6

A Fast, Flexible Network Interface Framework

Willy S. Liao, See-Mong Tan and Roy H. Campbell
University of Illinois at Urbana-Champaign
Department of Computer Science, University of Illinois at Urbana-Champaign, Digital Computer Laboratory, 1304 W. Springfield, Urbana, IL 61801, USA. Telephone: (217) 333-7937. Fax: (217) 333-3501. email: {liao, stan, roy}@cs.uiuc.edu

Abstract

The Network Interface Framework (NIF) is an object-oriented software architecture for providing networking services in the *Choices* object-oriented operating system. The NIF supports multiple client subsystems, provides clients with low-latency notification of received packets, and imposes no particular structure on clients. By contrast, traditional BSD UNIX-style networking does not meet the last two requirements, since it forces clients to use software interrupts and queueing. BSD UNIX cannot accomodate a process-based protocol subsystem such as the *x*-Kernel, whereas the NIF can. We have ported the *x*-Kernel to *Choices* by embedding it into the NIF. Using the standard *x*-Kernel protocol stack with NIF yields Ethernet performance comparable to BSD networking. The NIF is also flexible enough to support services that cannot easily be supported by traditional BSD, such as quality-of-service for multimedia. Preliminary performance results for asynchronous transfer mode (ATM) networks show that the NIF can be used to minimize jitter for continuous media data streams in the presence of non-realtime streams.

Keywords

ATM, Network Interface Framework, resource exchanger, *x*-Kernel

1 INTRODUCTION

The rising popularity of multimedia in distributed systems places greater burdens on the networking services in an operating system than in the past. An important problem is how the system should structure network services to support continuous media such as realtime audio and video efficiently. The requirements for a flexible network architecture are as follows. The operating system must have a software interface to the network that allows multiple users of the network hardware and a great deal of individual flexibility for a user when processing packets. The software interface must support *multiple clients*, it must support *active clients*, and it must be *flexible in client design* and not force any

W. Dabbous et al. (eds.), *Protocols for High-Speed Networks V*

particular processing structure on clients. *Multiple client* support is essential to allow independent users to share the network hardware simultaneously. For example, one client might be handling a realtime video stream while another client handles ordinary data traffic. The clients can differ in how they handle received buffers, and neither must be allowed to starve the other of buffers. This is essentially a resource management issue.

The other two requirements specify what the software interface must permit in terms of clients. Support for *active clients* means that clients must be able to perform some limited processing when their packets arrive. For minimum latency, clients must have access to the hardware interrupt handler. Some clients will perform very limited processing that should be done immediately, such as accumulating packet statistics, rather than requiring the overhead of context switching and synchronization to schedule processes to do it. *Flexibility in client design* requires that the software interface not impose any particular structure on clients. For example, the interface cannot force a client to use a queue on which packets wait to be processed; the client might need to take immediate action, such as adjusting process priorities in response to the packet. Clients must be free to design their processing models exactly to their needs.

BSD UNIX has influenced a great many contemporary UNIX versions on which multimedia applications such as vat and Mosaic run today. It is therefore germane to ask whether the traditional BSD-style networking used on these systems supports the requirements. Unfortunately, BSD networking (Leffler [1989]) does not satisfy the requirement that clients be active. BSD network clients are *passive* entities that are only indirectly invoked by the driver. The hardware interrupt handler takes a received packet, looks up its network-level protocol and places the packet on the per-protocol queue. The driver then posts a software interrupt, and the software interrupt handler performs receive processing for all queued packets. (A software interrupt is handled by the processor when there are no pending hardware interrupts; it is the lowest-priority interrupt in the system.) This arrangement does not satisfy the requirement that clients have access to the hardware interrupt handler, and it also imposes a particular structure, packet queueing, on clients.

We have constructed the object-oriented Network Interface Framework (NIF) to satisfy all the stated requirements. Buffer transport between the driver and a network client is handled efficiently and with low-latency; copying is avoided whenever possible. The NIF manages resources to prevent buffer starvation of one client by another. The NIF also lets clients hook into the hardware interrupt handler so that they are active entities. Finally, the NIF does not impose any particular client policy for received buffers. Unlike BSD, queueing or transfer of buffer ownership is not required.

The NIF provides a scaffold for embedding network clients, so the next step to having the equivalent of BSD's network subsystem is to embed a client providing a protocol stack. For a primary network client, we have chosen the *x*-Kernel of Hutchinson [1991] as a network protocol subsystem whose architecture supports arbitrary composition of protocols and easy construction of new ones. The *x*-Kernel defines an explicit structure for the protocol-protocol interface, which makes it possible to compose protocols in an arbitrary fashion. The *x*-Kernel uses a process-per-message approach, where each received packet is handled by a single process that shepherds it up through the protocol stack. Implementing the *x*-Kernel as an NIF client provides the system with common protocols like TCP/IP and a convenient network protocol architecture. Quality-of-service mechanisms for multimedia can be easily implemented using a process-based client like the *x*-Kernel. For example, in ATM each transport-level connection can be assigned a client

and a thread pool for processing incoming messages. Our results show that even with a static scheduling policy this technique reduces variance in inter-frame arrival time on a video stream in the presence of non-realtime network traffic.

We have implemented the NIF in our object-oriented operating system, *Choices* (Campbell [1993]). *Choices* is an object-oriented multiprocessor operating system written in the C++ programming language (Stroustrup [1986]). In *Choices* all system entities and resources (such as disks, processes, files, memory maps and so forth) are objects, with various frameworks specifying interactions among objects within and across different subsystems (Campbell [1992]). The word *framework* is used in the sense that building a framework involves designing software components that operate together. More precisely, a framework is the specification of the interactions that are permitted between components and their relationships to each other (Deutsch [1989]). (One well-known example is the Model-View-Controller framework of Krasner [1988] in the SmallTalk-80 programming environment.)

The remainder of this paper is organized as follows. Section 2 describes the architecture of the NIF. Section 3 describes how the network hardware drivers and the *x*-Kernel are implemented in *Choices* using the NIF. Section 4 compares the performance of the NIF and the *x*-Kernel against BSD networking on Ethernet. Section 5 illustrates how the NIF can be used to implement a quality-of-service mechanism for ATM, and how performance improves with this mechanism. Section 6 discusses the advantages of the NIF over BSD as they pertain to the design requirements. Finally, the Conclusion summarizes our findings.

2 ARCHITECTURE OF THE NETWORK INTERFACE FRAMEWORK

The Network Interface Framework (NIF) is an object-oriented framework consisting of three classes: NetworkDrivers, NetworkBuffers and NetworkClients. An overview of the NIF is given in Figure 1. NetworkBuffers are exchanged between NetworkClients and NetworkDrivers. The NetworkDriver acts as a multiplexor/demultiplexor in order to deliver received data to the proper NetworkClient or to transmit data. Each NetworkClient must maintain its own pool of NetworkBuffers for receiving and sending—buffers are not shared among clients.

The NIF is a realization of the Resource Exchanger design pattern put forth by Sane [1996]. This pattern, as applied here, states that one network client should not be able to starve the others by consuming all of the resources (buffers). Hence the key principle for buffer management is "conservation of buffers." A NetworkClient will be given a newly received packet only if it has already given the driver an empty NetworkBuffer for that packet. Otherwise the new packet is discarded. Buffers are generally exchanged between the driver and the client, but the NIF also supports clients that require data in particular buffers, using copying in these cases if necessary.

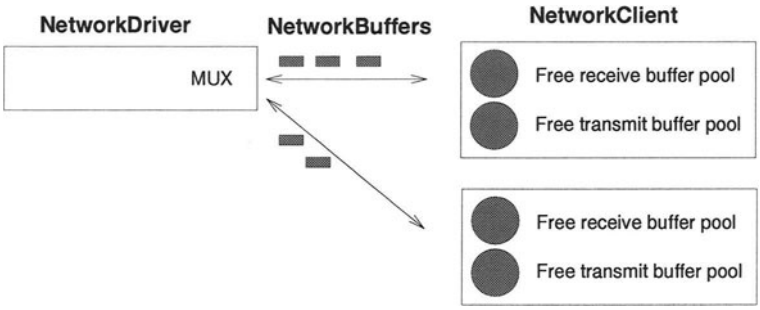

Figure 1 An overview of the NIF. NetworkBuffers flow between one NetworkDriver and many NetworkClients.

2.1 The NetworkClient Class

The NetworkClient base class encapsulates a subsystem that wishes to use the network hardware, which is represented by a NetworkDriver. Each NetworkClient interacts with a single NetworkDriver.

A subclass of NetworkClient defines the receive method to perform some activity when a new packet intended for that client arrives. The NetworkDriver class calls this method at interrupt time with a NetworkBuffer containing the new packet as the argument when it wants to deliver data to a client. The client can process the packet immediately or can defer processing to a later time through an internal queue. The NIF does not impose any queueing or scheduling policy, so low-latency clients can act immediately if they desire.

In order to manage buffers efficiently, the NIF requires that clients provide it with information about their expected source of incoming packets. Each NetworkClient must export the method getReceiveBufferPolicy, which returns the client's *receive buffer policy*. This policy must be one of SwapOrReturn, MustReturn or PeekOnly. The meaning of each policy is given below.

A SwapOrReturn client is indifferent to the source of the buffer that holds the data. The standard driver technique is a buffer Swap. The driver takes empty buffers from the client's receive buffer pool in exchange for buffers from a different location that contain data destined for the client. An alternate technique is buffer Return, where the driver delivers the data in a buffer that the driver earlier obtained from the client's own receive buffer pool.

A MustReturn client must have the data delivered in one of its own NetworkBuffers. A client can use this policy when it has its own subclass of buffer that it must use for reception and so cannot swap away for generic NetworkBuffers.

A PeekOnly client client wishes only to "peek" at the data at interrupt time. The driver immediately delivers the data to the client in some buffer owned by the driver. The data in the buffer will be valid only for the duration of the call to receive, as the driver will then reuse the buffer by storing later packets in it after that call returns. An example of a PeekOnly client is a client that needs only to accumulate some summary information about each incoming packet. PeekOnly clients do not have their own receive buffer pools.

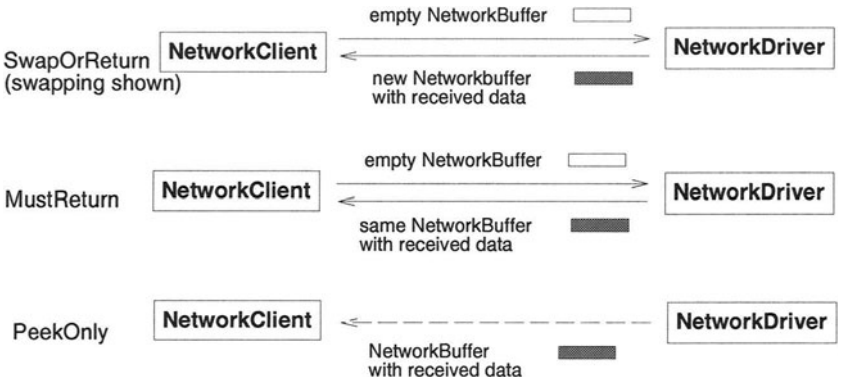

Figure 2 An illustration of client receive buffer policies. A solid arrow denotes a transfer of buffer ownership; a dotted arrow denotes a temporary loan only. Note that the driver's procurement of an empty buffer can occur before the interrupt handler invocation that delivers the new packet to the client.

Figure 2 depicts these policies. In the first two cases above, the client will own the buffer with the data after the interrupt handler completes. The PeekOnly case is applicable whenever a long-term transfer of ownership is unnecessary. Since the receive method is called at interrupt time, there is no process scheduling discipline imposed by the NIF on received packets.

2.2 The NetworkDriver Class

The NetworkDriver manages the host adapter for the network. This class is basically a multiplexor/demultiplexor for NetworkBuffers that flow to and from NetworkClients. As such it contains a registry, and any client that wishes to use a network interface must register itself with the driver. When registering, the client provides a packet type that indicates which packets the client is interested in receiving. Subclasses of NetworkDriver are free to specify the exact meaning of a "type." A special reserved value, DEFAULT_CLIENT_TYPE, indicates the default type, which only one client may register with any particular driver. The client registering this type will receive all packets for which no other client has registered.

The driver also handles the multiplexing of outbound packets from several different sources. Methods are provided for synchronous and asynchronous transmission. In the simplest case, these methods send the packets out using a first-come, first-serve order. More complex strategies can be used by subclasses, such as sorting buffers by some priority function and then transmitting higher priority requests before lower priority ones.

2.3 The NetworkBuffer Class

A NetworkBuffer object encapsulates a packet that is being passed between a driver and a client. Its use ensures a consistent abstract view of memory by all entities in the NIF. A NetworkBuffer represents a packet of data which may or may not be contiguous in memory. A buffer has a maximum length and a current length. The maximum length is fixed at the time the buffer is constructed, while the current length may be changed as new data is put into the object. Methods to copy a buffer to a contiguous memory region or to copy from a contiguous memory region into a buffer are also provided. It is also possible to iterate over the various contiguous subregions of a buffer, and to make the entire buffer contiguous.

The NIF provides a concrete LinearNetworkBuffer subclass that implements a buffer whose data is a single contiguous region in memory. A framework entity (client or driver) that requires no special optimizations can simply use this subclass. A client that wants data to be delivered in objects of its own subclass of NetworkBuffer will need to use the AlwaysReturn receive policy. The data in such subclasses of NetworkBuffer need not be contiguous in memory. The interface of NetworkBuffer has methods to determine whether a buffer object is contiguous and if so, what the address of the data is. The NIF requires a NetworkBuffer to be contiguous only when it is given to the driver by the client during received packet delivery. This is easy to accomplish in practice since the buffer must be empty and thus in the worst case only memory reallocation need be performed to make it contiguous, rather than copying.

3 USING THE NETWORK INTERFACE FRAMEWORK IN *Choices*

This section describes how networking is implemented in *Choices* with the NIF and the *x*-Kernel. Since the NIF is a proper object-oriented framework, subclassing is used to specialize objects for particular hardware and clients. Some brief background on the *Choices* operating system is necessary first, since this section refers to two versions of *Choices*. The native version runs on the SPARCStation 2 among other machines. There is also an emulated or "virtual" operating system that is called *Virtual Choices*. This version runs on top of UNIX as a user process and simulates hardware interrupts with signals and virtual memory through mmap system calls. *Virtual Choices* is extremely useful for debugging and preliminary development.

3.1 Drivers: Ethernet and ATM Support in *Choices*

The NIF supports Ethernet on the native SPARCStation 2 version of *Choices* through subclasses of NetworkDriver. There is an EthernetDriver abstract subclass which knows about Ethernet packet types and Ethernet-specific control operations such as toggling promiscuous mode. The concrete SS1Am7990 subclass of EthernetDriver implements the SPARCStation 2 Ethernet driver. A NetworkClient can register for all packets with the same 16-bit EtherType code; the default NetworkClient will receive all packets no other client has explicitly registered for. The SS1Am7990 class manages an internal pool of receive buffers located in a special DMA region of memory; any buffer that will be used to receive data must be in this memory region. The SS1Am7990 always uses the Swap

receive policy whenever possible. Clients who are willing to swap buffers with the driver allocate their own buffers from the DMA region to allow this. Buffers are always pulled out of the receive ring and given to a client in return for an empty buffer if the client has a SwapOrReturn receive policy. If the client has a MustReturn policy, the buffer is copied rather than swapped, and the original buffer remains in the ring. PeekOnly clients merely examine the buffer during the interrupt handler; as mentioned earlier, there is no structured way to do this in BSD.

The NIF supports ATM through a different scheme that exploits the early-demultiplexing nature of most ATM interface cards. The abstract ATMDriver subclass of NetworkDriver is connection-oriented, unlike the Ethernet driver, and treats each full-duplex virtual circuit (VC) as a "packet type." Each ATM client, when it registers with the NetworkDriver, opens a full-duplex connection to the given destination and thereafter receives and sends all packets on the VC for that connection.

A concrete subclass of ATMDriver implements this abstract design for a particular network card, such as the Fore System 100 and 200 series ATM adapters, but the basic design described below is the same for all subclasses. The ATM adapter delivers all packets received on a given VC into a specified region of memory that was earlier programmed into the hardware by the host's operating system when the VC was opened. Therefore, in its receive buffer pool an ATM client has a fixed number of buffers mapped into a region accessible by the ATM hardware. The ATM driver takes possession of all this memory at the time the client opens its connection so that the hardware can be notified of its location, and afterwards the driver hands back to the client buffers from this pool containing received packets. Clients generally must use a MustReturn policy. A PeekOnly client can use an internal buffer pool in the ATM driver set aside for PeekOnly clients.

3.2 Clients: *x*-Kernel as a NetworkClient in *Choices*

The *x*-Kernel can be embedded into the NIF as one or more clients. The structure of the clients is influenced by the hardware nature of the underlying driver. We first describe the late-demultiplexing client for the Ethernet driver (the (LateDemuxClient), and then we describe the early-demultiplexing client for the ATM driver (the EarlyDemuxClient).

On Ethernet, there is no notion of a connection at the network link level; any connections must be recovered by software (hence "late-demultiplexing"). Therefore it is convenient to instantiate a single client for the entire *x*-Kernel. This client, the LateDemuxClient, is registered for the default packet type (all packets not claimed by other clients), so that non-*x*-Kernel clients can still receive any individual packet types they desire. The LateDemuxClient manages a pool of threads which are used to shepherd received messages upwards through the *x*-Kernel. The LateDemuxClient exports a SwapOrReturn receive policy. LateDemuxClient processing at interrupt time is limited. Its receive method simply takes the NetworkBuffer and places it on a receive queue on which the client's pool of *x*-Kernel threads is waiting. One of these threads will awaken and dequeue the packet, create an *x*-Kernel Msg over it, and then shepherd it upward through the protocol stack. No copying of packet data is necessary to create the Msg.

The EarlyDemuxClient is designed for early-demultiplexing hardware such as ATM. Early-demultiplexing hardware is aware of link-level connections (VCs for ATM) and performs demultiplexing at receive time by putting a packet into a memory location that is a function of the packet's connection identifier. A EarlyDemuxClient represents one VC

Table 1 TCP throughput on Ethernet with 1K messages and 16K socket buffers.

platform	*receive*	*send*
x-Kernel 3.2 in *Choices*	950KB/sec	920KB/sec
BSD 4.4 in *Choices*	1030KB/sec	980KB/sec
SunOS 4.1.3	1080KB/sec	1050KB/sec

Table 2 TCP throughput on Ethernet with 4K messages and 16K socket buffers.

platform	*receive*	*send*
x-Kernel 3.2 in *Choices*	1080KB/sec	920KB/sec
BSD 4.4 in *Choices*	1060KB/sec	980KB/sec
SunOS 4.1.3	1080KB/sec	1060KB/sec

and manages a chunk of memory into which its packets are delivered. It uses a MustReturn policy, since the ATMDriver grabs this memory when the client's VC is opened and hands back buffers with received packts. Each EarlyDemuxClient manages its own pool of threads used for shepherding received packets up the *x*-Kernel protocol stack. This is in contrast to the LateDemuxClient, which is not associated with an individual connection. Interrupt time processing in the EarlyDemuxClient is the same as its late-demultiplexing counterpart. The primary difference in system behavior arises from the fact that there can only be a single LateDemuxClient for any packet type, whereas there can be many EarlyDemuxClients for a given packet type.

The LateDemuxClient and EarlyDemuxClient classes are generic clients for interfacing the NIF to the *x*-Kernel. Therefore both classes handle certain utility functions, such as conversion between *x*-Kernel's Msg objects and the NIF's NetworkBuffers. The *x*-Kernel uses a message library providing lazily-evaluated messages implemented as trees of pointers to data regions. The XKernelNetworkBuffer, a subclass of NetworkBuffer, is an NIF "wrapper" for an *x*-Kernel Msg. XKernelNetworkBuffers are used when transmitting so that the driver can use any hardware scatter-gather facilities, avoiding an extra copy.

4 PERFORMANCE COMPARISON OF BSD AND THE *x*-Kernel ON ETHERNET

This section describes the performance of the *Choices* implementation of the sockets interprocess communication facility (Leffler [1989] and Curry [1989]). The sockets implementation in *Choices* with *x*-Kernel is fully described in Liao [1995]. The data show that the *x*-Kernel in the NIF performs comparably to the BSD protocol stack, implying that moving away from the BSD model of network processing to a process-based model does not impose a great penalty.

User-level performance of the *Choices* sockets implementation was measured on SPARC-

Station 2 workstations with 16 megabytes of RAM on an isolated Ethernet. TCP throughput was tested with the public-domain benchmark application *ttcp*. We used TCP to send or receive from a reference machine, a SPARCStation 2 running SunOS 4.1.3. The socket buffers on the reference machine were set to 32KB to give it plenty of buffer space. SunOS 4.1.3 has a very fast networking subsystem based on BSD UNIX capable of achieving throughput near the theoretical maximum of Ethernet, so it can be excluded from consideration as a bottleneck in any comparison of throughput. The three operating systems tested against this reference machine were *Choices* with *x*-Kernel, *Choices* with BSD 4.4 networking, and SunOS 4.1.3. The version of *Choices* with BSD 4.4 networking was produced independently at an earlier date for experiments in file system caching, before we had elucidated our requirements for the networking architecture. All of these platforms have very similar TCP/IP protocol stacks, as the *x*-Kernel protocol stack is a modified version of the BSD UNIX source, and SunOS 4 networking is also based on BSD. The results obtained by running the benchmark in user mode on both ends are given in Tables 1 and 2.

The tables demonstrate that *x*-Kernel in *Choices* has good performance compared to BSD networking in *Choices*. Neither *Choices* version has throughput as high as SunOS. Aside from observing that SunOS is a mature commercial product that has been highly optimized, we can offer two explanations for this result. There are still a fair number of spin locks and semaphores in the *Choices* kernel, as it is designed to be preemptable and multiprocessor-compliant, unlike SunOS. Also, both *Choices* versions were built with the freely available GNU C++ version 2.5.8, whose optimizations are not as aggressive as a commercial compiler.

Comparing *x*-Kernel and BSD networking in *Choices* is useful since the two are embedded in the same underlying operating system, which factors out implementation differences in areas such as virtual memory. Overall, TCP throughput in *x*-Kernel is around 5–10% less than BSD. Receiving large message sizes is an apparent exception (see Table 2), since *x*-Kernel throughput is the same as the other two platforms. This exception exists only because all three operating systems are operatingly close to the maximum TCP throughput possible on our network; otherwise BSD and SunOS would probably be a little faster than *x*-Kernel. The Tables show that the *x*-Kernel has a higher per-message overhead than BSD.

BSD networking's minor performance advantage over the *x*-Kernel is actually smaller than shown by the Tables. The *x*-Kernel protocol stack is not as optimized as the BSD protocol stack. All BSD message routines are inline macros, whereas the *x*-Kernel data manipulation routines are currently implemented as ordinary functions. There are also only three layers in the BSD protocol stack (Ethernet, IP and TCP), while there are currently five protocol layers in the *x*-Kernel: one extra layer between IP and Ethernet due to the handling of address resolution using ARP, and another due to the separation of hardware-independent and hardware-dependent aspects of Ethernet processing into different layers. This arrangement is excellent for quick addition of Ethernet drivers for new hardware and for creating IP protocol variants. However, for maximum performance the protocol stack can be modified and collapsed into fewer modules to match the three in BSD. After these optimizations *x*-Kernel performance should be very similar to that of BSD.

The point of these figures is not to show that *x*-Kernel might be faster than BSD, but that the NIF can support a rather different model for network protocol processing

without suffering great performance losses. After all, one can always implement BSD-style networking using software interrupts for receive processing in the NIF by defining the appropriate subclasses. We have chosen the *x*-Kernel instead for continuing work because its performance is comparable to BSD and when combined with the NIF it is far more flexible for the work we are pursuing in multimedia networking. The next section gives more detail on the applicability of the NIF and the *x*-Kernel to multimedia and quality of service.

5 SUPPORTING QUALITY OF SERVICE ON ATM

Quality of service (QOS) in networks is an important issue in high-speed networking and continuous media. The NIF can support QOS through the use of multiple network clients connected to a process-based protocol stack such as *x*-Kernel. In this section we describe preliminary work in this area using the NIF to provide operating system support for multimedia and quality of service.

We are now using the NIF to implement quality of service mechanisms (Tan [1996]). The platform for this work is *μChoices*, the microkernel-based successor to *Choices*. The NIF and all clients are currently still implemented within the microkernel, so the platform is virtually identical to the original *Choices*. Consequently, the actual architecture and performance of the NIF on *μChoices* do not differ from the original *Choices*.

To illustrate why quality of service is necessary, consider Table 3. This Table gives a baseline measurement of *presentation jitter* on Ethernet when an application is receiving 10 video frames per second while the host machine is also receiving some background TCP streams (representing FTP sessions). The test platform is *Virtual μChoices* on a Sun Sparcserver 600MP, using the NIF and a LateDemuxClient with *x*-Kernel for protocol processing. Presentation jitter is the variance in interframe arrival time. The Table shows that jitter increases as the number of interfering of TCP streams rises. Jitter due to the network itself was measured separately to be only 5.2% of the presentation jitter with 3 interfering TCP streams, so virtually all the jitter is due to priority inversion in protocol processing. All incoming packets are being processed first-come, first-serve, which means a video packet must often wait for some FTP packets to be processed. This is inappropriate since each video frame has an implicit deadline which recurs every 100 ms, while an FTP packet has no such deadline for its processing. A video frame should not miss its deadline simply because it arrived after some FTP packets.

A quality of service mechanism is necessary at the operating system level in order to reduce jitter on the video stream. The background TCP streams are not time-critical, whereas the data received on the video stream is time-critical and becomes useless if delayed too long before reaching the application. Therefore processing for incoming packets for the video stream must be scheduled differently from processing for the FTP sessions. We realize this goal by giving each transport-level connection its own VC, following the IP over ATM model (Cole [1995]). Since VCs are not shared between higher-level connections, it is then possible to manipulate the thread priorities for the associated EarlyDemuxClients to provide quality of service.

Table 4 shows the same experiment repeated on ATM, with one EarlyDemuxClient per connection. This experiment used the Fore Systems SBA-200 series of ATM adapter boards on a Fore ASX-200 ATM network, with network links of 155 Mbit/s OC3C fiber.

Table 3 Presentation jitter statistics for a 10 frame per second video application in the presence of interfering TCP streams on Ethernet, using one LateDemuxClient with all threads having the same priority.

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

Table 4 Presentation jitter results for the use of end-to-end ATM VCI application communication over TCP/IP with one EarlyDemuxClient per VC, and static priority scheduling.

Video Application Statistics	No. of TCP Streams			
	0	1	2	3
Maximum interframe period (ms)	102	105	119	120
Minimum interframe period (ms)	94	91	90	89
Interframe variance	1	5	6	7
Improvement (old variance/new variance)	1.0	2.2	8.2	14.8

A simple static priority scheme was used: the threads used by the video stream's client were given a higher priority than those associated with the background TCP streams. A great reduction in presentation jitter is evident. Note that the change from Ethernet to ATM is not significant for this experiment, since in both experiments network-level jitter was insignificant and network bandwidth was nowhere near saturated.

More sophisticated scheduling algorithms incorporating deadlines will be implemented later, but our ATM implementation on the NIF is a clear case of an architecture that cannot be emulated with BSD-style networking. Even if the BSD software interrupt handler were modified to handle high priority packets first, all incoming packets would still have to be processed before exiting the handler. Receive-side protocol processing would thus retain absolute priority over all other normal software activities in the system. The NIF allows protocol processing to be intermingled with other system activities by allowing the use of a process-based client like x-Kernel. This flexibility in scheduling is needed to handle packets on different streams with differing degrees of urgency in the outbound as well as inbound directions.

Future work will focus on refining this approach to control jitter and delay for multimedia network traffic on ATM, using the EarlyDemuxClient-per-VC approach. Each client enqueues received data on its respective queue, but the realtime scheduler determines which thread(s) to run next based on the information given by the realtime clients. One applicable realtime scheduling discipline is the Deadline/Workahead Scheduling of Anderson [1990], where realtime processes are critical if they have unprocessed messages

that are within a deadline based on the message's logical (not actual) arrival time. An enhanced EarlyDemuxClient, when invoked by the driver at interrupt time, can check to see if the newly received message would cause the shepherd thread for that packet to go critical. If so, the client increases the priority of the process responsible for handling this message to critical.

6 DISCUSSION

As we have mentioned earlier, the NIF has two benefits over BSD-style networking: lower latency between reception and client processing, and less restrictive client design. This section elaborates further on these advantages.

The NIF has a lower latency between the time it receives a packet and the time a client begins processing, due to its multiple client support and policy-free structure. The BSD hardware interrupt handler places all received packets on a per-protocol queue corresponding to the packet type, and it then posts a software interrupt to perform the protocol processing. Thus the lower bound on packet latency consists of the following: determining the packet type and locating the per-protocol queue, enqueueing the packet, posting a software interrupt, ending the hardware interrupt handler and starting software interrupt processing. In reality, other packets may be ahead of this one on the queue so that the real latency is unpredictable and not within the client's ability to influence. The NIF's latency only includes determining the packet type and locating the client, in all cases. Even though it is often necessary to schedule a real process to do actual protocol processing, it is vital to have the opportunity to perform some activities at interrupt time. For example, the ATM implementation will eventually manipulate process priorities at interrupt time by communicating with a deadline scheduler. For this approach to work, the scheduler must receive notice of events such as packet arrivals as soon as possible after they occur.

The NIF is much more flexible in client design than BSD since it does not require queueing or the software interrupt model. Of course, the BSD model can be implemented in the NIF, but with multimedia operating systems a process-based system is preferable. In BSD all received packets are processed not by true processes, but by a single pseudo-process, the software interrupt handler. The original motivation for using software interrupts was to avoid an expensive process switch, since all processes were heavy-weight and each one ran in its own address space. In modern OSes like *Choices* this motivation no longer holds, since threads are supported at the kernel level and the overhead of a thread switch is not much greater than invoking a software interrupt handler. Threads offer the advantage of being adjustable in priority. The system designer can manipulate process priorities and deadlines to implement quality of service guarantees for network processing. Dynamic scheduling of this sort is just not possible with software interrupts. Even if the software interrupt handler had its own internal priority scheduling, the protocols in BSD are passive and so cannot be called in the hardware interrupt handler to interrogate them for scheduling information.

A further design restriction in BSD network clients is that the software interrupt handler cannot block, since this would have the undesirable side effect of also blocking the process that was running when the software interrupt was taken. Since it is not possible to block the software interrupt handler, the programmer must explicitly save all required state

when later processing is needed. Programming with processes where state is encapsulated in the stack is more convenient and flexible for the implementor. Another important advantage of a process-based system is that multiple processes can be concurrently executed and synchronized with each other in a multiprocessor system. Multiple software interrupt handlers executing concurrently cannot coordinate their activities since they must process all queued packets in a fixed order.

7 CONCLUSION

In summary, the NIF fully supports the requirements for flexible network services in an operating system, whereas BSD does not. The NIF provides an architecture for allowing independent network clients to access network hardware. It satisfies the requirement that the clients' buffer handling policies not interfere with each other through the Resource Exchanger paradigm. It also allows clients the opportunity to manipulate and react to received data at interrupt time, as the clients are *active* objects that share the network hardware. Most importantly, the NIF does not force any particular structure onto client software. This requirement is the major failing of BSD, which tightly binds its clients to its model of network services involving software interrupts and queueing. Moving away from this model does not impose significantly greater performance overheads; the NIF supports a process-based protocol stack (the *x*-Kernel) with performance comparable to BSD's.

The NIF is not tied to any one design for client software, and its ability to support process-based solutions such as the *x*-Kernel gives it greater flexibility than traditional BSD. New features of network hardware such as early-demultiplexing can be exploited by properly-designed NetworkClients. The NIF can support a quality of service mechanism for ATM through the use of one client per connection and static scheduling. Our results show that this mechanism reduces jitter on a realtime data stream in the presence of background, interfering non-realtime streams. This scheme can be generalized so that each client provides feedback to a QOS module that uses process priorities and deadlines to improve QOS even further. Our future work will involve building on top of the NIF and continuing to develop it to support quality-of-service and high-speed multimedia.

REFERENCES

Anderson, D.P. (1990) Meta-scheduling for distributed continuous media. Technical Report UCB/CSD 90/599, University of California at Berkeley EECS Department.

Campbell, R., Islam, N., Madany, P. and Raila, D. (1993) Designing and implementing Choices: an object-oriented system in C++. *Communications of the ACM*. September 1993.

Campbell, R., Islam, N. and Madany, P. (1992) Choices, frameworks and refinement revisited. Technical Report UIUCDCS-R-92-1769, Department of Computer Science, University of Illinois at Urbana-Champaign.

Cole, R.G., Shur, D.H. and Villamizar, C. (1995) IP over ATM: A framework document. Internet draft. October 1995.

Curry, D.A. (1989) *Using C on the UNIX System. A Guide to System Programming*, O'Reilly and Associates, Inc.

Deutsch, L.P. (1989) Design reuse and frameworks in the Smalltalk-80 programming system. *Software Reusability*, volume II, 55–71.

Govindan, R. and Anderson, D.P. (1991) Scheduling and IPC mechanisms for continuous media. *Proceedings of the 13th ACM Symposium on Operating System Principles.*

Hutchinson, N. and Peterson, L. (1991) The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1), 64–75.

Krasner, G.E. and Pope, S.T. (1988) A cookbook for using the model-view-controller paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 26–49.

Leffler, S.J., McKusick, M.K., Karels, M.J. and Quarterman, J.S. (1989) *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley.

Liao, W.S. (1995) *Operating System Support for Embedding Network Subsystems*, master's thesis, University of Illinois at Urbana-Champaign.

Sane, A. and Campbell, R. (1996) Resource exchanger: A behavioral pattern for low overhead concurrent resource management. *Pattern Languages of Program Design*, Addison-Wesley.

Stroustrup, B. (1986) *The C++ Programming Language.*

Tan, S.M., Liao, W.S. and Campbell, R.H. (1996) Multimedia network subsystem design. *Proceedings of the Sixth International Workshop on Network and Operating Systems Support for Digital Audio and Video.*