantees are provided via appropriate preemptive scheduling of channel handlers and non-preemptive scheduling of packet transmissions. While CPU scheduling can be priority-based (using relative channel priorities), we consider deadline-based scheduling for channel handlers and proxies. Execution deadline of a channel handler is inherited dynamically from the deadline of the message to be processed. Execution deadline of a channel proxy is derived from the logical arrival time of the message to be processed. Channel handlers are assigned to one of two run queues based on their type (best-effort or real-time), while channel proxies (representing early real-time traffic) are assigned to a separate run queue. The relative priority assignment for handler run queues is such that on-time real-time traffic gets the highest protocol processing priority, followed by best-effort traffic and early real-time traffic in that order.

Provision of QoS guarantees necessitates bounded delays in obtaining the CPU for protocol processing. As shown in [5], immediate preemption of an executing lower-priority handler results in significant overheads due to context switches and cache misses; channel admissibility is significantly improved if preemption overheads are amortized over the processing of several packets. The maximum number of packets processed in a block is a system parameter determined via experimentation on a given host architecture. Cooperative preemption provides a reasonable mechanism to bound CPU access delays while improving utilization, especially if all handlers execute within a single (kernel) address space.

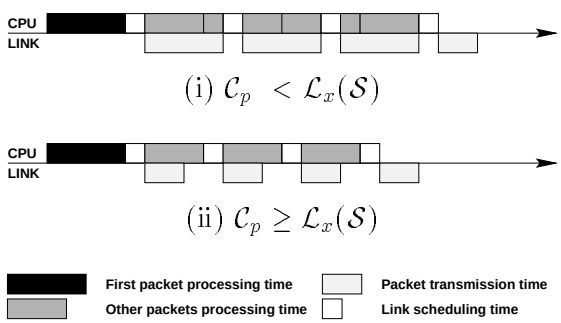
Link bandwidth is managed via multi-class non-preemptive EDF scheduling with link packet queues organized similar to CPU run queues. Link scheduling is non-work-conserving to avoid stressing resources at downstream hosts; in general, the link is allowed to "work ahead" in a limited fashion, as determined by the link *horizon* [4].

Overload protection: Per-channel traffic enforcement is performed when new messages are inserted into the message queue, and again when packets are inserted into the link packet queues. The per-channel message queue absorbs message bursts on a channel, preventing violations of B_{max} and R_{max} on this channel from interfering with other, well-behaved channels. During deadline assignment, new messages are checked for violations in M_{max} and R_{max}. Before inserting each message into the message queue, the inter-message spacing is enforced according to I_{min}. For violations in M_{max}, the (logical) inter-arrival time between messages is increased in proportion to the extra packets in the message.

The number of packet buffers available to a channel is determined by the product of the maximum number of packets constituting a message (derived from M_{max}) and the maximum allowable burst length

$\mathcal{C}_{sw}$	context switch time
$\mathcal{C}_{cm}$	cache miss penalty
$\mathcal{C}_p^{1st}$	first-packet CPU processing cost
$\mathcal{C}_p$	per-packet CPU processing cost
$\mathcal{C}_l$	per-packet link scheduling cost
$\mathcal{P}$	packets between preemption points
$\mathcal{S}$	maximum packet size in bytes
$\mathcal{L}_x(s)$	transmit time for packet size s

(a) Important system parameters



(b) CPU processing and link transmission overlap

Figure 4: Extensions to admission control procedure.

B_{max}. Under work-conserving processing, it is possible that the packets generated by a handler cannot be accommodated in the channel packet queue because all the packet buffers available to the channel are exhausted. A similar situation could arise in non-work-conserving processing with violations of M_{max}. Handlers of such violating channels are prevented from consuming excess processing and link capacity, either by blocking their execution or lowering their priority relative to well-behaved channels. Blocked handlers are subsequently woken up when the link scheduler indicates availability of packet buffers. Blocking handlers in this fashion is also useful in that a slowdown in the service provided to a channel propagates up to the application via the message queue. Once the message queue fills up, the application can be blocked until additional space becomes available. Alternately, messages overflowing the queue can be dropped and the application informed appropriately. Note that while scheduling of handlers and packets provides isolation between traffic on different channels, interaction between the CPU and link schedulers helps police per-channel traffic.

Fairness to best-effort traffic: Under non-work-conserving processing, early real-time traffic does not consume any resources at the expense of best-effort traffic. With work-conserving processing, best-effort traffic is given processing and transmission priority over early real-time traffic.

3.2 Accounting for CPU Preemption Delays and Overheads

The admission control procedure (`D_order`) must account for CPU preemption overheads, access delays due to cooperative preemption, and other overheads involved in managing resources. For each new channel to be admitted, `D_order` computes *message service time*, the worst-case time for which the CPU and link must be allocated to the channel for processing a message, and *wait time*, the worst-case time spent waiting for a lower-priority handler to relinquish the CPU and link. Accordingly, it must account for the overlap between CPU processing and link transmission/reception, and hence the relative bandwidths of the CPU and link. The deadlines assigned to messages are derived from the worst-case service and wait times computed by `D_order`. In recent work [5], we developed extensions to `D_order` to account for the above-mentioned factors at a sending host; the extensions corresponding to interrupt-

mode link scheduling are included below for completeness. Admission control extensions for receiving hosts, for two packet input mechanisms, namely, interrupt mode and polled mode, are outlined in [7].

Suppose the CPU is reallocated to a waiting handler, if needed, every $\mathcal{P}$ packets; that is, up to $\mathcal{P}$ packets are processed between successive preemption points. Further, (see Figure 4(a)), suppose that the maximum packet size is $\mathcal{S}$, context switch overhead between handlers is $\mathcal{C}_{sw}$, cache miss penalty due to a context switch is $\mathcal{C}_{cm}$, and packet transmission time is $\mathcal{L}_x(\mathcal{S})$ for packet size $\mathcal{S}$. Per-packet protocol processing cost is $\mathcal{C}_p$ and per-packet (link) scheduling overhead of selecting a packet and initiating transmission is $\mathcal{C}_l$. For the following discussion, Figure 4(b) illustrates the overlap between processing and transmission of packets, where the link scheduler is invoked as a function call either by the currently executing handler or in interrupt context (after packet transmission).

Consider a message of size $\mathcal{M}$ bytes that has $\mathcal{N}_p = \lceil \frac{\mathcal{M}}{\mathcal{S}} \rceil$ packets, with $(\mathcal{N}_p - 1)$ packets of size $\mathcal{S}$ and the last packet of size $\mathcal{S}^{last} = (\mathcal{M} \bmod \mathcal{S})$ if $(\mathcal{M} \bmod \mathcal{S}) \neq 0$, else $\mathcal{S}^{last} = \mathcal{S}$. Thus, the link transmission time is $\mathcal{L}_x(\mathcal{S})$ for all but the last packet and $\mathcal{L}_x(\mathcal{S}^{last})$ for the last packet. Protocol processing cost for the first packet is $\mathcal{C}_p^{1st}$, while subsequent fragments each incur a lower cost $\mathcal{C}_p$. $\mathcal{C}_p^{1st}$ includes the fixed cost of obtaining the message for processing, timestamp overhead, and the cost of preparing the first packet, which is higher because our fragmentation protocol traverses a slower path for messages larger than $\mathcal{S}$ bytes. Both $\mathcal{C}_p^{1st}$ and $\mathcal{C}_p$ include the cost of network-level encapsulation.

The worst-case message service time $\mathcal{T}_s$ is given by:

$$\mathcal{T}_s = \begin{cases} \mathcal{C}_p^{1st} + \mathcal{L}_x^m + \mathcal{C}_l^m + \mathcal{C}_{pr} & \text{if } \mathcal{C}_p < \mathcal{L}_x(\mathcal{S}) \\ \mathcal{C}_p^m + \mathcal{C}_l^m + \mathcal{L}_x(\mathcal{S}^{last}) + \mathcal{C}_{pr} & \text{otherwise} \end{cases}$$

where $\mathcal{L}_x^m = (\mathcal{N}_p - 1)\mathcal{L}_x(\mathcal{S}) + \mathcal{L}_x(\mathcal{S}^{last})$ is the total link transmission time for the message, $\mathcal{C}_l^m = \mathcal{N}_p\mathcal{C}_l$ is the total link scheduling overhead for the message, $\mathcal{C}_{pr} = (\frac{\mathcal{N}_p - 1}{\mathcal{P}})\mathcal{C}_{csp}$ is the total cost of preemption during the processing of the message ($\mathcal{C}_{csp} = \mathcal{C}_{cm} + \mathcal{C}_{sw}$), and $\mathcal{C}_p^m = \mathcal{C}_p^{1st} + (\mathcal{N}_p - 1)\mathcal{C}_p$ is the total protocol processing cost for the message. If $\mathcal{C}_p < \mathcal{L}_x(\mathcal{S})$, there is at least the cost of processing the first packet. Since link transmission time dominates the time to process subsequent packets (see Figure 4(b):(i)), message service time is determined by the link transmission time for the message and the total link scheduling and preemption overheads incurred. If $\mathcal{C}_p \geq \mathcal{L}_x(\mathcal{S})$ (Figure 4(b):(ii)), however, message service time corresponds to the total protocol processing time for the message plus the time to transmit the last packet, in addition to the total link scheduling and preemption overheads.

The worst-case CPU wait time due to a lower-priority handler is given by

$$\mathcal{T}_w^{cpu} = \mathcal{C}_b^{1st} + \lceil \frac{\mathcal{C}_b^{1st}}{\mathcal{L}_x(\mathcal{S})} \rceil \mathcal{C}_l + \mathcal{C}_{cm} + \mathcal{C}_{sw},$$

where $\mathcal{C}_b^{1st}$ is the worst-case processing time for a block of up to $\mathcal{P}$ packets and is given by $\mathcal{C}_b^{1st} = \mathcal{C}_p^{1st} + (\max(\mathcal{N}_p, \mathcal{P}) - 1)\mathcal{C}_p$. During this time $\mathcal{C}_b^{1st}$, up to $\lceil \frac{\mathcal{C}_b^{1st}}{\mathcal{L}_x(\mathcal{S})} \rceil$ packets could complete transmission. Due to non-preemptive packet transmission, the worst-case link wait time is simply $\mathcal{T}_w^{link} = \mathcal{L}_x(\mathcal{S})$. Thus, the total message wait time is $\mathcal{T}_w = \mathcal{T}_w^{cpu} + \mathcal{T}_w^{link}$.

3.3 Determination of $\mathcal{P}$, $\mathcal{S}$, and $\mathcal{L}_x$

$\mathcal{P}$ and $\mathcal{S}$ determine the granularity at which the CPU and link, respectively, are multiplexed between channels; the choice of these parameters therefore determines channel admissibility at the host [5].

Packets between preemptions: Selection of $\mathcal{P}$ is governed by the architectural characteristics of the host CPU, as captured by the parameters listed in Figure 4(a). For a given message (and packet) size, small values of $\mathcal{P}$ imply a higher number of preemptions, increasing the total overhead incurred and reducing the CPU bandwidth available to channel handlers; this in turn reduces channel admissibility. Large values of $\mathcal{P}$, on the other hand, increase the temporal granularity at which the CPU is multiplexed between channel handlers and hence the window of non-preemptibility. This may also reduce the number of channels admitted for service. For a given host architecture, $\mathcal{P}$ can be selected such that channel admissibility is maximized while delivering reasonable data transfer throughput.

Packet size: $\mathcal{S}$ can be selected either using end-to-end transport protocol performance or host/adapter design characteristics. End-to-end protocol performance has been used to determine packet size in IP and IP-over-ATM networks for optimum TCP performance. However, since data transfer on real-time channels is unidirectional and unreliable, end-to-end protocol performance may not be the best guide for selection of $\mathcal{S}$. A particular choice for $\mathcal{S}$ determines the number of packets constituting a message, and hence total CPU and link bandwidth required to process and transmit it. In general, the latency and throughput characteristics of the adapter as a function of packet size can be used to pick a packet size that minimizes $\mathcal{L}_x$(see below) while delivering reasonable data transfer throughput. However, the value chosen for $\mathcal{S}$ must be less than the MTU of the attached network. Note that in channels spanning heterogeneous networks, $\mathcal{S}$ can be different at each hop, as long as the cost of additional fragmentation within the network is accounted for when determining end-to-end delays.

Packet transmission time: For a typical network adapter, the transmission time for a packet of size s, $\mathcal{L}_x(s)$, depends primarily on the overhead of initiating transmission and the time to transfer the packet to the adapter and on the link. The latter is a function of the packet size and the data transfer bandwidth available between host and adapter memories. The data transfer bandwidth itself is determined by host/adapter design features such as pipelining, on-board queueing on the adapter, and the raw link bandwidth. If $\mathcal{C}_x$ is the overhead to initiate transmission on an adapter feeding a link of bandwidth $\mathcal{B}_l$ bytes/second, $\mathcal{L}_x(s)$ can be approximated as $\mathcal{L}_x(s) = \mathcal{C}_x + \frac{s}{\min(\mathcal{B}_l, \mathcal{B}_x)}$, where $\mathcal{B}_x$ is the data transfer bandwidth available to/from host memory. $\mathcal{B}_x$ is determined by factors such as the mode (direct memory access (DMA) or programmed IO) and efficiency of data transfer, and the degree to which the adapter pipelines packet transmissions. $\mathcal{C}_x$ includes the cost of setting up DMA transfer operations, if any. Our experience with adapter design and the implications for packet transmission time are highlighted in [26].

Routines	Invoked By	Function Performed
`rtc_init`	receiving task	create local queue to receive messages
`rtc_create`	sending task	create real-time channel with given parameters to remote end point (queue); return channel ID
`rtc_send`	sending task	send message on the specified real-time channel
`rtc_recv`	receiving task	receive message from real-time message queue
`rtc_close`	sending task	close specified real-time channel

Table 1: Routines constituting the real-time channel API

4 Prototype Implementation

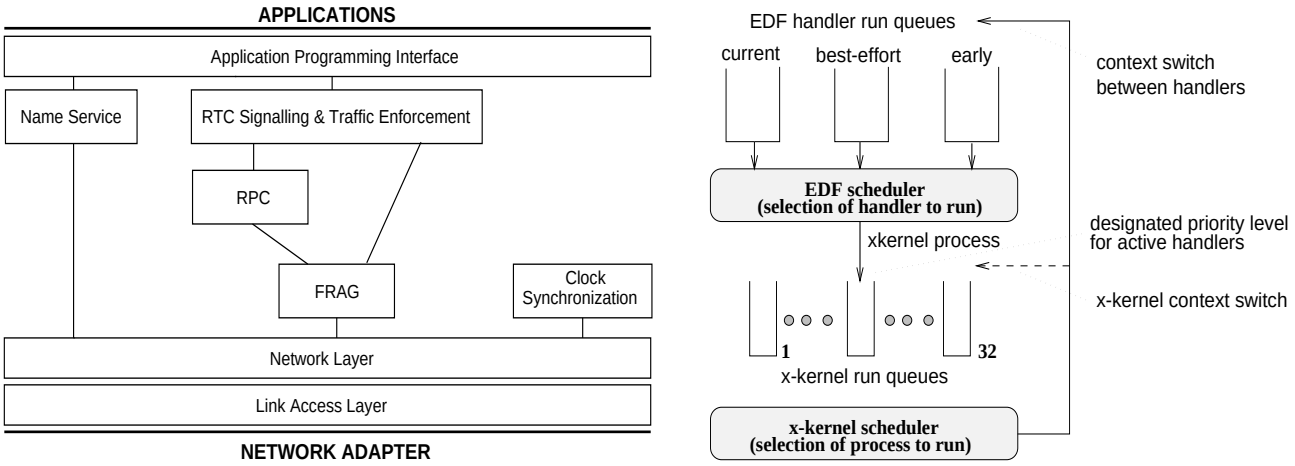
We have implemented the proposed architecture using a communication executive derived from *x*-kernel (v3.1) [6] that exercises complete control over a 25 MHz Motorola 68040 CPU. Accordingly, CPU bandwidth is consumed only by communication-related activities, facilitating admission control and resource management for real-time channels.[1] *x*-kernel (v3.1) employs a process-per-message protocol-processing model and a priority-based non-preemptive scheduler with 32 priority levels; the CPU is allocated to the highest-priority runnable thread, while scheduling within a priority level is FIFO.

4.1 Architectural Configuration

Real-time communication is accomplished via a connection-oriented protocol stack in the communication executive (see Figure 5(a)). The API exports routines for real-time channel establishment, channel teardown, and data transfer (see Table 1); it also supports routines for best-effort data transfer (not shown here). Network transport for signalling is provided by a (resource reservation) protocol layered on top of a remote procedure call (RPC) protocol derived from *x*-kernel's CHAN protocol. Network transport for data is provided by a fragmentation (FRAG) protocol, which packetizes large messages so that communication resources can be multiplexed between channels on a packet-by-packet basis. The FRAG transport protocol is a modified, unreliable version of *x*-kernel's BLAST protocol with timeout and data retransmission operations disabled. The protocol stack also provides protocols for clock synchronization and network layer encapsulation. The network layer protocol is connection-oriented and provides network-level encapsulation for data transport across a point-to-point communication network. The link access layer provides support for link scheduling and includes the network device driver.

Our choice of protocols was based on the perceived requirements for guaranteed-QoS communication. An alternative approach would be to utilize the TCP/IP suite of protocols used on the Internet. Most TCP/IP stacks do not provide a sequenced, unreliable message transport protocol that supports fragmentation. TCP is a reliable byte-stream protocol while UDP does not fragment outbound messages

[1] Implementation of the reception-side architecture is a slight variation of the transmission-side architecture.



(a) Real-time communication protocol stack (x-kernel) (b) Layered EDF CPU scheduler

Figure 5: Implementation environment.

or support message sequencing. Thus, while TCP is suitable for transporting best-effort traffic, it is not suitable for guaranteed-QoS communication. Further, IP is a connectionless protocol and would require either modifications to make it connection-oriented or mechanisms to classify packets. Note that the proposed architecture does not preclude employing TCP to transport best-effort traffic, but this would require IP as the network layer. More details on the protocol stack are provided in [26].

4.2 Realizing a QoS-Sensitive Architecture

Process-per-channel model: On successful establishment, a channel is allocated a channel handler, space for its message and packet queues, and the message and packet queue semaphores. If work-conserving protocol processing is desired, a channel proxy is also allocated to the channel. A channel handler is an x-kernel process (which provides its thread of control) with additional attributes such as the type of channel (best-effort or real-time), flags encoding the state of the handler, its execution priority or deadline, and an event identifier corresponding to the most recent x-kernel timer event registered by the handler. In order to suspend execution until a message is current, a handler utilizes x-kernel's timer event facility and an *event semaphore* which is signaled when the timer expires. A channel proxy is also an x-kernel process with an execution priority or deadline. The states of all established channels are maintained in a linked list that is updated during channel signalling.

We extended x-kernel's process management and semaphore routines to support handler creation, termination, and synchronization with events such as message insertions and availability of packet buffers after packet transmissions. Each packet of a message must inherit the transmission deadline assigned to the message. We modified the BLAST protocol and message manipulation routines in x-kernel to associate the message deadline with each packet. Each outgoing packet carries a global channel identifier, allowing efficient packet demultiplexing at a receiving node.

CPU scheduling: Two policies are available for scheduling channel handlers on the CPU: (i) multi-class EDF scheduling and (ii) fixed-priority scheduling with 32 priority levels. The following discussion applies to (i). Three distinct run queues are maintained for channel handlers, one for each of the three classes mentioned in Section 3.1, similar to the link packet queues. Q1 is a priority queue implemented as a heap ordered by handler deadline while Q2 is implemented as a FIFO queue. Q3, utilized only when the protocol processing is work-conserving, is a priority queue implemented as a heap ordered by the logical arrival time of the message being processed by the handler. Channel proxies are also realized as x-kernel threads and are assigned to Q3. Since Q3 has the lowest priority, proxies do not interfere with the execution of channel handlers.

The multi-class EDF scheduler is layered above the x-kernel scheduler, as illustrated in Figure 5(b). When a channel handler or proxy is selected for execution from the EDF run queues, the associated x-kernel process is inserted into a designated x-kernel priority level for CPU allocation by the x-kernel scheduler. To realize this design, we modified x-kernel's context switch, semaphore, and process management routines appropriately. For example, a context switch between channel handlers involves enqueuing the currently-active handler in the EDF run queues and picking another runnable handler, before invoking the normal x-kernel code to switch process contexts. To support cooperative preemption, we added new routines to check the EDF and x-kernel run queues for waiting higher-priority handlers or native x-kernel processes, respectively, and yield the CPU accordingly.

Link scheduling: The implementation can be configured such that link scheduling is performed:

Option 1: via a function call in the currently executing handler's context or in interrupt context,

Option 2: by a dedicated process/thread, or

Option 3: by a new thread created after each packet transmission.

As demonstrated in [5], option 1 gives the best performance in terms of throughput and sensitivity of channel admissibility to $\mathcal{P}$ and $\mathcal{S}$; accordingly, we focus on option 1 in the discussion below.

The organization of link packet queues is similar to that of handler run queues, except that Q3 is used for early packets when protocol processing is work-conserving. After inserting a packet into the appropriate link packet queue, channel handlers invoke the scheduler directly as a function call. If the link is busy, i.e., a packet transmission is in progress, the function returns immediately and the handler continues execution. If the link is idle, the processing shown in Figure 6 is performed. Scheduler processing is repeated when the network adapter indicates completion of packet transmission or the wakeup event for early packets expires. Additional packets can be kept outstanding on the network adapter as long as packet transmission time is bounded and predictable.

Per-channel traffic enforcement: A channel's message queue semaphore is initialized to B_{max}; messages overflowing the message queue are dropped. The packet queue semaphore is initialized to $B_{max} \cdot \mathcal{N}_{pkts}$, the maximum number of outstanding packets permitted on a channel. Upon completion of packet transmission, the corresponding channel's packet queue semaphore is signalled to indicate availability of packet buffers and enable execution of a blocked handler. If the overflow is due to

1. Mark the link as busy.
2. Examine Q3; transfer (via pointer manipulations) all packets that are current to Q1.
3. Transmit packet at *head*(Q1) if Q1 non-empty, else transmit packet at *head*(Q2).
4. If Q1 and Q2 are both empty, and packet at *head*(Q3) is not current, mark the link as idle, and register wakeup event with x-kernel for the time *head*(Q3) becomes current.

Figure 6: Processing done by the link scheduler.

a violation in M_{max}, the priority (or deadline) of the handler is degraded in proportion to the extra packets in its payload, so that further consumption of CPU bandwidth does not affect other well-behaved channels. Table 2(a) summarizes the policies and options available in the implementation.

4.3 System Parameterization

Table 2(b) lists system parameter settings for our implementation. Selection of $\mathcal{P}$ and $\mathcal{S}$ is based on the tradeoff between available resource capacity and channel admissibility [5]. To use the model of packet transmission time presented in Section 3.3, $\mathcal{C}_x$ and $\mathcal{B}_x$ must be determined for a given network adapter and host architecture. This in turn involves experimentally determining the latency-throughput characteristics of the adapter. Using our implementation, a parameterization of the networking hardware available to us revealed significant performance-related deficiencies such as poor data transfer throughput and high, unpredictable packet transmission time [26]. Since these deficiences were due to adapter design, they severely limited our ability to demonstrate the capabilities of our architecture and implementation. Given our primary focus on unidirectional data transfer, it suffices to ensure that transmission of a packet of size s takes $\mathcal{L}_x(s)$ time units. This can be achieved by *emulating* the behavior of a network adapter such that $\mathcal{L}_x(s)$ time units are consumed for each packet being transmitted. We have implemented such a device emulator, referred to as the *null device*, that can be configured to emulate any desired packet transmission time.

The device emulator is a thread that, once signalled, tracks time by consuming CPU resources for $\mathcal{L}_x(s)$ time units before signalling completion of packet transmission. This emulator is implemented on a separate processor that is connected via a backplane system bus to the processor implementing the communication subsystem (the host processor). Upon expiration of $\mathcal{L}_x(s)$ time units (i.e., completion of packet transmission) the emulator issues an interrupt to the host processor, similar to the mechanism employed in typical network adapters. We have used the emulator to study a variety of tradeoffs, most importantly the effects of the relationship between CPU and link processing bandwidth, in the context of QoS-sensitive protocol processing [5]. We experimentally determined $\mathcal{C}_x$ to be $\approx 40\mu s$. For the experiments, we select $\min(\mathcal{B}_l, \mathcal{B}_x)$ to correspond to a link (and data transfer) speed of 50 ns per byte. This corresponds to an effective packet transmission bandwidth (for 4KB packets) of 16 MB/s.

Note that using the device emulator is not completely accurate since no packet data is actually transferred from host memory. If packet data were transferred from host memory via DMA, there would be additional contention for the system bus and main memory, resulting in somewhat higher packet processing time and cache miss penalties upon resumption of execution after preemption. This would

Category	Available Policies	
Protocol processing model	process-per-channel	
	work-conserving	non-work-conserving
CPU scheduling	fixed-priority with 32 priority levels	
	multi-class earliest-deadline-first	
Handler execution	cooperative preemption with configurable number of packets between preemptions	
Link scheduling	multi-class EDF (options 1, 2 and 3)	
Overload protection	block handler, decay handler deadline, enforce I_{min}, drop overflow messages	

(a) Available policies

Symbol	Value	Unit
$\mathcal{C}_{sw}^{xk}$	20	μs
$\mathcal{C}_{sw}^{edf}$	55	μs
$\mathcal{C}_{cm}$	90	μs
$\mathcal{C}_{p}^{1st}$	420	μs
$\mathcal{C}_{p}$	170	μs
$\mathcal{C}_{l}$	160	μs
$\mathcal{P}$	4	$packets$
$\mathcal{S}$	4096	$bytes$
$\mathcal{L}_x(\mathcal{S})$	245	μs

(b) System parameters

Table 2: Policies and system parameters in the current implementation.

result in optimistic estimates of the message service and wait times, and hence channel admissibility. However, a performance degradation of this nature would affect all real-time channels and best-effort traffic more or less equally, for option 1 as well as option 2.

Therefore, while the absolute performance observed may not be entirely accurate, the observed *trends* and performance *comparisons* reported here continue to be valid. Also, it may be possible to extend the message service time computation to accurately account for the potential perturbation caused by the DMA transfers via careful analysis [27]. We note that at least two other efforts have employed such artificial sources and sinks of data, namely, the virtual network device in [28] that resides on a separate processor, and the "in-memory" network device used in [29].

5 Experimental Evaluation

We evaluate the efficacy of the proposed architecture in isolating real-time channels from each other and from best-effort traffic. The evaluation is conducted for a subset of the policies listed in Table 2, under varying degrees of traffic load and traffic specification violations. In particular, we evaluate the process-per-channel model with non-work-conserving multi-class EDF CPU scheduling and non-work-conserving multi-class EDF link scheduling using option 1 (Section 4.2). Overload protection for packet queue overflows is provided via blocking of channel handlers; messages overflowing the message queues are dropped. The parameter settings given in Table 2(b) are used for the evaluation.

5.1 Methodology and Metrics

Using the null device, the performance of the proposed architecture is compared with and without features such as cooperative preemption and traffic enforcement. We choose a workload that stresses

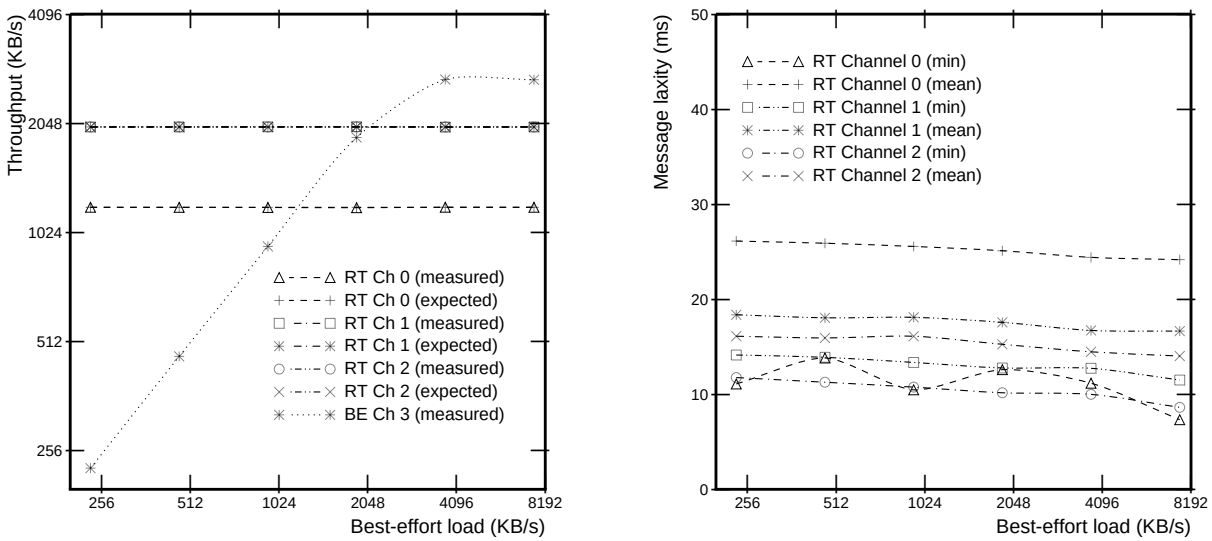
Channel	Type	Traffic Specification				Deadline (ms)
		M_{max} (KB)	B_{max} (messages)	R_{max} (KB/s)	I_{min} (ms)	
0	real-time (RT)	60	12	1200	50	40
1	real-time (RT)	60	8	2000	30	25
2	real-time (RT)	60	1	2000	30	30
3	best-effort (BE)	60	10	variable	–	–

Table 3: Workload used for the evaluation.

the resources on our platform, and is shown in Table 3. Similar results were obtained for other workloads, including a large number of channels with a wide variety of deadlines and traffic specifications. Message size is fixed at 60 KB; experiments with other message sizes reveal that our architecture is insensitive to message size. However, if we relax any of the constraints of our architecture, larger messages tend to introduce greater QoS violations. Three real-time channels are established (channel establishment here is strictly local) with different traffic specifications. Channels 0 and 1 are bursty while channel 2 is periodic in nature. Best-effort traffic is realized as channel 3, with a variable load depending on the experiment, and has similar semantics as the real-time traffic, i.e., it is unreliable with no retransmissions under packet loss.

Messages on each real-time channel are generated by an x-kernel process, running at the highest priority, as specified by a linear bounded arrival process with bursts of up to B_{max} messages. Violations in the specified rate (R_{max}) are realized by generating messages at rates that are multiples of R_{max}. The best-effort traffic generating process is similar, but runs at a priority lower than that of the real-time generating processes and higher than the x-kernel priority assigned to channel handlers. Each experiment's duration corresponds to the transmission of 32K packets; the first 2K and last 2K packets are ignored so that the evaluation is based on steady-state behavior. All the results are obtained after averaging over multiple runs; different runs consistently gave similar results, and hence low standard deviation.

Our evaluation focuses on the efficacy of the proposed architecture and the need for cooperative preemption. All the experiments reported here have traffic enforcement and deadline-based CPU and link scheduling enabled. We use the following metrics measuring per-channel performance. *Throughput* refers to the service received by each real-time channel and best-effort traffic. It is calculated by counting the number of packets successfully transmitted within the experiment duration. *Message laxity* is the difference between the transmission deadline of a real-time message and the actual time that it completes transmission. *Deadline misses* measures the number of real-time packets missing deadlines. Recall all packets of a message inherit the deadline of the message. Deadline misses are detected by checking the actual transmission time of a real-time packet against its deadline. Finally, *Packet drops* measures the number of packets dropped for both real-time and best-effort traffic. Deadline misses and packet drops account for the QoS violations on individual channels.



(a) Throughput

(b) Message laxity

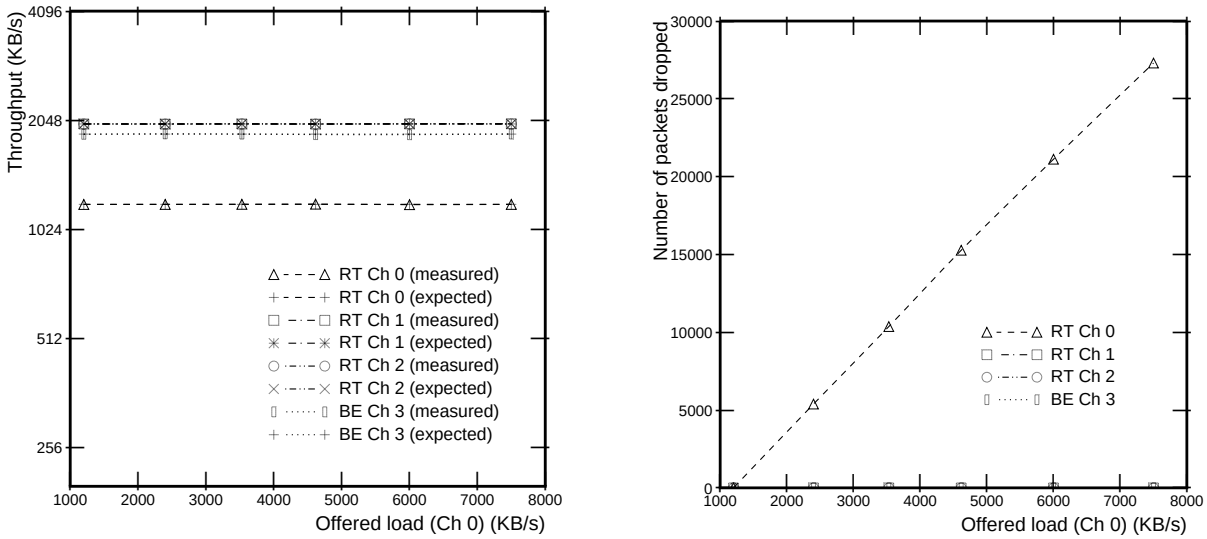
Figure 7: Maintenance of QoS guarantees when traffic specifications are honored.

5.2 Efficacy of the Proposed Architecture

Figure 7 depicts the efficacy of the proposed architecture in maintaining QoS guarantees when all channels honor their traffic specifications. Figure 7(a) plots the throughput received by each real-time channel and best-effort traffic as a function of (offered) best-effort load. Several conclusions can be drawn from the observed trends. First, all real-time channels receive their desired level of throughput; since no packets were dropped (not shown here) or late (Figure 7(b)), the QoS requirements of all real-time channels are met. Increase in offered best-effort load has no effect on the service received by real-time channels. Second, the service received by best-effort traffic continues to increase linearly until the system capacity is exceeded. That is, real-time traffic (early as well as current) does not deny service to best-effort traffic. Third, even under extreme overload conditions, best-effort throughput saturates and declines slightly due to packet drops. However, performance of real-time traffic is not affected.

Figure 7(b) plots the message laxity for real-time traffic, also as a function of offered best-effort load. As can be seen, no messages miss their deadlines, since minimum laxity is non-negative for all channels. In addition, the mean laxity for real-time messages is largely unaffected by an increase in best-effort load, regardless of whether the channel is bursty or not.

Figure 8 demonstrates the same behavior even in the presence of traffic specification violations by real-time channels. Channel 0 generates messages at a rate faster than specified while best-effort traffic is fixed at $\approx$ 1900 KB/s. In Figure 8(a), not only do well-behaved real-time channels and best-effort traffic continue to receive their expected service, channel 0 also receives only its expected service. The laxity behavior is similar to that shown in Figure 7(b). No real-time packets miss deadlines, including



(a) Throughput (b) Number of packets dropped

Figure 8: Maintenance of QoS guarantees under violation of R_{max}.

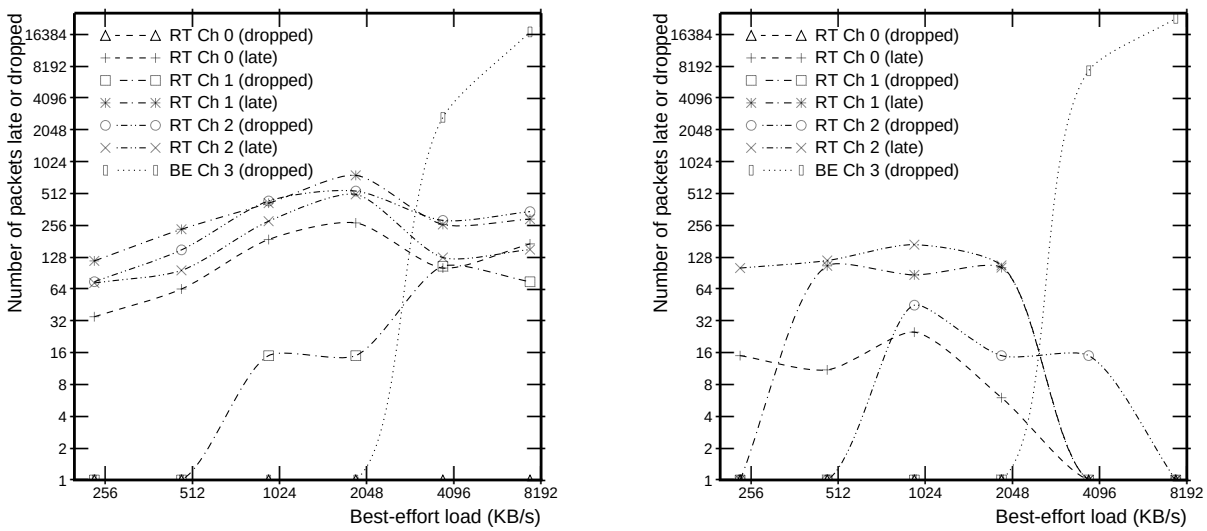
those of channel 0. However, as can be from Figure 8(b), channel 0 overflows its message queue and drops excess messages. None of the other real-time channels or best-effort traffic incur any packet drops.

5.3 Need for Cooperative Preemption

The preceding results demonstrate that the features provided in the architecture are sufficient to maintain QoS guarantees. The following results demonstrate that these features are also necessary.

In Figure 9(a), protocol processing for best-effort traffic is non-preemptive. Even though best-effort traffic is processed at a lower priority than real-time traffic, once the best-effort handler obtains the CPU, it continues to process messages from the message queue regardless of any waiting real-time handlers. That is, CPU scheduling is QoS-insensitive. As can be seen, this introduces a significant number of deadline misses and packet drops, even at low best-effort loads. The deadline misses and packet drops increase with best-effort load until the system capacity is reached. At this point, all excess best-effort traffic is dropped, while the drops and misses for real-time channels decline. The behavior is largely unpredictable, in that different real-time channels are affected differently, and depends on the mix of channels. Further, this behavior is exacerbated by an increase in the amount of buffer space allocated to best-effort traffic; the best-effort handler now runs longer before blocking due to buffer overflow, thereby increasing the window of non-preemptibility.

Figure 9(b) shows the effect of processing real-time messages with preemption only at message boundaries. In addition, early handlers are allowed to execute in a work-conserving fashion but at a



(a) Non-preemptive best-effort processing

(b) Non-preemptive real-time processing

Figure 9: Violation of QoS guarantees with cooperative preemption disabled.

priority higher than best-effort traffic. Note that all real-time traffic is still being shaped since logical arrival time is enforced. As before, we observe significant deadline misses and packet drops for all real-time channels. In this case, best-effort throughput also declines due to early real-time traffic having higher processing priority. This behavior worsens when the window of non-preemptibility is increased by draining the message queue each time a handler executes.

5.4 Discussion

The above results demonstrate the need for cooperative preemption, in addition to traffic enforcement and CPU scheduling, for access to the CPU. While CPU and link scheduling were always enabled, CPU access by real-time channels was also shaped due to traffic enforcement. If traffic was not shaped, one would observe significantly worse real-time and best-effort performance due to non-conformant traffic. We also note that a fully-preemptive kernel is likely to have larger, unpredictable costs for context switches and cache misses. This is because preemption due to unrelated, even lower priority, activities can occur frequently and at arbitrary instants.

Not only does this result in loss of CPU capacity to unnecessary context switches, it also increases the likelihood of disturbing the footprint in the cache [30], unless the cache is suitably partitioned [31]. This is particularly true for preemption caused by external events such as network interrupts. One can account for the cache miss penalty due to preemption via careful schedulability analysis [32], but frequent preemption still degrades available CPU capacity, as also observed in [33]. Moreover, an important implication of arbitrary preemption for the proposed architecture is that a handler may

get preempted just before initiating transmission, even though it had finished preparing a packet for transmission, thus idling the link. Cooperative preemption, on the other hand, provides greater control over preemption points, which in turn improves utilization of resources that may be used concurrently. With cooperative preemption, a handler can initiate transmission on the link before yielding to any higher priority activity.

6 Related Work

This paper (an earlier version appeared in [34]) deals with run-time communication resource management for provision of QoS guarantees at end hosts. The proposed QoS-sensitive architecture and admission control extensions for sending as well as receiving hosts have been fully implemented in the ARMADA guaranteed-QoS communication service [10]. While we have focused on host communication subsystem design to implement real-time channels, our implementation methodology is applicable to other proposals for providing QoS guarantees in packet-switched networks, a survey of which can be found in [1,2]. A number of proposed host and network QoS architectures are compared and contrasted in [35]. Below we highlight related work most relevant to our present work.

Network and protocol support for QoS on the Internet: Similar issues are being examined for provision of integrated services on the Internet [36, 37]. Several classes of service are being considered, including guaranteed service (similar to our work) which provides guaranteed delay [38], predictive service [39], and controlled load [40], the latter two having more relaxed QoS requirements. The expected QoS requirements of applications and issues involved in sharing link bandwidth across multiple classes of traffic are explored in [41, 42]. Much support being provided on the Internet is geared towards multicast communication, in contrast with our work on unicast real-time channels. The signalling required to set up reservations for application flows can be provided by RSVP [43], which initiates reservation setup at the receiver, or ST-II [44], which initiates reservation setup at the sender; RSVP, in particular, provides special provisions for multicast communication. We note that our proposed architecture is applicable to unicast as well as multicast sessions, for both sender-initiated and receiver-initiated signalling.

The issues involved in providing QoS support in IP-over-ATM networks are also being explored [45, 46]. The Tenet real-time protocol suite [47] is an implementation of real-time communication on wide-area networks (WANs), but it did not consider incorporation of protocol processing overheads into the network-level resource management policies. In particular, it has not addressed the problem of QoS-sensitive protocol processing inside hosts. Further, they do not consider incorporation of implementation constraints and overheads, and simultaneous management of CPU and link bandwidth for transmission and reception.

End host architectures for QoS: A number of projects have focused on developing appropriate end host architectures capable of supporting QoS guarantees. The OMEGA [48] end point architecture provides support for end-to-end QoS guarantees. In this architecture, application QoS requirements are translated to network QoS requirements by the QoS Broker [49], which negotiates for the necessary host

and network resources. The OMEGA approach assumes appropriate support from the network subsystem for provision of transport-to-transport layer guarantees, and hence can utilize the architecture and mechanisms proposed in this paper. QoS-A [50] is a layered architecture focusing on provision of QoS within the communication subsystem and the network. It provides features such as end-to-end admission control, resource reservation, QoS translation between layers, and QoS monitoring and maintenance. QoS-A specifies a functionally rich and general architecture supporting networked multimedia applications. Practical realization of QoS-A, however, would necessitate architectural mechanisms and extensions similar in flavor to the ones demonstrated in this paper. We note that these mechanisms, while developed for provision of deterministic QoS guarantees, can also be utilized to realize QoS adaptive communication subsystems [51].

An RSVP-based QoS architecture supporting integrated services in TCP/IP protocol stacks, running on legacy (e.g., Token Ring and Ethernet) LAN interfaces as well as high-speed ATM networks, is described in [52]. A native-mode ATM transport layer has been designed and implemented in [53]. The design of a QoS-controlled communications system for ATM networks is described in [54]. However, implementation overheads and constraints are not incorporated in the resource management policies. Moreover, no performance impact of supporting QoS in communication is reported.

Real-time upcalls (RTUs) [55] are a mechanism to schedule protocol processing for networked multimedia applications via an extended version of the rate monotonic (RM) scheduling policy [23]. Similar to our approach, delayed preemption is adopted to reduce the number of context switches. Our approach differs from RTUs in that we use a thread-based execution model for protocol processing, schedule threads via a modified earliest-deadline-first (EDF) policy [23], and extend resource management policies within the communication subsystem to account for a number of important implementation overheads and constraints. Similar to our approach, rate-based flow control of multimedia streams via kernel-based communication threads is also proposed in [56]. In contrast to our notion of per-connection threads, however, a coarser notion of per-process kernel threads is adopted. This scheme does not seem suitable for an application with multiple QoS connections, each with different QoS requirements and traffic characteristics. Mechanisms for scheduling multiple communication threads, and the issues involved in reception side processing, are not considered. More importantly, the architecture outlined in [56] does not consider provision of signaling and resource management services within the communication subsystem.

Communication subsystem design and performance optimization: Several recent efforts have focused on optimizing the performance of the data transfer path in TCP/IP protocol stacks, via improvement of protocol processing latency [57–59], and user-level handling of network data [14–16, 60, 61] to increase throughput via data copy minimization. Several researchers have studied the issues affecting the design and performance of network adapters [8, 16, 62], and communication subsystems in general [24, 63]. All of these efforts are geared towards traditional best-effort traffic with the primary goal of increasing data transfer throughput. These efforts are complementary to our work, which focuses on provision of QoS guarantees for communication.

Handling of incoming traffic: Proper provisioning of QoS guarantees requires appropriate handling of incoming traffic at a receiving host. While a number of recent efforts have focused on efficient packet classification [64–66], attention has also been given to the problem of receive livelock [8]. Receive livelock is a phenomenon in which, under network input overload, a host or router is swamped with processing and discarding arriving packets to the extent that the effective throughput of the system falls to zero. While techniques for receive livelock elimination are described in [67], an approach to preventing receive livelock, Lazy Receiver Processing (LRP) is presented in [68]. In LRP, an incoming packet is classified and enqueued, but not processed, until the application receives the data. While this works well for best-effort traffic, appropriate OS support is still needed to ensure the application is scheduled to run in a QoS-sensitive fashion. Furthermore, since received packets may not be processed immediately on arrival, LRP cannot exploit the overlap between protocol processing and packet arrival. More importantly, architectural support similar to the one presented in this paper is needed to multiplex resources across multiple connections originating from the same application.

Our approach also utilizes early demultiplexing and channel-specific queueing of incoming packets. However, packet processing and message reassembly is performed in a QoS-sensitive fashion via EDF scheduling of channel handlers, as and when communication capacity is made available. Demultiplexing incoming packets early and absorbing bursts in distinct per-connection queues is an attractive way to prevent receive livelock, an observation also made in the context of paths (see below) in Scout [69]. Coupled with the admission control extensions developed in [7], our architectural approach facilitates provision of QoS guarantees while preventing receive livelock.

"Paths" through the communication subsystem: The Path abstraction realized in Open Software Foundation's CORDS framework [70] greatly facilitates development of real-time communication services for distributed applications, as demonstrated with the ARMADA guaranteed-QoS communication service that we have developed [10]. Using this abstraction, unique *paths* can be defined through the communication subsystem, and path-specific allocation performed for resources such as packet buffers, input packet queues, and input shepherd threads. Similarly, the Scout operating system proposes the use of explicit paths as an important abstraction in operating system design to improve performance [69]. However, while paths in [70] are envisioned primarily as a static, relatively coarse-grain mechanism, paths in [69] are not allocated communication resources and assigned deadlines or priorities via admission control. Our QoS-sensitive communication subsystem architecture for sending and receiving hosts effectively defines, and associated QoS guarantees with, unique paths through the communication subsystem.

Modeling of real-time operating systems: Since the analysis presented in Section 3.2 is geared towards the real-time communication needs of distributed systems, it compliments recent efforts to bridge the gap between the theory and practice for real-time systems [33, 71–73]. The implications of priority inversion due to non-preemptible critical sections was studied in [74]; however, preemption costs (context switches and cache misses) and the resulting degradation in useful resource capacity were not considered.

OS support for QoS-sensitive computation and communication: The need for scheduling protocol pro-

cessing at priority levels consistent with those of the communicating application was highlighted in [75] and some implementation strategies demonstrated in [76]. More recently, processor capacity reserves in Real-Time Mach [11] have been combined with user-level protocol processing [14] for predictable protocol processing inside hosts [12]. Unlike our approach, the approach outlined in [12], does not derive the protocol processing priority from a connection's QoS and traffic specifications, nor does it exploit the overlap between protocol processing and link transmission/reception. Moreover, only a single communication library thread is associated with an application; data arriving on multiple connections associated with the application are all processed by this thread. While this realizes per-application QoS guarantees, it does not facilitate provision of *per-connection* QoS guarantees.

Several scheduling algorithms for integrated scheduling of multimedia soft real-time computation and traditional hard real-time tasks on a multiprocessor multimedia server are proposed and evaluated in [77]. Operating system support for multimedia communication also is explored in [78], where the focus is on provision of preemption points and EDF scheduling in the kernel, and in [79], which also focuses on the scheduling architecture. However, no support is provided for traffic enforcement or decoupling of protocol processing priority from application priority. Several QoS-sensitive CPU scheduling policies have been proposed recently [17–19]. These schemes do not provide support for QoS guarantees on communication-related CPU processing, although our architecture can be integrated with these policies for application-level QoS.

7 Conclusions and Future Work

We have proposed and evaluated a QoS-sensitive communication subsystem architecture for end hosts that supports guaranteed-QoS connections. The architecture provides various services for managing communication resources, such as admission control, traffic enforcement, buffer management, and CPU & link scheduling. Using our implementation of real-time channels, we demonstrated the efficacy with which the architecture maintains per-channel QoS guarantees and delivers reasonable performance to best-effort traffic. While we demonstrated the need for specific features and policies in the architecture for a relatively lightweight stack, such support will be even more important if computationally intensive services such as coding, compression, or checksums are added to the protocol stack. The usefulness of these features also depends on the relationship between CPU and link bandwidths.

Our work assumes that the network adapter (i.e., the underlying network) does not provide any explicit support for QoS guarantees, other than providing a bounded and predictable packet transmission time. This assumption is valid for a large class of networks prevalent today, such as FDDI and switch-based networks. Thus, link scheduling is realized in software, requiring lower layers of the protocol stack to be cognizant of the delay-bandwidth characteristics of the network. A software-based implementation also enables experimentation with a variety of link sharing policies, especially if multiple service classes are supported. For example, alternative approaches such as setting aside a certain minimum CPU and link bandwidth for best-effort traffic can be explored. The architecture can also be extended to networks

providing explicit support for QoS guarantees, such as ATM. However, the communication software may need to track adapter buffer usage in order to schedule the transfer of outgoing packets to the adapter.

The architectural framework and methodology adopted in this paper is applicable to other host platforms as well. This requires that the host communication subsystem be parameterized accurately to capture overheads and processing costs that comprise the abstraction of the underlying communication subsystem. While we have implemented the architecture on an x-kernel platform, the architecture and the implementation do not utilize any features specific to this platform. We argue that the underlying resource management policies can be supported on any operating system platform and communication subsystem.

We are extending this work in several directions. We have extended the null device into a sophisticated network device emulator providing link bandwidth management [80], to explore issues involved when interfacing to adapters with support for QoS guarantees. For true end-to-end QoS guarantees, scheduling of channel handlers must be integrated with application scheduling. We have extended and reimplemented the proposed architecture in OSF Mach MK, a microkernel-based operating system, using OSF's x-kernel framework [70]. This platform is being used to explore issues in integration of the proposed architecture with applications and OSF Mach. We are also exploring the issues involved in implementing *statistical* real-time channels, as opposed to the deterministic real-time channel implementation described in this paper. Statistical QoS guarantees can potentially be useful to a large class of distributed multimedia applications. Finally, we have extended this architecture to shared-memory multiprocessor multimedia servers [81].

References

[1] C. M. Aras, J. F. Kurose, D. S. Reeves, and H. Schulzrinne, "Real-time communication in packet-switched networks," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 122–139, January 1994.

[2] H. Zhang, "Service disciplines for guaranteed performance service in packet-switching networks," *Proceedings of the IEEE*, vol. 83, no. 10, , October 1995.

[3] D. Ferrari and D. C. Verma, "A scheme for real-time channel establishment in wide-area networks," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 3, pp. 368–379, April 1990.

[4] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time communication in multi-hop networks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1044–1056, October 1994.

[5] A. Mehra, A. Indiresan, and K. Shin, "Resource management for real-time communication: Making theory meet practice," in *Proc. of 2nd Real-Time Technology and Applications Symposium*, pp. 130–138, June 1996.

[6] N. C. Hutchinson and L. L. Peterson, "The x-Kernel: An architecture for implementing network protocols," *IEEE Trans. Software Engineering*, vol. 17, no. 1, pp. 1–13, January 1991.

[7] A. Mehra, A. Indiresan, and K. Shin, "Resource management for real-time communication: Making theory meet practice," In submission to IEEE Transactions on Networking, June 1997.

[8] K. K. Ramakrishnan, "Performance considerations in designing network interfaces," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 203–219, February 1993.

[9] *ARMADA: A Real-Time Middleware Architecture for Distributed Applications.* http://www.eecs.umich.edu/RTCL/armada/.

[10] A. Mehra, A. Shaikh, T. Abdelzaher, Z. Wang, and K. Shin, "Realizing guaranteed-QoS communication services on a micro-kernel operating system," In preparation, August 1997.

[11] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves for multimedia operating systems," in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[12] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar, "Predictable communication protocol processing in Real-Time Mach," in *Proc. of 2nd Real-Time Technology and Applications Symposium*, June 1996.

[13] D. D. Clark and D. L. Tennenhouse, "Architectural considerations for a new generation of communication protocols," in *Proc. of ACM SIGCOMM*, pp. 200–208, September 1990.

[14] C. Maeda and B. N. Bershad, "Protocol service decomposition for high-performance networking," in *Proc. ACM Symp. on Operating Systems Principles*, pp. 244–255, December 1993.

[15] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. Lazowska, "Implementing network protocols at user level," in *Proc. of ACM SIGCOMM*, pp. 64–73, San Francisco, California, October 1993.

[16] P. Druschel, L. L. Peterson, and B. S. Davie, "Experiences with a high-speed network adaptor: A software perspective," in *Proc. of ACM SIGCOMM*, pp. 2–13, London, UK, August 1994.

[17] C. Waldspurger, *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*, PhD thesis, Technical Report, MIT/LCS/TR-667, Laboratory for CS, MIT, September 1995.

[18] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton, "A proportional share resource allocation algorithm for real-time time-shared systems," in *Proc. 17th Real-Time Systems Symposium*, pp. 288–299, December 1996.

[19] P. Goyal, X. Guo, and H. M. Vin, "A hierarchical CPU scheduler for multimedia operating systems," in *Proc. 2nd OSDI Symposium*, pp. 107–121, October 1996.

[20] A. Mehra, Z. Wang, and K. Shin, "Self-parameterizing protocol stacks for guaranteed quality of service," In preparation, August 1997.

[21] R. L. Cruz, *A Calculus for Network Delay and a Note on Topologies of Interconnection Networks*, PhD thesis, University of Illinois at Urbana-Champaign, July 1987. available as technical report UILU–ENG–87–2246.

[22] D. P. Anderson, S. Y. Tzou, R. Wahbe, R. Govindan, and M. Andrews, "Support for continuous media in the DASH system," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 54–61, 1990.

[23] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in hard real-time environment," *Journal of the ACM*, vol. 1, no. 20, pp. 46–61, January 1973.

[24] D. C. Schmidt and T. Suda, "Transport system architecture services for high-performance communications systems," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 4, pp. 489–506, May 1993.

[25] J. Liebeherr, D. E. Wrege, and D. Ferrari, "Exact admission control for networks with a bounded delay service," *IEEE/ACM Trans. Networking*, vol. 4, no. 6, pp. 885–901, December 1996.

[26] A. Indiresan, A. Mehra, and K. Shin, "Design tradeoffs in implementing real-time channels on bus-based multiprocessor hosts," Technical Report CSE-TR-238-95, University of Michigan, April 1995.

[27] T.-Y. Huang, J. W. Liu, and D. Hull, "A method for bounding the effect of DMA I/O interference on program execution time," in *Proc. 17th Real-Time Systems Symposium*, pp. 275–285, December 1996.

[28] M. Bjorkman and P. Gunningberg, "Locking effects in multiprocessor implementations of protocols," in *Proc. of ACM SIGCOMM*, pp. 74–83, September 1993.

[29] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley, "Performance issues in parallelized network protocols," in *Proc. USENIX Symp. on Operating Systems Design and Implementation*, pp. 125–137, November 1994.

[30] J. Mogul and A. Borg, "The effect of context switches on cache performance," in *Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 75–85, April 1991.

[31] J. Liedtke, H. Hartig, and M. Hohmuth, "OS-controlled cache predictability for real-time systems," in *Proc. Real-Time Technology and Applications Symposium*, pp. 213–223, June 1997.

[32] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," in *Proc. 17th Real-Time Systems Symposium*, pp. 264–274, December 1996.

[33] R. Gopalakrishnan and G. M. Parulkar, "Bringing real-time scheduling theory and practice closer for multimedia computing," in *Proc. of ACM SIGMETRICS*, pp. 1–12, May 1996.

[34] A. Mehra, A. Indiresan, and K. Shin, "Structuring communication software for quality of service guarantees," in *Proc. 17th Real-Time Systems Symposium*, pp. 144–154, December 1996.

[35] A. T. Campbell, C. Aurrecoechea, and L. Hauw, "A review of qos architectures,", 1996.

[36] D. D. Clark, S. Shenker, and L. Zhang, "Supporting real-time applications in an integrated services packet network: Architecture and mechanism," in *Proc. of ACM SIGCOMM*, pp. 14–26, August 1992.

[37] R. Braden, D. Clark, and S. Shenker, "Integrated services in the Internet architecture: An overview," *Request for Comments RFC 1633*, July 1994. Xerox PARC.

[38] S. Shenker, C. Partridge, and R. Guerin, "Specification of guaranteed quality of service," *Internet Draft draft-ietf-intserv-guaranteed-svc-05.txt*, June 1996. Xerox/BBN/IBM.

[39] S. Jamin, P. Danzig, S. Shenker, and L. Zhang, "A measurement-based admission control algorithm for integrated services packet networks," in *Proc. of ACM SIGCOMM*, pp. 2–13, August 1995.

[40] J. Wroclawski, "Specification of controlled-load network element service," *Internet Draft draft-ietf-intserv-ctrl-load-svc-03.txt*, June 1996. MIT.

[41] S. Shenker, D. Clark, and L. Zhang, "A scheduling service model and a scheduling architecture for an integrated services packet network," *Working Paper*, August 1993. Xerox PARC.

[42] S. Floyd and V. Jacobson, "Link-sharing and resource management models for packet networks," *IEEE/ACM Trans. Networking*, vol. 3, no. 4, , August 1995.

[43] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A new resource ReSerVation Protocol," *IEEE Network*, pp. 8–18, September 1993.

[44] L. Delgrossi and L. Berger, "Internet stream protocol version 2 (ST-2) protocol specification - version ST2+," *Request for Comments RFC 1819*, August 1995. ST2 Working Group.

[45] M. Borden, E. Crawley, B. Davie, and S. Batsell, "Integration of real-time services in an IP-ATM network architecture," *Request for Comments RFC 1821*, August 1995. Bay Networks, Bellcore, NRL.

[46] M. Perez, F. Liaw, A. Mankin, E. Hoffman, D. Grossman, and A. Malis, "ATM signaling support for IP over ATM," *Request for Comments RFC 1755*, February 1995. ISI, Fore, Motoral Codex, Ascom Timeplex.

[47] A. Banerjea, D. Ferrari, B. Mah, M. Moran, D. C. Verma, and H. Zhang, "The Tenet real-time protocol suite: Design, implementation, and experiences," *IEEE/ACM Trans. Networking*, vol. 4, no. 1, pp. 1–11, February 1996.

[48] K. Nahrstedt and J. M. Smith, "Design, implementation and experiences of the OMEGA end-point architecture," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1263–1279, September 1996.

[49] K. Nahrstedt and J. M. Smith, "The QoS broker," *IEEE Multimedia*, vol. 2, no. 1, pp. 53–67, Spring 1995.

[50] A. T. Campbell, G. Coulson, and D. Hutchison, "A quality of service architecture," *Computer Communication Review*, April 1994.

[51] A. T. Campbell and G. Coulson, "QoS adaptive transports: Delivering scalable media to the desktop," *IEEE Network Magazine*, pp. 18–27, March/April 1997.

[52] T. Barzilai, D. Kandlur, A. Mehra, D. Saha, and S. Wise, "Design and implementation of an RSVP-based quality of service architecture for integrated services Internet," in *Proc. Int'l Conf. on Distributed Computing Systems*, May 1997.

[53] R. Ahuja, S. Keshav, and H. Saran, "Design, implementation, and performance of a native mode ATM transport layer," in *Proc. IEEE INFOCOM*, pp. 206–214, March 1996.

[54] G. Coulson, A. Campbell, P. Robin, G. S. Blair, M. Papathomous, and D. Shepherd, "The design of a QoS-controlled ATM-based communications system in Chorus," *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 4, pp. 686–699, May 1995.

[55] R. Gopalakrishnan and G. M. Parulkar, "A real-time upcall facility for protocol processing with QoS guarantees," in *Proc. ACM Symp. on Operating Systems Principles*, p. 231, December 1995.

[56] D. K. Y. Yau and S. S. Lam, "An architecture towards efficient OS support for distributed multimedia," in *Proc. Multimedia Computing and Networking (MMCN '96)*, January 1996.

[57] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley, "Analysis of techniques to improve protocol processing latency," in *Proc. of ACM SIGCOMM*, pp. 73–84, October 1996.

[58] T. Blackwell, "Speeding up protocols for small messages," in *Proc. of ACM SIGCOMM*, pp. 85–95, October 1996.

[59] R. van Renesse, "Masking the overhead of protocol layering," in *Proc. of ACM SIGCOMM*, pp. 96–104, October 1996.

[60] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Clamvokis, and C. Dalton, "User-space protocols deliver high performance to applications on a low-cost Gb/s LAN," in *Proc. of ACM SIGCOMM*, pp. 14–24, London, UK, August 1994.

[61] V. Buch, T. von Eicken, A. Basu, and W. Vogels, "U-Net: A user-level network interface for parallel and distributed computing," in *Proc. ACM Symp. on Operating Systems Principles*, pp. 40–53, December 1995.

[62] P. A. Steenkiste, "A systematic approach to host interface design for high-speed networks," *IEEE Computer*, pp. 47–57, March 1994.

[63] P. Druschel, M. B. Abbott, M. Pagels, and L. L. Peterson, "Network subsystem design: A case for an integrated data path," *IEEE Network Magazine*, pp. 8–17, July 1993.

[64] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss, "Efficient packet demultiplexing for multiple endpoints and large messages," in *Proc. of Winter USENIX*, 1994.

[65] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar, "PATHFINDER: A pattern-based packet classifier," in *Proc. of ACM SIGCOMM*, pp. 115–123, London, UK, August 1994.

[66] D. Engler and M. F. Kaashoek, "DPF: Fast, flexible message demultiplexing using dynamic code generation," in *Proc. of ACM SIGCOMM*, pp. 53–59, October 1996.

[67] J. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," in *Winter USENIX Conference*, January 1996.

[68] P. Druschel and G. Banga, "Lazy receiver processing (LRP): A network subsystem architecture for server systems," in *Proc. 2nd OSDI Symposium*, pp. 261–275, October 1996.

[69] D. Mosberger and L. L. Peterson, "Making paths explicit in the Scout operating system," in *Proc. USENIX Symp. on Operating Systems Design and Implementation*, pp. 153–167, October 1996.

[70] F. Travostino, E. Menze, and F. Reynolds, "Paths: Programming with system resources in support of real-time distributed applications," in *Proc. IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, February 1996.

[71] D. Katcher, H. Arakawa, and J. K. Strosnider, "Engineering and analysis of fixed priority schedulers," *IEEE Trans. Software Engineering*, vol. 19, no. 9, pp. 920–934, September 1993.

[72] A. Burns, K. Tindell, and A. Wellings, "Effectice analysis for engineering real-time fixed priority schedulers," *IEEE Trans. Software Engineering*, vol. 21, no. 5, pp. 475–480, May 1995.

[73] K. A. Kettler, D. I. Katcher, and J. K. Strosnider, "A modeling methodology for real-time/multimedia operating systems," in *Proc. of the Real-Time Technology and Applications Symposium*, pp. 15–26, May 1995.

[74] C. W. Mercer and H. Tokuda, "Preemptibility in real-time operating systems," in *Proc. Real-Time Systems Symposium*, December 1992.

[75] D. P. Anderson, L. Delgrossi, and R. G. Herrtwich, "Structure and scheduling in real-time protocol implementations," Technical Report TR–90–021, International Computer Science Institute, Berkeley, CA, June 1990.

[76] R. Govindan and D. P. Anderson, "Scheduling and IPC mechanisms for continuous media," in *Proc. ACM Symp. on Operating Systems Principles*, pp. 68–80, 1991.

[77] H. Kaneko, J. A. Stankovic, S. Sen, and K. Ramamritham, "Integrated scheduling of multimedia and hard real-time tasks," in *Proc. 17th Real-Time Systems Symposium*, pp. 206–217, December 1996.

[78] O. Hagsand and P. Sjodin, "Workstation support for real-time multimedia communication," in *Winter USENIX Conference*, pp. 133–142, January 1994. Second Edition.

[79] C. Vogt, R. G. Herrtwich, and R. Nagarajan, "HeiRAT: The Heidelberg resource administration technique design philosophy and goals," Research Report 43.9213, IBM Research Division, IBM European Networking Center, Heidelberg, Germany, 1992.

[80] A. Indiresan, A. Mehra, and K. Shin, "The *END*: An *E*mulated *N*etwork *D*evice for evaluating adapter design," in *Proc. 3rd Intl. Workshop on Performability Modeling of Comput er and Communication Systems (PMCCS3)*, pp. 90–94, September 1996.

[81] A. Mehra and K. Shin, "QoS-sensitive protocol processing in shared-memory multiprocessor multimedia servers," in *Proc. of 3rd IEEE Workshop on Architecture and Implementation of High-Performance Communication Subsystems*, pp. 163–169, August 1995.