An Architecture Towards Efficient OS Support for Distributed Multimedia*

David K.Y. Yau and Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

December 4, 1995

Abstract

In designing operating system support for distributed multimedia, we target three areas for improvement: reduced copying, reduced reliance on explicit kernel/user interactions, and provision of rate-based flow control. Towards these goals, we propose an architecture that includes the concept of *I/O efficient buffers* for reduced copying, the concept of *fast system calls* for low latency network access, and the concept of *kernel threads* for flow control. Also included is a concept called *direct media streaming* which is suitable for applications that require limited user processing of media data. These concepts have been implemented as an extension to SunOS 5.3 (the operating system component of Solaris 2.3). We report some experimental results on the performance of our current system.

1 Introduction

The Unix network I/O architecture is illustrated in Figure 1. Note that control is transferred between kernel and user space through system calls and returns. There are two kinds of system calls: *synchronous* and *asynchronous*. A synchronous call blocks until its operation can be performed. For example, a read that finds an empty connection blocks until the connection has packets. On the other hand, an asynchronous call that does not succeed immediately returns with an indication that the user process should try again later. The retry may be facilitated by a signal delivered to the user process when the blocking condition is lifted. In either case, an I/O operation succeeds only when the system call successfully returns.

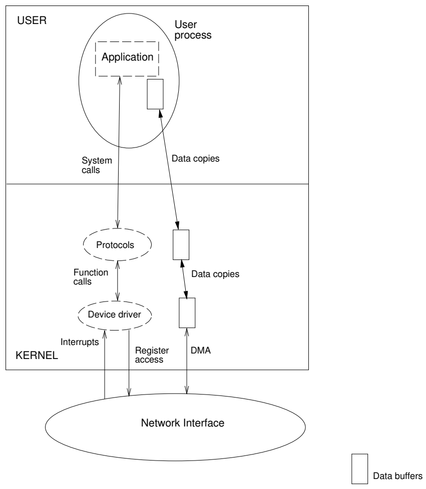

Figure 1: Unix I/O architecture.

Data transfer goes through intermediary kernel buffers by memory-to-memory data copies. We illustrate by describing the read/write paths between a user process and the Ethernet interface in SunOS 5.3, whose networking subsystem is implemented using streams [9]. When a `write()` system call occurs, the user data are copied by the stream head into stream buffers; the data are segmented into multiple stream messages if needed (e.g., to conform to Ethernet's packet size). The stream messages are then queued to the Ethernet driver.

*To appear in *Proceedings MMCN '96*, January 1996. Research supported in part by National Science Foundation under grant no. NCR-9506048 and by a grant from AT&T Foundation.

The driver copies the data of each stream message to another kernel buffer area already mapped for DMA transfer, appends the buffer descriptor to a queue, and then writes to a control register in the network interface to effect DMA transfer. Altogether two memory-to-memory data copies are made. The receive data path is similar, but in reverse. Data, already transferred from the network interface into main memory, are copied from DMA buffers to stream buffers,[1] and then from stream buffers to user buffers.

The problem with copying is that it incurs a per-byte overhead. As network bandwidth increases, it is approaching the same order of magnitude as memory bandwidth [5]. Therefore, memory bandwidth may become the communication bottleneck if memory copies are not carefully minimized.

Data flows generated by multimedia applications are more predictable than conventional data traffic. In a server for video playback, for example, a picture is made available for sending once every frame period. For such isochronous traffic, it would be unnecessary to use explicit system calls to transfer control between kernel and user. Furthermore, the delivery of media data would require real-time QoS (such as delay and throughput) guarantees to ensure smooth playback. We therefore expect that future multimedia applications will negotiate *reserved-rate connections* with networks—such as those considered in [2] and [8]. To obtain QoS guarantees, the packet arrivals to a reserved-rate connection are required to satisfy certain flow specifications.

To better understand the characteristics of multimedia applications, we studied a working video-conferencing tool called `nv` (for *n*etwork *v*ideo), which is widely used in mbone multicasts. We found that `nv` sends each picture as a sequence of smaller buffers of about 1040 bytes. Each buffer corresponds to a packet of RTP, which is an application level protocol for multimedia transport over the Internet. Figure 2 profiles `nv`'s use of the `send()` function call for sending to the network. This profile shows that `nv` makes fairly frequent use of system calls for data transfer in a periodic manner.

[1] However, an optimization is made to remove this first copy if the number of free DMA buffers exceeds a threshold.

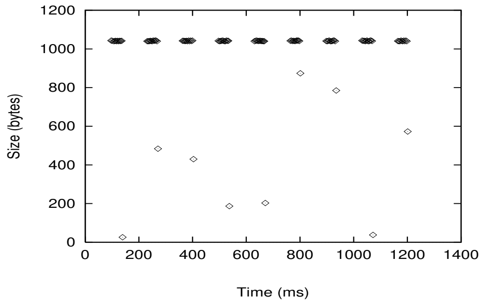

Figure 2: Send profile of an `nv` session.

Aside from being isochronous, distributed multimedia applications require bulk data transfer. For example, an MPEG-2 source, targeted at 4-9 Mbps for broadcast TV quality video, would provide about 1 Mbyte of data per second.

The balance of this paper is as follows. In Section 2, we present an overview of an architecture designed to efficiently support distributed multimedia applications. In Section 3, we elaborate upon *I/O efficient buffers* for reducing the need for memory-to-memory data copy. In Section 4, we introduce a new *fast system call* for low latency network access. In Section 5, we describe *kernel threads* intended for reducing the use of system calls and for rate-based flow control. In Section 6, *direct media streaming* is introduced for applications that are mainly concerned with the transport—rather than processing—of media data. In Section 7, we present performance measurements of the primitive operations in the architecture. In Section 8, we illustrate how to apply the primitive operations for reducing the use of system calls and for implementing rate-based flow control in a distributed multmedia system.

2 Architecture

We propose an architecture designed to provide OS support for multimedia networking. Specifically, it aims to 1) efficiently transfer data between user and kernel (or network) via I/O efficient buffers, 2) efficiently transfer control between user and kernel by using independently scheduled kernel threads instead of explicit system calls, and 3) provide rate-based flow control to a reserved-rate network connection. Our architecture is illustrated in Figure 3.

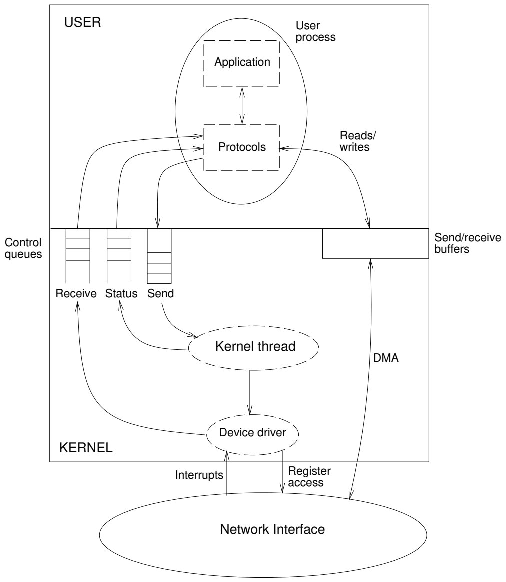

Figure 3: Architecture.

There are various ways to structure protocols in an operating system. We assume a user-level protocol implementation called the protocol server model [11], which facilitates protocol development and customization. From a performance point of view, application and performance-critical protocol code run in the same protection domain; communication between them does not need kernel intervention.

We have left device management in kernel space. There are two reasons. First, the kernel can appropriately schedule use of a device among a set of active users. Second, a user process is prevented from performing operations with global impact on the device (such as resetting the device).

We next discuss the kernel/user interface in our design. When a user process opens a network connection, the kernel automatically allocates control queues (for send, receive and status) to the connection. Receive data buffers are implicitly allocated as packets arrive, while send data buffers are explicitly allocated using a system call interface (see Section 3.1). The data buffers and control queues are *shared* between kernel and user. The kernel enables this sharing by co-mapping[2] the shared areas to both kernel space and the address space of the user process. The cost of establishing this co-mapping is kept low by continually reusing a shared buffer for the same user process.

[2]We assume that a shared buffer is mapped for both reading and writing unless otherwise noted.

In our current system, receives from the network are driven by the network device's receive interrupts. The device driver informs the user process of data available for reading by appending read notifications to the receive control queue. A read notification includes a pointer to a receive buffer of the connection holding a packet to be received. The user process can access both the read notification and the packet by using ordinary memory reads.

When a user process has data to send, it appends a send request to the send control queue. The send control queue is periodically checked by a *kernel thread*, which is a schedulable thread of control created in kernel space. It runs in kernel space as an agent to manage users' sends to the network. In particular, it may perform flow control specified by network connections on behalf of user processes.

A kernel thread is *dedicated* if it serves one user process (its *client*) exclusively; it is *shared* if it can serve multiple clients. If a dedicated kernel thread is used, the data to be sent are found in user buffers that can be anywhere in the user's address space. If a shared kernel thread is used, the data to be sent are found in shared data buffers allocated to the user.

When a user process opens a network connection, the user communicates to its kernel thread a *flow schedule* consisting of two parameter values: 1) a delay to the time when the first check should be scheduled, and 2) the period for rescheduling (e.g., every 30 ms). The kernel thread checks the user's send control queue according to the flow schedule.

When a packet has been transmitted by the network interface, the kernel reports its transmission status by appending a notification to the status queue of the user process that is the packet's source.

In our current system, kernel threads are used only for sending. They can also be used for receiving if the network device is *polled* for packet arrivals (as opposed to generating interrupts).

3 I/O Efficient Buffers

The three different types of buffers in our architecture are classified as follows:

- *User buffers* are only mapped to a user domain.
- *Kernel buffers* (short for *kernel mapped buffers*) are co-mapped to user and kernel. Kernel buffers can be used by a user process to efficiently communicate with the kernel.
- *Network buffers* (short for *network mapped buffers*) are kernel buffers that are additionally mapped for DMA transfers. They can be used for efficient network I/O.

Kernel and network buffers are collectively referred to as I/O efficient buffers. In our current system, kernel buffers are used for control queues, while network buffers are used for data transfers.

3.1 User management of buffers

Our system allows a user process to manage I/O efficient data buffers. A user process explicitly requests an allocation of network buffers for sending by using a system call interface.[3] On the receive side, network buffers used to hold network data destined for a particular user process are implicitly[4] allocated to that user process.

The maximum number of buffers that can be allocated (implicitly or explicitly) is configurable and is enforced by the kernel. The user process is then expected to manage the allocated I/O efficient buffers itself. For example, it is responsible for distinguishing them from ordinary data buffers, and reusing them as much as possible for efficient network I/O. When a user process no longer needs a network buffer, the buffer is returned to the system with a system call.[5]

We believe that it is reasonable to expect an application to manage I/O efficient buffers. We make two observations about multimedia applications: 1) code for network access is typically centralized, and 2) buffers used for network I/O are localized. For example, there is only one place in nv that writes to the network, and all writes are from the same buffer address.

[3] In our current system, this is done through an ioctl() call to the network driver.

[4] In that the user process has not explicitly requested the allocation.

[5] Our implementation uses an ioctl() call to the network driver.

3.2 Protection issues

Giving user code direct access to kernel data structures naturally raises the question of data integrity. Simply put, we would like to prevent a user process from reading and modifying data that belong to another process.

First, we observe that send, receive and status control queues present no problem since they are allocated and mapped on a per connection basis and are private to the user process that owns the connection. Send data buffers are likewise safe, as long as they are allocated in pages, and read/write permissions to the pages are properly granted.

Receive buffers are safe provided that the network controller supports *link level demultiplexing* in hardware.[6] Link level demultiplexing means that the link level header identifies a process level connection endpoint. This enables the network controller to inspect the address information in a packet and decide which receive buffer to store the packet's data. For example, we expect future ATM interface controllers to have this capability, since the VCI/VPI pair of signaling information is adequate to identify a communication endpoint at the process (as opposed to host) level.

4 Fast System Call

We have defined a new system call based upon the use of I/O efficient buffers,

`fast_write(int fd);`

where `fd`, a handle for the network connection for sending, is its only parameter. The actual request containing send parameters is appended to the send control queue. The data being sent are placed in the user's allocated network buffers.

`fast_write()` is designed for low-latency sending. To illustrate, consider the latency[7] of a conventional `write()` system call, which is made up of

[6] In addition, it is necessary to use explicit allocation of I/O buffers for receiving, as well as for sending.

[7] This latency can be interpreted as a measure of service time, i.e., time when the CPU is occupied. As such, it is also a measure of efficiency.

the following components: 1) executing trap handling code for crossing the user/kernel boundary, 2) validating the send parameters, and 3) copying the data being sent from user to kernel space. Note that the potentially costly third component is eliminated in fast_write().

We expect that fast_write() will be useful for transaction-type data where low latency is desired but the data volume is relatively small. In Section 8.1, we illustrate how fast_write() can also be used in conjunction with kernel threads to reduce system call overheads without incurring the delay penalty of a periodically scheduled kernel thread.

5 Kernel Threads

We elaborate on the concept of dedicated and shared kernel threads introduced in Section 3 for send processing.

As its name suggests, a *dedicated kernel thread* serves only one user process (the *client*). It is created in the context of the client and, therefore, has access to the client's entire address space. As a result, any user buffer (as opposed to just network buffers) can be used for sending. However, one memory-to-memory copy is needed from user buffers to network buffers. The kernel thread ensures that only user data are sent by performing a simple range check.[8] In our system, a dedicated kernel thread automatically prepends link level headers[9] to user data and segments user data into multiple link level messages if needed.

Dedicated kernel threads also allow closer cooperation with the user. In particular, upcalls (i.e. kernel directly giving control to user code without process context switching) are simpler to perform.

A *shared kernel thread* serves multiple clients, and cannot assume a client's address space context. Although less flexible from a programming point of view, a shared kernel thread is more efficient than dedicated kernel threads. In our system, we assume that network buffers are used for sending, and the user directly supplies link level packets. Hence there is no memory-to-memory copy.

[8] Checking is needed since a user process could have submitted a kernel address for sending.

[9] Ethernet headers in our system.

Consider a shared kernel thread that maintains a set of "active" clients sharing a network connection. While this set is not empty, the kernel thread is periodically scheduled as a high priority system service. (We assume that the shared kernel thread runs at a higher priority than any of its clients.) When the kernel thread runs at the beginning of a period, its function is to schedule packets of its clients to be transmitted by the network interface for the duration of the period. As a result, send side scheduling of packets is mostly decoupled from CPU scheduling.

6 Direct Media Streaming

For special applications that are mainly concerned with the *transport* of media data over a network, without processing the data, we propose *direct media streaming*. It satisfies the efficiency goals of our architecture by taking advantage of specialized application needs.

In the streaming approach, the user opens a media device (such as an audio or a video board) and a network connection, and links the two together with a system call. The kernel then takes over for the subsequent continuous data transfer. In recording, the kernel streamlines the captured media data directly to the network interface. In playback, it directly writes data received over a specified network connection to the media device. The data path is efficient because

- the streamline is constructed to avoid unnecessary data copies, and
- data transfers do not incur system call overhead.

Note also that the user process retains control over the data path. By using a set of system calls, it can reconfigure the media device, reconfigure the network connection or close the device-network streamline.

7 Performance of Primitives

We have an implementation of our proposed architecture on Sun SPARCstation 10's equipped with the LANCE Ethernet controller and the Dual Basic Rate ISDN (dbri) audio interface.

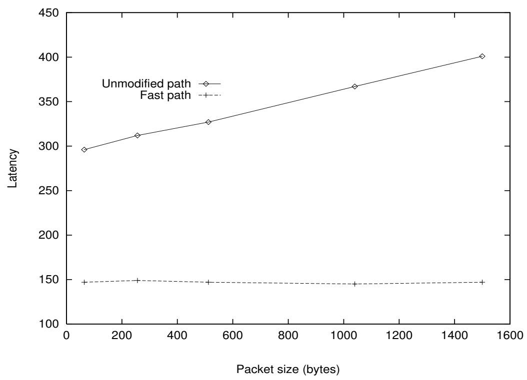

Figure 4: Read path packet latency (μs).

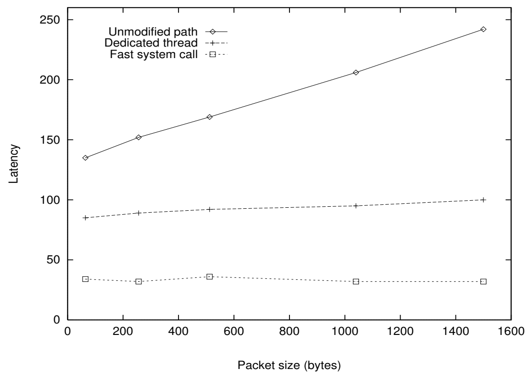

Figure 5: Write path packet latency (μs).

Our current system has been implemented as an extension to SunOS 5.3. Most of the modifications/additions are to the Ethernet driver and the kernel/user interface. Support for direct media streaming has been added to the device independent part of the audio driver. In this section, we report performance measurements of several primitive operations in the architecture.

7.1 Read path packet latency

A set of experiments was performed to measure the per-packet latency of read paths. The fast read path is as described in Section 2 for our architecture. The unmodified read path is as described in Section 1 for SunOS 5.3. In each experiment, a user process blocks trying to read from an empty connection. A packet is then sent from a remote host to the connection to wake up the user process performing the read. The per-packet latency is measured from when the interrupt handler gets control to when the user process wakes up and has access to the received packet. Averaged results over 50 trials are reported. Individual results do not vary much.

The results in Figure 4 show that the latency of the fast read path stays more or less constant at about 147 μs. In contrast, the latency of the unmodified read path increases with packet size.

7.2 Write path packet latency

In this set of experiments, we measured the per packet latency of send side processing. The unmodified write path is as described in Section 1 for SunOS 5.3. Fast system call is as described in Section 4, and dedicated kernel thread in Section 5.

Latency is measured from when the user submits a packet for sending to when the kernel, or kernel thread, appends the packet to the network interface queue.[10] The latency results are shown in Figure 5, which show that fast system call has the lowest latency of the three approaches studied. From Table 1, we see why the latency of fast system call is lower than the latency of unmodified write path (because memory-to-memory copies are avoided) and lower than the latency of dedicated kernel thread (because context switching and memory-to-memory copy are avoided).

7.3 Write path train latency

In this set of experiments, the write path latency for a train of packets was measured. We chose a packet size of 1040 bytes from the observation made in Section 1 about `nv`'s packet size. The number of packets in a train was varied over 20, 30, and 40. Note that if each train represents a video frame, and the frame rate is 30 per second, then the train sizes correspond to video bit rates of 5 Mbps, 7.5 Mbps and 10 Mbps, respectively. This range roughly covers the target bit rates of MPEG-2 video. The reported packet-train latency was measured from when the user process sent the first packet to when control was returned to the user after the kernel, or kernel thread, had appended the last packet to the network interface queue.

[10] Actually, the queue is made up of buffer descriptors for packets.

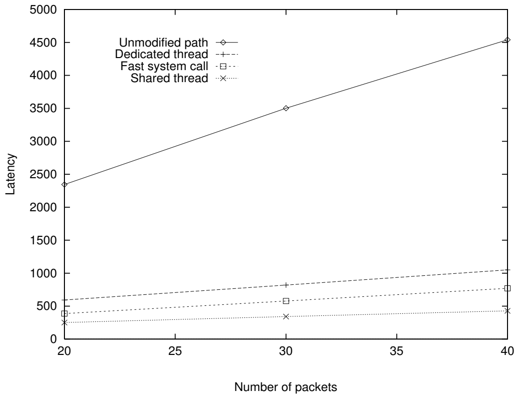

Figure 6: Write path packet-train latency (μs).

Path	Context switch	System call	Copying
Unmodified	none	per packet	twice
Dedicated	one	per train	once
Shared	one	per train	none
Fast sys calls	none	per packet	none

Table 1: Overheads in write paths.

Figure 6 shows that both kernel thread write paths performed significantly better than the unmodified write path. From Table 1, we can see two reasons: 1) reduced memory-to-memory copies, and 2) reduced system call overhead (even though there is an additional context switch).

Observe that shared kernel thread performs better than fast system call. Moreover, the latency of shared kernel thread increases extremely slowly with the number of packets, showing that shared kernel thread is particularly efficient for handling a long sequence of sends (such that the amortized context switch overhead becomes negligible).

7.4 Direct media streaming

We have also evaluated the performance of an audio write path to the network. In our implementation, audio data captured by the SS10 dbri audio device are directly sent by the kernel to the Ethernet controller. Since both the audio device and the network interface support DMA, the data path consists of two DMA transfers: 1) from audio device to main memory, and 2) from main memory to network. No other intermediary data copies are made.

We arranged the audio device to interrupt once every 33 ms. The streaming latency was measured from when the audio interrupt handler gained control to when the kernel appended the audio data to the network interface queue. It was found to be about 67 μs.

8 Applications

In this section, we discuss two specific applications of our architecture to support distributed multimedia.

8.1 Reduced system calls

Multimedia applications may send packets in bursts. When a user process sends a burst of packets to its network connection that has a prespecified reserved rate, the packets cannot be transmitted on the network connection as fast as they arrive. Many of the packets (actually their send requests) will be waiting in the send control queue of the user process.[11] By using a kernel thread to move these packets from the send control queue to the network interface queue (see Figure 7), the overhead of a system call is eliminated for each of these packets without incurring additional delay. In our design that exploits this observation, a user process either sends a packet directly to the network interface, by first appending a request to the send control queue and then calling fast_write(), or just lets the packet wait in the send control queue (without calling fast_write()). The path chosen is determined by an *eager flag*. The eager flag is set by the kernel, but a user process can map the flag read-only in a kernel buffer. Reading the flag thus requires no kernel intervention.

We next present the specifications of two algorithms, one for each of the two paths in Figure 7. When the kernel thread is scheduled, it executes Algorithm KT specified in Figure 8. When fast_write() is called by the user process, Algorithm FS specified in Figure 9 is executed. In the specifications, *curtime* (in ms) denotes the time when an algorithm begins execution, S denotes the media stream sent over a network connection

[11] The send control queue is made up of the user's send requests rather than data packets, which are stored in network buffers.

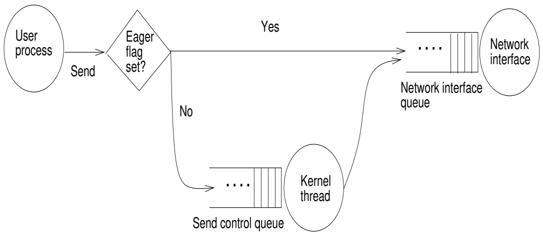

Figure 7: Design for reducing system calls.

Algorithm KT

$nextsched := curtime + period;$
while $(active(S)$ **and** $vclock < nextsched)$
    $vclock :=$ max $(vclock, curtime)+$
        $l(head(S))/BW;$
    netsend $head(S);$
end while;

if $(vclock < nextsched)$
    $eager\_flag :=$ **true**;
else
    $eager\_flag :=$ **false**;

Figure 8: Specification of Algorithm KT.

with reserved rate BW (in kbytes/s), *period* (in ms) denotes the scheduling period of the kernel thread, and, for each packet, say p, $l(p)$ denotes its length in bytes. Define

- $active(S)$ to be true iff S has a packet in the send control queue, and
- if $active(S)$ is true, $head(S)$ to be the packet at the head of the send control queue for S.

The algorithms maintain two state variables: *vclock*, initialized to 0, and *nextsched*, initialized to the time when the kernel thread is first scheduled. *vclock* holds the expected completion time (in ms) of the packet most recently moved from the send control queue to the network interface queue; *nextsched* holds the time (in ms) when the kernel thread is next scheduled to run. The operation, netsend $head(S)$, moves the first packet in the send control queue of S to the network interface queue.

The algorithms work as follows. When enough packets have been put into the network interface queue (by either KT or FS) that will keep

Algorithm FS

{precondition: $(active(S) =$ **true**$) \wedge$
$(eager\_flag =$ **true**$) \wedge (vclock < nextsched)$}

$vclock :=$ max$(vclock, curtime)+$
        $l(head(S))/BW;$
netsend $head(S);$
if $(vclock >= nextsched)$
    $eager\_flag :=$ **false**;

Figure 9: Specification of Algorithm FS.

the reserved-rate connection busy until the kernel thread next runs, the eager flag is cleared; otherwise, the eager flag is set. A cleared eager flag informs the user process that a system call can be avoided after a send request has been appended to the send control queue. Conversely, a set eager flag gives a user process the option to notify the kernel of a send request by using a fast_write() system call instead of waiting for the kernel thread to handle its request.

We evaluated experimentally the above design using 10 seconds of video recorded in a tele-conferencing setting. The video sequence is MPEG-1 encoded with a pattern of IPPP.[12] The average bit rate of the sequence is about 5 Mbps, and the largest picture is about four times the size of the smallest picture. The user process sent a picture every 40 ms[13] over a network connection that had a reserved rate of 5 Mbps. The video sequence was sent using a combination of fast_write() and kernel thread. The packet size was chosen to be 1040 bytes (as in Section 7.3). Table 2 shows the percentage reduction in system calls in four trial runs. The kernel thread was scheduled with a period of 40 ms in all runs. The relative starting times of the user process and the kernel thread were not controlled in the experiments and this accounts for the performance variations in Table 2.

Lastly, we note that applications which are not time critical, such as video playback applications, can altogether ignore the eager flag. The resulting delay penalty is the duration of a checking period

[12] All I (similarly P) pictures are roughly of the same size in the video sequence.

[13] The frame rate of 25 per second was chosen for ease of experimentation. The video segment was recorded at 30 frames per second.

Packets sent	No. of system calls	% reduction
6021	2071	66
6021	378	94
6021	381	94
6021	1857	69

Table 2: Reduction in system calls for sending.

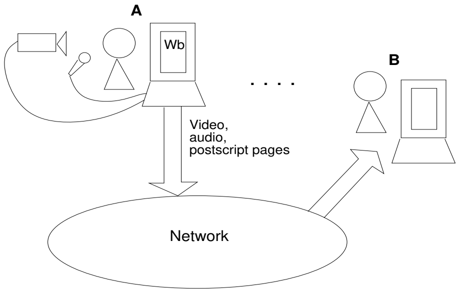

Figure 10: Example tele-conferencing setup.

of the kernel thread in the worst case, and half that duration on the average.

8.2 Rate-based flow control

In this section, we illustrate the use of a kernel thread to implement rate-based flow control for a tele-conferencing scenario. The setup for illustration is shown in Figure 10, where three media streams from participant site A to participant site B are indicated.

In what follows, the tele-conferencing scenario we assume is first described. We next present results from a simulation experiment to illustrate how, in the absence of rate-based flow control, media streams can interfere with one another. The KT algorithm in Figure 8 is then extended to include rate-based flow control. Some measurement results from experimenting with the algorithm implementation are then presented to show the performance of our rate-based flow control algorithm.

Tele-conferencing scenario

Video is captured at a frame rate of 25 per second. Every 40 ms, a video board digitizes a picture captured from the video camera and MPEG encodes it using an IPPP pattern. A video application scheduled at 40 ms intervals reads each encoded picture and sends it to the network as a sequence of 1500 byte packets.

An audio application is invoked every 10 ms to read from an audio device which digitizes incoming audio signal and PCM encodes it at 64 kbps. The application performs silence detection on the encoded data, and a non-silent buffer of about 80 bytes is sent as a single packet. Note that audio is sent with a lower delay (10 ms) than video (40 ms) to facilitate real-time interactions between conference participants.

In addition to video and audio, conference participants use a "whiteboard" application to retrieve pages from a postscript document. A retrieved page is sent over the network by invoking a send function provided in the application. Postscript pages only need to be loosely synchronized (say to about 1 second) with video and audio.

Consider site A in Figure 10. To benefit from statistical multiplexing, video, audio and postscript pages together share a reserved-rate network connection; sharing allows the reserved rate to be better utilized. We assume that video is sent at close to the bit rate of I pictures for minimal delay. As a result, we know that postscript pages can be transported with acceptable delay using the residual reserved rate not needed by P pictures.

Even though sharing a reserved-rate network connection achieves economy for the user, it presents a problem when the different streams interfere with one another. For example, a burst of packets from an I picture can block out audio packets for the entire picture duration of 40 ms, defeating our intention that audio be sent with lower delay. Similarly, a burst of packets from a postcript page can occupy the network connection for an extended period of time, thereby disrupting the continuity requirements of video and audio.

For the experiments described below, we used a sample 4 second sequence of sends from a participant site (such as A in Figure 10). The video stream is active throughout the sample sequence. Its I pictures have a rate of about 4.8 Mbps, while its P picures have a rate of about 1.8 Mbps. For audio, a very short talkspurt is active from about 150 to 650 ms. Figure 11 profiles the sequence of sends made by the video and audio applications from 100 to 800 ms. In addition, roughly 2,175

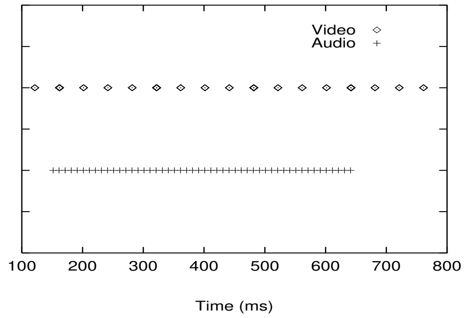

Figure 11: Send profile from 100–800 ms.

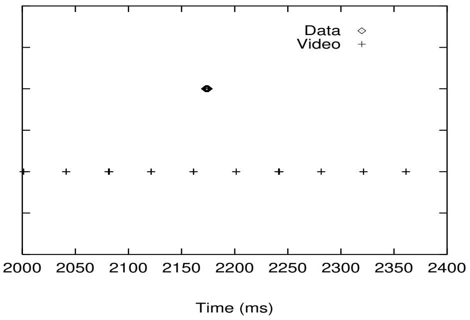

Figure 12: Send profile from 2–2.4 seconds.

ms into the sequence, a postscript page of about 60,000 bytes is sent by the whiteboard application. This profile is shown in Figure 12.

Simulation results

We first illustrate the problem of media flows interfering with one another when there is no rate-based flow control.

The applications generating audio, video, and postscript pages were assigned fixed priorities (audio has highest priority, and postscript pages lowest) in CPU scheduling. We recorded traces of times when send requests were made by these applications. The traces were then used to simulate packet arrivals to the network interface queue. Without rate-based flow control, packets belonging to audio, video, and postscript pages were transmitted by the network interface in FIFO order.

For the 4-second sequence of sends, we recorded their simulated arrival times at a destination site at the other end of a network connection with a reserved rate of 5 Mbps. The arrival times of audio and video packets at the destination site are shown in Figures 13 and 14. Figure 13 shows that the sequence of audio packets is no longer

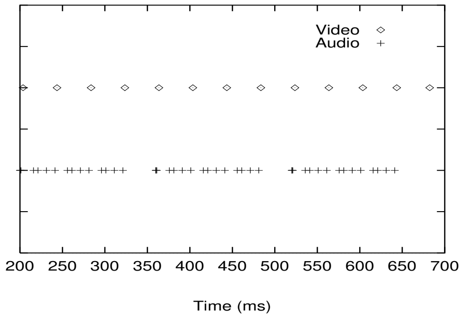

Figure 13: Disrupted sequence of audio arrivals.

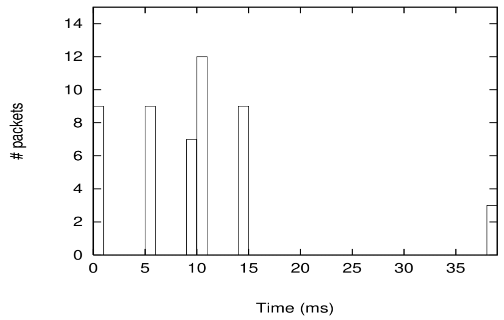

Figure 14: Distribution of inter-packet arrival times for audio.

"smooth". The distribution of inter-arrival times between audio packets (Figure 14) shows that some of the audio packets are delayed by as much as 40 ms because of a preceding I picture. Similarly, Figure 15 shows how the arrivals of video pictures can be interrupted by the burst of packets belonging to a postscript page. For simplicity, only the arrival time of the first packet of each picture is shown in Figure 15; subsequent packets of a picture arrive back-to-back afterwards.

Algorithm implementation and measurements

We tackle the problem by recognizing that each media stream should be assigned a reserved rate based upon its throughput requirement. In the experiment described below, the video stream was assigned a reserved rate of 4.8 Mbps, which is its I picture rate. The audio stream has an inherent bit rate of 64 kbps. However, jitters from CPU scheduling may result in two consecutive audio packets to arrive slightly less than 10 ms apart. Hence, the audio stream was assigned a reserved rate of 80 kbps. Finally, since we expect postscript pages to ultilize the slack bandwidth

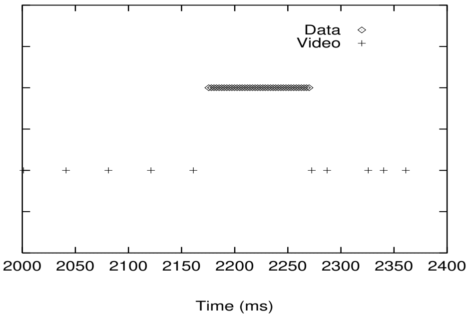

Figure 15: Disrupted sequence of video arrivals.

left by P pictures, it was assigned a nominal reserved rate of 1 kbps.

Having assigned a reserved rate to each media stream, we rely on rate control by a shared kernel thread to provide firewall protection between media streams. For this purpose, Algorithm KT has been extended[14] to Algorithm KT_RC specified in Figure 16.

In addition to the notation in Section 8.1, we use the following: $active$ denotes $\exists Y, active(Y)$. For each media stream, say S, sharing the network connection, define

- $rate(S)$ to be the reserved rate (in kbytes/s) of S, and
- $vclock(S)$ to be a priority value, initially 0, associated with S.

The aggregate reserved rate of the media streams sharing the network connection is less than or equal to BW, the reserved rate of the network connection (which was 5 Mbps in our experiment). $budget$ is equal to the number of bytes that can be sent at BW kbytes/s during a kernel thread period. $sent$ denotes a counter which is initially zero. The kernel thread maintains a set $inactset$ of media streams. Initially, the set contains all of the media streams sharing the network connection.

Algorithm KT_RC is specified such that the kernel thread attempts to send packets in order of increasing *expected finishing time*. Informally, the expected finishing time of a packet is the time (in ms) when the packet would be completely sent if its media stream were served at precisely

[14] For simplicity, we have presented an algorithm for the kernel thread only, without considering the use of fast_write().

Algorithm KT_RC

```
for each S in inactset
    if (active(S))
        vclock(S) :=max(vclock(S), curtime)+
                    l(head(S))/rate(S);
        delete S from inactset;
    fi;

while (active and sent < budget)
    choose S s.t. vclock(S) =
                  min{vclock(Y)|active(Y)};
    sent := sent + l(head(S));
    netsend head(S);
    if (active(S))
        vclock(S) := vclock(S)+
                     l(head(S))/rate(S);
    else
        add S to inactset;
    fi;
end while;
sent := max(sent − budget, 0);
```

Figure 16: Specification of Algorithm KT_RC.

the assigned reserved rate. In practice, the kernel thread is a periodic, rather than a continuous server. It is possible for a higher priority packet (a packet with a smaller expected finishing time) to arrive after the kernel thread has moved some lower priority packets to the network interface queue. However, because the kernel thread does not attempt to schedule ahead by more than its checking period, a higher priority packet cannot be blocked by lower priority packets for longer than the checking period.

We close this section by reporting some measurement results from experimenting with an implementation of Algorithm KT_RC. In this experiment, the network interface sent the sequence of packets from the previous tele-conferencing scenario (Figures 11 and 12) over a dedicated Ethernet and the actual arrival times of packets at a destination host were measured. The shared kernel thread was scheduled with the same period (10 ms) as the audio stream. Figure 17 shows the effects of rate control on the distribution of inter-arrival times between audio packets. Compared with Figure 14, the values are much more concentrated around the expected value of 10 ms. We also observed minimal impact from the burst

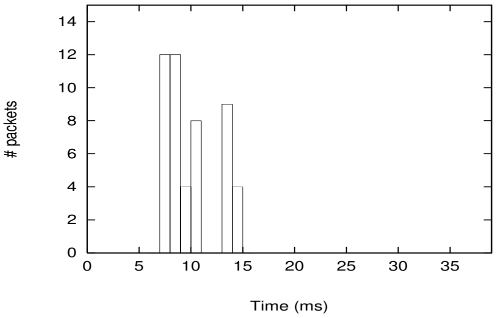

Figure 17: Distribution of inter-packet arrival times for audio with rate control.

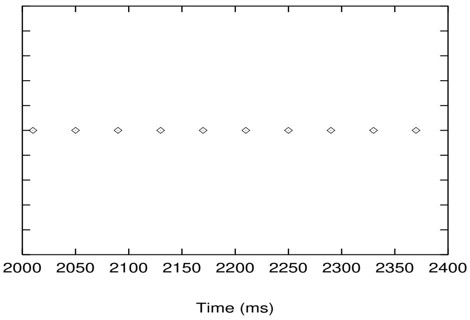

Figure 18: Sequence of video arrivals with rate control (only first packet of each picture is shown).

of postscript packets on the smoothness of video arrivals (Figure 18).

9 Conclusion

We have proposed an architecture including several concepts designed to provide efficient support for multimedia networking. The concept of I/O efficient buffers is similar to various proposals for minimizing data copies, e.g., [4], [6], and [10]. Independent threads of control for network I/O have been used in several systems, such as [7]. We have shown how a thread of control implemented in kernel space can 1) efficiently support sends to the network by user processes, and 2) provide rate-based flow control to a shared network connection. The scheduling algorithms presented in Section 8 are extensions of rate-based proposals for scheduling packets in a network switch [12, 13], and for metascheduling of continuous media [1]. Our algorithms have been designed for send scheduling by a periodic kernel thread. Elimination of system calls for sending has also been proposed in [3] where requests to send are polled in the clock interrupt handler. Issues such as order of interleaving of sends by different user processes as well as impact on end-to-end delay are not considered in [3]. An approach similar to direct media streaming has been considered in [5]. We have implemented an audio write path in SunOS 5.3 and studied its performance. Furthermore, we have described and evaluated two specific applications of our architecture for supporting distributed multimedia.

References

[1] David P. Anderson. Metascheduling for continuous media. *ACM Trans. Computer Systems*, 11(3):226–252, August 1993.

[2] Flavio Bonomi and Kerry W. Fendick. The rate-based flow control framework for the available bit rate ATM service. *IEEE Network*, March/April 1995.

[3] C. Cranor and G.M. Parulkar. Design of universal continuous media I/O. In *Proc. NOSSDAV '95*, pages 83–86, April 1995.

[4] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network*, pages 36–43, July 1993.

[5] Peter Druschel. Operating system support for high-speed networking. Technical Report 94-24 (Ph.D. Dissertation), The University of Arizona, Tucson, Arizona, August 1994.

[6] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 189–202, 1993.

[7] Paul W. Jardetzky, Cormac J. Sreenan, and Roger M. Needham. Storage and synchronization for distributed continuous media. *Multimedia Systems*, 3:151–161, 1995.

[8] C. Partridge. A proposed flow specification. Internet RFC 1363, September 1992.

[9] D.M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[10] Jonathan M. Smith, C. Brendan, and S. Traw. Giving applications access to Gb/s networking. *IEEE Network*, pages 44–52, July 1993.

[11] C.A. Thekkath, T.D. Nguyen, E. Moy, and E.D. Lazowska. Implementing network protocols

at user level. *IEEE/ACM Trans. Networking*, 1(5):554–565, October 1993.

[12] Geoffrey G. Xie and Simon S. Lam. Delay guarantee of Virtual Clock server. Technical Report TR-94-24, October 1994. Available from http://www.cs.utexas.edu/users/lam/NRL; to appear in *IEEE/ACM Transactions on Networking*, December 1995.

[13] Lixia Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. *ACM Transactions on Computer Systems*, 9(2):101–124, May 1991.