Structuring Communication Software for Quality-of-Service Guarantees

Ashish Mehra, Atri Indiresan and 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

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. We evaluate the implementation to demonstrate the efficacy with which the architecture maintains QoS guarantees while adhering to the stated design goals. The evaluation also demonstrates the need for specific features and policies provided in the architecture.*

1. Introduction

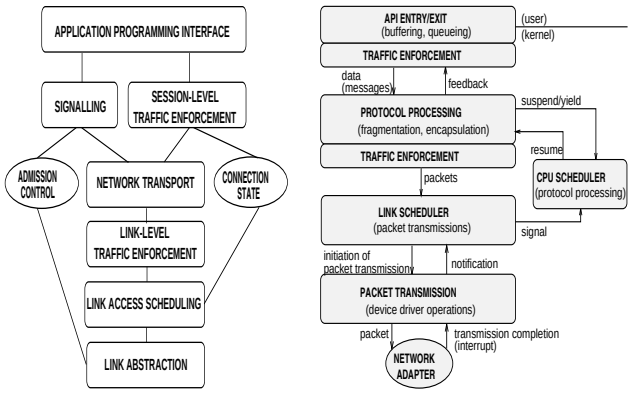
Distributed multimedia applications (e.g., video conferencing, video-on-demand, digital libraries) and distributed real-time command/control systems require certain quality-of-service (QoS) guarantees from the underlying network. QoS guarantees may be specified in terms of parameters such as the end-to-end delay, delay jitter, and bandwidth delivered on each connection; additional requirements regarding packet loss and in-order delivery can also be specified. 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.

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 [3], we focus on *real-time channels*, a paradigm for guaranteed-QoS communication services in packet-switched networks [16].

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

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.



(a) Overall architecture (b) Protocol processing

Figure 1. Desired software architecture.

We have implemented this architecture using a modified x-kernel 3.1 [14] communication executive exercising 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 implementation under different traffic loads, 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.

For end-to-end guarantees, resource management within the communication subsystem must be integrated with that for applications. The proposed architecture is directly applicable if a portion of the host processing capacity can be reserved for communication-related activities [21, 17]. 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 standalone 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. Integration of the proposed architecture with resource management for applications will be addressed in a forthcoming paper.

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 these requirements, 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 [3], 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.

Figure 1(a) illustrates a generic software architecture for guaranteed-QoS communication services at the host. The components constituting this architecture are as follows.
Application programming interface (API): The API must export routines 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 transport: Protocols are needed for unidirectional (reliable and unreliable) data transfers.
Traffic enforcement: This provides overload protection between established connections by forcing an application to conform to its traffic specification. This is required at the session level, and may also be required at the link level.
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 be bounded and predictable.

Assuming support for signalling, we focus 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. 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. Conformant data (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* with length smaller than a certain maximum (typically the MTU of the attached network). Additional computationally intensive services (e.g., coding, compression, or checksums) may also be performed during protocol processing. QoS-sensitive allocation of processing bandwidth necessitates multiplexing 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 attributes such as frame size for multimedia applications, the same for best-effort traffic may not be known *a priori*. Thus, mechanisms to suspend and resume protocol processing on a connection are needed. Protocol processing on a connection may also need to be suspended if it has no available packet buffers.

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. It should not be unduly penalized by non-conformant real-time traffic, especially under work-conserving processing.

2.2. Real-time channels

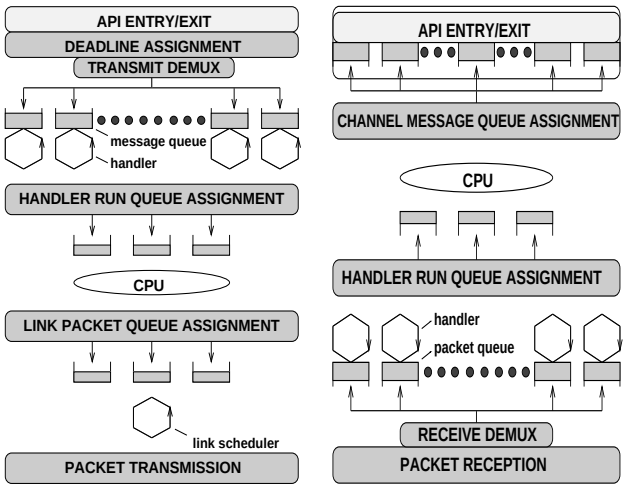
Several models have been proposed for guaranteed-QoS communication in packet-switched networks [3]. While the architectural mechanisms proposed in this paper are applicable to most of the proposed models, we focus on *real-time channels* [9, 16]. A real-time channel is a simplex, fixed-route, virtual 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* [8, 2] 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 a 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 [16] for more details.

Resource Management: Admission control for real-time channels is provided by Algorithm `D_order` [16], which uses fixed-priority scheduling for computing the worst-case delay experienced by a channel at a link. Run-time link scheduling, on the other hand, is governed by a multi-class variation of the earliest-deadline-first (EDF) policy.

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. Further, resource management policies must maximize the number of connections accepted for



(a) Source host (b) Destination host

Figure 2. Proposed architecture.

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 [19], we have studied the extent to which these factors affect admissibility in the context of real-time channels.

3. A QoS-sensitive communication architecture

In the *process-per-message* model [23], a process or thread shepherds a message through the protocol stack. Besides eliminating extraneous context switches encountered in the *process-per-protocol* model [23], 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 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 becomes unpredictable, making worst-case estimates either overly conservative or impossible to provide.

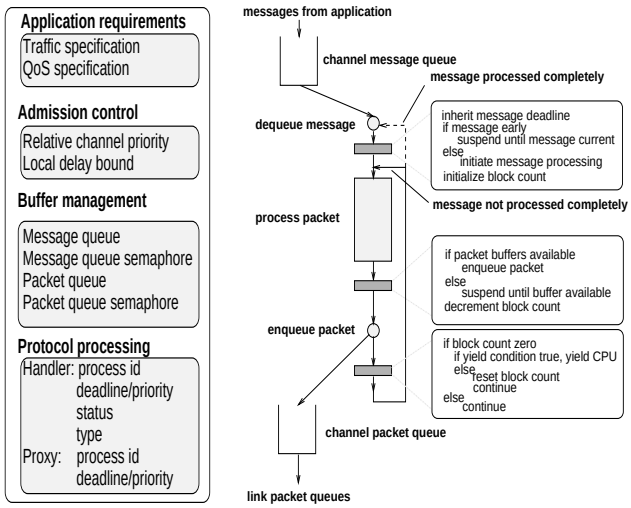
Since QoS guarantees are specified on a per-channel basis, it suffices to have a single thread coordinate access to resources for all messages on a given channel. We employ a *process-per-channel* model, which is a QoS-sensitive extension of the *process-per-connection* model [23]. In the process-per-channel 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 shown. Associated with each channel is a *message queue*, a FIFO queue of messages to be processed by the channel handler (at the source) or to be received by the application (at the destination). 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) or to be reassembled by the channel handler (at the destination).

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. 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, from where the application retrieves the message via the API's receive routine.

At intermediate nodes, the link scheduler relays arriving packets to the next node along the route. While we focus on transmission-side processing at the sending host, the following discussion also applies to reception-side processing.



(a) Channel state (b) Handler execution profile

Figure 3. Channel state and handler profile.

3.1. Salient features

Figure 3(a) illustrates a portion of the state associated with a channel at the host upon successful establishment. 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. 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. The packets created 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. Handler execution employs *cooperative preemption*, where 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 ensures that the handler is made runnable if the 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 [19], immediate preemption of an executing lower-priority handler results in expensive 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 per the link *horizon* [16].

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 message queue absorbs message bursts on a channel, preventing violations of B_{max} and R_{max}

$\mathcal{C}_{sw}$	context switch time	55 μs
$\mathcal{C}_{cm}$	cache miss penalty	90 μs
$\mathcal{C}_p^{1st}$	first-packet CPU processing cost	420 μs
$\mathcal{C}_p$	per-packet CPU processing cost	170 μs
$\mathcal{C}_l$	per-packet link scheduling cost	160 μs
$\mathcal{P}$	packets between preemption points	4 $pkts$
$\mathcal{S}$	maximum packet size	4K $bytes$
$\mathcal{L}_x(\mathcal{S})$	transmit time for packet size $\mathcal{S}$	245 μs

Table 1. Important system parameters.

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 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: 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. 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. In addition, it must account for the overlap between CPU processing and link transmission, and hence the relative bandwidths of the CPU and link. In a companion paper [19], we presented extensions to `D_order` to account for the above-mentioned factors. Table 1 lists the important system parameters used in the extensions.

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, and thus determine channel admissibility at the host [19]. Selection of $\mathcal{P}$ is governed by the architectural characteristics of the host CPU (Table 1). For a given host architecture, $\mathcal{P}$ is selected such that channel admissibility is maximized while delivering reasonable data transfer throughput. $\mathcal{S}$ is selected either using end-to-end transport protocol performance or host/adapter design characteristics. 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 $\mathcal{L}_x$ while delivering reasonable data transfer throughput.

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 packet size and the data transfer bandwidth available between host and adapter memories. Data transfer bandwidth itself is determined by host/adapter design features (pipelining, 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.

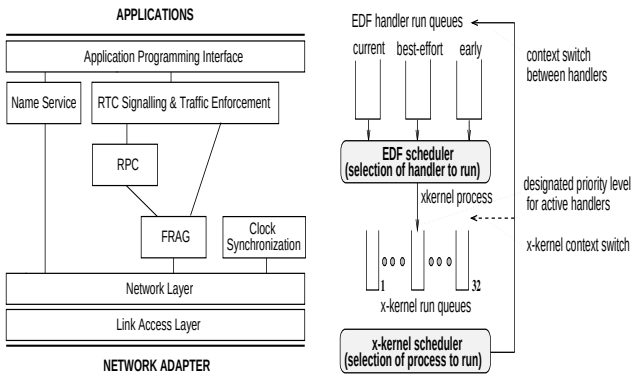
4. Implementation

We have implemented the proposed architecture using a modified x-kernel 3.1 communication executive [14] that exercises complete control over a 25 MHz Motorola 68040 CPU. 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 4(a)). The API exports routines for channel establishment, channel teardown, and data transfer; it also supports routines for best-effort data transfer. Network transport for signalling is provided by a

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



(a) x-kernel protocol stack (b) Layered EDF scheduler

Figure 4. Implementation environment.

Category	Available Policies
Protocol processing	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 (configurable number of packets between preemptions)
Link scheduling	multi-class earliest-deadline-first (options 1, 2 and 3)
Overload protection	block handler, decay handler deadline, enforce I_{min}, drop overflow messages

Table 2. Implementation policies.

(resource reservation) protocol layered on top of a remote procedure call (RPC) protocol derived from x-kernel's CHAN protocol. Network data transport 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 link scheduling and includes the network device driver. More details on the protocol stack are provided in [15].

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 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: For multi-class EDF scheduling, 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 (Figure 4(b)). When a channel handler or proxy is selected 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, picking another runnable handler, and 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 via a function call in the currently executing handler's context or in interrupt context (option 1), or by a dedicated process/thread (option 2), or by a new thread after each packet transmission (option 3). As demonstrated in [19], option 1 gives the best performance in terms of throughput and sensitivity of channel admissibility to $\mathcal{P}$ and $\mathcal{S}$; we focus on option 1 below.

The organization of link packet queues is similar to that of handler run queues, except that Q3 is used for early pack-

ets 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, current packets (if any) are transferred from Q3 to Q1, and the highest priority packet is selected for transmission from Q1 or Q2. If Q1 and Q2 are empty, a wakeup event is registered for the time when the packet at the head of Q3 becomes current. Scheduler processing is repeated when the network adapter indicates completion of packet transmission or the wakeup event for early packets expires.

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. On completion of a packet's transmission, its 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 a violation in M_{max}, the handler's priority/deadline 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 summarizes the available policies and options.

4.3. System parameterization

Table 1 lists the system parameters for our implementation. Selection of $\mathcal{P}$ and $\mathcal{S}$ is based on the tradeoff between available resources and channel admissibility [19]. The packet transmission time model presented in Section 3.3 requires that $\mathcal{C}_x$ and $\mathcal{B}_x$ be determined for a given network adapter and host architecture. An evaluation of the available networking hardware revealed significant performance-related deficiencies (poor data transfer throughput; high and unpredictable packet transmission time) [15]. These deficiences in the adapter design severely limited our ability to demonstrate the capabilities of our architecture. Given our 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* a network adapter by consuming $\mathcal{L}_x(s)$ time units for each packet being transmitted.

We have implemented such a device emulator, the *null device* [19], that can be configured to emulate a desired packet transmission time. We have used it to study a variety of tradeoffs,such as the effects of the relationship between CPU and link processing bandwidth, in the context of QoS-sensitive protocol processing [19]. 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, for an effective packet transmission bandwidth (for 4KB packets) of 16 MB/s.

5. 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 of Table 1 are used for the evaluation.

Ch	Traffic Specification				Deadline (ms)
	M_{max} (KB)	B_{max} (msgs)	R_{max} (KB/s)	I_{min} (ms)	
0, RT	60	12	1200	50	40
1, RT	60	8	2000	30	25
2, RT	60	1	2000	30	30
3, BE	60	10	varied	–	–

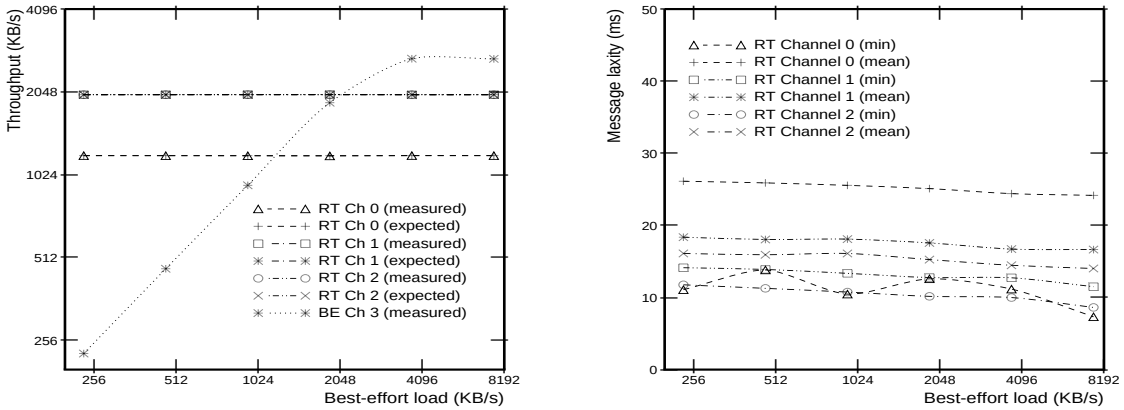
Table 3. Workload used for the evaluation.

5.1. Methodology and metrics

We chose a workload that stresses 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. 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. Rate violations 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 32K packet transmissions; only steady-state behavior is evaluated by ignoring the first 2K and last 2K packets.

All experiments reported here have traffic enforcement and CPU and link scheduling enabled. The following metrics measure per-channel performance. *Throughput* refers to the service received by each 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. *Packet drops* measures the number of packets dropped for both real-time and best-effort traffic. Deadline misses and packet drops account for QoS violations on individual channels.



(a) Throughput (b) Message laxity

Figure 5. Maintenance of QoS guarantees when traffic specifications are honored.

5.2. Efficacy of the proposed architecture

Figure 5 depicts the efficacy of the proposed architecture in maintaining QoS guarantees when all channels honor their traffic specifications. Figure 5(a) plots the throughput of 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 bandwidth; since no packets were dropped (not shown here) or late (Figure 5(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, best-effort traffic throughput increases linearly until system capacity is exceeded; real-time traffic (early and 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, without affecting real-time traffic.

Figure 5(b) plots the message laxity for real-time traffic, also as a function of offered best-effort load. 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.

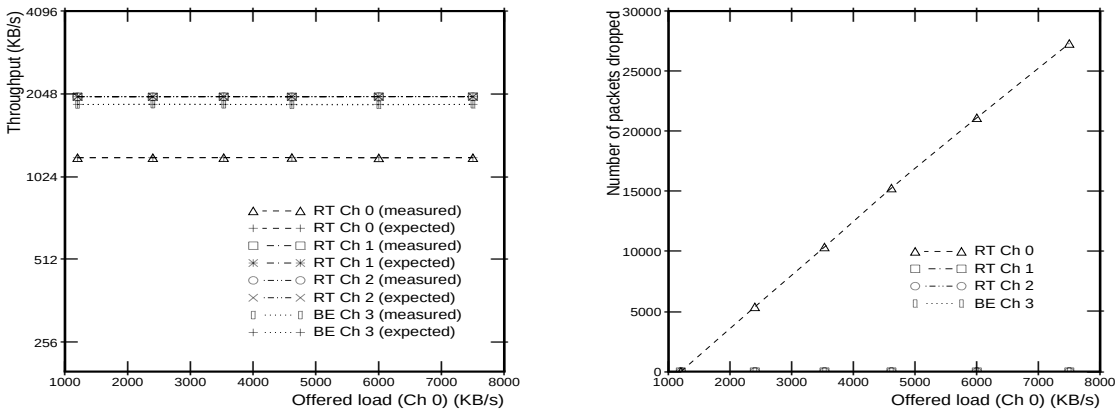
Figure 6 demonstrates the same behavior even in the presence of traffic specification violations by real-time channels. Channel 0 generates messages at a rate faster than specified while best-effort traffic is fixed at $\approx$ 1900 KB/s. In Figure 6(a), not only do well-behaved real-time channels and best-effort traffic receive their expected service, channel 0 also receives only its expected service. The laxity behavior is similar to that shown in Figure 5(b). No real-time packets miss deadlines, including those of channel 0. However, channel 0 overflows its message queue and drops excess messages (Figure 6(b)). 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 architectural features provided are sufficient to maintain QoS guarantees. The following results demonstrate that these features are also necessary. In Figure 7(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, making CPU scheduling 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. Subsequently, 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 channels are affected differently, and depends on the mix of channels. This behavior is exacerbated by an increase in the buffer space allocated to best-effort traffic; the best-effort handler now runs longer before blocking due to buffer overflow, thus increasing the window of non-preemptibility.

Figure 7(b) shows the effect of processing real-time messages with preemption only at message boundaries. Early handlers are allowed to execute in a work-conserving fashion but at a priority higher than best-effort traffic. Note that all real-time traffic is still being shaped since logical arrival time is enforced. Again, we observe significant deadline misses and packet drops for all real-time channels. 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.

Discussion: The above results demonstrate the need for cooperative preemption, in addition to traffic enforcement and CPU scheduling. While CPU and link scheduling were always enabled, real-time traffic was also shaped via 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. Cooperative preemption provides greater control over preemption



(a) Throughput (b) Number of packets dropped

Figure 6. Maintenance of QoS guarantees under violation of R_{max}.

points, which in turn improves utilization of resources that may be used concurrently. For example, a handler can initiate transmission on the link before yielding to any higher priority activity; arbitrary preemption may occur before the handler initiates transmission, thus idling the link.

6. Related work

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 detailed survey of the proposed techniques can be found in [3].

Similar issues are being examined for provision of integrated services on the Internet [7, 6]. The expected QoS requirements of applications and issues involved in sharing link bandwidth across multiple classes of traffic are explored in [24, 10]. The issues involved in providing QoS support in IP-over-ATM networks are also being explored [5, 22]. The Tenet protocol suite [4] provides real-time communication on wide-area networks (WANs), but does not incorporate protocol processing overheads into their network-level resource management policies. In particular, it does not provide QoS-sensitive protocol processing inside end hosts.

The need for scheduling protocol processing at priority levels consistent with the communicating application was highlighted in [1] and some implementation strategies demonstrated in [12]. Processor capacity reserves in Real-Time Mach [21] have been combined with user-level protocol processing [18] for predictable protocol processing inside hosts [17]. Operating system support for multimedia communication is explored in [25, 13]. However, no explicit support is provided for traffic enforcement or decoupling of protocol processing priority from application priority. The Path abstraction [11] provides a rich framework for development of real-time communication services.

7. Conclusions and future work

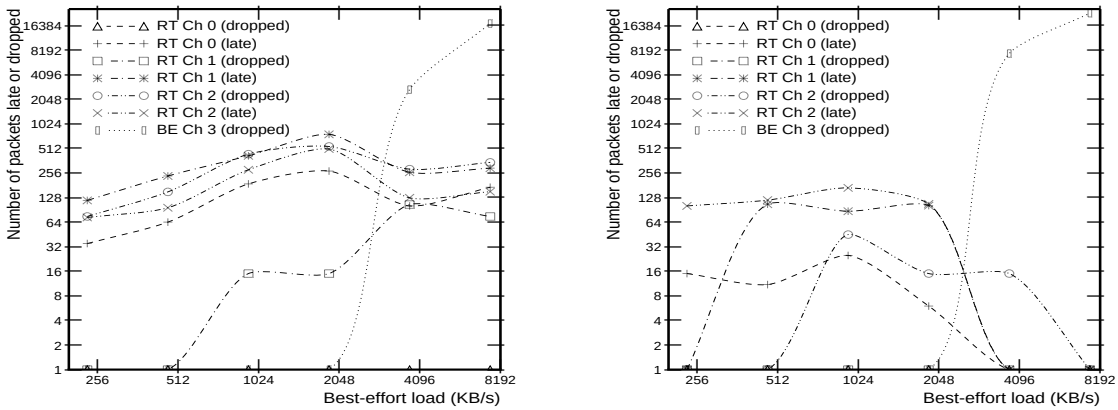
We have proposed and evaluated a QoS-sensitive communication subsystem architecture for end hosts that supports guaranteed-QoS connections. Using our implementation of real-time channels, we demonstrated the efficacy with which the architecture maintains QoS guarantees and delivers reasonable performance to best-effort traffic. While we evaluated the architecture for a relatively lightweight stack, such support would be necessary if computationally intensive services such as coding, compression, or checksums are added to the protocol stack. The usefulness of the features also depends on the relative bandwidths of the CPU and the link. The proposed architectural features are independent of our platform, and are generally applicable.

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. The architecture can also be extended to networks providing explicit support for QoS guarantees, such as ATM.

We are now extending the null device into a sophisticated network device emulator providing link bandwidth management, 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 are currently implementing the proposed architecture in OSF Mach-RT, a microkernel-based uniprocessor real-time operating system. Finally, we have extended this architecture to shared-memory multiprocessor multimedia servers [20].

References

[1] D. P. Anderson, L. Delgrossi, and R. G. Herrtwich. Structure and scheduling in real-time protocol implementations. Tech-



(a) Non-preemptive best-effort processing

(b) Non-preemptive real-time processing

Figure 7. Violation of QoS guarantees with cooperative preemption disabled.

nical Report TR–90–021, International Computer Science Institute, Berkeley, June 1990.

[2] 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*, pages 54–61, 1990.

[3] C. M. Aras, J. F. Kurose, D. S. Reeves, and H. Schulzrinne. Real-time communication in packet-switched networks. *Proc. of the IEEE*, 82(1):122–139, January 1994.

[4] A. Banerjea, D. Ferrari, B. Mah, M. Moran, D. C. Verma, and H. Zhang. The Tenet real-time protocol suite: Design, implementation, and experiences. Technical Report TR-94-059, ICSI, Berkeley, CA, November 1994.

[5] 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.

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

[7] 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*, pages 14–26, August 1992.

[8] 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.

[9] D. Ferrari and D. C. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, SAC-8(3):368–379, April 1990.

[10] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Trans. Networking*, 3(4), August 1995.

[11] 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*, Feb. 1996.

[12] R. Govindan and D. P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proc. ACM Symp. on Oper. Sys. Principles*, pages 68–80, 1991.

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

[14] N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Trans. Software Engineering*, 17(1):1–13, January 1991.

[15] 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.

[16] D. D. Kandlur, K. G. Shin, and D. Ferrari. Real-time communication in multi-hop networks. *IEEE Trans. on Parallel and Distributed Systems*, 5(10):1044–1056, October 1994.

[17] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar. Predictable communication protocol processing in Real-Time Mach. In *Proc. of 2nd Real-Time Tech. and Appl. Symp.*, June 1996.

[18] C. Maeda and B. N. Bershad. Protocol service decomposition for high-performance networking. In *Proc. ACM Symp. on Oper. Sys. Principles*, pages 244–255, December 1993.

[19] A. Mehra, A. Indiresan, and K. Shin. Resource management for real-time communication: Making theory meet practice. In *Proc. of 2nd Real-Time Tech. and Appl. Symp.*, June 1996.

[20] A. Mehra and K. Shin. QoS-sensitive protocol processing in shared-memory multiprocessor multimedia servers. In *Proc. of 3rd IEEE Workshop on Arch. and Impl. of High-Perf. Comm. Subsystems*, pages 163–169, Aug. 1995.

[21] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. In *Proc. of IEEE Intl. Conf. on Multimedia Comp. and Systems*, May 1994.

[22] 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.

[23] D. C. Schmidt and T. Suda. Transport system architecture services for high-performance communications systems. *IEEE Journal on Selected Areas in Communications*, 11(4):489–506, May 1993.

[24] 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.

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