The following paper was originally published in the
Proceedings of the USENIX 2nd Symposium on
Operating Systems Design and Implementation
Seattle, Washington, October 1996

Lazy Receiver Processing (LRP):
A Network Subsystem Architecture for Server Systems

Peter Druschel and Gaurav Banga
Rice University

Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems

Peter Druschel and Gaurav Banga*

Department of Computer Science
Rice University
Houston, TX 77005

Abstract

The explosive growth of the Internet, the widespread use of WWW-related applications, and the increased reliance on client-server architectures places interesting new demands on network servers. In particular, the operating system running on such systems needs to manage the machine's resources in a manner that maximizes and maintains throughput under conditions of high load. We propose and evaluate a new network subsystem architecture that provides improved fairness, stability, and increased throughput under high network load. The architecture is hardware independent and does not degrade network latency or bandwidth under normal load conditions.

1 Introduction

Most work on operating system support for high-speed networks to date has focused on improving message latency and on delivering the network's full bandwidth to application programs [1, 5, 7, 21]. More recently, researchers have started to look at resource management issues in network servers such as LAN servers, firewall gateways, and WWW servers [16, 17]. This paper proposes a new network subsystem architecture based on *lazy receiver processing (LRP)*, which provides stable overload behavior, fair resource allocation, and increased throughput under heavy load from the network.

State of the art operating systems use sophisticated means of controlling the resources consumed by application processes. Policies for dynamic scheduling, main memory allocation and swapping are designed to ensure graceful behavior of a timeshared system under various load conditions. Resources consumed during the processing of network traffic, on the other hand, are generally not controlled and accounted for in the same manner. This poses a problem for network servers that face a large volume of network traffic, and potentially spend considerable amounts of resources on processing that traffic.

In particular, UNIX based operating systems and many non-UNIX operating systems use an interrupt-driven network subsystem architecture that gives strictly highest priority to the processing of incoming network packets. This leads to scheduling anomalies, decreased throughput, and potential resource starvation of applications. Furthermore, the system becomes unstable in the face of overload from the network. This problem is serious even with the relatively slow current network technology and will grow worse as networks increase in speed.

We propose a network subsystem architecture that integrates network processing into the system's global resource management. Under this system, resources spent in processing network traffic are associated with and charged to the application process that causes the traffic. Incoming network traffic is scheduled at the priority of the process that receives the traffic, and excess traffic is discarded early. This allows the system to maintain fair allocation of resources while handling high volumes of network traffic, and achieves system stability under overload.

Experiments show that a prototype system based on LRP maintains its throughput and remains responsive even when faced with excessive network traffic on a 155 Mbit/s ATM network. In comparison, a conventional UNIX system collapses under network traffic conditions that can easily arise on a 10 Mbit/s Ethernet. Further results show increased fairness in resource allocation, traffic separation, and increased throughput under high load.

The rest of this paper is organized as follows. Section 2 gives a brief overview of the network subsystem found in

*This work supported in part by National Science Foundation Grant CCR-9503098

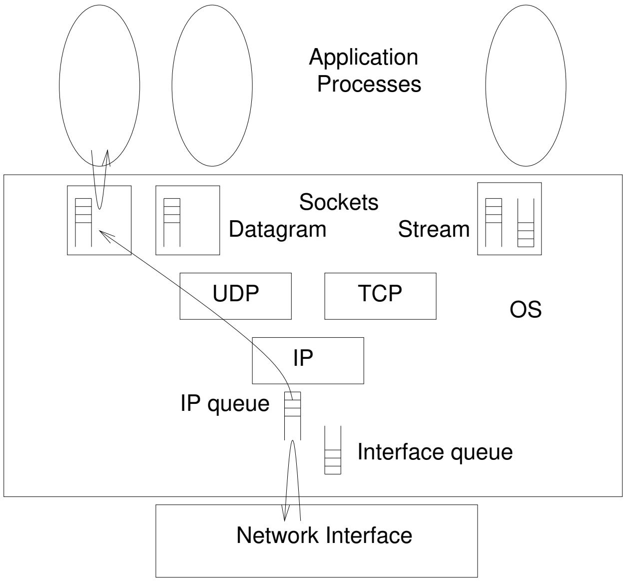

Figure 1: BSD Architecture

BSD UNIX-derived systems [13] and identifies problems that arise when a system of this type is used as a network server. The design of the LRP network architecture is presented in Section 3. Section 4 gives a quantitative performance evaluation of our prototype implementation. Finally, Section 5 covers related work and and Section 6 offers some conclusions.

2 UNIX Network Processing

This section starts with a brief overview of network processing in UNIX operating systems. It then points out problems that arise when a system of this type faces large volumes of network traffic. Finally, we argue that these problems are important by discussing common sources of high network traffic.

To simplify the discussion, we focus on the TCP/UDP/IP protocol suite, and on BSD-derived UNIX systems [13]. Similar problems arise with other protocol suites, in System V-derived UNIX systems, and in many commercial non-UNIX operating systems. Figure 1 illustrates the BSD networking architecture.

2.1 Overview

On the receiving side, the arrival of a network packet is signaled by an interrupt. The interrupt handler, which is part of the network interface device driver, encapsulates the packet in an *mbuf*, queues the packet in the *IP queue*, and posts a software interrupt. In the context of this software interrupt, the packet is processed by IP. After potential reassembly of multiple IP fragments, UDP's or TCP's input function is called, as appropriate. Finally—still in the context of the software interrupt—the packet is queued on the *socket queue* of the socket that is bound to the packet's destination port. The software interrupt has higher priority than any user process; therefore, whenever a user process is interrupted by a packet arrival, the protocol processing for that packet occurs *before* control returns to the user process. On the other hand, software interrupts have lower priority than hardware interrupts; thus, the reception of subsequent packets can interrupt the protocol processing of earlier packets.

When an application process performs a receive system call[1] on the socket, the packet's data is copied from the mbufs into the application's address space. The mbufs are then dequeued and deallocated. This final processing step occurs in the context of the user process performing a system call.

On the sending side, data written to a socket by an application is copied into newly allocated mbufs. For datagram sockets (UDP), the mbufs are then handed to UDP and IP for transmission. After potential fragmentation, the resulting IP packets are then transmitted, or—if the interface is currently busy—placed in the driver's interface queue. All of these actions are executed in the context of the user process that performed the send system call on the socket. Packets queued in the interface queue are removed and transmitted in the context of the network interface's interrupt handler.

For stream sockets (TCP), the mbufs are queued in the socket's outgoing socket queue, and TCP's output function is called. Depending on the state of the TCP connection and the arguments to the send call, TCP makes a logical copy of all, some, or none of the queued mbufs, processes them for transmission, and calls IP's output function. The resulting IP packets are then transmitted or queued on the interface queue. Again, this processing occurs in the context of the application process performing a system call. As for UDP packets, data is removed from the interface queue and transmitted in the context of the network interface's interrupt handler.

Processing of any remaining data in the socket queue typically occurs in the context of a software interrupt. If TCP receives an acknowledgment, more data from the socket queue may be sent in the context of the software interrupt that was posted to process the incoming acknowledgment. Or, data may be sent in the context of a software interrupt that was scheduled by TCP to indicate a timeout. Data is not removed from the socket queue until its reception was acknowledged by the remote receiver.

CPU time consumed during the processing of network I/O is accounted for as follows. Any processing that

[1] We use the term *receive system call* to refer to any of the five system calls available to read data from a socket. The term *send system call* is used analogously to refer to system calls that write data to a socket.

occurs in the context of a user process performing a system call is charged to that process as system time. CPU time spent in software or hardware interrupt handlers is charged to the user process *that was interrupted*. Note that in general, the interrupted process may be unrelated to the network communication that caused the interrupt.

2.2 Problems

We now turn to describe several problems that can arise when a system with conventional network architecture faces high volumes of network traffic. Problems arise because of four aspects of the network subsystem:

Eager receiver processing
Processing of received packets is strictly interrupt-driven, with highest priority given to the capture and storage of packets in main memory; second highest priority is given to the protocol processing of packets; and, lowest priority is given to the applications that consume the messages.

Lack of effective load shedding Packet dropping as a means to resolve receiver overload occurs only after significant host CPU resources have already been invested in the dropped packet.

Lack of traffic separation Incoming traffic destined for one application (socket) can lead to delay and loss of packets destined for another application (socket).

Inappropriate resource accounting CPU time spent in interrupt context during the reception of packets is charged to the application that happens to execute when a packet arrives. Since CPU usage, as maintained by the system, influences a process's future scheduling priority, this is unfair.

Eager receiver processing has significant disadvantages when used in a network server. It gives highest priority to the processing of incoming network packets, regardless of the state or the scheduling priority of the receiving application. A packet arrival will always interrupt a presently executing application, even if any of the following conditions hold true: (1) the currently executing application is not the receiver of the packet; (2) the receiving application is not blocked waiting on the packet; or, (3) the receiving application has lower or equal priority than the currently executing process. As a result, overheads associated with dispatching and handling of interrupts and increased context switching can limit the throughput of a server under load.

Under high load from the network, the system can enter a state known as *receiver livelock* [20]. In this state, the system spends all of its resources processing incoming network packets, only to discard them later because no CPU time is left to service the receiving application programs. For instance, consider the behavior of the system under increasing load from incoming UDP packets [2]. Since hardware interface interrupt and software interrupts have higher priority than user processes, the socket queues will eventually fill because the receiving application no longer gets enough CPU time to consume the packets. At that point, packets are discarded when they reach the socket queue. As the load increases further, the software interrupts will eventually no longer keep up with the protocol processing, causing the IP queue to fill. The problem is that early stages of receiver processing have strictly higher priority than later stages. Under overload, this causes packets to be dropped only after resources have been invested in them. As a result, the throughput of the system drops as the offered load increases until the system finally spends all its time processing packets only to discard them.

Bursts of packets arriving from the network can cause scheduling anomalies. In particular, the delivery of an incoming message to the receiving application can be delayed by a burst of *subsequently* arriving packets. This is because the network processing of the entire burst of packets must complete before any application process can regain control of the CPU. Also, since all incoming IP traffic is placed in the shared IP queue, aggregate traffic bursts can exceed the IP queue limit and/or exhaust the mbuf pool. Thus, traffic bursts destined for one server process can lead to the delay and/or loss of packets destined for other sockets. This type of traffic interference is generally unfair and undesirable.

2.3 Sources of High Network Load

Network protocols and distributed application programs use flow control mechanisms to prevent a sender process from generating more traffic than the receiver process can handle. Unfortunately, flow control does not necessarily prevent overload of network server machines. Some reasons for this are:

- simultaneous requests from a large number of clients
- misbehaved distributed applications
- incorrect client protocol implementations
- malicious denial-of-service attacks
- broadcast and multicast traffic

TCP connection establishment requests (TCP SYN packets) from a large number of clients can flood a WWW server. This is true despite TCP's flow control mechanism

[2] Similar problems can arise under load from TCP connection establishment request packets.

(which regulates traffic on established connections) and TCP's exponential backoff strategy for connection establishment requests (which can only limit the rate of retries). The maximal rate of SYN packets is only bounded by the capacity of the network. Similar arguments apply for any server that serves a virtually unlimited client community such as the Internet.

Distributed applications built on top of a simple datagram service such as UDP must implement their own flow and congestion control mechanisms. When these mechanisms are deficient, excessive network traffic can result. Incorrect implementations of flow-controlled protocols such as TCP—not uncommon in the PC market—can have the same effect. The vulnerability of network servers to network traffic overload can be and has been exploited for security attacks[3]. Thus, current network servers have a protection and security problem, since untrusted application programs running on clients can cause the failure of the shared server.

There are many examples of real-world systems that are prone to the problems discussed above. A packet filtering application-level gateway, such as a firewall, establishes a new TCP connection for every flow that passes through it. An excessive flow establishment rate can overwhelm the gateway. Moreover, a misbehaving flow can get an unfair share of the gateway's resources and interfere with other flows that pass through it. Similar problems can occur in systems that run several server processes, such as Web servers that use a process per connection; or, single process servers that use a kernel thread per connection. Scheduling anomalies, such as those related to bursty data, can be ill-afforded by systems that run multimedia applications. Apart from the above examples, any system that uses eager network processing can be livelocked by an excess of network traffic—this need not always be part of a denial of service attack, and can simply be because of a program error.

These problems make it imperative that a network server be able to control its resources in a manner that ensures efficiency and stability under conditions of high network load. The conventional, interrupt-driven network subsystem architecture does not satisfy this criterion.

3 Design of the LRP Architecture

In this section, we present the design of our network subsystem architecture based on lazy receiver processing (LRP). We start with an overview, and then focus on details of protocol processing for UDP and TCP.

[3]Often, a denial-of-service attack is used as part of a more elaborate security attack.

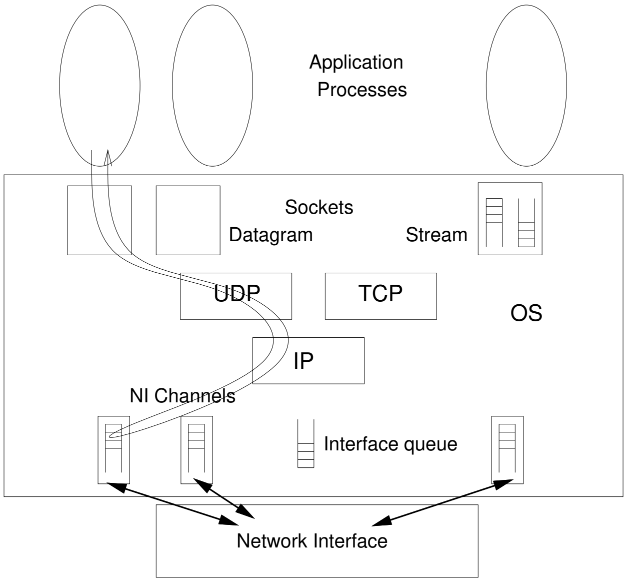

Figure 2: LRP Architecture

The proposed architecture overcomes the problems discussed in the previous section through a combination of techniques: (1) The IP queue is replaced with a per-socket queue that is shared with the network interface (NI). (2) The network interface demultiplexes incoming packets according to their destination socket, and places the packet directly on the appropriate receive queue[4]. Packets destined for a socket with a full receiver queue are silently discarded (early packet discard). (3) Receiver protocol processing is performed at the priority of the receiving process[5]. (4) Whenever the protocol semantics allow it, protocol processing is performed lazily, in the context of the user process performing a receive system call. Figure 2 illustrates the LRP architecture.

There are several things to note about the behavior of this architecture. First, protocol processing for a packet in many cases does not occur until the application requests the packet in a receive system call. Packet processing no longer interrupts the running process at the time of the packet's arrival, unless the receiver has higher scheduling priority than the currently executing process. This avoids inappropriate context switches and can increase performance.

Second, the network interface separates (demultiplexes) incoming traffic by destination socket and places packets directly into per-socket receive queues. Combined with the receiver protocol processing at application

[4]The present discussion assumes that the network interface has an embedded CPU that can be programmed to perform this task. Section 3.2 discusses how LRP can be implemented with an uncooperative NI.

[5]For a shared or multicast socket, this is the highest of the participating processes' priorities.

priority, this provides feedback to the network interface about application processes' ability to keep up with the traffic arriving at a socket. This feedback is used as follows: Once a socket's receive queue fills, the NI discards further packets destined for the socket until applications have consumed some of the queued packets. Thus, the NI can effectively shed load without consuming significant host resources. As a result, the system has stable overload behavior and increased throughput under high load.

Third, the network interface's separation of received traffic, combined with the receiver processing at application priority, eliminates interference among packets destined for separate sockets. Moreover, the delivery latency of a packet cannot be influenced by a subsequently arriving packet of equal or lower priority. And, the elimination of the shared IP queue greatly reduces the likelihood that a packet is delayed or dropped because traffic destined for a different socket has exhausted shared resources.

Finally, CPU time spent in receiver protocol processing is charged to the application process that receives the traffic. This is important since the recent CPU usage of a process influences the priority that the scheduler assigns a process. In particular, it ensures fairness in the case where application processes receive high volumes of network traffic.

Early demultiplexing—a key component of LRP's design—has been used in many systems to support application-specific network protocols [11, 23], to avoid data copying [6, 21], and to preserve network quality-of-service guarantees for real-time communication [10]. Demultiplexing in the network adaptor and multiple NI channels have been used to implement low-latency, high-bandwidth, user-level communication [1, 5]. Protocol processing by user-level threads at application priority has been used in user-level network subsystem implementations [10, 11, 23]. What is new in LRP's design is (1) the lazy, delayed processing of incoming network packets, and (2) the combination and application of the above techniques to provide stability, fairness, and increased throughput under high load. A full discussion of related work is given in Section 5.

It is important to note that the two key techniques used in LRP—lazy protocol processing at the priority of the receiver, and early demultiplexing—are both necessary to achieve stability and fairness under overload. Lazy protocol processing trivially depends on early demultiplexing. To see this, observe that the receiver process of an incoming packet must be known to determine the time and priority at which the packet should be processed.

Conversely, early demultiplexing by itself is not sufficient to provide stability and fairness under overload. Consider a system that combines the traditional eager protocol processing with early demultiplexing. Packets are dropped immediately in case their destination socket's receive queue is full. One would expect this system to remain stable under overload, since traffic arriving at an overloaded endpoint is discarded early. Unfortunately, the system is still defenseless against overload from incoming packets that do not contain valid user data. For example, a flood of control messages or corrupted data packets can still cause livelock. This is because processing of these packets does not result in the placement of data in the socket queue, thus defeating the only feedback mechanism that can effect early packet discard.

In addition, early demultiplexing by itself lacks LRP's benefits of reduced context switching and fair resource allocation, since it shares BSD's resource accounting and eager processing model. A quantitative comparison of both approaches is given in Section 4. We proceed with a detailed description of LRP's design.

3.1 Sockets and NI Channels

A *network interface (NI) channel* is a data structure that is shared between the network interface and the OS kernel. It contains a receiver queue, a free buffer queue, and associated state variables. The NI determines the destination socket of any received packets and queues them on the receive queue of the channel associated with that socket. Thus, the network interface effectively demultiplexes incoming traffic to their destination sockets.

When a socket is bound to a local port (either implicitly or explicitly by means of a bind() system call), an NI channel is created. Also, when a connected stream socket is created, it is allocated its own NI channel. Multiple sockets bound to the same UDP multicast group share a single NI channel. All traffic destined for or originating from a socket passes through that socket's NI channel.

3.2 Packet Demultiplexing

LRP requires that the network interface be able to identify the destination socket of an incoming network packet, so that the packet can be placed on the correct NI channel. Ideally, this function should be performed by the NI itself. Incidentally, many commercial high-speed network adaptors contain an embedded CPU, and the necessary demultiplexing function can be performed by this CPU. We call this approach LRP with *NI demux*. In the case of network adaptors that lack the necessary support (e.g., inexpensive Fast Ethernet adaptors), the demultiplexing function can be performed in the network driver's interrupt handler. We call this approach *soft demux*. Here, some amount of host interrupt processing is necessary to demultiplex incoming packets. Fortunately, with current technology, this overhead appears to be small enough to still maintain good stability under overload. The advantage of this approach is that it will work with any network adaptor, i.e., it is hardware independent. We will quantitatively evaluate both demultiplexing approaches in Section 4.

Our demultiplexing function is self-contained, and has minimal requirements on its execution environment (non-blocking, no dynamic memory allocation, no timers). As such, it can be readily integrated in a network interface's firmware, or the device's host interrupt handler. The function can efficiently demultiplex all packets in the TCP/IP protocol family, including IP fragments. In rare cases, an IP fragment does not contain enough information to allow demultiplexing to the correct endpoint. This happens when the fragment containing the transport header of a fragmented IP packet does not arrive first. In this case, the offending packet is placed on a special NI channel reserved for this purpose. The IP reassembly function checks this channel queue when it misses fragments during reassembly.

Throughout this paper, whenever reference is made to actions performed by the network interface, we mean that the action is performed either by the NI processor (in the case of NI demux), or the host interrupt handler (in the case of soft demux).

3.3 UDP protocol processing

For unreliable, datagram-oriented protocols like UDP, network processing proceeds as follows: The transmit side processing remains largely unchanged. Packets are processed by UDP and IP code in the context of the user process performing the send system call. Then, the resulting IP packet(s) are placed on the interface queue.

On the receiving side, the network interface determines the destination socket of incoming packets and places them on the corresponding channel queue. If that queue is full, the packet is discarded. If the queue was previously empty, and a state flag indicates that interrupts are requested for this socket, the NI generates a host interrupt[6]. When a user process calls a receive system call on a UDP socket, the system checks the associated channel's receive queue. If the queue is non-empty, the first packet is removed; else, the process is blocked waiting for an interrupt from the NI. After removing a packet from the receive queue, IP's input function is called, which will in turn call UDP's input function. Eventually the processed packet is copied into the application's buffer. All these steps are performed in the context of the user process performing the system call.

There are several things to note about the receiver processing. First, protocol processing for a packet does not occur until the application is waiting for the packet, the packet has arrived, and the application is scheduled to run. As a result, one might expect reduced context switching and increased memory access locality. Second, when the rate of incoming packets exceeds the rate at which the receiving application can consume the packets, the channel receive queue fills, causing the network interface to drop packets. This dropping occurs before significant host resources have been invested in the packet. As a result, the system has good overload behavior: As the offered rate of incoming traffic approaches the capacity of the server, the throughput reaches its maximum and stays at its maximum even if the offered rate increases further[7].

It is important to realize that LRP does *not* increase the latency of UDP packets. The only condition under which the delivery delay of a UDP packet could increase under LRP is when a host CPU is idle between the time of arrival of the packet and the invocation of the receive system call that will deliver the packet to the application. This case can occur on multiprocessor machines, and on a uniprocessor when the only runnable application blocks on an I/O operation (e.g., disk) before invoking the receive system call. To eliminate this possibility, an otherwise idle CPU should always perform protocol processing for any received packets. This is easily accomplished by means of a kernel thread with minimal priority that checks NI channels and performs protocol processing for any queued UDP packets.

3.4 TCP protocol processing

Protocol processing is slightly more complex for a reliable, flow-controlled protocol such as TCP. As in the original architecture, data written by an application is queued in the socket queue. Some data may be transmitted immediately in the context of the user process performing the send system call. The remaining data is transmitted in response to arriving acknowledgments, and possibly in response to timeouts.

The main difference between UDP and TCP processing in the LRP architecture is that receiver processing cannot be performed only in the context of a receive system call, due to the semantics of TCP. Because TCP is flow controlled, transmission of data is paced by the receiver via acknowledgments. Achieving high network utilization and throughput requires timely processing of incoming acknowledgments. If receiver processing were performed only in the context of receive system calls, then at most one TCP congestion window of data could be transmitted between successive receive system calls, resulting in poor performance for many applications.

The solution is to perform receiver processing for TCP sockets asynchronously when required. Packets arriving on TCP connections can thus be processed even when

[6] With soft demux, a host interrupt always occurs upon packet arrival.

[7] With soft demux, the throughput diminishes slightly as the offered load increases, due to the demultiplexing overhead.

the application process is not blocked on a receive system call. Unlike in conventional architectures, this asynchronous protocol processing does not take strict priority over application processing. Instead, the processing is scheduled at the priority of the application process that uses the associated socket, and CPU usage is charged back to that application[8]. Under normal conditions, the application has a sufficiently high priority to ensure timely processing of TCP traffic. If an excessive amount of traffic arrives at the socket, the application's priority will decay as a result of the high CPU usage. Eventually, the protocol processing can no longer keep up with the offered load, causing the channel receiver queue to fill and packets to be dropped by the NI. In addition, protocol processing is disabled for listening sockets that have exceeded their listen backlog limit, thus causing the discard of further SYN packets at the NI channel queue. As shown in Section 4, TCP sockets enjoy similar overload behavior and traffic separation as UDP sockets under LRP.

There are several ways of implementing asynchronous protocol processing (APP). In systems that support (kernel) threads (i.e., virtually all modern operating systems), an extra thread can be associated with application processes that use stream (TCP) sockets. This thread is scheduled at its process's priority and its CPU usage is charged to its process. Since protocol processing always runs to completion, no state needs to be retained between activations. Therefore, it is not necessary to assign a private runtime stack to the APP thread; a single per CPU stack can be used instead. The resulting per-process space overhead of APP is one thread control block. This overhead can be further reduced through the use of continuations [3]. The exact choice of a mechanism for APP greatly depends on the facilities available in a particular UNIX kernel. In our current prototype implementation, a kernel process is dedicated to TCP processing.

3.5 Other protocol processing

Processing for certain network packets cannot be directly attributed to any application process. In the TCP/IP suite, this includes processing of some ARP, RARP, ICMP packets, and IP packet forwarding. In LRP, this processing is charged to daemon processes that act as proxies for a particular protocol. These daemons have an associated NI channel, and packets for such protocols are demultiplexed directly onto the corresponding channel. For example, an IP forwarding daemon is charged for CPU time spent on forwarding IP packets, and its priority controls resources spent on IP forwarding[9]. The IP daemon competes with other processes for CPU time.

4 Performance

In this section, we present experiments designed to evaluate the effectiveness of the LRP network subsystem architecture. We start with a description of the experimental setup and the prototype implementation, and proceed to present the results of various experiments.

4.1 Experimental Setup

All experiments were performed on Sun Microsystems SPARCstation 20 model 61 workstations (60MHz SuperSPARC+, 36KB L1, 1MB L2, SPECint92 98.2). The workstations are equipped with 32MB of memory and run SunOS 4.1.3_U1. A 155 Mbit/s ATM local area network connects the workstations, using FORE Systems SBA-200 network adaptors. These network adaptors include an Intel i960 processor that performs cell fragmentation and reassembly of protocol data units (PDUs). Note that LRP does not depend on a specific network adaptor or ATM networks. SOFT-LRP can be used with any network and NI.

The LRP architecture was implemented as follows. We modified the TCP/UDP/IP network subsystem that comes with the 4.4 BSD-Lite distribution [24] to optionally implement LRP. The resulting code was then downloaded into the SunOS kernel as a loadable kernel module and attached to the socket layer as a new protocol family (PF_LRP). A custom device driver was developed for the FORE network adaptor. The 4.4 BSD-Lite networking subsystem was used because of its performance and availability in source form. (We did not have access to SunOS source code.) The 4.4 BSD networking code was slightly modified to work with SunOS mbufs. At the time of this writing, the prototype implementation uses a kernel process to perform asynchronous protocol processing for TCP.

Since we were unable to obtain source code for the SBA-200 firmware, we could not integrate our own demultiplexing function in this network adaptor. However, we know enough about the interface's architecture to be confident that the function could be easily integrated, given the source code. To evaluate packet demultiplexing in the network adaptor (NI demux), we used instead the SBA-200 firmware developed by Cornell University's U-Net project [1]. This firmware performs demultiplexing based on the ATM virtual circuit identifier (VCI). A signaling scheme was used that ensures that a separate

[8] In UNIX, more than one process can wait to read from a socket. In this case, the process with the highest priority performs the protocol processing.

[9] QoS attributes or IPv6 flows could be used in an LRP based IP gateway to provide more fine-grained resource control.

ATM VCI is assigned for traffic terminating or originating at each socket. The resulting implementation of NI-LRP is fully functional.

4.2 Experimental Results

All experiments were performed on a private ATM network between the SPARCstations. The machines were running in multiuser mode, but were not shared by other users.

The first experiment is a simple test to measure UDP latency and throughput, and TCP throughput. Its purpose is to demonstrate that the LRP architecture is competitive with traditional network subsystem implementations in term of these basic performance criteria. Moreover, we include the results for an unmodified SunOS kernel with the Fore ATM device driver for comparison. Latency was measured by ping-ponging a 1-byte message between two workstations 10,000 times, measuring the elapsed time and dividing to obtain round-trip latency. UDP throughput was measured using a simple sliding-window protocol (UDP checksumming was disabled.) TCP throughput was measured by transferring 24 Mbytes of data, with the socket send and receive buffers set to 32 KByte. Table 1 shows the results.

The numbers clearly demonstrate that LRP's basic performance is comparable with the unmodified BSD system from which it was derived. That is, LRP's improved overload behavior does not come at the cost of low-load performance. Furthermore, both BSD and LRP with our device driver perform significantly better than SunOS with the Fore ATM driver in terms of latency and UDP bandwidth. This is due to performance problems with the Fore driver, as discussed in detail in [1].

SunOS exhibits a performance anomaly that causes its base round-trip latency—as measured on otherwise idle machines—to drop by almost 300 μsecs, when a compute-bound background process is running on both the client and the server machine. We have observed this effect in many different tests with SunOS 4.1.3_U1 on the SPARCstation 20. The results appear to be consistent with our theory that the cost of dispatching a hardware/software interrupt and/or the receiver process in SunOS depends on whether the machine is executing the idle loop or a user process at the time a message arrives from the network. Without access to source code, we were unable to pinpoint the source of this anomaly.

Since our modified systems (4.4BSD, NI-LRP, SOFT-LRP) are all based on SunOS, they were equally affected by this anomaly. Apart from affecting the base round-trip latency, the anomaly can perturb the results of tests with varying rates and concurrency of network traffic, since these factors influence the likelihood that an incoming packet interrupts a user process. To eliminate this variable, some of the experiments described below were run with low-priority, compute-bound processes running in the background, to ensure that incoming packets never interrupt the idle loop.

The next experiment was designed to test the behavior of the LRP architecture under overload. In this test, a client process sends short (14 byte) UDP packets to a server process on another machine at a fixed rate. The server process receives the packets and discards them immediately. Figure 3 plots the rate at which packets are received and consumed by the server process as a function of the rate at which the client transmits packets.

With the conventional 4.4 BSD network subsystem, the throughput increases with the offered load up to a maximum of 7400 pkts/sec. As the offered load increases further, the throughput of the system decreases, until the system approaches livelock at approximately 20,000 pkts/sec. With NI-LRP, on the other hand, throughput increases up to the maximum of 11,000 pkts/sec and remains at that rate as the offered load increases further. This confirms the effectiveness of NI-LRP's load shedding in the network interface, before any host resources have been invested in handling excess traffic. Instrumentation shows that the slight drop in NI-LRP's delivery rate beyond 19,000 pkts/sec is actually due to a reduction in the delivery rate of our ATM network, most likely caused by congestion-related phenomena in either the switch or the network interfaces.

SOFT-LRP refers to the case where demultiplexing is performed in the host's interrupt handler (soft demux). The throughput peaks at 9760 pkts/sec, but diminishes slightly with increasing rate due to the overhead of demultiplexing packets in the host's interrupt handler. This confirms that, while NI-LRP eliminates the possibility of livelock, SOFT-LRP merely postpones its arrival. However, on our experimental ATM network hardware/software platform, we have been unable to generate high enough packet rates to cause livelock in the SOFT-LRP kernel, even when using an in-kernel packet source on the sender.

For comparison, we have also measured the overload behavior of a kernel with early demultiplexing only (Early-Demux). The system performs demultiplexing in the interrupt handler (as in SOFT-LRP), drops packets whose destination socket's receiver queue is full, and otherwise schedules a software interrupt to process the packet. Due to the early demultiplexing, UDP's PCB lookup was bypassed, as in the LRP kernels. The system displays improved stability under overload compared with BSD, a result of early packet discard. The rate of decline under overload is comparable to that of SOFT-LRP, which is consistent with their use of the same demultiplexing mechanism. However, the throughput of the Early-Demux kernel is only between 40–65% of SOFT-

System	round-trip latency (μsecs)	UDP throughput (Mbps)	TCP throughput (Mbps)
SunOS, Fore driver	1006	64	63
4.4 BSD	855	82	69
LRP (NI Demux)	840	92	67
LRP (Soft Demux)	864	86	66

Table 1: Throughput and Latency

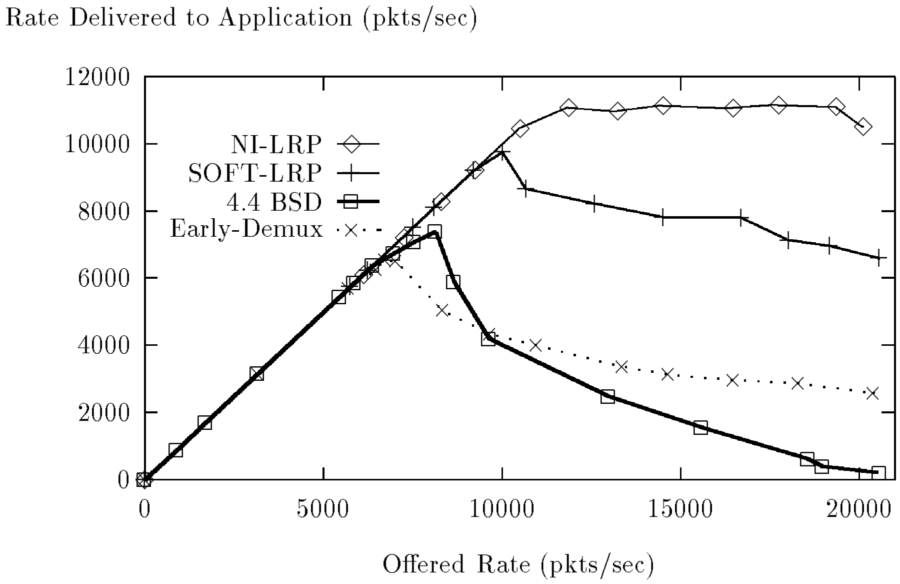

Figure 3: Throughput versus offered load

LRP's throughput across the overload region.

Both variants of LRP display significantly better throughput than both the conventional 4.4 BSD system, and the Early-Demux kernel. The maximal delivered rate of NI-LRP is 51% and that of SOFT-LRP is 32% higher than BSD's maximal rate (11163 vs. 9760 vs. 7380 pkts/sec). Note that the throughput with SOFT-LRP at the maximal offered rate is within 12% of BSD's maximal throughput.

In order to understand the reasons for LRP's throughput gains, we instrumented the kernels to capture additional information. It was determined that the Maximum Loss Free Receive Rate (MLFRR) of SOFT-LRP exceeded that of 4.4BSD by 44% (9210 vs. 6380 pkts/sec). 4.4BSD and LRP drop packets at the socket queue or NI channel queue, respectively, at offered rates beyond their MLFRR. 4.4BSD additionally starts to drop packets at the IP queue at offered rates in excess of 15,000 pkts/sec. No packets were dropped due to lack of mbufs.

Obviously, early packet discard does not play a role in any performance differences at the MLFRR. With the exception of demultiplexing code (early demux in LRP versus PCB lookup in BSD) and differences in the device driver code, all four kernels execute the same 4.4BSD networking code. Moreover, the device driver and demultiplexing code used in Early-Demux and SOFT-LRP are identical, eliminating these factors as potential contributors to LRP's throughput gains. This suggests that the performance gains in LRP must be due in large part to factors such as reduced context switching, software interrupt dispatch, and improved memory access locality.

Our next experiment measures the latency that a client experiences when contacting a server process on a machine with high network load. The client, running on machine A, ping-pongs a short UDP message with a server process (ping-pong server) running on machine B. At the same time, machine C transmits UDP packets at a fixed rate to a separate server process (blast server) on machine B, which discards the packets upon arrival. Figure 4 plots the round-trip latency experienced by the client as a function of the rate at which packets are transmitted from machine C to the blast server (background load). To avoid the abovementioned performance anomaly in SunOS, the machines involved in the ping-pong exchange were each running a low-priority (nice +20) background process executing an infinite loop.

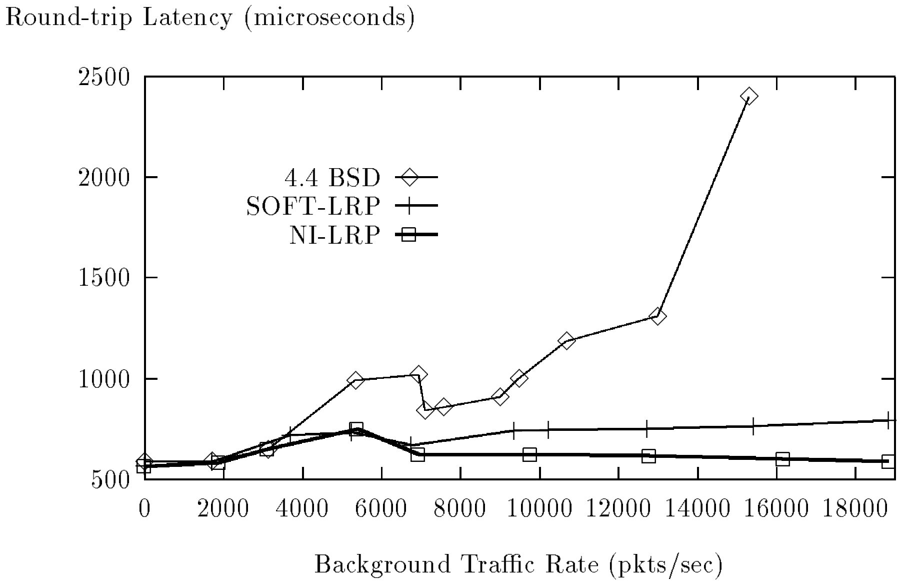

Figure 4: Latency with concurrent load

In all three systems, the measured latency varies with the background traffic rate. This variation is caused by the arrival of background traffic packets during the software processing of a ping-pong packet on the receiver. Arrivals of background traffic delay the processing of the request and/or the transmission of the response message, thus causing an increase in the round-trip delay. The magnitude of this delay is determined by two factors: The rate of arrivals, and the length of the interruptions caused by each arrival. This length of interruptions consists of the fixed interrupt processing time (hardware interrupt in LRP, hardware plus software interrupt in BSD), plus the optional time for scheduling of the blast server, and delivery of the message. This last component only occurs when the blast server's priority exceeds that of the ping-pong server, i.e., it is a function of SunOS's scheduling policy.

Instrumentation and modeling confirmed that the two main factors shaping the graphs are (1) the length of the fixed interrupt processing and (2) the scheduling-dependent overhead of delivering messages to the blast receiver. The fixed interrupt overhead causes a non-linear increase in the latency as the background traffic rises. Due to the large overhead (hardware plus software interrupt, including protocol processing, approximately 60μsecs), the effect is most pronounced in 4.4BSD. SOFT-LRP's reduced interrupt overhead (hardware interrupt, including demux, approx. 25μsecs), results in only a gradual increase. With NI-LRP (hardware interrupt with minimal processing), this effect is barely noticeable.

The second factor leads to an additional increase in latency at background traffic rates up to 7000 pkts/sec. The UNIX scheduler assigns priorities based on a process's recent CPU usage. As a result, it tends to favor a process that had been waiting for the arrival of a network packet, over the process that was interrupted by the packet's arrival. At low rates, the blast receiver is always blocked when a blast packet arrives. If the arrival interrupts the ping-pong server, the scheduler will almost always give the CPU to the blast receiver, causing a substantial delay of the ping-pong message. At rates around 6000 pkts/sec, the blast receiver is nearing saturation, thus turning compute-bound. As a result, its priority decreases, and the scheduler now preferentially returns the CPU to the interrupted ping-pong server immediately, eliminating this effect at high rates.

The additional delay caused by context switches to the blast server is much stronger in BSD as in the two LRP systems (1020 vs. 750μsecs peak). This is a caused by the mis-accounting of network processing in BSD. In that system, protocol processing of blast messages that arrive during the processing of a ping-pong message is charged to the ping-pong server process. This depletes the priority of the ping-pong server, and increases the likelihood that the scheduler decides to assign the CPU to the blast server upon arrival of a message. Note that in a system that supports fixed-priority scheduling (e.g., Solaris), the influence of scheduling could be eliminated by assigning the ping-pong server statically highest priority. The result is nevertheless interesting in that it displays the effect of CPU mis-accounting on latency in a system with a dynamic scheduling policy.

With BSD, packet dropping at the IP queue makes latency measurements impossible at rates beyond 15,000

RPC	System	Worker elapsed time (secs)	Server (RPCs/sec)
Fast	4.4BSD	49.7	3120
	SO-LRP	38.7	3133
	NI-LRP	34.6	3410
Medium	4.4BSD	47.1	2712
	SO-LRP	37.9	2759
	NI-LRP	34.1	2783
Slow	4.4BSD	43.9	2045
	SO-LRP	38.5	2134
	NI-LRP	35.7	2208

Table 2: Synthetic RPC Server Workload

pkts/sec. In the LRP systems, no dropped latency packets were observed, which is due to LRP's traffic separation.

Our next experiment attempts to more closely model a mix of workloads typical for network servers. Three processes run on a server machine. The first server process, called the worker, performs a memory-bound computation in response to an RPC call from a client. This computation requires approximately 11.5 seconds of CPU time and has a memory working set that covers a significant fraction (35%) of the second level cache. The remaining two server processes perform short computations in response to RPC requests. A client on the other machine sends an RPC request to the worker process. While the worker RPC is outstanding, the client sends RPC requests to the remaining server processes in such a way that (1) each server has a number of outstanding RPC requests at all times, and (2) the requests are distributed near uniformly in time. (1) ensures that the RPC server processes never block on receiving from the network[10]. The purpose of (2) is to make sure there is no correlation between the scheduling of the server processes, and the times at which requests are issued by the client. Note that in this test, the clients generate requests at the maximal throughput rate of the server. That is, the server is not operating under conditions of overload. The RPC facility we used is based on UDP datagrams.

Table 2 shows the results of this test. The total elapsed time for completion of the RPC to the worker process is shown in the third column. The rightmost column shows the rate at which the servers process RPCs, concurrently with each other and the worker process. "Fast", "Medium" and "Slow" correspond to tests with different amounts of per-request computations performed in the two RPC server processes. In each of the tests, the server's throughput (considering rate of RPCs completed and worker completion time) is lowest with BSD, higher with SOFT-LRP (SO-LRP), and highest with NI-LRP. In the "Medium" case, where the RPC rates are within 3% for each of the systems, the worker completion time with SOFT-LRP is 20% lower, and with NI-LRP 28% lower than with BSD. In the "Fast" case, NI-LRP achieves an almost 10% higher RPC rate and a 30% lower worker completion time than BSD. This confirms that LRP-based servers have increased throughput under high load. Note that packet discard is not a factor in this test, since the system is not operating under overload. Therefore, reduced context switching and improved locality must be responsible for the higher throughput with LRP.

Furthermore, the LRP systems maintain a fair allocation of CPU resources under high load. With SOFT-LRP and NI-LRP, the worker process's CPU share (CPU time / elapsed completion time) ranges from 29% to 33%, which is very close to the ideal $1/3$ of the available CPU, compared to 23%–26% with BSD. This demonstrates the effect of mis-accounting in BSD, which tends to favor processes that perform intensive network communication over those that do not. Observe that this effect is distinct from, and independent of, the UNIX scheduler's tendency to favor I/O-bound processes.

Finally, we conducted a set of experiments with a real server application. We configured a machine running a SOFT-LRP kernel as a WWW server, using the NCSA httpd server software, revision 1.5.1. A set of informal experiments show that the server is dramatically more stable than a BSD based server under overload. To test this, Mosaic clients were contacting the server, while a test program running on a third machine was sending packets to a separate port on the server machine at a high rate (10,000 packets/sec). An HTTP server based on 4.4 BSD freezes completely under these conditions, i.e., it no longer responds to any HTTP requests, and the server console appears dead. With LRP and soft demux, the server responds to HTTP requests and the console is responsive, although some increase in response time is noticeable.

The results of a quantitative experiment are shown in Figure 5. In this test, eight HTTP clients on a single machine continually request HTTP transfers from the server. The requested document is approximately 1300 bytes long. The eight clients saturate the HTTP server. A second client machine sends fake TCP connection establishment requests (SYN packets) to a dummy server running on the server machine that also runs the HTTP server. No connections are ever established as a result of these requests; TCP on the server side discards most of them once the dummy server's listen backlog is exceeded. To avoid known performance problems with BSD's PCB lookup function in HTTP servers [16], the TCP TIME_WAIT period was set to 500ms, instead of the default 30 seconds. The test were run for long periods

[10] This is to ensure that the UNIX scheduler does not consider these server processes I/O-bound, which would tend to give them higher scheduling priority.

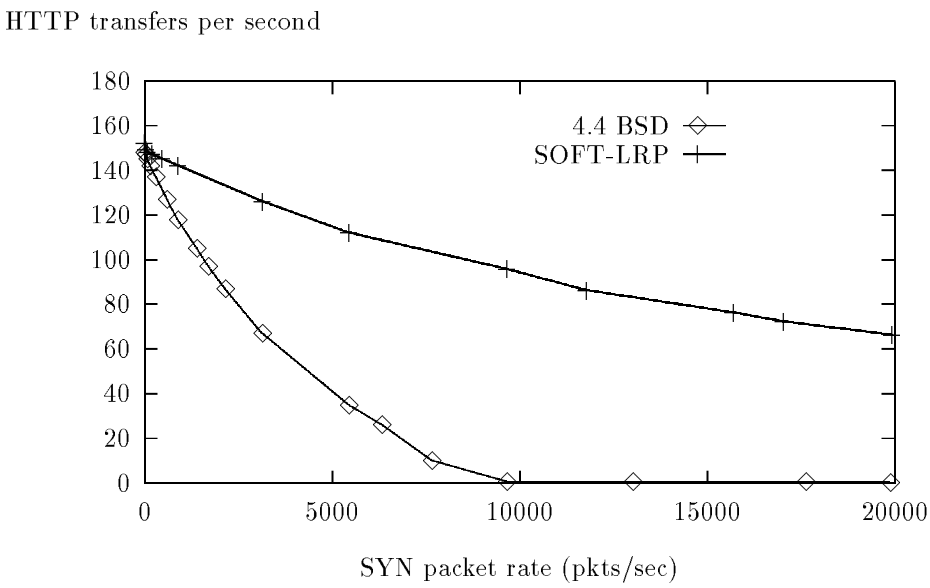

Figure 5: HTTP Server Throughput

of time to ensure steady-state behavior. Furthermore, the LRP system performed a redundant PCB lookup to eliminate any bias due to the greater efficiency of the early demultiplexing in LRP. Note that the results of this test were not affected by the TCP bug described in RFC 1948.

The graphs show the number of HTTP transfers completed by all clients, as a function of the rate of SYN packets to the dummy server, for 4.4 BSD and SOFT-LRP. The throughput of the 4.4 BSD-based HTTP server sharply drops as the rate of background requests increases, entering livelock at close to 10,000 SYN pkts/sec. The reason is that BSD's processing of SYN packets in software interrupt context starves the httpd server processes for CPU resources. Additionally, at rates above 6400 SYN pkts/sec, packets are dropped at BSD's shared IP queue. This leads to the loss of both TCP connection requests from real HTTP client and traffic on established TCP connections. Lost TCP connection requests cause TCP on the client side to back off exponentially. Lost traffic on established connections cause TCP to close its congestion window. However, the dominant factor in BSD's throughput decline appears to be the starvation of server processes.

With LRP, the throughput decreases relatively slowly. At a rate of 20,000 background requests per second, the LRP server still operates at almost 50% of its maximal throughput[11]. With LRP, traffic on each established TCP connection, HTTP connection requests, and dummy requests are all demultiplexed onto separate NI channels and do not interfere. As a result, traffic to the dummy server does not cause the loss of HTTP traffic at all. Furthermore, most dummy SYN packets are discarded early at the NI channel queue. The predominant cause of the decline in the SOFT-LRP based server's throughput is the overhead of software demultiplexing.

It should be noted that, independent of the use of LRP, an Internet server must limit the number of active connections to maintain stability. A related issue is how well LRP works with a large number of established connections, as has been observed on busy Internet servers [15]. SOFT-LRP uses one extra mbuf compared to 4.4BSD for each established TCP connection, so SOFT-LRP should scale well to large numbers of active connections. NI-LRP, on the other hand, dedicates resources on the network interface for each endpoint and is not likely to scale to thousands of allocated NI channels. However, most of the established connections on a busy web server are in the TIME_WAIT state. This can be exploited by deallocating an NI channel as soon as the associated TCP connection enters the TIME_WAIT state. Any subsequently arriving packets on this connection are queued at a special NI channel which is checked by TCP's *slowtimo* code. Since such traffic is rare, this does not affect NI-LRP's behavior in the normal case.

5 Related Work

Experiences with DEC's 1994 California Election HTTP server reveal many of the problems of a conventional network subsystem architecture when used as a busy HTTP

[11] Note that a (slow) T1 link is capable of carrying almost 5000 SYN packets per second. With the emerging faster network links, routers, and a sufficiently large user community, a server could easily be subjected to such rates.

server [15]. Mogul [16] suggests that novel OS support may be required to satisfy the needs of busy servers.

Mogul and Ramakrishnan [17] devise and evaluate a set of techniques for improving the overload behavior of an interrupt-driven network architecture. These techniques avoid receiver livelock by temporarily disabling hardware interrupts and using polling under conditions of overload. Disabling interrupts limits the interrupt rate and causes early packet discard by the network interface. Polling is used to ensure progress by fairly allocating resources among receive and transmit processing, and multiple interfaces.

The overload stability of their system appears to be comparable to that of NI-LRP, and it has an advantage over SOFT-LRP in that it eliminates—rather than postpones—livelock. On the other hand, their system does not achieve traffic separation, and therefore drops packets irrespective of their destination during periods of overload. Their system does not attempt to charge resources spent in network processing to the receiving application, and it does not attempt to reduce context switching by processing packets lazily. A direct quantitative comparison between LRP and their system is difficult, because of differing hardware/software environments and benchmarks.

Many researchers have noted the importance of *early demultiplexing* to high-performance networking. Demultiplexing immediately at the network interface point is necessary for maintaining network quality of service (QoS) [22], it enables user-level implementations of network subsystems [2, 7, 11, 21, 23], it facilitates copy-avoidance by allowing smart placement of data in main memory [1, 2, 5, 6], and it allows proper resource accounting in the network subsystem [14, 19]. This paper argues that early demultiplexing also facilitates fairness and stability of network subsystems under conditions of overload. LRP uses early demultiplexing as a key component of its architecture.

Packet filters [12, 18, 25] are mechanisms that implement early demultiplexing without sacrificing layering and modularity in the network subsystem. In the most recent incarnations of packet filters, dynamic code generation is used to eliminate the overhead of the earlier interpreted versions [8].

Architecturally, the design of LRP is related to *user-level network subsystems*. Unlike LRP, the main goal of these prior works is to achieve low communication latency and high bandwidth by removing protection boundaries from the critical send/receive path, and/or by enabling application-specific customization of protocol services. To the best of our knowledge, the behavior of user-level network subsystems under overload has not been studied.

U-Net [1] and Application Device Channels (ADC) [4, 5] share with NI-LRP the approach of using the network interface to demultiplex incoming packets and placing them on queues associated with communication endpoints. With U-Net and ADCs, the endpoint queues are mapped into the address space of application processes. More conventional user-level networking subsystems [7, 11, 23] share with SOFT-LRP the early demultiplexing of incoming packets by the OS kernel (software). Demultiplexed packets are then handed to the appropriate application process using an upcall. In all user-level network subsystems, protocol processing is performed by user-level threads. Therefore, network processing resources are charged to the application process and scheduled at application priority.

Based on the combination of early demultiplexing and protocol processing by user-level threads, user-level network subsystems can be in principle expected to display improved overload stability. Since user-level threads are normally prioritized to compete with other user and kernel threads, protocol processing cannot starve other applications as in BSD. A user-level network subsystem's resilience to livelock depends then on the overhead of packet demultiplexing on the host. When demultiplexing and packet discard are performed by the NI as in [1, 5], the system should be free of livelock. When these tasks are performed by the OS kernel as in [7, 11, 23], the rate at which the system experiences livelock depends on the overhead of packet demultiplexing (as in SOFT-LRP). Since the systems described in the literature use interpreted packet filters for demultiplexing, the overhead is likely to be high, and livelock protection poor. User-level network subsystems share with LRP the improved fairness in allocating CPU resources, because protocol processing occurs in the context of the receiver process.

User-level network subsystems allow applications to use application-specific protocols on top of the raw network interface. The performance (i.e., latency, throughput) of such protocols under overload depends strongly on their implementation's processing model. LRP's technique of delaying packet processing until the application requests the associated data can be applied to such protocols. The following discussion is restricted to user-level implementations of TCP/IP.

The user-level implementations of TCP/IP described in the literature share with the original BSD architecture the eager processing model. That is, a dedicated user thread (which plays the role of the BSD software interrupt) is scheduled as soon as a packet arrives, regardless of whether or not the application is waiting for the packet. As in BSD, this eager processing can lead to additional context switching, when compared to LRP.

The single shared IP queue in BSD is replaced with a per-application IP queue that is shared only among multiple sockets in a single application. As a result, the system ensures traffic separation among traffic destined for different applications, but not necessarily among traffic destined for different sockets within a single application. Depending on the thread scheduling policy and the relative priority of the dedicated protocol processing thread(s) and application thread(s), it is possible that incoming traffic can cause an application process to enter a livelock state, where the network library thread consumes all CPU resources allocated to the application, with no CPU time left for the application threads. Traffic separation and livelock protection within an application process are important, for instance, in single-process HTTP servers.

Finally, UNIX based user-level TCP/IP implementations revert to conventional network processing under certain conditions (e.g., whenever a socket is shared among multiple processes.) In this case, the system's overload behavior is similar to than that of a standard BSD system.

In summary, we expect that user-level network implementations—while designed with different goals in mind—share some but not all of LRP's benefits with respect to overload. This paper identifies and evaluates techniques for stability, fairness, and performance under overload, independent of the placement of the network subsystem (application process, network server, or kernel). We fully expect that LRP's design principles can be applied to improve the overload behavior of kernelized, server-based, and user-level implementations of network subsystems.

Livelock and other negative effects of BSD's interrupt-driven network processing model can be viewed as an instance of a priority inversion problem. The real-time OS community has developed techniques for avoiding priority inversion in communication systems in order to provide quality of service guarantees for real-time data streams [9, 10]. RT-Mach's network subsystem [10], which is based on the Mach user-level network implementation [11], performs early demultiplexing, and then hands incoming packets for processing to a real-time thread with a priority and resource reservation appropriate for the packet's stream. Like LRP, the system employs early demultiplexing, schedules protocol processing at a priority appropriate to the data's receiver, and charges resources to the receiver. Unlike LRP, it does not attempt to delay protocol processing until the data is requested by the application. Moreover, the overhead of the Mach packet filter is likely to make RT-Mach vulnerable to overload. We fully expect that LRP, when combined with real-time thread scheduling, is applicable to real-time networking, without requiring user-level protocols.

6 Conclusion

This paper introduces a novel network subsystem architecture suitable for network server systems. Performance evaluations indicate that under conditions of high load, the architecture offers increased throughput, stable overload behavior, and reduced interference among traffic destined for separate communication endpoints.

More specifically, LRP's lazy, delayed processing of received network packets reduces context switching and can result in increased server throughput under high load. LRP's combination of early packet demultiplexing, early packet discard, and the processing of incoming network packets at the receiver's priority provide improved traffic separation and stability under overload.

A public release of our SunOS-based prototype is planned for the Fall of 1996. The source code, along with additional technical information can be found at "http://www.cs.rice.edu/CS/Systems/LRP/".

Acknowledgments

We are indebted to our OSDI shepherd Jeff Mogul and the anonymous reviewers, whose comments have helped to improve this paper. Also, thanks to Thorsten von Eicken and the U-Net group at Cornell for making the U-Net NI firmware available to us.

References

[1] A. Bas, V. Buch, W. Vogels, and T. von Eicken. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 40–53, 1995.

[2] G. Buzzard, D. Jacobson, S. Marovich, and J. Wilkes. Hamlyn: A high-performance network interface with sender-based memory management. In *Proceedings of the Hot Interconnects III Symposium*, Palo Alto, CA, Aug. 1995.

[3] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 122–36. Association for Computing Machinery SIGOPS, October 1991.

[4] P. Druschel. Operating systems support for high-speed networking. Technical Report TR 94-24, Department of Computer Science, University of Arizona, Oct. 1994.

[5] P. Druschel, B. S. Davie, and L. L. Peterson. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings of the SIGCOMM '94 Conference*, pages 2–13, London, UK, Aug. 1994.

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

[7] 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. In *Proceedings of the SIGCOMM '94 Conference*, pages 14–23, London, UK, Aug. 1994.

[8] D. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of the SIGCOMM '96 Conference*, pages 53–59, Palo Alto, CA, Aug. 1996.

[9] K. Jeffay. On Latency Management in Time-Shared Operating Systems. In *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 86–90, Seattle, WA, May 1994.

[10] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar. Predictable communication protocol processing in real-time Mach. *In the proceedings of IEEE Real-time Technology and Applications Symposium*, June 1996.

[11] C. Maeda and B. N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244–255, 1993.

[12] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX '93 Winter Conference*, pages 259–269, Jan. 1993.

[13] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, 1996.

[14] J. C. Mogul. Personal communication, Nov. 1992.

[15] J. C. Mogul. Network behavior of a busy web server and its clients. Technical Report WRL 95/5, DEC Western Research Laboratory, Palo Alto, CA, 1995.

[16] J. C. Mogul. Operating system support for busy internet servers. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[17] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *Proc. of the 1996 Usenix Technical Conference*, pages 99–111, 1996.

[18] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, Nov. 1987.

[19] A. B. Montz et al. Scout: A communications-oriented operating system. Technical Report TR 94-20, Department of Computer Science, University of Arizona, June 1994.

[20] K. K. Ramakrishnan. Scheduling issues for interfacing to high speed networks. In *Proc. Globecom'92 IEEE Global Telecommunications Conference*, pages 622–626, Orlando, FL, Dec. 1992.

[21] J. M. Smith and C. B. S. Traw. Giving applications access to Gb/s networking. *IEEE Network*, 7(4):44–52, July 1993.

[22] D. L. Tennenhouse. Layered multiplexing considered harmful. In H. Rudin and R. Williamson, editors, *Protocols for High-Speed Networks*, pages 143–148, Amsterdam, 1989. North-Holland.

[23] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. In *Proceedings of the SIGCOMM '93 Symposium*, pages 64–73, Sept. 1993.

[24] G. Wright and W. Stevens. *TCP/IP Illustrated Volume 2*. Addison-Wesley, Reading, MA, 1995.

[25] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Winter 1994 Usenix Conference*, pages 153–165, Jan. 1994.