Design and Evaluation of a QoS-Sensitive Communication Subsystem Architecture

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

There are a growing number of real-time applications (e.g., real-time controls, and audio/video conferencing) that require certain quality of service (QoS) guarantees from the underlying communication subsystem. The communication subsystem (host as well as network) must support real-time communication services that can be used to provide the required QoS of these applications, while providing reasonably good performance for best-effort traffic. In this paper 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 various 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 for protocol processing 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 CPU scheduling and overload protection. We evaluate the implementation to demonstrate that this architecture maintains QoS guarantees while adhering to the stated design goals. The evaluation also demonstrates convincingly the need for specific features and policies provided in the architecture.

Key Words — Real-time communication, QoS-sensitive protocol processing, traffic enforcement, 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.

Figure 1 illustrates a generic communication software architecture for real-time communication services at the host. The components constituting this architecture are briefly discussed below.
Application programming interface (API): The API must export routines that an application can use to set up and teardown guaranteed-QoS connections (with appropriate traffic specifications and associated QoS requirements), and perform data transfer on the established connections.
Signalling and admission control: A signalling protocol is required to establish/tear down guaranteed-QoS connections. The communication subsystem must keep track of communication resources (CPU and link bandwidth, buffers) and perform admission control to admit new connections.
Network data transport: The communication subsystem must provide protocols for request-response and unidirectional (reliable and unreliable) data transfers, including fragmentation.
Traffic enforcement: Modules for traffic enforcement are necessary to provide overload protection between established connections.
Link access scheduling and link abstraction: Link bandwidth must be managed such that all active connections receive their associated QoS. This necessitates abstracting the link in terms of transmission delay and bandwidth, and scheduling all outgoing data on real-time connections for network access. The minimum requirement from the underlying network for provision of QoS guarantees is

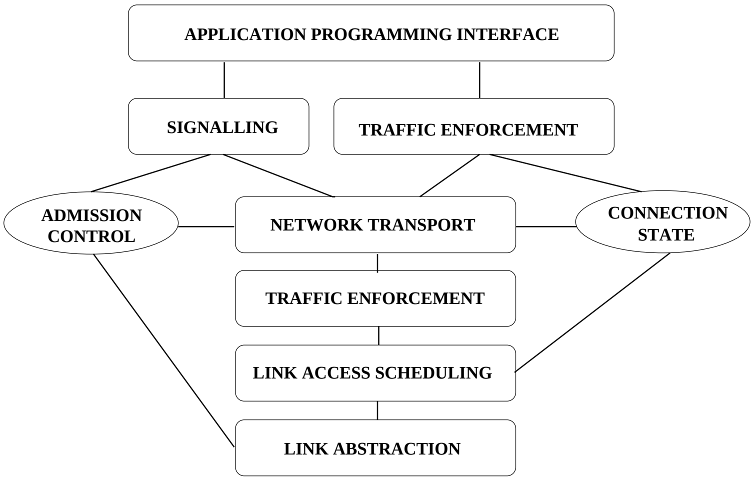


Figure 1: Desired software architecture.

that the packet transmission time be bounded and predictable.
Assuming support for signalling, our primary focus in this paper is on the components involved in data transfer, namely, traffic enforcement, network data transport (including CPU bandwidth management for protocol processing), and link access scheduling at the host.

Specifically, we propose and evaluate a QoS-sensitive communication subsystem architecture for guaranteed-QoS connections.[1] The proposed architecture is centered around the concept of a *real-time channel*, a paradigm for real-time communication services in packet-switched networks [1]. Our focus is on the run-time traffic management architecture used within the communication subsystem to satisfy the QoS requirements of all channels, while facilitating coexistence of best-effort traffic (for which no QoS guarantees are given) without undue degradation in performance. This model of real-time channels is similar to other proposals for guaranteed-QoS connections [2].

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. Associated with each channel is a first-in-first-out (FIFO)[2] queue of messages to be processed by the channel handler (at the source host) or received by the application (at the destination host). QoS guarantees on each channel are provided via appropriate deadline-based CPU scheduling of channel handlers and link scheduling of packet transmissions. While scheduling handlers and packets provides traffic isolation between channels, per-channel traffic is policed via 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 their 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.

We have implemented the proposed architecture using a communication executive derived from x-kernel 3.1 [3] 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. Using this implementation, we evaluate the proposed architecture, under varying degrees of traffic load, and demonstrate the efficacy with which it

[1]For end-to-end guarantees, resource management within the communication subsystem must be integrated with that for applications; the issues involved in such integration are beyond the scope of this paper.
[2]FIFO service *within* a channel together with priority service *between* channels provides QoS guarantees.

maintaining QoS guarantees on real-time channels and provides fair performance for best-effort traffic, even in the presence of ill-behaved real-time channels.

The rest of the paper is organized as follows. A brief description of the real-time channel model of guaranteed-QoS communication is provided in Section 2, which also discusses the requirements for QoS-sensitive protocol processing in the context of this model. Section 3 motivates and presents a communication subsystem architecture realizing QoS-sensitive protocol processing and packet transmission, and Section 4 describes its implementation. Section 5 experimentally evaluates the efficacy of the proposed architecture. Our results are compared and contrasted with other related work in Section 6. Finally, Section 7 concludes the paper.

2 Guaranteed-QoS Communication Using Real-Time Channels

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. The data flow on real-time channels is unidirectional, from source to sink via intermediate nodes, with successive messages delivered in the order they were generated. 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, data transfer on real-time channels has unreliable-datagram semantics with no acknowledgements and retransmissions.

An application requests a real-time channel by specifying 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 any kind of performance 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. A local bound, called the link delay deadline in [1], which is the worst-case transit delay seen by a packet on this channel plus a certain slack, is also assigned at each node. Once the channel is successfully established, the communication subsystem and the network maintain QoS guarantees via resource management and traffic enforcement policies. When the application requests that the channel be destroyed, all resources allocated for the channel are released by the network and the communication subsystems at the source and destination hosts.

2.1 Traffic Model and QoS

The traffic generation model in real-time channels is based on a *linear bounded arrival process* [4, 5], which is characterized by three parameters: maximum message size (M_{max} bytes), maximum message rate (R_{max} messages/second), and maximum burst size (B_{max} messages). In any interval of length δ, the number of messages generated is bounded by $B_{max} + \delta \cdot R_{max}$. Message generation rate is bounded by R_{max}, and its reciprocal, I_{min}, is the minimum inter-generation time between messages. The burst parameter, B_{max}, bounds the allowed short-term variation in message generation, and partially determines the buffer space requirement of the real-time channel. To ensure that a real-time channel does not use more resources than it reserved at the expense of other channels' QoS guarantees, this model uses the notion of *logical arrival time* to enforce a minimum separation I_{min} between messages on the same real-time channel. The logical arrival time, $\ell(m)$, of a message

m is defined as:

$$\begin{aligned}\ell(m_0) &= t_0 \\ \ell(m_i) &= max\{(\ell(m_{i-1}) + I_{min}),\ t_i\},\end{aligned}$$

where t_i is the actual generation time of message m_i; $\ell(m_i)$ is the time at which m_i would have arrived (generated) if the maximum message rate constraint was strictly obeyed.

The QoS of a real-time channel is specified in terms of a deterministic, worst-case bound on the end-to-end delay experienced by a message of the channel. If d is the desired end-to-end delay bound for a channel, message m_i generated at the source is guaranteed to be delivered at the sink by time $\ell(m_i) + d$. More details on the real-time channel model can be found in [1].

2.2 Resource Management

There are two related aspects to resource management for guaranteed-QoS communication [2]: admission control and run-time resource scheduling. Real-time channels employ the following admission control procedure and link scheduling policy.

Signalling and Admission Control: Admission control for real-time channels is provided by algorithm `D_order` [1], 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 priority assigned to the new channel depends upon the characteristics of the other channels going through the link. 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; the permissible message delay is split proportionally among the different links. The priority assignment algorithm ensures that the new channel does not affect the QoS promised to existing channels. `D_order` assumes that for all channels, the worst-case delay at each link for any channel does not exceed I_{min}; the total end–to–end delay of the channel, however, can exceed I_{min}.

Link Scheduling: Link bandwidth is allocated via (non-preemptive) scheduling of packet transmissions, assuming bounded and predictable packet transmission time on the link. At the source host (and intermediate nodes), the link scheduler maintains three packet queues: Q1 (ordered by deadline) for current real-time packets, Q2 (FIFO) for best-effort packets, and Q3 (ordered by logical arrival time) for early real-time packets. Q3 packets are transferred to Q1 as they become current [1]. Q1 gets the highest transmission priority, followed by Q2 and Q3 in that order; this ensures that early real-time traffic does not unduly penalize best-effort traffic.

2.3 Requirements for QoS-Sensitive Protocol Processing

In order to incorporate management of CPU bandwidth in this model, we consider the following three main requirements for QoS-sensitive protocol processing.

Maintaining Per-Channel QoS Guarantees**:** In the presence of multiple real-time channels and best-effort traffic, protocol-processing bandwidth must be consumed in a QoS-sensitive fashion, i.e., in an order consistent with the QoS requirements of the active channels. Assuming a process-per-message model [3], first-in-first-out (FIFO) scheduling of protocol threads (or message handlers) does not meet these requirements. QoS-sensitive CPU scheduling of protocol threads, with bounded latency in acquiring the CPU for protocol processing, is necessary to maintain QoS guarantees. While FIFO scheduling is inherently QoS-insensitive, CPU access latency can be exacerbated by non-preemptive execution of protocol threads, which makes the CPU access latency a function of the message payload (i.e., the processing needed by a message). The payload associated with a message is determined by the number of packets constituting the message and the per-packet processing cost. Alternative preemptable scheduling mechanisms are therefore needed for QoS-sensitive protocol processing on each channel. Any delays in obtaining the CPU for protocol processing (at the source and destination nodes) must be accounted for in the admission control and traffic management functions.

***Overload Protection via Per-Channel Traffic Enforcement*:** The communication subsystem must police and/or shape per-channel traffic such that traffic violations on a channel do not affect other channels. A channel violating its traffic specification should not be allowed to consume protocol-processing bandwidth over and above that reserved for it. Messages violating the traffic specification should be prevented from consuming protocol processing bandwidth if the generated packets would overflow the link packet queues. Similarly, early messages (due to bursts) should not penalize best-effort traffic and on-time messages on other real-time channels. CPU utilization should be maximized such that early messages can consume protocol processing bandwidth if there are no best-effort or on-time real-time messages to process. In contrast, the link scheduler must not transmit early packets unless sufficient network resources are available downstream.

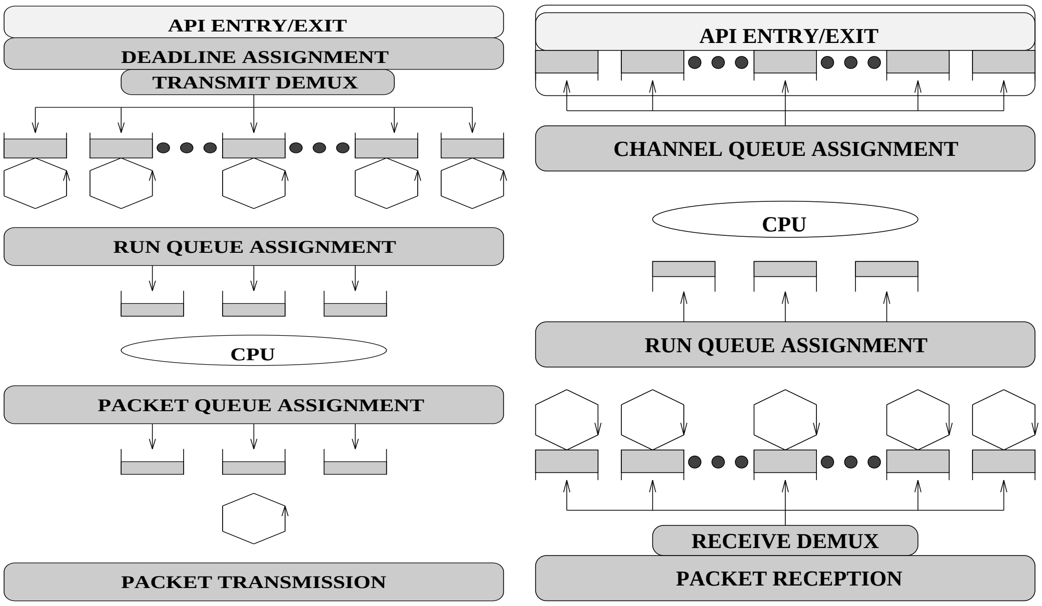
***Fairness to Best-Effort Traffic*:** Best-effort traffic includes data transfers that are given best-effort service (such as those transported by conventional protocols like TCP and UDP) and the signalling required for real-time channels. The communication subsystem should facilitate coexistence of best-effort traffic with real-time traffic without significant degradation in performance.

3 Architecture for QoS-Sensitive Protocol Processing

In a *process-per-message* protocol-processing model [6], a process or thread shepherds a message through the protocol stack. Not only does it eliminate extraneous context switches encountered in the *process-per-protocol* model [6], it also facilitates protocol processing for messages to be scheduled according to a variety of policies, as opposed to the software-interrupt level processing in BSD Unix. While the process-per-message model suffices for best-effort messages, it 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.

Since QoS guarantees are specified on a per-channel basis, it suffices to have a single active thread that coordinates 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 [6]. We now present the proposed QoS-sensitive communication subsystem architecture that uses the process-



(a) Source host (b) Destination host

Figure 2: Proposed communication subsystem architecture.

per-channel model.

3.1 Transmission and Reception

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. 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. 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).

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. The channel handler is assigned to one of three CPU run queues for execution. It executes in an infinite loop, dequeueing messages one at a time from the message queue and performing protocol processing (including fragmentation). The packets generated by each handler are inserted in one of three (outbound) link packet queues, again based on channel type and traffic generation, to be transmitted by the link scheduler at the appropriate time.

Reception-Side Processing: In Figure 2(b), a received packet is demultiplexed to the corresponding channel's *packet queue*, a FIFO queue of packets waiting to be processed by the handler, for subsequent processing and reassembly. The channel handler executes in an infinite loop, waiting for packets to arrive in the channel packet queue. As in transmission-side processing, channel handlers are assigned to one of three CPU run queues for execution. 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.

3.2 Salient Features

Channel handlers are classified into three classes: current real-time, early real-time, and best-effort. A current real-time handler is one that is processing an *on-time* message (obeying the channel's rate specification), while an early real-time handler is one that is processing an *early* message (violating the channel's rate specification). A best-effort handler is one that is processing messages on a best-effort channel. Channel handlers must be scheduled for execution such that early real-time traffic does not affect the performance of current real-time and best-effort traffic. Note that with per-channel handlers, the scheduling overhead is only a function of the number of *active* channels.

Maintenance of QoS guarantees: Per-channel QoS guarantees are provided via appropriate preemptive CPU scheduling of channel handlers and link scheduling of packet transmissions as described in Section 2. While link scheduling is deadline-based, CPU scheduling can be deadline-based (using message deadlines) or priority-based (using relative channel priorities). Channel handlers are assigned to one of three run queues based on their type (best-effort or real-time) and traffic generation (early or on-time). The relative priority assignment for the handler run queues is similar to that for the link packet queues, with on-time real-time traffic getting the highest protocol processing priority. We consider non-work-conserving link scheduling to avoid stressing resources at downstream hosts; in general, real-time channels allow the link to "work ahead" in a limited fashion, as determined by the link *horizon* [1].

Provision of QoS guarantees necessitates bounded delays in obtaining the CPU for protocol processing. Immediate preemption of an executing lower-priority handler can result in significant overheads due to context switches and cache misses. Accordingly, 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 up to a certain number of packets; this amortizes preemption overheads over the processing of several packets. Section 3.3 presents the modifications made to `D_order` to account for CPU preemption delays and overheads.

Overload protection: Per-channel traffic enforcement is performed when new messages are inserted into the message queue, and when packets are inserted into the link packet queues. The per-channel message queue serves to absorb message bursts on the 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 B_{max}. Under work-conserving processing, it is possible that the packets generated by a handler cannot be accommodated in the link packet queues because all the link 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 con-

Symbol	Description
$\mathcal{C}^{xk}_{switch}$	time to switch contexts between x-kernel threads (original)
$\mathcal{C}^{edf}_{switch}$	time to switch contexts between channel handlers (EDF)
$\mathcal{C}_{cache}$	penalty due to cache misses resulting from a context switch
$\mathcal{P}$	number of packets processed between preemption points
$\mathcal{C}_{prot}$	per-packet protocol processing cost
$\mathcal{C}_{link}$	per-packet link scheduling cost
$\mathcal{S}$	maximum packet size in bytes
$\mathcal{L}_{xmit}(s)$	packet transmission time for packet size s

Table 1: Important system parameters in the proposed architecture

suming 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.

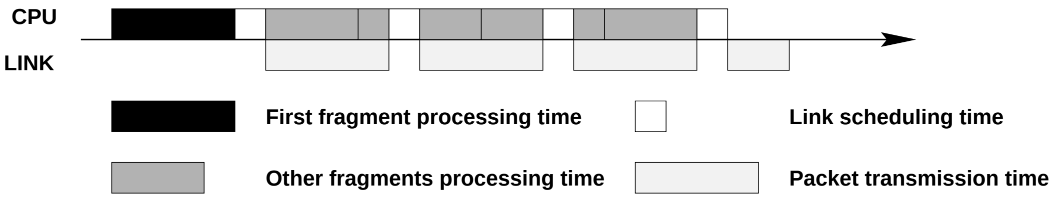
Fairness to best-effort traffic: To ensure that best-effort traffic is not unduly penalized by ill-behaved real-time channels, protocol processing of an early message can be delayed (by buffering it) till it becomes current, in which case protocol processing is non-work-conserving; the above discussion corresponds to this option. Alternately, protocol processing can be work-conserving, with CPU scheduling mechanisms ensuring QoS-sensitive allocation of CPU bandwidth to channel handlers. In this case, best-effort traffic is given processing and transmission priority over early real-time channels. Work-conserving protocol processing can potentially improve CPU utilization, since the CPU does not idle when there is work to do. 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 early real-time traffic, if any, assuming there is no pending best-effort traffic. This can free up protocol processing capacity for subsequent messages. In the absence of best-effort traffic, work-conserving protocol processing can also improve the average QoS delivered to real-time channels, especially if link scheduling is work-conserving.

3.3 Accounting for CPU Preemption Delays and Overheads

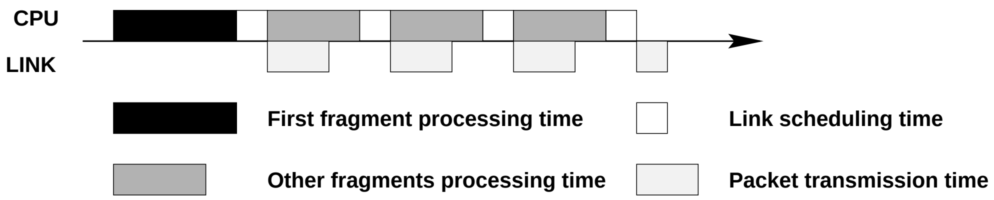
Preemption overheads effectively reduce usable resource capacity, lowering channel admissibility at the host.[3] In a companion paper [7], we conducted an in-depth study of the tradeoff between usable resource capacity and channel admissibility. Below we present how `D_order` was modified to account for preemption overheads and the window of non-preemptibility in handler execution.

Suppose a handler processes up to $\mathcal{P}$ packets before yielding the CPU to a waiting higher-priority channel handler. Further, suppose that the packet size is $\mathcal{S}$, per-packet protocol processing

[3] Channel admissibility (number of channels admitted) here is a local metric, assuming perfect admissibility at other network elements.



(a) $\mathcal{C}_{prot} < \mathcal{L}_{xmit}(\mathcal{S})$



(b) $\mathcal{C}_{prot} \geq \mathcal{L}_{xmit}(\mathcal{S})$

Figure 3: Protocol processing and link transmission overlap

cost is $\mathcal{C}_{prot}$, context switch overhead is $\mathcal{C}^{edf}_{switch}$, cache miss penalty due to context switches is $\mathcal{C}_{cache}$, and packet transmission time on the link is $\mathcal{L}_{xmit}(s)$ for packet size s. These system parameters are listed in Table 1, which distinguishes the context switch overhead between x-kernel threads ($\mathcal{C}^{xk}_{switch}$) from the context switch overhead between channel handlers using EDF scheduling policy for CPU access ($\mathcal{C}^{edf}_{switch}$). For each new channel to be admitted, `D_order` computes the *message service time*, the worst-case time for which the CPU and link must be allocated to the channel for processing a message, and the *wait time*, the worst-case time spent waiting for a lower-priority handler to relinquish the CPU and link. For the following discussion, Figure 3 illustrates the processing and transmission of packets, where the link scheduler is invoked a function call either in the currently executing handler or in interrupt context (after packet transmission) (option 1 of Section 4.4).

Consider a message of size $\mathcal{M}$ bytes that has $\mathcal{N}_{pkts} = \lceil \frac{\mathcal{M}}{\mathcal{S}} \rceil$ packets, with $\mathcal{N}_{pkts} - 1$ 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 $\mathcal{L}_{xmit}(\mathcal{S})$ for all but the last packet and $\mathcal{L}_{xmit}(\mathcal{S}^{last})$ for the last packet. The protocol processing cost for the first packet is $\mathcal{C}^{first}_{prot}$ while subsequent fragments each incur a lower cost $\mathcal{C}_{prot}$. $\mathcal{C}^{first}_{prot}$ includes the fixed cost of obtaining the message for processing, timestamp overhead, and the cost of preparing the first packet.[4] Both $\mathcal{C}^{first}_{prot}$ and $\mathcal{C}_{prot}$ include the cost of network-level encapsulation.

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

$$\mathcal{T}_{service} = \begin{cases} \mathcal{C}^{first}_{prot} + \mathcal{L}^{msg}_{xmit} + \mathcal{C}^{msg}_{link} + \mathcal{C}_{preempt} & \text{if } \mathcal{C}_{prot} < \mathcal{L}_{xmit}(\mathcal{S}) \\ \mathcal{C}^{msg}_{prot} + \mathcal{C}^{msg}_{link} + \mathcal{L}_{xmit}(\mathcal{S}^{last}) + \mathcal{C}_{preempt} & \text{otherwise} \end{cases}$$

where $\mathcal{C}^{msg}_{prot}$ represents the total protocol processing cost for the message and is given by $\mathcal{C}^{msg}_{prot} = \mathcal{C}^{first}_{prot} + (\mathcal{N}_{pkts} - 1)\mathcal{C}_{prot}$, $\mathcal{C}_{preempt}$ represents the total cost of preemption during the processing of the message and is given by $\mathcal{C}_{preempt} = (\frac{\mathcal{N}_{pkts}-1}{\mathcal{P}})(\mathcal{C}^{edf}_{switch} + \mathcal{C}_{cache})$. $\mathcal{C}^{msg}_{link} = \mathcal{N}_{pkts}\mathcal{C}_{link}$ represents

[4] Our fragmentation protocol traverses a slower path for messages larger than S bytes and hence the higher processing cost for the first packet.

the total link scheduling overhead for the message, and $\mathcal{L}_{xmit}^{msg} = (\mathcal{N}_{pkts} - 1)\mathcal{L}_{xmit}(\mathcal{S}) + \mathcal{L}_{xmit}(\mathcal{S}^{last})$ represents the total link transmission time for the message.

If $\mathcal{C}_{prot} < \mathcal{L}_{xmit}(\mathcal{S})$, there is at least the cost of processing the first packet. Since the link transmission time dominates the time to process subsequent packets (see Figure 3(a)), the message service time is determined by the link transmission time for the message and the total link scheduling and preemption overheads incurred. If $\mathcal{C}_{prot} \geq \mathcal{L}_{xmit}(\mathcal{S})$ (Figure 3(b)), however, the 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

$$\mathcal{T}_{wait}^{cpu} = \mathcal{C}_{block}^{first} + \lceil \frac{\mathcal{C}_{block}^{first}}{\mathcal{L}_{xmit}(\mathcal{S})} \rceil \mathcal{C}_{link} + \mathcal{C}_{cache} + \mathcal{C}_{switch}^{edf},$$

where $\mathcal{C}_{block}^{first}$ is the worst-case processing time for a block of up to $\mathcal{P}$ packets and is given by $\mathcal{C}_{block}^{first} = \mathcal{C}_{prot}^{first} + (\max(\mathcal{N}_{pkts}, \mathcal{P}) - 1)\mathcal{C}_{prot}$. During this time $\mathcal{C}_{block}^{first}$, up to $\lceil \frac{\mathcal{C}_{block}^{first}}{\mathcal{L}_{xmit}(\mathcal{S})} \rceil$ packets could complete transmission. Similarly, the worst-case link wait time is $\mathcal{T}_{wait}^{link} = \mathcal{L}_{xmit}(\mathcal{S})$. This is because a lower-priority transmission could be started just before the currently-executing handler makes a packet ready for transmission. The packet generated by the handler will have to wait for the lower-priority packet to complete transmission. Thus, the total message wait time is $\mathcal{T}_{wait} = \mathcal{T}_{wait}^{cpu} + \mathcal{T}_{wait}^{link}$.

3.4 Determination of $\mathcal{P}$, $\mathcal{S}$, and $\mathcal{L}_{xmit}$

$\mathcal{P}$ 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.

Packets between preemptions: Selection of $\mathcal{P}$ is governed by the architectural characteristics of the host CPU, as captured by the parameters listed in Table 1. 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 reduce the number of channels admitted for service. For a given host architecture and system parameters listed in Table 1, $\mathcal{P}$ can be selected such that channel admissibility is maximized while delivering reasonable data transfer throughput.

Packet size: Selection of $\mathcal{S}$ can be done using either 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 $\mathcal{L}_{xmit}$ (see below) while delivering reasonable data transfer throughput. Note that in channels spanning heterogeneous networks, $\mathcal{S}$ can be different at each hop. This is acceptable 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, $\mathcal{L}_{xmit}(s)$ depends primarily on two aspects, namely, 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 $\mathcal{C}_{startup}$ is the overhead to initiate transmission on an adapter feeding a link of bandwidth $\mathcal{B}_{link}$ bytes/second, the packet transmission time can be approximated as

$$\mathcal{L}_{xmit}(s) = \mathcal{C}_{startup} + \frac{s}{\min(\mathcal{B}_{link}, \mathcal{B}_{xfer})},$$

where $\mathcal{B}_{xfer}$ is the data transfer bandwidth available to/from host memory. $\mathcal{B}_{xfer}$ is determined by a variety of factors including the mode (direct memory access (DMA) or programmed IO) and efficiency of data transfer, and the degree to which packet transmissions can be pipelined on the adapter. $\mathcal{C}_{startup}$ includes the cost of setting up any DMA transfer operations, if any. Our experience with adapter design and the implications for packet transmission time are highlighted in [8].

4 Implementation

We have implemented the proposed architecture using a communication executive derived from x-kernel (v3.1) [3] 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.[5] 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. The API implements routines for channel establishment, channel teardown, and data transfer; it also supports routines for best-effort data transfer. Referring to Figure 1, network transport for signalling is provided by a resource reservation protocol layered on top of a remote procedure call (RPC) protocol. Network transport for data transfer 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 includes the network device driver and support for link scheduling.

Our choice of protocols was based on the perceived requirements for real-time channels. 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 heavy-weight reliable byte-stream protocol while UDP does not fragment outbound messages or support message sequencing. Moreover, IP is a connectionless protocol and

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

would have required either modifications to make it connection-oriented or mechanisms to classify incoming and outgoing packets. More details on the protocol stack are provided in [8].

4.2 Realizing Process-Per-Channel Model

On successful establishment, a channel is allocated a channel handler and space for its message and packet queues. 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, protocol processing priority or deadline, and an event identifier corresponding to the most recent x-kernel event registered by the handler. In addition, two semaphores are allocated to each channel handler, one to synchronize with message insertions into the channel's message queue, and other to synchronize with availability of buffers for outbound packets at the link. Best-effort traffic is also handled by the corresponding channel handlers. The execution priority of a handler can either be derived dynamically from the deadline of the message it processes, or statically from that computed by `D_order` [1]. 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 message manipulation routines in x-kernel to associate the message deadline with each packet, enabling the link scheduler to correctly order packet transmissions. Each outgoing packet carries a global channel identifier, allowing efficient packet demultiplexing at a receiving node.

4.3 Multi-Class EDF Scheduling of Channel Handlers

Two policies are available for scheduling channel handlers on the CPU: (i) multi-class earliest-deadline-first (EDF) scheduling and (ii) fixed-priority scheduling with 32 priority levels; the following discussion applies to (i).[6] Three distinct run queues are maintained for channel handlers, one for each of the three classes mentioned above, 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. For non-work-conserving protocol processing, a channel handler is blocked till the message it has dequeued for processing becomes current. For work-conserving protocol processing, in addition to blocking the handler as before, a *channel proxy* is created on its behalf and added to Q3. A channel proxy is an x-kernel 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. Since Q3 has the lowest priority, proxies do not interfere with the execution of channel handlers.

This scheduling policy is implemented by layering the multi-class EDF scheduler above the x-kernel scheduler. 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

[6]Fixed-priority scheduling maps channel priorities to 32 priority levels, to provide lower scheduling overheads at the expense of increased inter-channel interference.

```
1. Mark the link as busy.
2. Examine Q3; transfer 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 4: Processing done by the link scheduler.

between two channel handlers is realized by first enqueueing the currently-active handler in the EDF run queues and picking another that is eligible to run, before invoking the normal x-kernel code to switch process contexts.

Cooperative preemption of handlers provides a reasonable mechanism to bound CPU access delays while improving utilization, especially when all handlers execute within a single (kernel) address space. To support cooperative preemption with EDF scheduling of channel handlers, 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.

4.4 Link Scheduling of Packet Transmissions

The implementation can be configured to perform link scheduling as per one of three options:

1. link scheduler is invoked as a function call either in the context of the currently executing handler or in interrupt context (after packet transmission),
2. link scheduler is realized as a dedicated process/thread, and
3. link scheduling is performed by a new thread created after each packet transmission.

As demonstrated in [7], 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.

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 4 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.

4.5 Per-Channel Traffic Enforcement

A channel's message queue is initialized to B_{max}; messages overflowing the queue are dropped. Blocking of handlers under buffer overflow is achieved by associating a counting *buffer semaphore* with each channel. The 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 buffer 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 priority (or deadline) of the handler is degraded in proportion to the extra packets in its payload, so that further consumption of protocol-processing bandwidth does not affect other well-behaved channels. Table 2(a)

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
$\mathcal{C}^{xk}_{switch}$	20	μs
$\mathcal{C}^{edf}_{switch}$	55	μs
$\mathcal{C}_{cache}$	90	μs
$\mathcal{C}^{first}_{prot}$	420	μs
$\mathcal{C}_{prot}$	170	μs
$\mathcal{C}_{link}$	160	μs
$\mathcal{P}$	4	*packets*
$\mathcal{S}$	4096	*bytes*
$\mathcal{L}_{xmit}(\mathcal{S})$	245	μs

(b) System parameters

Table 2: Policies and system parameters in the current implementation

summarizes the policies and options available in the implementation.

4.6 System Parameterization

We parameterized the protocol stack to determine the system parameters for our implementation. Selection of $\mathcal{P}$ and $\mathcal{S}$ is based on the tradeoff between available resource capacity and channel admissibility [7]. Table 2(b) summarizes the system parameter settings.

In order for the (simple) model of packet transmission time (presented in Section 3.4) to be useful, $\mathcal{C}_{startup}$ and $\mathcal{B}_{xfer}$ 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 real-time channel 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 [8]. While these deficiences were due to the design of the adapter, they severely limited our ability to demonstrate the capabilities of our architecture and implementation. It suffices to ensure that transmission of a packet of size s takes $\mathcal{L}_{xmit}(s)$ time units. This can be achieved by *emulating* the behavior of a network adapter such that $\mathcal{L}_{xmit}(s)$ 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 simply a thread that, once signalled, tracks time by consuming CPU resources for $\mathcal{L}_{xmit}$ 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 $\mathcal{L}_{xmit}$ time units (and hence completion of packet transmission) the emulator issues an interrupt to the host processor, similar to the mechanism employed in typical network adapters. The emulator allows us 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. However, it is not completely accurate since no packet data is actually transferred from host memory. Still, this does not affect the trends observed and performance comparisons reported here. We experimentally determined $\mathcal{C}_{startup}$ to be $\approx 40\mu s$. For the experiments, we select $\min(\mathcal{B}_{link}, \mathcal{B}_{xfer})$ 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.

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 4.4). Overload protection for packet buffer 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. The workload used for the evaluation is specified in Table 3.[7] Three real-time channels were established[8] 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 the linear bounded arrival process with bursts of up to B_{max} messages. This behavior is appropriately modified to create traffic specification violations. 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 has a duration corresponding to the transmission of a total of 32K packets. The first 2K and last 2K packets are ignored so that performance measurements are based on steady-state behavior.

The following metrics measuring per-channel performance are used for the evaluation.

Throughput: 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: the difference between the transmission deadline of a real-time message and the actual time that it completes transmission.

Deadline misses: the number of real-time packets missing deadlines.

Packet drops: the number of packets dropped for both real-time and best-effort traffic.

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 received by each real-time channel and best-effort traffic as a function of (offered) best-effort load. Several conclusions

[7] Similar results were obtained for experiments performed for a variety of workloads, including large number of channels with a wide variety of deadlines and traffic specifications.

[8] Note that channel establishment here is strictly local.

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

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 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, 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.

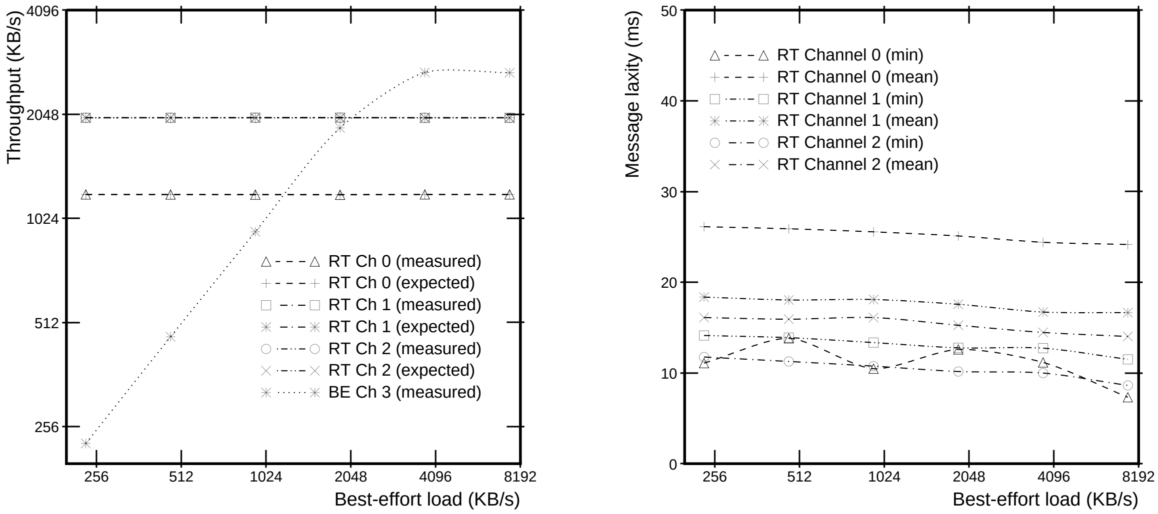
Figure 5(b) plots the message laxity for real-time traffic, also as a function of offered best-effort load. As can be seen, no messages miss their deadlines, since minimum laxity is non-negative for all channels. In addition, the mean laxity for real-time messages is largely unaffected by an increase in best-effort load, regardless of whether the channel is bursty or not.

Figure 6 demonstrates the same performance behavior even in the presence of traffic specification violations by real-time channels. In this case, channel 0 generates messages at a rate faster than specified. Best-effort traffic is fixed at $\approx$ 1900 KB/s. 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 5(b). No real-time packets miss deadlines, including those of channel 0. However, as can be from Figure 6(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 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. 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



(a) Throughput (b) Message laxity

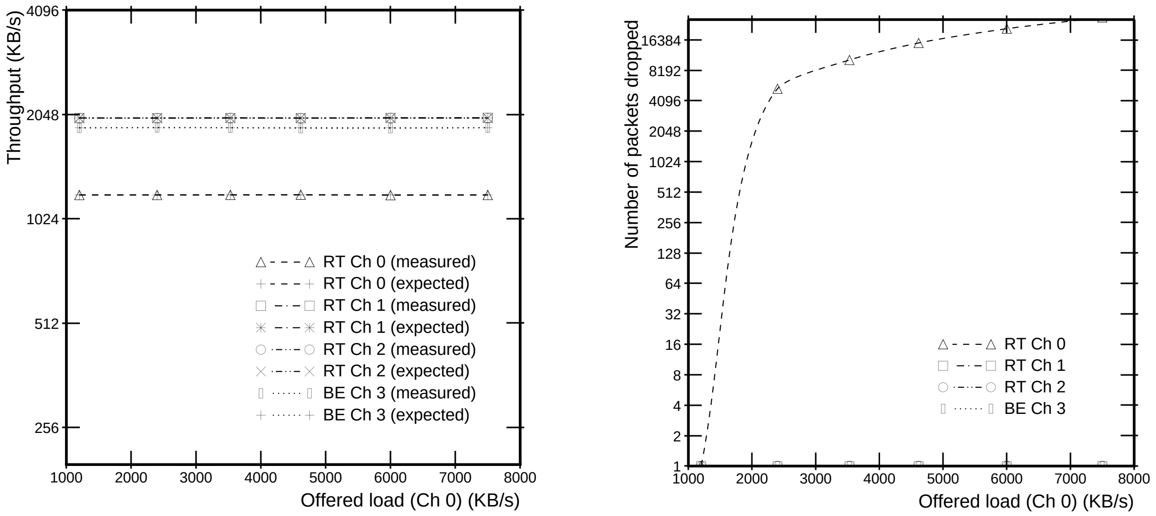
Figure 5: Maintenance of QoS guarantees when traffic specifications are honored

before blocking due to buffer overflow, thereby increasing the window of non-preemptibility.

Figure 7(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 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 convincingly demonstrate the need for cooperative preemption, in addition to traffic enforcement and CPU scheduling, for access to the CPU. While link scheduling was 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. In the course of performing these experiments, we have realized the necessity of accurately accounting for all possible overheads that can affect timing. In one instance, we overlooked a per-packet timestamp (expensive on our platform) needed for logging, resulting in numerous QoS violations. We have also identified a shortcoming in the way link packet queues are organized in the present implementation. Since all outgoing packets are immediately inserted into the link queues, heap insertion/deletion overhead becomes a function of the number of packets, making it relatively high and unpredictable. In this case, it is all the more important to hold back early messages in the message queue. We can improve performance by using per-channel packet FIFOs on transmission side as well, so that each channel has at most one packet in the link packet queues. This makes heap insertion/deletion overhead a function of the number of active channels,



(a) Throughput (b) Number of packets dropped

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

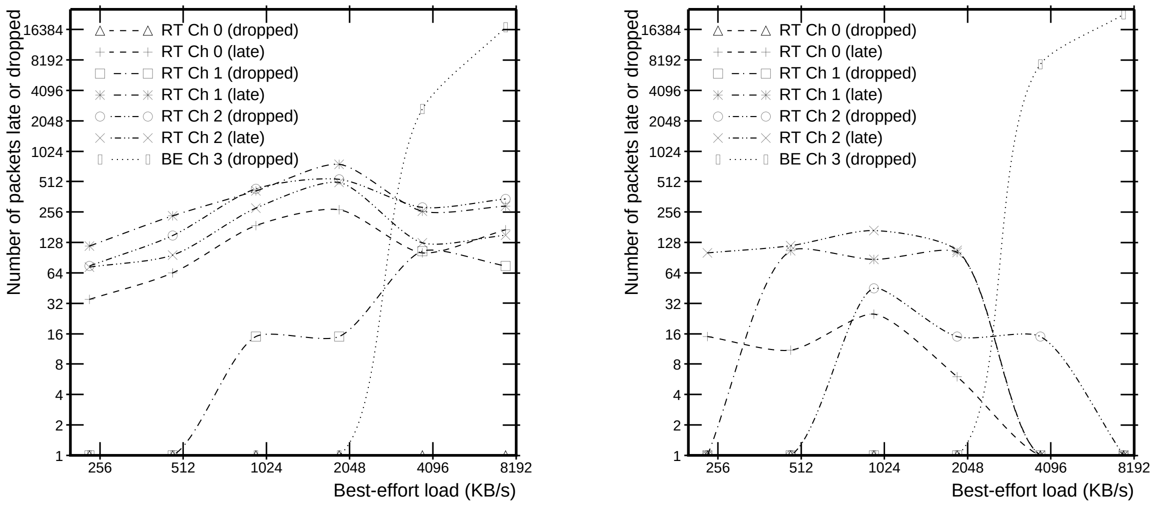
as in the CPU scheduling queues.

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 [2].

Similar issues are being examined for provision of integrated services on the Internet [9, 10]. Several classes of service are being considered, including guaranteed service (similar to our work) which provides guaranteed delay, and predictive service [11] which has 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 [12, 13]. 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 [14], which initiates reservation setup at the receiver, or ST-II [15], which initiates reservation setup at the sender; RSVP, in particular, provides special provisions for multicast communication. The issues involved in providing QoS support in IP-over-ATM networks are also being explored [16, 17]. The Tenet real-time protocol suite [18] 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.

The need for scheduling protocol processing at priority levels consistent with those of the communicating application was highlighted in [19] and some implementation strategies demonstrated in [20]. More recently, processor capacity reserves in Real-Time Mach [21] have been combined



(a) Non-preemptive best-effort processing (b) Non-preemptive real-time processing

Figure 7: Violation of QoS guarantees with cooperative preemption disabled

with user-level protocol processing [22] for predictable protocol processing inside hosts [23]. Our approach decouples the protocol processing priority from that of the application, allowing the former to be derived from the QoS requirements, traffic characteristics and run-time communication behavior of the application. Operating system support for multimedia communication is explored in [24], where the focus is on provision of preemption points and earliest-deadline-first scheduling in the kernel, and in [25], which also focuses on the scheduling architecture. However, no explicit support is provided for traffic enforcement or decoupling of protocol processing priority from application priority. The Path abstraction [26] provides a rich framework for development of real-time communication services for distributed applications.

7 Conclusions and Future Work

In this paper we have presented and evaluated a QoS-sensitive communication subsystem architecture for end hosts that supports guaranteed-QoS connections. To achieve this goal, 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 implementation maintains per-channel QoS guarantees and delivers reasonable performance to best-effort traffic. The evaluation also highlighted the necessity of specific features and policies provided in the architecture. While we demonstrated the need for QoS-sensitive protocol processing 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 extent to which QoS-sensitive protocol processing is useful depends on the relative speeds of the CPU and the network.

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.

This work can extend in several areas of research. We are now extending the null device into a more sophisticated network device emulator providing link bandwidth management. This will allow us to explore the 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. The analysis presented is directly applicable if a portion of the host processing capacity can be reserved for communication-related activities [21,23]. Within this framework, 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.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] A. Mehra, A. Indiresan, and K. Shin, "Resource management for real-time communication: Making theory meet practice," Technical Report CSE-TR-281-96, University of Michigan, January 1996.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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, International Computer Science Institute, Berkeley, CA, November 1994.

[19] 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, June 1990.

[20] 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.

[21] 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.

[22] 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.

[23] C. W. Mercer, J. Zelenka, and R. Rajkumar, "On predictable operating system protocol processing," Technical Report CMU-CS-94-165, Carnegie Mellon University, May 1994.

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

[25] 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.

[26] 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.