

Extended Abstract: Multimedia Network Subsystem Design

See-Mong Tan Willy S. Liao Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
Digital Computer Laboratory
1304 W. Springfield
Urbana, IL 61801
{*stan,liao,roy*}@cs.uiuc.edu

1 Introduction

Modern operating systems must provide time-sensitive performance to networked multimedia applications. Traditional architectures for networking subsystems in operating systems ignore the temporal requirements of real time multimedia data like video and audio. For example, video frames received from a networked multimedia source must be displayed at a given frame rate within certain jitter bounds in order to achieve acceptable playback quality. When packets arrive at the network interface, data delivery to the application level is commonly *first-come-first serve*. Other traffic streams that are not time-sensitive, such as file transfer streams, compete for message handling time with real time data in the network subsystem. This leads to large skews and extreme jitter observed by the multimedia application presenting the data in the presence of interfering data streams.

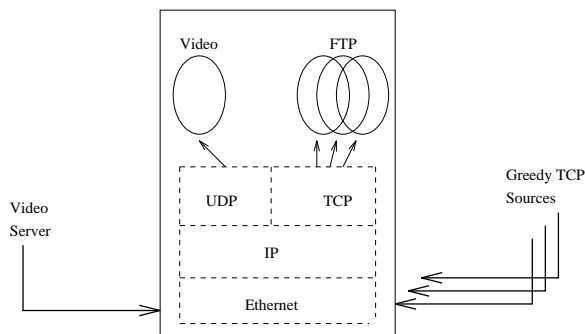


Figure 1: Experimental setup for presentation jitter measurement.

We measured the *presentation level jitter* of a

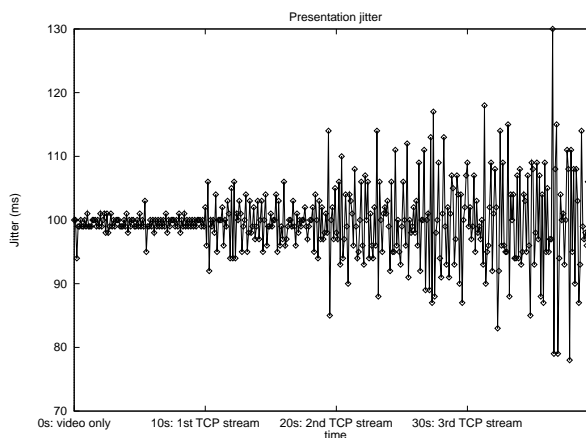


Figure 2: Single video stream and multiple interfering greedy TCP streams.

typical 10 frame per second (fps) video application in the presence of interfering greedy¹ TCP streams for a BSD-like protocol stack. We define presentation jitter here as the difference in the times at which subsequent frames are presented to the application after traversing the system's network protocol stack. The experimental setup (figure 1) consisted of a Sun Sparcstation 20 running the μ Choices operating system[1] in virtual mode[12], similar to the Hive operating system in SimOS[10]. μ Choices includes the *x*-Kernel[7] protocol stack as part of its network subsystem. The *x*-Kernel is similar to traditional protocol stacks that do not include explicit support for continuous media. The experiment consisted of a video source sending to a video client at

¹These streams attempt to send as much data as possible; hence "greedy."

Video Application Statistics	No. of TCP Streams			
	0	1	2	3
Maximum jitter (ms)	103	114	119	130
Minimum jitter (ms)	94	85	79	77
Jitter variance	1	11	49	80

Table 1: Presentation jitter statistics for a 10 frame per second video application in the presence of interfering TCP streams.

a rate of 10 fps on a 10 Mbps Ethernet. Figure 2 depicts the results graphically, while table 1 tabulates the jitter statistics. A separate measurement indicated that network level jitter of video packets received directly off the network interface without going through protocol processing was only a maximum of 5.2% of the presentation-level jitter with 3 interfering TCP streams. The results clearly show that traditional network protocol processing systems do not adequately support networked continuous media. Indeed, efforts to boost the performance of OS networking have so far been limited to affecting higher throughput[4, 5].

In this paper, we examine what is required for the support of time-sensitive data in the network subsystem of a general purpose operating system. The environment we consider is one of workstations connected to an Asynchronous Transfer Mode (ATM) network. We have implemented our design in the network subsystem of the $\mu Choices$ object-oriented operating system. Our experience shows that proper architectural decisions in the design of the networking subsystem are necessary to support time-sensitive real time video and audio. Experiments show close to fifteen-fold improvements in reducing jitter variance between video frames in the presence of interfering non-real time data streams.

2 Network Subsystem Architecture

Packet Demultiplexing Arriving network packets consume CPU resources for message and protocol processing. It is thus crucial that the system differentiates between packets belonging to different streams as early as possible. Emerging network technologies such as ATM networks use connection identifiers (virtual channel identifiers or VCIs) that are accessible at the link level. Much other work also recognize the value of this so-called “early-demultiplexing” of

incoming packets[5, 2]. In $\mu Choices$, VCIs represent end-to-end application communication. Logical streams are always mapped to unique VCIs, thus a continuous media stream is always distinguishable at the network device interface level from other non-time constrained streams.

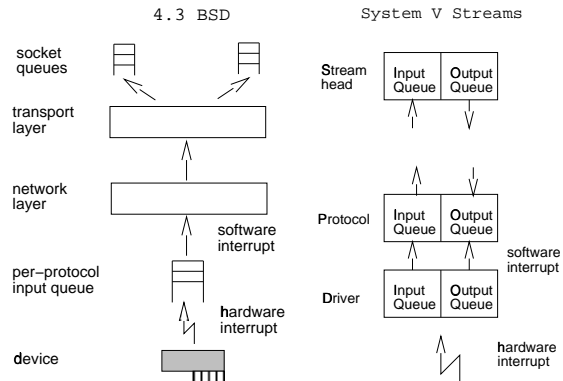


Figure 3: The 4.3BSD and System V Streams Network Subsystem.

Interrupt Processing The network subsystem must make use of VCI information in order to prioritize the handling of data packets. Traditional protocol stacks for systems such as the BSD[11] and System V Streams[3] variants of UNIX, shown in figure 3, fail to do so. In BSD, the device interrupt handler places the new message into a per-protocol input queue. In Streams, the device interrupt handler inserts the message in the Streams queue and puts the queue on a list of queues requesting service. In both systems, a device interrupt on a network device causes the system to post a *software interrupt* (software interrupts run when no higher priority device interrupts occur). In BSD, the software interrupt handler processes messages put into the per-protocol input queues in a first-in, first-out basis. The Streams software interrupt handler similarly processes messages on a queue by queue basis. This approach has several limitations. An urgent message representing a video frame may not overtake a low priority message in the queue. Relative urgencies are not communicated to the level of the software interrupt handler.

The needs of multimedia networking call for prioritizable message processing. The x -Kernel[7] architecture, depicted in figure 4, associates a thread of execution with each message. Threads *shepherd* messages up the protocol stack, which is made up of protocol objects and session objects. Protocol objects in the x -Kernel encapsulate shared data for network

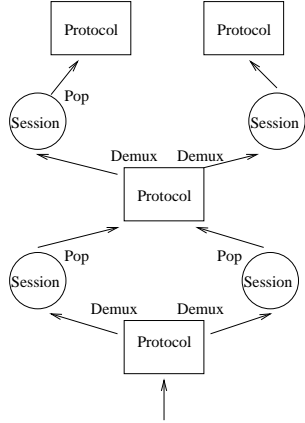


Figure 4: The x -Kernel network architecture.

protocols, while session objects encapsulate the state of individual connections at each level of the protocol stack. A “thread-per-message” model simplifies the task of scheduling and prioritizing message handling and protocol processing. However, the x -Kernel environment removes concurrency and the need for synchronization by making the subsystem a monitor in which only one thread may execute simultaneously. Since the monitor is non-preemptable, this leads to priority-inversion problems when a thread of higher priority becomes ready to execute while a thread of lower priority is executing within the monitor. Finer grained synchronization may be obtained in the x -Kernel by removing the monitor constraint, however, since messages are propagated up the protocol stack with the *Demux* operation (see figure 4) on protocol objects, locks are required to synchronize access between multiple connections to shared state in the protocol objects.

μ Choices Network Architecture The μ Choices network architecture is based on

- virtual connections as end-to-end application communication,
- early demultiplexing in the Network Interface Framework (NIF)[9] with ATM virtual connection identifiers,
- the use of a “thread-per-message” model of message handling, similar to the x -Kernel,
- a parallel protocol stack,
- the integration of shepherd thread execution with the system’s process management and scheduling system, and

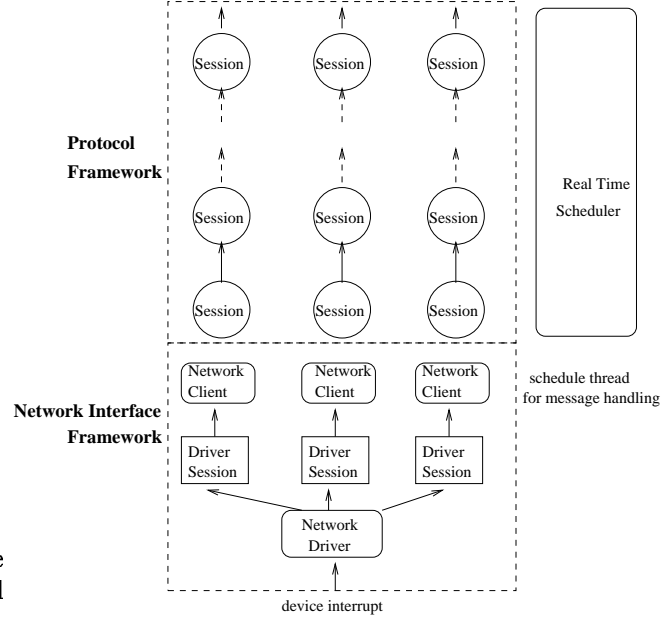


Figure 5: The μ Choices parallel network architecture.

- the specification of Quality of Service (QOS) parameters and its mapping into the framework’s model.

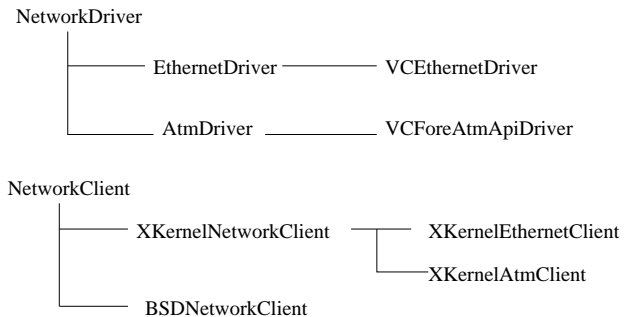


Figure 6: NIF class hierarchy.

Network Interface Framework The NIF in μ Choices is an object-oriented software architecture that supports low latency notification of received packets with flexible data placement and buffer exchange policies. It provides the basis for the “thread-per-message” mode of message handling and its integration with the μ Choices’s process management and scheduling subsystem. QOS specifications for connections are mapped into NIF parameters such as:

- size of receive and transmit buffer pools,
- number of shepherd threads, and

- thread scheduling parameters, eg. priorities, or periods and deadlines.

Figure 6 depicts a portion of the class hierarchy for the NIF. Support is included for “late-demultiplexing” networks like Ethernet with concrete subclasses of the abstract classes *EthernetDriver* and *EthernetClient*, and “early-demultiplexing” networks with the ATM subclasses.

Protocol Processing The NIF imposes no particular structure on the higher-level protocol processing system. Unlike UNIX, where protocol processing is performed at software interrupt time, protocol processing in *μChoices* is accomplished through *fully preemptable threads* dispatched within the NIF. Protocol implementations can thus block and include synchronization constructs such as semaphores, permitting greater flexibility in the implementation of network protocols. We are building parallel versions of IP, UDP and TCP. These implementations contain no shared state between connections. No locking is needed for the access of shared variables. This completely eliminates the priority inversion problem due to locks around shared state variables during protocol processing. In addition, the stack is suitable for multiprocessor implementation. Threads from separate connections can execute independently on different CPUs without interfering with one another.

Scheduling The “thread-per-message” model of message processing introduces potential concurrency problems and possible race conditions in protocol implementations. The integrated network and process subsystems in *μChoices* allows users of the network subsystem to limit the number of simultaneously active threads in a connection. By having only one active thread per connection at any time, protocol implementations can eliminate the use of expensive locks and other synchronization constructs.

Protocol Stack Location *μChoices* locates the protocol stack within the operating system kernel. While the major arguments for user-level protocol libraries have been for ease of maintenance and development, we note that *μChoices* runs in virtual mode specifically for the purposes of system prototyping. We have found this approach to be very beneficial in the development of the operating system itself and this argument naturally extends itself to the protocol processing framework. A kernel implementation also facilitates the integration of scheduling with the NIF that would not be easily attainable

at the application-level. In other work, we developed *customizable message passing frameworks*[8] that allow users to tailor parts of the message system in application-specific ways. We intend to extend these ideas to the network subsystem. Finally, kernel implementations avoid many performance overheads incurred by user-level implementations, for example [6].

3 Results

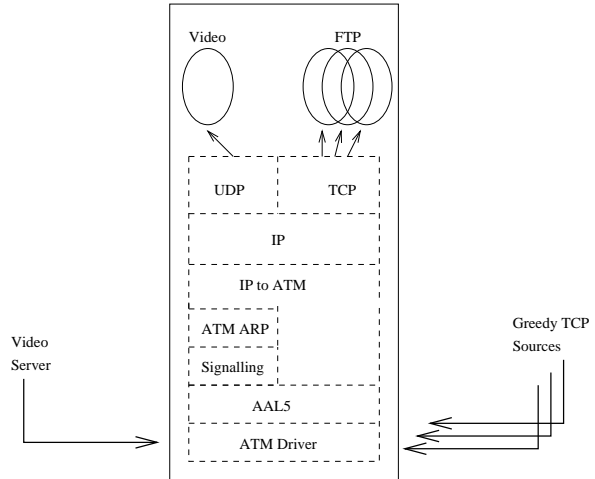


Figure 7: Experimental setup for presentation jitter measurement with ATM protocol stack.

Video Application Statistics	No. of TCP Streams			
	0	1	2	3
Maximum jitter (ms)	102	105	119	120
Minimum jitter (ms)	94	91	90	89
Jitter variance	1	5	6	6
Improvement (old variance/new variance)	1.0	2.2	8.2	14.8

Table 2: Preliminary presentation jitter results for the use of end-to-end ATM VCI application communication over TCP/IP, early demultiplexing, and static priority scheduling in the *μChoices* network subsystem.

Preliminary results were obtained with a version of *μChoices* running in virtual mode on a Sun Sparcstation 20 with the Fore SBA-200 series of ATM adapter boards on a Fore ASX-200 based ATM network of Sparcstations. We repeated the presentation jitter experiment described in section 1 with

early demultiplexing based on ATM VCIs. TCP/IP in the *x*-Kernel protocol stack was extended to use ATM virtual channels as end-to-end application communication. Although the protocol stack executed in *x*-Kernel monitor-mode, static priority was used for thread scheduling. Threads for the 10 fps video stream always ran at a higher priority than the other greedy TCP threads. The results are summarized in table 2. Compared to the vanilla network protocol stack, improvements are seen in both minimum and maximum jitter, as well as jitter variance. Since threads for the video data stream always run at a higher priority than threads for the greedy TCP streams, the presentation jitter is limited to priority inversion caused by at most one non-real time thread executing within the protocol stack when a real time thread becomes ready.

4 Conclusion

Network jitter in modern ATM networks is relatively negligible. Presentation level jitter arises from multiple data streams competing for message processing time within the operating system. Application writers have traditionally had to explicitly buffer video and audio data in order to reduce jitter. As transmission rates and video resolutions increase to high definition standards in the future, such buffering will become increasingly costly. Secondary effects such as decreased memory reference locality due to the larger buffers may also negatively impact system performance. The sensible alternative is for the operating system to handle efficiently networked continuous media.

Without recognizing QOS parameters in network connections and without appropriate structural considerations, the network subsystem cannot adequately support multimedia traffic. *μChoices*'s network subsystem is designed to reduce cross interference between competing message streams. Thread scheduling for the "thread-per-message" style of message handling ensures proper prioritization of incoming packets. A parallel protocol implementation eliminates possible priority inversion due to locking. The preliminary results presented in section 3 show that the approach is indeed tenable. We expect that the full framework, including completely parallel protocol stacks and real time scheduling policies, such as Earliest Deadline First, will significantly minimize presentation jitter in situations where *multiple* real time streams with different QOS parameters execute in parallel with non-real time streams. We also intend a full evaluation of the subsystem's efficiency in

multiprocessor environments.

References

- [1] Roy H. Campbell and See-Mong Tan. *μChoices: An Object-Oriented Multimedia Operating System*. In *Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, Washington, May 1995. IEEE Computer Society.
- [2] Geoffrey Coulson. *Multimedia Application Support in Open Distributed Systems*. PhD thesis, Computing Department, Lancaster University, April 1993.
- [3] D. M. Ritchie. A Stream Input Output System. *AT&T Bell Laboratories Technical Journal*, 63(8):1987–1910, October 1984.
- [4] P. Druschel, M. B. Abbot, M. A. Pagels, and L. L. Peterson. Network Subsystem Design. *IEEE Network Magazine*, July 1993.
- [5] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross domain transfer facility. In *Fourteenth ACM Symposium on Operating Systems Principles*, pages 189–202, Dec 1993.
- [6] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Calamvokis, and C. Dalton. User-space protocols deliver high performance to applications on a low-cost Gb/s LAN. *SIGCOMM '94*, August 1994.
- [7] Norman Hutchinson and Larry Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–75, January 1991.
- [8] Nayeem Islam, Robert E. McGrath, and Roy Campbell. "Parallel Distributed Application Performance and Message Passing: A case study". In *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, San Diego, California, September 1993.
- [9] Willy Liao. Operating System Support for Embedding Network Protocols. Master's thesis, University of Illinois at Urbana-Champaign, 1994.
- [10] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete Computer Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, Fall 1995.
- [11] S. J. Leffler and M. K. McKusick and M. J. Karels and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1988.
- [12] S. M. Tan, D . K. Raila, W. S. Liao, and R. H. Campbell. Virtual Hardware for Operating System Development. Technical report, University of Illinois at Urbana-Champaign, Department of Computer Science, September 1995. <http://choices.cs.uiuc.edu/srg/stan/vchoices.ps>.