Resource Management for Real-Time Communication: Making Theory Meet Practice

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

This paper focuses on bridging the gap between theory and practice in the management of host CPU and link resources for real-time communication. Using our implementation of real-time channels, *a paradigm for real-time communication in packet-switched networks, we illustrate the tradeoff between resource capacity and channel admissibility, which determines the number and type of real-time channels that can be accepted for service and the performance delivered to best-effort traffic. We demonstrate that this tradeoff is affected significantly by the choice of implementation paradigms and the grain at which CPU and link resources are multiplexed amongst active channels. To account for this effect, we extend the admission control procedure for real-time channels originally proposed using idealized resource models. Our results show that practical considerations significantly reduce channel admissibility compared to idealized resource models. Further, the optimum choice of multiplexing grain depends on several factors such as resource preemption overheads, the relationship between CPU and link bandwidth, and the interaction between link bandwidth allocation and CPU bandwidth allocation.*

1. Introduction

The advent of high-speed networks has generated an increasing demand for a new class of distributed applications that require certain quality-of-service (QoS) guarantees from the underlying network. QoS guarantees may be specified in terms of several parameters such as end-to-end delay, delay jitter, and bandwidth delivered on each active 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) and distributed real-time command/control systems. To support these applications, both the communication subsystem in end hosts and the network must be designed to provide QoS guarantees on individual connections.

Given appropriate support within the network, host communication resources must be managed in a *QoS-sensitive* fashion, i.e., according to the relative importance of the connections requesting service. For a sending host,[1] communication resources include CPU bandwidth for protocol processing and link bandwidth for packet transmissions, assuming sufficient availability of buffer space. QoS-sensitive management of communication resources necessitates admission control and resource scheduling policies to ensure that each connection obtains at least its required QoS. These resource management policies are typically formulated using idealized models of the resources being managed. For example, it may be assumed that a given resource is immediately preemptible or the cost of preemption is negligible. More importantly, it may be assumed that a required set of resources can be accessed, and hence allocated, independent of one another. However, the above assumptions can be violated when implementing resource management policies, since the performance characteristics of the hardware and software components employed can deviate significantly from those of the idealized resource models.

In this paper, we focus on bridging the gap between theory and practice in the management of host CPU and link resources for real-time communication. For this purpose we utilize *real-time channels*, a paradigm for real-time communication in packet-switched networks [12], similar to other proposals for guaranteed-QoS connections [2]. Using our implementation of real-time channels [16], we illustrate the tradeoff between useful resource capacity, which is the pro-

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. .

[1]The issues involved in resource management at the receiving host are similar, but beyond the scope of this paper.

portion of the raw resource capacity that can be utilized effectively, and channel admissibility. In the context of real-time channels, useful resource capacity determines the number and type of real-time channels accepted for service and the performance delivered to best-effort traffic.

A channel requires a portion of the available CPU bandwidth to process each generated message and packetize it. Similarly, it requires a portion of the available link bandwidth to transmit each packet on the link. Since these two resources typically differ in their performance characteristics, they present different tradeoffs for resource management. We demonstrate that these tradeoffs are affected significantly by the choice of implementation paradigms and the (temporal) grain at which CPU and link resources are multiplexed amongst active channels. To account for this effect, we extend the admission control procedure for real-time channels originally proposed using idealized resource models. Our results show that, compared to idealized resource models, practical considerations significantly reduce channel admissibility. Further, the optimum choice of the multiplexing grain depends on several factors such as resource preemption overheads, the relationship between CPU and link bandwidth, and the interaction between CPU bandwidth allocation and link bandwidth allocation.

The rest of the paper is organized as follows. Section 2 provides a brief description of guaranteed-QoS communication using real-time channels. Section 3 discusses the issues involved in managing CPU and link bandwidth for QoS-sensitive protocol processing and packet transmissions, respectively. The modifications required in the admission control procedure to manage CPU and link bandwidth simultaneously are presented in Section 4. Section 5 studies the tradeoff between useful resource capacity and channel admissibility. Our work is contrasted with related work in Section 6. Finally, Section 7 concludes the paper.

2. 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 not sufficient time to recover from errors. Data transfer on real-time channels has unreliable-datagram semantics, i.e., occurs without acknowledgements and retransmissions.

2.1. Traffic Generation and QoS

Traffic generation on real-time channels is based on a *linear bounded arrival process* [7, 1] characterized by three parameters: maximum message size (S_{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} between messages on the same real-time channel. This ensures that a channel does not use more resources than it reserved at the expense of other channels' QoS guarantees. 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 [12] for more details.

2.2. Resource Management

As with other proposals for guaranteed-QoS communication [2], there are two related aspects to resource management for real-time channels: admission control and (run-time) scheduling. Admission control for real-time channels is provided by Algorithm `D_order` [12], which uses fixed-priority scheduling for computing the worst-case delay experienced by a channel at a link. When a channel is to be established at a link, the worst-case response time for a message (when the message completes transmission on the link) on this channel is estimated based on non-preemptive fixed-priority scheduling of packet transmissions. The total response time, which is the sum of the response times over all the links on the route of the channel, is checked against the maximum permissible message delay and the channel can be established only if the latter is greater. Contrary to the approach for admission control, run-time link scheduling is governed by a variation of the multi-class earliest-deadline-first policy [12]. The overheads and effectiveness of link scheduling in our implementation are discussed in [16].

The above approach only accounts for management of link bandwidth at the host. As discussed in Section 3, it cannot be applied directly to CPU bandwidth management.

2.3. Implementation

We have implemented a QoS-sensitive communication subsystem architecture featuring real-time channels [16]. Our implementation employs a communication executive derived from x-kernel 3.1 [11] exercising complete control over a Motorola 68040 CPU.

QoS-Sensitive CPU and Link Scheduling: The implementation provides a *process-per-channel* model of protocol processing adapted from the process-per-message model provided by x-kernel. In this model, a unique handler is associated with each channel to perform protocol processing for all messages generated on the channel. Channel handlers are scheduled for execution using a multi-class earliest-deadline-first (EDF) scheduler layered above the x-kernel scheduler (which provides fixed-priority non-preemptive scheduling with 32 priority levels). Since all channel handlers execute within a single (kernel) address space, the preemption model employed for handler execution is that of *cooperative preemption*. That is, the currently executing handler yields the CPU to a waiting higher-priority handler after processing up to a certain (configurable) number of packets (the preemption granularity). Besides bounding CPU access latency, this allows us to study the influence of preemption granularity and overheads on channel admissibility.

In order to support real-time communication, network adapters must provide a bounded, predictable transmission time for a packet of a given size. Since network adapters are typically best-effort in nature, their design is optimized for throughput and may be unsuitable for real-time communication, even with a bounded and predictable packet transmission time. Even when explicit support for real-time communication is provided, on-board buffer space limitations may necessitate staging of outgoing traffic in host memory, for subsequent transfer to the adapter. To support real-time communication on these adapters, link scheduling must be provided in software on the host processor. In our implementation, packets created by channel handlers are scheduled for transmission by a non-preemptive multi-class EDF link scheduler. See [16] for more details of the protocol stack and the real-time channel implementation.

Null Network Device: In order to explore the effects of the relationship between CPU and link bandwidth, we have implemented a device emulator, referred to as the *null device*, that can be configured to emulate any desired packet transmission time $\mathcal{L}_x$. $\mathcal{L}_x$ is the minimum time that must elapse between successive packet transmissions on the link. Thus, it suffices to ensure that successive packet transmissions can be invoked every $\mathcal{L}_x$ time units apart. This can be achieved by *emulating* the behavior of a network adapter such that $\mathcal{L}_x$ time units are consumed for each packet being transmitted.

The device emulator is simply a thread that, once signalled, tracks time by consuming CPU resources for $\mathcal{L}_x$ 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 expiry of $\mathcal{L}_x$ time units (completion of packet transmission), the emulator issues an interrupt to the host processor, similar to the mechanism employed in typical network adapters. While the emulator allows us to study a variety of tradeoffs, including the effects of the relationship between CPU and link bandwidth, it is not completely accurate since no packet data is actually transferred from host memory. However, this does not invalidate the trends observed and performance comparisons reported here.

3. Managing CPU and Link Bandwidth

As mentioned earlier, Algorithm `D_order` [12] computes the worst-case response time for a message. This response time has two components: the time spent waiting for resources and the time spent consuming resources. At the host, the time spent consuming resources is equal to the *message service time*, the time required to process and transmit all the packets constituting the message. To calculate the time spent waiting for resources, one must consider the preemption model used for resource access.

The real-time channel model presented in [12] accounts for non-preemptive packet transmissions, but assumes an ideal preemption model for CPU access, i.e., the CPU can be allocated to a waiting higher-priority handler immediately at no extra cost. Under this assumption, message service time is determined solely by the CPU processing bandwidth required to packetize the message, and the link bandwidth required to transmit all the packets. The time spent waiting for resources is calculated by accounting for resource usage by messages from all higher-priority channels, and the one-packet delay (due to non-preemptive packet transmission) in obtaining the link. However, as explained below, implementation issues necessitate extensions to the model to account for implementation overheads and constraints.

3.1. Implementation Issues

Several implementation issues impact resource management policies. These include handler execution requirements, implementation of link scheduling, and the relationship between CPU and link bandwidth.

Handler Execution: Preemption of an executing process/thread comes with a significant cost due to context switch and cache miss penalty. Preemption effectively increases the CPU usage attributed to a channel, which in turn reduces the CPU processing bandwidth available for real-time channels; immediate preemption is thus too expensive. It is desirable to limit the number of times a handler is forced to preempt the CPU in the course of processing a message. At the other extreme, non-preemptive execution of handlers implies that the CPU can be reallocated to a waiting handler only after processing an entire message. This results in a coarser (temporal) grain of channel multiplexing on the CPU and makes admission control less effective. More importantly, admission control must consider the largest possible message size across all real-time and best-effort channels; maximum message size for best-effort traffic may not even be known *a priori*. An intermediate solution is to preempt the CPU only at specific preemption points, if needed. Since message processing involves packetization, the CPU can be preempted after processing every packet. The important parameter here is the number of packets processed between preemption points, which determines the (temporal) grain at which the CPU can be multiplexed between channels. Admission control must account for the extra delay in obtaining the CPU which may be currently allocated to a lower-priority handler.

In the absence of per-byte copying overheads, the total CPU time required to process a message is directly proportional to the number of packets constituting the message. Clearly, assuming that the communication subsystem does not copy message data unnecessarily (true for our implementation), the CPU processing time will be minimum if a single packet constituted the entire message, i.e., if the packet size was the same as the message size. However, as explained below, the total time required to transmit a packet on the link is determined primarily by the size of the packet, although initiation of transmission involves non-zero per-packet overhead. If the set of channels requesting service have identical traffic specifications, and hence the same maximum message size, then single-packet messages maximize channel admissibility. However, under a heterogeneous mix of real-time channels (with large and small messages), a large packet size would significantly reduce the admissibility for channels with messages smaller than the chosen packet size. Packet size, therefore, also plays a significant role in determining channel admissibility.

Implementation of Link Scheduling: An assumption often made when formulating resource management policies for communication is that CPU and link bandwidth can be independently allocated to a channel. This assumption may get violated in an implementation depending on the paradigm used to implement link scheduling. We consider three options for implementing link scheduling in software:

- **O1** Packets are scheduled for transmission either in the context of the currently executing channel handler (via a function call) or in interrupt context after each packet transmission.
- **O2** Packets are scheduled for transmission by a dedicated process that executes at the highest priority and is signalled via semaphore operations.
- **O3** Packets are scheduled for transmission either in the context of the currently executing channel handler or in the context of a new thread that is fired up after every packet transmission.

O1 and O2 differ significantly in the implications for CPU and link bandwidth allocation, since with O2 the link scheduler must also be scheduled for execution on the CPU. Since O3 presents tradeoffs similar to O2, we focus on O1 and O2 in the discussion below.

Selecting a packet for transmission incurs some overhead in addition to that of initiating transmission on the link. Additional overhead may be involved if the link scheduler must transfer packets between link packet queues [16]. In O1, the scheduler is frequently invoked from the interrupt service routine (ISR) announcing completion of packet transmission. Since the scheduling overhead involved can be substantial in the worst case, it is undesirable to incur this penalty in the ISR, since this prolongs the duration for which network interrupts are disabled. If the host is also receiving data from the network, there is now a greater likelihood of losing incoming data.

O2, on the other hand, does not suffer from this problem; since scheduler processing is scheduled for execution, it is performed outside the ISR. In addition to keeping the ISR short, this paradigm also has some software structuring benefits such as a relatively cleaner implementation. However, because the link scheduler is itself scheduled for execution on the CPU, there is now an additional overhead of a context switch and the accompanying (instruction) cache miss penalty for each packet transmission. More importantly, allocation of CPU and link bandwidth is closely coupled in O2. This coupling can potentially lower the utilization of the link and, as demonstrated in Section 5, significantly reduces channel admissibility while making it unpredictable.

Relationship Between CPU and Link Bandwidth: A conservative estimate of message service time can be obtained by adding the total CPU processing time and the total link transmission time. However, this ignores the overlap between CPU processing and link transmission of packets constituting the same message. The extent of this overlap depends largely on the relationship between the CPU and link bandwidth, i.e., on the relative speed of the two. To improve channel admissibility, message service time must be calculated to account for this overlap. The extent of the overlap also depends on the implementation option used for link scheduling. While O1 allows link utilization to be kept relatively high, O2 can cause the link to idle even when there are packets available for transmission. From another perspective, O2 forces link scheduling to be non-work-conserving while O1 allows for work-conserving packet transmissions.

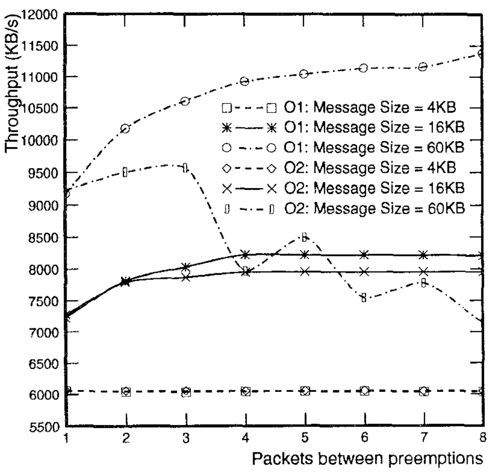
While admission control can utilize the overlap between CPU processing and link transmission of packets belonging to a message, it cannot do so for the potential overlap between CPU processing and link transmission of packets belonging to *different* messages. Since message arrivals serve as system renewal points, no *a priori* assumptions can be made about the presence of messages in the system.

Determination of $\mathcal{L}_x$: As mentioned earlier, the packet transmission time $\mathcal{L}_x(s)$ for a packet of size s measures the delay between initiation and completion of packet transmission on the network adapter. It determines the minimum time between successive packet transmission invocations by the link scheduler. For a typical network adapter, this delay 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 the host memory and the adapter. If $\mathcal{C}_x$ is the overhead to initiate transmission on an adapter feeding a link of bandwidth $\mathcal{B}_l$ bytes/second, then the transmission time of a packet (of size s) can be approximated as

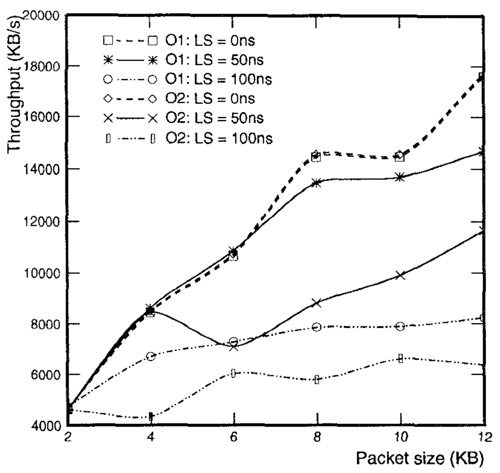
$$\mathcal{L}_x(s) = \mathcal{C}_x + \frac{s}{\min(\mathcal{B}_l, \mathcal{B}_x)},$$

where $\mathcal{B}_x$ is the data transfer bandwidth available to/from host memory. $\mathcal{B}_x$ is determined by a variety of 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. Note that with non-preemptive packet transmissions on the link, $\mathcal{L}_x(\mathcal{S})$ is also the delay experienced by a waiting highest-priority packet to commence transmission, where $\mathcal{S}$ is the maximum packet size.

To use this model of packet transmission time, $\mathcal{C}_x$ and $\mathcal{B}_x$ must be determined for a given network adapter and host architecture. This involves experimentally determining the latency-throughput characteristics of the adapter. Since we want to explore the effects of the relationship between CPU and link bandwidth, we select $\min(\mathcal{B}_l, \mathcal{B}_x)$ to conform to a desired link (and data transfer) speed, measured in nanoseconds (ns) required to transfer one byte. On the null device, $\mathcal{C}_x$ is determined by the granularity of time-keeping and overhead of communication with the host processor; the measured value of $\mathcal{C}_x$ is $\approx 40\ \mu s$.



(a) Packets between preemptions (LS = $0ns$)



(b) Packet size and link speed (PBP = 1)

Figure 1. Throughput as a function of packets between preemptions, packet size and link speed

3.2. Performance Implications

To illustrate the performance implications of some of the above-mentioned issues, we ran several experiments using our real-time channel implementation to measure system throughput (kilobytes(KB)/second) as a function of parameters such as the number of packets processed between preemption points, the packet size, and link speed. For a given CPU processing power, varying the speed of the link allows us to explore the relationship between CPU and link bandwidth. In the experiments reported here, four best-effort channels were created and messages generated on these channels continuously; each experimental run involved transmission of over 25,000 packets. Multiple runs produced consistently repeatable results. The results reported are representative in that similar trends were obtained with more channels and other parameter settings. An experimental parameterization of our implementation yielded the values listed in Table 1.

Figure 1(a) shows system throughput (useful resource capacity) as a function of the number of packets processed between preemption points (PBP) for several message sizes. The packet size is fixed at 4 KB and the link speed (LS) is set at 0 ns per byte, i.e., the link is "fast" relative to the CPU. For O1, changing PBP has no effect when the message size is 4 KB; this is expected for single-packet messages. As message size increases, so do the number of packets and throughput increases until PBP equals the number of packets in the message. After this point PBP has no effect on throughput. As can be seen, for large messages an increase in PBP improves throughput significantly and consistently.

O2 reveals the same behavior as O1 for small- to medium-sized messages. However, for large messages throughput rises initially as PBP increases. Subsequently, throughput starts falling sharply, in a non-linear fashion. The decline in throughput is due to increasingly poor utilization of the link bandwidth and a corresponding increase in the time to transmit all the packets belonging to the message. The oscillations in throughput are due to subtle interactions between the CPU preemption window and link transmission, as investigated in Section 4.

Figure 1(b) shows the measured system throughput as a function of packet size and link speed. Three values of link speed are considered: 0 ns per byte (fast link), 50 ns per byte (medium-speed link), and 100 ns per byte (slow link). For each link speed, we fix PBP at 1 and message size at 32 KB; packet size is varied from 2–12 KB.

Consider system throughput for O1. For a given packet size (i.e., fixed CPU processing time), an increase in link speed results in higher throughput for O1 and O2, with O1 outperforming O2. For a given link speed, throughput increases with packet size since the CPU processing time reduces due to a reduction in the number of packets constituting the message. An increase in packet size from 8 KB to 10 KB does not change the number of packets and the throughput remains unchanged. As the link becomes slower, however, there is a saturation in the achieved throughput due to the link tending to become a bottleneck. After a certain packet size, for a given link speed, link transmission time exceeds the protocol processing time; thus any gains from a higher PBP cease to matter and the two curves converge. From Figure 1(b), this occurs at a packet size of 8 KB for link speed of 50 ns per byte and at 4 KB for link speed of 100 ns per byte.

Consider system throughput for O2. The trends are similar to those observed when the link is either very fast (CPU is the bottleneck) or very slow (link is the bottleneck) since CPU and link processing overlap almost completely. For a medium speed link (CPU and link bandwidths are more balanced), however, throughput behavior is more non-linear.

Subtle interactions between CPU preemption window and link transmission time cause the link to idle until the next preemption point. This explains the drop in throughput at a packet size of 6 KB. Subsequently, throughput climbs because link utilization improves and CPU requirements continue to decrease. This effect is analyzed in Section 4.

4. Worst-Case Service and Wait Times

For a channel requesting admission, D_order can compute the worst-case message response time (the *system time requirement* in [12]) by accounting for three components:

- worst-case waiting time ($\mathcal{T}_w$) due to lower-priority handlers or packets,
- worst-case service time for the message ($\mathcal{T}_s$),
- worst-case waiting time due to message arrivals on all existing higher-priority channels ($\mathcal{T}_t^{hp}$).

We show below how $\mathcal{T}_w$ and $\mathcal{T}_s$ can be estimated to account for the implementation-related issues highlighted above; $\mathcal{T}_t^{hp}$ can then be recomputed using $\mathcal{T}_s$.

Suppose the CPU is reallocated to a waiting handler, if needed, every $\mathcal{P}$ packets; that is, up to $\mathcal{P}$ packets are processed between successive preemption points. Further, (see Table 1) suppose that the maximum packet size is $\mathcal{S}$, context switch overhead between handlers is $\mathcal{C}_{sw}$ (this includes scheduling overhead to select a handler for execution), cache miss penalty due to a context switch is $\mathcal{C}_{cm}$, and packet transmission time is $\mathcal{L}_x(\mathcal{S})$ for packet size $\mathcal{S}$. Per-packet protocol processing cost is $\mathcal{C}_p$ and per-packet (link) scheduling overhead of selecting a packet and initiating transmission is $\mathcal{C}_l$.

4.1. Estimating Service Time

Consider a message of size $\mathcal{M}$ bytes, i.e., $\mathcal{N}_p = \lceil \frac{\mathcal{M}}{\mathcal{S}} \rceil$ packets, with $(\mathcal{N}_p - 1)$ packets of size $\mathcal{S}$ and the last packet of size $\mathcal{S}^{last} = (\mathcal{M} \bmod \mathcal{S})$ if $(\mathcal{M} \bmod \mathcal{S}) \neq 0$, else $\mathcal{S}^{last} = \mathcal{S}$. Thus, link transmission time is $\mathcal{L}_x(\mathcal{S})$ for all but the last packet and $\mathcal{L}_x(\mathcal{S}^{last})$ for the last packet. Protocol processing cost for the first packet is $\mathcal{C}_p^{1st}$ while subsequent fragments each incur a lower cost $\mathcal{C}_p$. $\mathcal{C}_p^{1st}$ includes the fixed cost of obtaining the message for processing, the cost of a timestamp, and the cost of preparing the first packet.[2] Both $\mathcal{C}_p^{1st}$ and $\mathcal{C}_p$ include the cost of network-level encapsulation. We estimate $\mathcal{T}_s$ for O1 and O2 separately. The reader is referred to [17] for explanations (omitted here due to space constraints) of the following derivations.

Option O1: Given the system parameters listed in Table 1, the worst-case service time for O1 is given by

$$\mathcal{T}_s^{O1} = \begin{cases} \mathcal{C}_p^{1st} + \mathcal{L}_x^m + \mathcal{C}_l^m + \mathcal{C}_{pr} & \text{if } \mathcal{C}_p < \mathcal{L}_x(\mathcal{S}) \\ \mathcal{C}_p^m + \mathcal{C}_l^m + \mathcal{L}_x(\mathcal{S}^{last}) + \mathcal{C}_{pr} & \text{otherwise} \end{cases}$$

where $\mathcal{L}_x^m = (\mathcal{N}_p - 1)\mathcal{L}_x(\mathcal{S}) + \mathcal{L}_x(\mathcal{S}^{last})$ is the total link transmission time for the message, $\mathcal{C}_l^m = \mathcal{N}_p\mathcal{C}_l$ is the total link scheduling overhead for the message, $\mathcal{C}_{pr} = (\frac{\mathcal{N}_p - 1}{\mathcal{P}})\mathcal{C}_{csp}$ is the total cost of preemption during the processing of the message ($\mathcal{C}_{csp} = \mathcal{C}_{cm} + \mathcal{C}_{sw}$), and $\mathcal{C}_p^m = \mathcal{C}_p^{1st} + (\mathcal{N}_p - 1)\mathcal{C}_p$ is the total protocol processing cost for the message. Protocol processing and link transmission overlap in O1 is illustrated in Figure 2(O1:(i) and O1:(ii)).

[2]Our fragmentation protocol traverses a slower path for messages larger than $\mathcal{S}$ bytes; the first packet thus has a higher processing cost.

$\mathcal{C}_{sw}$	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
$\mathcal{S}$	maximum packet size	4 KB
$\mathcal{L}_x(\mathcal{S})$	transmit time for packet size $\mathcal{S}$	245 μs

Table 1. Important system parameters

Option O2: Calculation of the worst-case service time for O2 is done similarly; however, we must now consider the processing of *blocks* of packets with each block comprising no more than $\mathcal{P}$ packets. The number of blocks in a message with $\mathcal{N}_p$ packets is given by $\mathcal{N}_b = \lfloor \frac{\mathcal{N}_p - 1}{\mathcal{P}} \rfloor + 1$. The protocol processing cost for the first block is given by $\mathcal{C}_b^{1st} = \mathcal{C}_p^{1st} + (\max(\mathcal{N}_p, \mathcal{P}) - 1)\mathcal{C}_p$, while the cost of processing the last block of packets is given by

$$\mathcal{C}_b^{last} = \begin{cases} \mathcal{C}_b & \text{if } (\mathcal{N}_p \bmod \mathcal{P}) = 0 \\ (\mathcal{N}_p \bmod \mathcal{P})\mathcal{C}_p + \mathcal{C}_{csp} & \text{otherwise} \end{cases}$$

where $\mathcal{C}_b = \mathcal{P}\mathcal{C}_p + \mathcal{C}_{csp}$ is the cost of processing the other blocks, if any. The worst-case service time is given by

$$\mathcal{T}_s^{O2} = \begin{cases} \mathcal{T}^A + \mathcal{T}_w^{O2,cpu} & \text{if } \mathcal{C}_b < \mathcal{L}_x(\mathcal{S}) \\ \mathcal{T}^B & \text{otherwise} \end{cases}$$

where $\mathcal{T}^A = \mathcal{C}_b^{1st} + \mathcal{L}_x^m + \mathcal{C}_l^m$ and

$$\mathcal{T}_w^{O2,cpu} = \mathcal{C}_p^{1st} + (\mathcal{P} - 1)\mathcal{C}_p + \mathcal{C}_{csp} + \mathcal{C}_l',$$

with $\mathcal{C}_l' = \mathcal{C}_l + \mathcal{C}_{csp}$. $\mathcal{T}^B$ is given by

$$\mathcal{T}^B = \begin{cases} \mathcal{T}^A & \text{if } \mathcal{N}_b = 1 \\ \mathcal{T}_a^C + \mathcal{T}_b^C & \text{otherwise} \end{cases}$$

where $\mathcal{T}_a^C = \mathcal{C}_b^{1st} + (\mathcal{N}_b - 2)\mathcal{C}_b + \max(\mathcal{C}_b^{last}, \mathcal{L}_x(\mathcal{S}))$ and $\mathcal{T}_b^C = (\mathcal{N}_p - \mathcal{N}_b)\mathcal{L}_x(\mathcal{S}) + \mathcal{L}_x(\mathcal{S}^{last}) + \mathcal{C}_l^m$. Protocol processing and link transmission overlap in O2 is illustrated in Figure 2(O2:(i) and O2:(ii)).

4.2. Estimating Wait Time

To compute the total message wait time, we first consider the time spent waiting for a lower-priority handler to relinquish the CPU, followed by the time spent waiting for the link.

Option O1: The worst-case CPU time for a block of packets is $\mathcal{C}_b^{1st}$, during which up to $\lceil \frac{\mathcal{C}_b^{1st}}{\mathcal{L}_x(\mathcal{S})} \rceil$ packets could complete transmission. Thus, the worst-case CPU wait time is

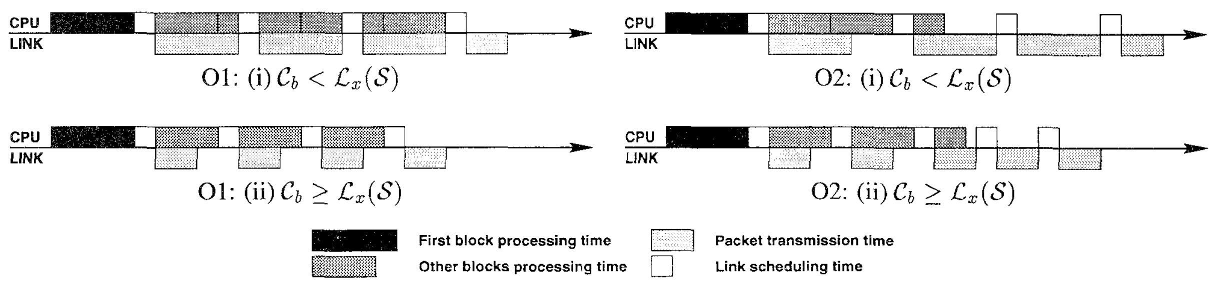

CPU
LINK
O1: (i) $\mathcal{C}_b < \mathcal{L}_x(\mathcal{S})$
O2: (i) $\mathcal{C}_b < \mathcal{L}_x(\mathcal{S})$
O1: (ii) $\mathcal{C}_b \geq \mathcal{L}_x(\mathcal{S})$
O2: (ii) $\mathcal{C}_b \geq \mathcal{L}_x(\mathcal{S})$
First block processing time
Other blocks processing time
Packet transmission time
Link scheduling time

Figure 2. Protocol processing and link transmission overlap in O1 and O2

$$\mathcal{T}_w^{O1,cpu} = \mathcal{C}_b^{1st} + (\lceil \frac{\mathcal{C}_b^{1st}}{\mathcal{L}_x(\mathcal{S})} \rceil)\mathcal{C}_l + \mathcal{C}_{csp}.$$

Due to non-preemptive packet transmission, the worst-case link wait time is simply $\mathcal{T}_w^{O1,link} = \mathcal{L}_x(\mathcal{S})$, and $\mathcal{T}_w^{O1} = \mathcal{T}_w^{O1,cpu} + \mathcal{T}_w^{O1,link}$

Option O2: The worst-case CPU wait time equals the time to process up to $\mathcal{P}$ packets on a lower-priority channel followed by a context switch to the link scheduler, followed by another context switch to the waiting handler. Thus,

$$\mathcal{T}_w^{O2,cpu} = \mathcal{C}_p^{1st} + (\mathcal{P} - 1)\mathcal{C}_p + \mathcal{C}_{csp} + \mathcal{C}_l^{'},$$

where $\mathcal{C}_l^{'} = \mathcal{C}_l + \mathcal{C}_{csp}$. Similarly, $\mathcal{T}_w^{O2,link}$ is derived as follows. If $\mathcal{L}(\mathcal{S}) <= \mathcal{C}_p^{1st}$, $\mathcal{T}_w^{O2,link} = 0$; else $\mathcal{T}_w^{O2,link} = \mathcal{T}_w^{O2,cpu}$. Thus, $\mathcal{T}_w^{O2} = \mathcal{T}_w^{O2,cpu} + \mathcal{T}_w^{O2,link}$.

4.3. Experimental Validation

Our implementation provides admission control based on the above estimates of service and wait times. While these estimates are geared towards real-time guarantees, and therefore are necessarily conservative, it is insightful to compare the throughput predicted by these estimates and the best-effort throughput measured using the real-time channel implementation. For this purpose, we parameterized the communication subsystem, including the protocol stack, extensively to determine the system parameter values listed in Table 1. We validated the implementation as a function of packet size, for different values of link speed; the main results are summarized below. See [17] for more details.

Predicted throughput tracks measured throughput well for both O1 and O2. However, for O1 with medium link speeds, the predicted and measured throughputs diverge significantly; we attribute this to overly conservative estimates of $\mathcal{C}_{sw}$ and $\mathcal{C}_{cm}$. The estimates are necessarily conservative in accounting for worst-case times which, though necessary for real-time traffic, may be relatively small on average. These validation experiments reveal certain shortcomings in determining the system parameter values listed in Table 1; part of the discrepancy stems from the unpredictability introduced by caches. More refined experiments are necessary to select accurate values for $\mathcal{C}_p$, $\mathcal{C}_{sw}$ and $\mathcal{C}_{cm}$.

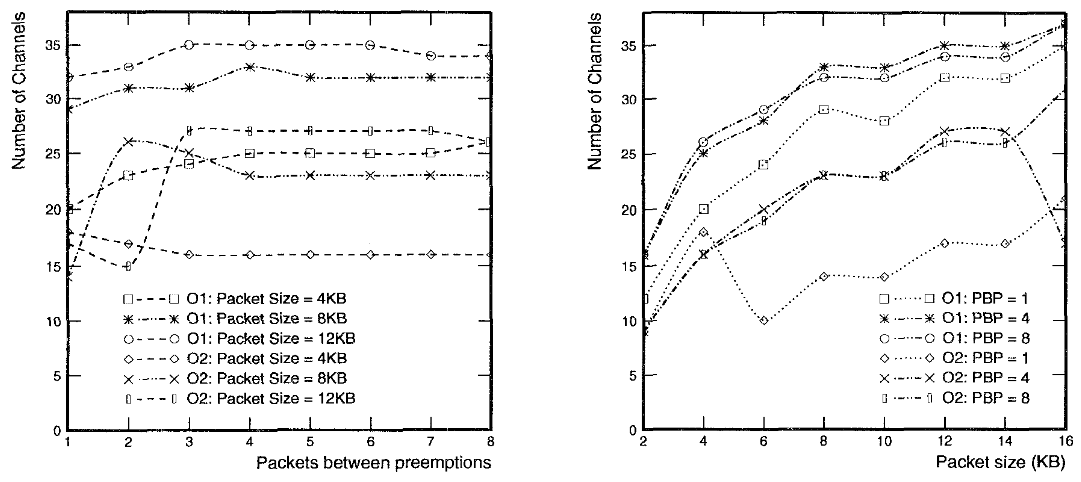
5. Channel Admissibility

In this section, we demonstrate that the tradeoff between resource capacity and channel admissibility is influenced significantly by $\mathcal{P}$, the number of packets between preemptions, and $\mathcal{S}$, the packet size. As expected, the mechanism employed to implement link scheduling and the relationship between CPU and link bandwidth also have a profound effect on channel admissibility. We studied channel admissibility for O1 and O2 for a range of link speeds, message sizes, rates and deadlines. In the following, we present and compare the results for a link speed of 50 ns per byte, message size of 32 KB, and message inter-arrival of 100 ms. We admit as many channels as possible with deadline of 100 ms.

5.1. Channel Admissibility in O1

From Figure 3, channel admissibility in O1 rises with both $\mathcal{P}$ and $\mathcal{S}$ due to the accompanying reduction in protocol processing cost and work-conserving packet transmissions. As $\mathcal{P}$ rises (Figure 3(a)), protocol processing costs decline, resulting in a small increase in channel admissibility. As $\mathcal{P}$ continues to rise, the marginal benefits in protocol processing costs decline. Due to an increase in the window of non-preemptibility, channel admissibility either saturates or shows a small decline. Figure 3(b) shows that increasing $\mathcal{S}$ increases channel admissibility substantially, since the reduction in the required CPU bandwidth more than compensates for the increase in the non-preemptibility window.

The above results might suggest that arbitrary-sized packets, i.e., sending each message as a single packet, are desirable to maximize channel admissibility. While this is true if all channels carry same-sized messages, the same cannot be said for channels with smaller (single-packet) messages. Increasing $\mathcal{P}$ and $\mathcal{S}$ arbitrarily only serves to increase the window of non-preemptibility with no reduction in CPU requirements for small messages. Large values of $\mathcal{P}$ and $\mathcal{S}$ lower admissibility for channels with small messages, especially those with tight deadlines. Selection of $\mathcal{P}$ and $\mathcal{S}$ therefore depends on system parameters as well as the targetted mix of communication traffic.



(a) Effect of $\mathcal{P}$ (b) Effect of $\mathcal{S}$

Figure 3. Effect of $\mathcal{P}$ and $\mathcal{S}$ on channel admissibility in O1 and O2

5.2. Channel Admissibility in O2

In contrast to O1, channel admissibility when using O2 to schedule packets is significantly lower and the behavior is highly non-linear. This is explained easily using our preemption model. Consider the effect of $\mathcal{P}$ on channel admissibility (Figure 3(a)) for 8 KB packets. With the given link speed and $\mathcal{P} = 1$, link transmission time is greater than protocol processing time for a block of packets. The model in Figure 2(O2:(i)) applies, making the channel susceptible to long idle periods (Section 4). For $\mathcal{P} = 2$, the transmission time for a packet remains unchanged, but the processing time for a preemption block increases, making it more than the link transmission time. This results in the scenario in Figure 2(O2:(ii)), in which the worst-case transmission time is reduced substantially, thereby increasing channel admissibility.

As $\mathcal{P}$ increases further, the nature of overlap between CPU processing and link transmission remains unchanged. Channel admissibility either remains unchanged or declines slightly due to an increase in the window of non-preemptibility. This transition occurs for all but the smallest packet sizes. In general, the larger the packet, the greater the $\mathcal{P}$ that causes a change in the nature of overlap, namely, from packet transmission time being slower to it being faster than the processing time for the longest block of packets.

Figure 3 (b) presents the same information as a function of $\mathcal{S}$. As packet size increases, there is an initial increase in admissibility due to reduced protocol processing load. At a certain value of $\mathcal{S}$, link transmission time becomes larger than block processing time, changing the scenario from that in Figure 2(O2:(ii)) to that in Figure 2(O2:(i)). Further increase in packet size slowly increases channel admissibility due to reduced CPU bandwidth requirements. As seen from Figure 3, the best operating point for O2 depends critically on system parameters. Since a change in channel characteristics will significantly change channel admissibility, a system parameterized and optimized for a particular workload is unlikely to perform well under a heterogeneous workload.

Using a model of ideal resources, i.e., with no CPU preemption cost and an immediately preemptible CPU, we found $\approx$ 40% improvement in channel admissibility over and above O1 with $\mathcal{P} = 1$. Thus, it is necessary to account for non-ideal characteristics (context switch overhead, cache miss penalty) of real systems.

6. Related Work

This paper extends the policies proposed in [12], focusing on CPU and link bandwidth management for admission control. We have implemented a QoS-sensitive architecture [16] that provides admission control and run-time support for real-time channels using the proposed extensions.

Our implementation methodology and analysis is applicable to other proposals for guaranteed real-time communication in packet-switched networks, a survey of which can be found in [2]. Similar issues are being explored for provision of integrated services on the Internet [6, 4, 8]. The Tenet protocol suite [3] is an advanced implementation of real-time communication on wide-area networks; however, they have not considered incorporation of protocol processing overheads into network-level resource management policies. In particular, they do not address the problem of making protocol processing inside the host QoS-sensitive.

The gap between theory and practice for real-time systems has received significant attention in recent years [13, 5, 14]. Our work is complimentary to these efforts in that we focus on communication needs of distributed real-time systems. Scheduling of protocol processing at priority levels consistent with those of the communicating application was considered in [9]. More recently, processor capacity reserves in RT-Mach [18] have been combined with user-level protocol processing [15] to make protocol processing inside hosts predictable [19]. Operating system support for multimedia communication is explored in [10] and [20]. In [10] the focus is on provision of preemption points and earliest-deadline-first scheduling in the kernel. Similarly, the focus of [20] is on the scheduling architecture.

7. Conclusion and Future Work

In this paper, we focused on management of host communication resources for real-time communication. In particular, we identified the issues involved in extending and implementing resource management policies originally formulated using idealized resource models. Using our real-time channel implementation, we extended the admission control procedure to account for protocol processing and implementation overheads, for two implementation paradigms realizing link scheduling. The extensions were validated against measured performance of the implementation and used to study the implications for channel admissibility.

The issues of simultaneous management of CPU and link bandwidth for real-time communication are of wide-ranging interest. Our present work is applicable to other proposals for real-time communication and QoS guarantees [2]. The proposed extensions are general and applicable to other host and network architectures. While we only considered management of communication resources, the present work can be extended to incorporate application scheduling as well. Our analysis is directly applicable if a portion of the host processing capacity can be reserved for communication-related activities [18, 19].

As part of future work, we plan to conduct more extensive validation of the proposed extensions. This would involve relaxing some of the fairly-conservative assumptions about worst-case scenarios without compromising real-time guarantees. Lastly, we plan to extend the null device into a more sophisticated network device emulator.

References

[1] 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.

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

[3] 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.

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

[5] A. Burns, K. Tindell, and A. Wellings. Effectice analysis for engineering real-time fixed priority schedulers. *IEEE Trans. Software Engineering*, 21(5):475–480, May 1995.

[6] 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.

[7] 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.

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

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

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

[11] 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.

[12] 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.

[13] D. Katcher, H. Arakawa, and J. K. Strosnider. Engineering and analysis of fixed priority schedulers. *IEEE Trans. Software Engineering*, 19(9):920–934, Sept. 1993.

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

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

[16] A. Mehra, A. Indiresan, and K. Shin. Design and evaluation of a QoS-sensitive communication subsystem architecture. Technical Report CSE-TR-280-96, University of Michigan, January 1996.

[17] 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.

[18] 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.

[19] 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.

[20] 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.