Signaled Receiver Processing

José Brustoloni, Eran Gabber, Abraham Silberschatz and Amit Singh
Information Sciences Research Center
Lucent Technologies — Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974, USA
{jcb, eran, avi, amitsingh}@research.bell-labs.com

Abstract

Protocol processing of received packets in BSD Unix is interrupt-driven and may cause scheduling anomalies that are unacceptable in systems that provide quality of service (QoS) guarantees. We propose an alternative mechanism, Signaled Receiver Processing (SRP), that generates a signal to the receiving process when a packet arrives. The default action of this signal is to perform protocol processing asynchronously. However, a receiving process may catch, block, or ignore the signal and defer protocol processing until a subsequent receive call. In any case, protocol processing occurs in the context of the receiving process and is correctly charged. Therefore, SRP allows the system to enforce and honor QoS guarantees. SRP offers several advantages over Lazy Receiver Processing (LRP), a previous solution to BSD's scheduling anomalies: SRP is easily portable to systems that support neither kernel threads nor Resource Containers (e.g., FreeBSD); gives applications control over the scheduling of protocol processing; uses a demultiplexing strategy that is appropriate for both hosts and gateways; and easily enables real-time or proportional-share scheduling.

1 Introduction

Many Internet protocols, including TCP and IP, first appeared on the BSD Unix operating system [19]. BSD implementations were widely circulated, evaluated, and debugged both in industry and academia, thus influencing many other implementations [23]. Such influence promoted interoperability, but also disseminated artifacts that are neither desirable nor part of the protocol standards.

One of these artifacts is that, in BSD-inspired implementations, protocol processing of received packets typically occurs in the context of a software interrupt, before the system demultiplexes packets to socket receive queues. In many implementations, this processing is charged to whatever application was interrupted, even if the latter is unrelated to the packets. In other implementations, this processing is not charged at all. In either case, protocol processing may thus cause scheduling anomalies; the system cannot enforce CPU allocations and therefore cannot provide quality of service (QoS) guarantees to applications. Furthermore, BSD's scheme always prioritizes protocol processing of a received packet over application processing, even if the respective socket receive queue is full and therefore the system will have to drop the packet. In fact, under high receive loads, the system may waste all CPU time processing packets that will have to be dropped (a phenomenon known as *receive livelock* [20]): The system gives applications no CPU time and therefore applications cannot empty the receive queues. Receive livelock can be exploited in denial of service attacks.

BSD's scheduling anomalies can be avoided by an alternative scheme, Lazy Receiver Processing (LRP) [14]. LRP's techniques are transparent to applications and do not violate protocol standards. LRP combines several mechanisms to guarantee that resources used in a packet's protocol processing are charged to the application that will receive that packet. LRP uses *early demultiplexing*, i.e., demultiplexes packets to the respective receiving applications *before* protocol processing. In the case of UDP packets, LRP always processes protocols *synchronously*, i.e., when the receiving application issues a receive call. On the other hand, LRP always processes TCP packets *asynchronously*, i.e., when packets arrive. For correct resource accounting, LRP may associate with each process an extra kernel thread that asynchronously processes incoming TCP packets for the respective process, and has its resource utilization charged to the process [14]. Alternatively, LRP may process all incoming TCP packets in a single process, and use a Resource Containers facility to charge resource usage to the Containers of the respective receiving applications [1].[1]

Unfortunately, LRP may also present significant difficulties. First, many operating systems support neither kernel threads (e.g., FreeBSD) nor Resource Containers (e.g., most existing systems). Therefore, it can be difficult to port LRP to such systems. Second, LRP's UDP processing is always synchronous, whereas LRP's TCP processing is always asynchronous and shares resources equally with the receiving application. However, for some applications, different protocol scheduling or resource apportionment may be preferable. Third, LRP and Resource Containers were designed for *hosts* (as acknowledged by the reference to "server systems" in the titles of the respective papers [14, 1]). However, scheduling and resource management in *gateways* is becoming as important as in hosts. Gateways no longer simply forward packets; they increasingly also need to run applications such as routing protocols, network management [10], firewalling, Network Address Translation (NAT) [15], load balancing [22], reservation protocols [4], or billing [12]. Extensible routers [21] and active networks [9] suggest a number of other ways in which it may be advantageous to run application-specific code on gateways.

LRP's use in these modern gateways faces two problems. First, LRP's early demultiplexing does not provide the required flexibility. In gateways, each packet may need to be processed not by a single receiving application, but by a variable series of applications, each of which may modify the packet's header and affect what other applications need to process the packet. Second, LRP and Resource Containers were described and evaluated in detail only in conjunction with time-sharing scheduling. However, this type of scheduling may be inadequate for gateways. For example, it would be improper to penalize IP forwarding according to its CPU usage, as a typical time-sharing scheduler would. On the other hand, giving IP forwarding a "real-time" priority may also be inadequate, because it could lead to the starvation of time-sharing or other lower-priority applications (e.g., in FreeBSD 3.0, real-time priorities are fixed and are always higher than time-sharing priorities).

This paper contributes a new scheme, Signaled Receiver Processing (SRP), that overcomes both BSD's and LRP's mentioned shortcomings. When an incoming packet is demultiplexed to a given process, SRP *signals* that process. The *default* action on such signal is to perform protocol processing asynchronously. However, a process may choose to synchronize protocol processing by *catching*, *blocking*, or *ignoring* SRP's signals. In the latter cases, protocol processing is deferred until a later receive call. In all cases, protocol processing occurs in the context of and is charged to the receiving process.

SRP has several advantages over LRP. First, SRP uses signals and not kernel threads nor Resource Containers. Therefore, SRP can be easily ported to most existing systems, including FreeBSD. Second, SRP gives applications considerable control over the scheduling and resource apportionment of protocol processing. For example, an application may catch SRP's signals to control the time spent doing protocol processing; block SRP's signals to avoid interruptions while processing some urgent event; or ignore SRP's signals to make TCP processing synchronous. Synchronous TCP processing can improve memory locality. Such control is not possible in LRP because LRP was designed to be transparent to applications. Third, SRP supports modern gateways. SRP uses a multi-stage demultiplexing function that, unlike LRP's simple early demultiplexing, allows packets to be examined and modified by a multiple and possibly variable series of applications. Moreover, SRP supports proportional-share scheduling. In a gateway, proportional-share scheduling can guarantee to each application (e.g., IP forwarding, load balancing, or billing) a minimum share of the CPU, without penalties for usage and without starvation of other applications. We implemented SRP as part of Eclipse/BSD, a new operating system that is derived from FreeBSD and that provides QoS guarantees via proportional-share scheduling of each system resource, including CPU, disk, and network output link bandwidth [7].

The rest of this paper is organized as follows. Sections 2 and 3 describe in greater detail how BSD and LRP process received packets, respectively. Section 4 reports the difficulties we encountered when

[1] In many operating systems, including FreeBSD, the notions of resource principal and protection domain coincide in the process abstraction. Resource Containers are a proposal to separate these notions, making resource management more flexible. For example, a given client's resource consumption may be represented by a single Resource Container. In this case, resources used by different servers on behalf of the client may be charged to that client's Resource Container.

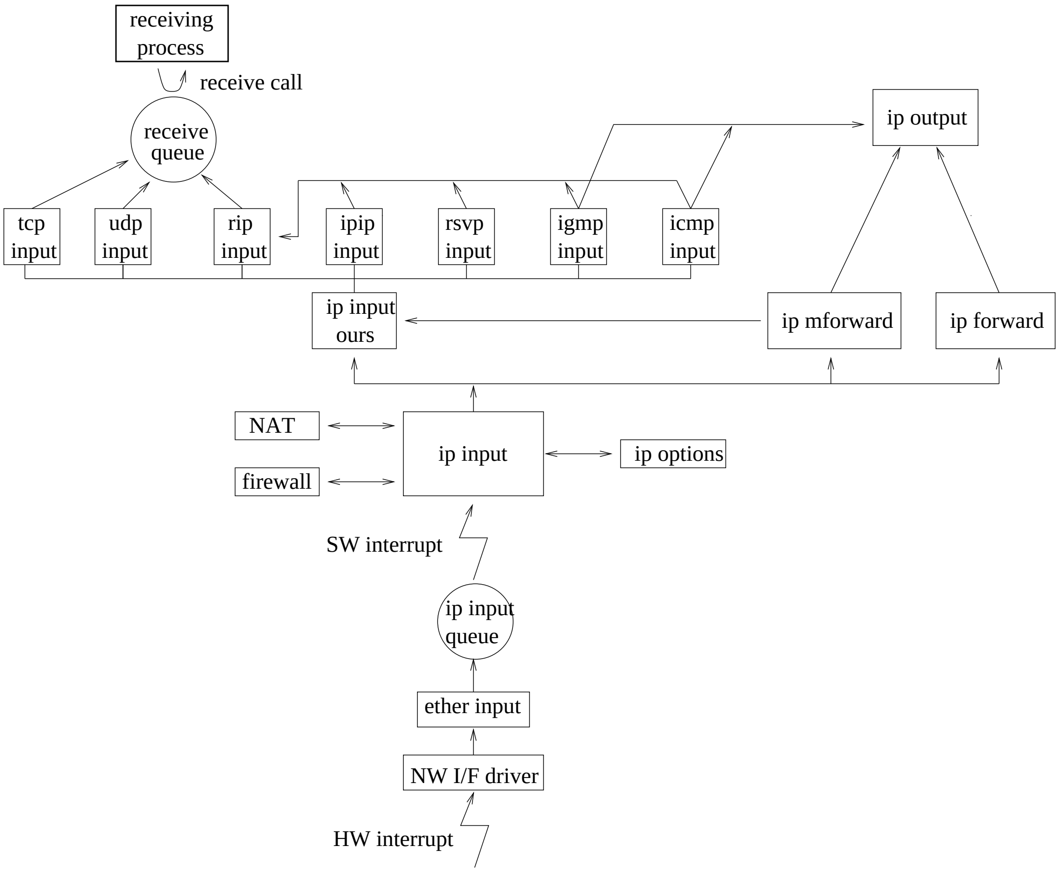

Figure 1: In FreeBSD, protocol processing of received packets occurs in the context of a hardware or software interrupt at priority higher than that of any application, and is charged to whatever process was interrupted.

porting LRP to FreeBSD for the implementation of Eclipse/BSD. Those difficulties led to the design of SRP, described in Section 5. Experiments in Section 6 demonstrate that, like LRP, SRP prevents receive livelock. However, the experiments also show that SRP supports Eclipse/BSD's QoS guarantees, that such support is desirable for gateway functionality, such as IP forwarding, and that SRP, unlike LRP, allows any application to control the scheduling of protocol processing, e.g. to achieve better memory locality. Section 7 discusses related work, and Section 8 concludes.

2 BSD receiver processing

This section discusses how FreeBSD processes IP packets received from an Ethernet. This discussion is representative also of protocol processing in other derivatives of BSD and for other protocol families and networks.

As shown in Figure 1, packet arrival causes a hardware interrupt that transfers CPU control to a network interface driver. The driver retrieves the packet from the network interface hardware, prepares the hardware for receiving a future packet, and passes the received packet to the `ether_input` routine. `ether_input` places the packet in IP's input queue without demultiplexing: All IP packets go in the same queue. `ether_input` then issues a network software interrupt. This software interrupt has priority higher than that of any application, but lower than that of the hardware interrupt.

FreeBSD handles the network software interrupt by dequeuing each packet from IP's input queue and calling the `ip_input` routine. `ip_input` checksums the packet's IP header and submits the packet to preliminary processing: firewalling and/or NAT, if configured in the system, and IP options, if present in the packet header. This preliminary processing may drop, modify, or forward the packet. `ip_input` then checks the packet's destination IP address. If that address is the same as one of the host's addresses, `ip_input` reassembles the packet and passes it to the input routine of the higher-layer protocol selected in the packet header (i.e., TCP, UDP, ICMP, IGMP, RSVP, IPIP, or, in remaining cases, raw IP). Otherwise, if the destination is a multicast address, `ip_input` submits the packet to a higher-layer protocol, for local delivery, and to multicast forwarding, if the system is configured as a multicast router. Finally, if the destination matches neither the host's nor a multicast address, and the system is configured as a gateway, `ip_input` submits the packet to IP forwarding; otherwise, `ip_input` drops the packet.

TCP's and UDP's input routines checksum the packet and then demultiplex it. They find the protocol control block (PCB) that corresponds to the destination port selected in the packet header, append the packet to the respective socket receive queue, and wake up processes that are waiting for that queue to be non-empty. However, if the socket receive queue is full, FreeBSD drops the packet. Note that, because demultiplexing occurs so late in FreeBSD, packets destined to the host are dropped *after* protocol processing has already occurred.

Note also that, in FreeBSD, protocol processing of a received packet is asynchronous relative to the respective receiving process. On receive calls, the receiving process checks the socket receive queue. If the queue is empty, the process sleeps; otherwise, the process dequeues the data and copies it out to application buffers.

However, processes only get a chance to run if the receive load is not so high that all CPU time is spent processing network hardware or software interrupts (receive livelock). Moreover, even at moderate receive loads, process scheduling may be disturbed by the fact that the CPU time spent processing network interrupts is charged to whatever process was interrupted, even if that process is unrelated to the received packets.

3 LRP

Although popular, BSD's scheme for processing received packets can cause scheduling anomalies, as discussed in the previous section. LRP [14] has been proposed as a remedy to such anomalies. This section reviews how LRP achieves that.

As illustrated in Figure 2, LRP uses *channels* instead of a single IP input queue. A channel is a packet queue; LRP associates one channel to each socket. The network interface hardware or driver examines packet headers and enqueues each packet directly in the corresponding channel (early demultiplexing). Following a hardware interrupt, LRP wakes up the processes that are waiting for the channel to be non-empty. However, if the channel is full, the network interface drops the packet immediately, *before* further protocol processing. LRP handles TCP and UDP packets differently, as discussed in the following subsections.

3.1 UDP

In the UDP case, on receive calls, the receiving process performs the following loop while there is not enough data in the socket receive queue: While the corresponding channel is empty, sleep; then dequeue each packet from the channel and submit the packet to `ip_input`, which calls `udp_input`, which finally enqueues the packet in the socket receive queue. The receiving process then dequeues the data from the socket receive queue and copies it out to application buffers. Therefore, for UDP, LRP is *synchronous* relative to the receiving process's receive calls.

3.2 TCP

Unlike the UDP case, in the TCP case, LRP is *asynchronous* relative to the receiving process. LRP was designed to be completely transparent to applications and, in some applications, performing TCP processing synchronously relative to application receive calls could cause large or variable delays in TCP acknowledgments, adversely affecting throughput. In order to process TCP asynchronously without resorting to software interrupts, LRP may associate with each process an extra kernel thread that is scheduled at the process's priority and has its resource utilization charged to the process, as shown in Figure 2. This kernel thread continuously performs the following loop: While the process's TCP channels are empty, sleep; then dequeue each packet from a non-empty TCP channel and submit the packet to `ip_input`, which calls `tcp_input`, which finally en-

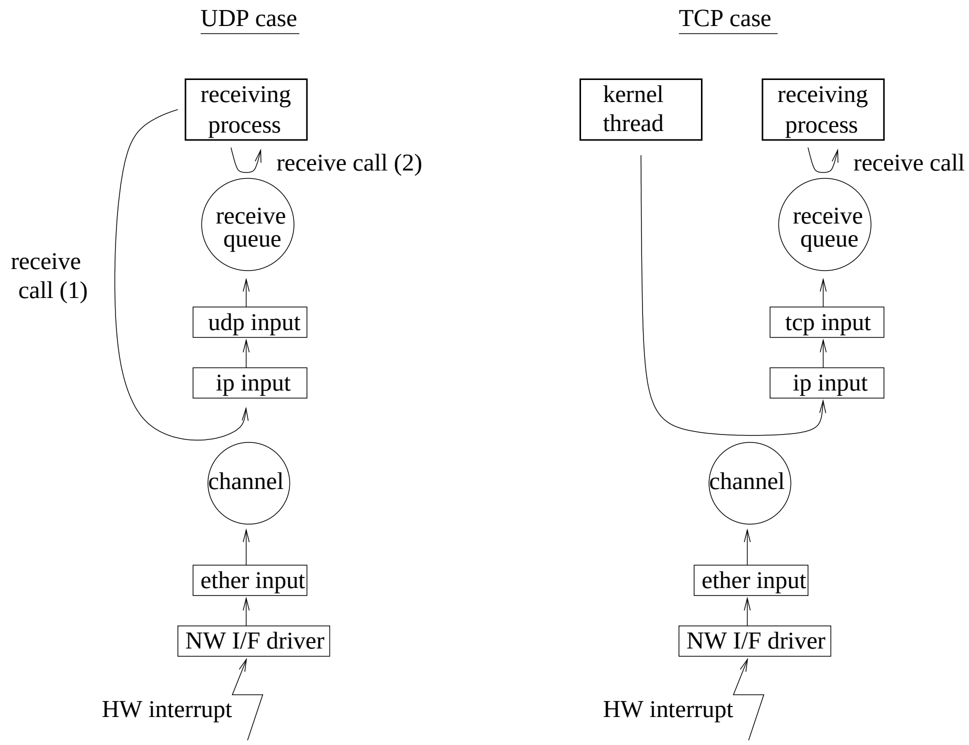

Figure 2: LRP processes received UDP packets in the context of the receiving process, when the latter issues a receive call. In the TCP case, LRP may use a kernel thread scheduled at the same priority as the receiving process.

queues the packet in the respective socket receive queue.

Instead of kernel threads, LRP may use a single process for similarly handling packets from *all* TCP channels in the system. In this case, LRP uses a Resource Containers facility to charge the resources used in processing each packet to the Container of the respective receiving application [1].

In either case, LRP handles TCP receive calls similarly to BSD: the receiving process simply checks the socket receive queue and, if the queue is empty, sleeps; otherwise, the process dequeues the data and copies it out to application buffers.

4 LRP's difficulties, shortcomings, and open questions

LRP was initially developed for a workstation time-sharing operating system, SunOS. However, LRP's ability to prevent certain networking-related scheduling anomalies caught our attention when we were modifying FreeBSD to implement Eclipse/BSD [7]. Eclipse/BSD is a new operating system that provides QoS guarantees using proportional-share scheduling. Eclipse/BSD is intended for both hosts and gateways. Our initial intention was to use LRP, but we ran into the problems reported in this section.

Our most obvious difficulty was that FreeBSD supports neither kernel threads nor Resource Containers, which are used in LRP's TCP processing. We could have chosen to build either piece of infrastructure, but decided against that because (1) the effort required would be nontrivial, and (2) we found that the redesign described in the next section (SRP) both leads to an easier implementation and solves several shortcomings and open questions in LRP, as discussed in the following subsections.

4.1 Inflexibility

Although LRP is an architecture with several implementation options, it is unnecessarily inflexible.

First, implementors may choose to perform early demultiplexing either in hardware (network interface card) or in software (network interface driver). However, in either case, each packet is demultiplexed only once, directly to the application that will consume the packet's data. These options do not support gateways, where applications like NAT may modify packet headers, and packets may need to be demultiplexed multiple times.

Second, implementors may choose to implement LRP's asynchronous TCP processing using, e.g., an additional kernel thread per process, or a system-wide TCP process and Resource Containers. However, LRP's TCP processing is always asynchronous. In some cases, synchronous TCP processing could give better performance, e.g. because of better memory locality, but the LRP architecture does not enable such option.

4.2 Interaction with real-time schedulers

The LRP and Resource Container papers [14, 1] suggest that the respective techniques can be used in conjunction with real-time or proportional-share schedulers, but do not satisfactorily explain how to achieve that. Those papers describe in reasonable detail only how LRP and Resource Containers are used in conjunction with time-sharing schedulers. The latter schedulers typically assign to each process a dynamic priority based on an average of the process's CPU usage [19]; no process has a fixed priority or CPU share.

Consider, in contrast, a real-time scheduler and processes that have fixed priorities. If LRP's asynchronous TCP processing is implemented with kernel threads, what should be the priority of those threads? By analogy to the time-sharing case, a TCP thread's priority would be the same as that of the respective process. However, in this case, process latencies and cumulative service [5] may suffer: Process events cannot preempt the TCP thread and may be delayed until the end of the latter's CPU quota.

On the other hand, if LRP is implemented with a single TCP process and Resource Containers, what should be the priority of the TCP process? Resource Containers calculate each thread's priority based on the thread's *scheduler binding*, i.e., the set of Containers recently served by the thread. In order to enable asynchronous TCP processing for all Containers, it appears that the TCP process's priority should be no less than the maximum priority of Containers in its scheduler binding. Therefore, similarly to the previous case (TCP kernel threads), application latencies and cumulative service may suffer. To reduce such degradation and prevent priority inversions and other scheduling anomalies, the TCP process's priority would have to be recomputed and the system would need to be rescheduled each time (1) the TCP process performs a *resource binding*, i.e., makes a call informing what Container it is about to serve, or (2) a packet arrives for a previously idle Container. These modifications may increase the Resource Container overhead considerably.

4.3 Interaction with proportional-share schedulers

The interaction of LRP with proportional-share schedulers similarly raises a number of questions. If LRP's asynchronous TCP processing is implemented with kernel threads, what CPU shares should those threads have? By analogy with the time-sharing case, (1) a TCP thread should have the same share as that of the respective process, and (2) if a TCP thread does not fully utilize its allocation, the respective process should be given the excess allocation (and vice-versa). To meet these requirements, LRP would need a *hierarchical* (not a *flat*) proportional-share scheduler. In a hierarchical proportional-share scheduler, LRP can take a CPU share and split it into subordinate equal shares for a process and the respective TCP thread [7]. Flat proportional-share schedulers, however, do not allow the hierarchical subdivision of shares, and divide excess allocations among *all* processes or threads, even if they are unrelated. Thus, LRP cannot prevent scheduling anomalies if it is used with such schedulers.

On the other hand, if LRP is implemented with a single TCP process and Resource Containers, how should scheduling be performed? In the time-sharing case, to avoid excessive context switching, systems with Resource Containers schedule *threads*, not *Containers*. A thread is scheduled based on a priority computed according the thread's scheduler binding. It is unclear what the analogous construct would be in the hierarchical proportional-share case. In particular, what would be the mapping between the hierarchy and shares of, on the one hand, Containers, and on the other hand, the threads that may dynamically serve those Containers? Proportional-share schedulers often compute fairly elaborate virtual time or virtual work functions in order to approximate GPS (Generalized Processor Sharing) scheduling. These functions may depend not only on shares, but also on requests' start and finish times [3, 2, 17, 7]. How would virtual time or work in the hierarchical Container space map to equivalents in the thread space? Furthermore, Resource Containers require each thread to do its own scheduling of requests from different Containers served by the thread. In the time-sharing case, thread-level scheduling is according to Container priority [1]. But in a proportional-share case, how would thread-level Container scheduling be integrated with system-level thread scheduling so as to approximate a global hierarchical proportional-share scheduling of Containers?[2]

5 SRP

As discussed in the previous section, the use of LRP in a system based on FreeBSD and with proportional-share scheduling, such as Eclipse/BSD, is fraught with difficulties and open problems. To circumvent those problems, we designed the alternative solution, SRP, described in this section. We discuss in the following subsections SRP's protocol organization, packet demultiplexing, packet notification, and protocol scheduling. Finally, we summarize the advantages of our approach.

5.1 Protocol organization

This subsection gives an overview of how SRP organizes the protocol processing of received packets.

SRP does not require modifications in the network interface hardware or driver. As illustrated in Figure 3, packet arrival causes a network hardware interrupt and transfers CPU control to the network interface driver, which passes the packet to `ether_input`.

In order to accommodate gateway functionality, such as firewalling and NAT, SRP organizes protocol processing in *stages*, where each stage comprises one or more protocol functions. Stages can be *preliminary* (including the `ether_input`, firewalling, NAT, and IP option handling stages) or *final* (including the end-application, ICMP, IGMP, RSVP, IPIP, raw IP, multicast, and IP forwarding stages). Preliminary stages invoke SRP's *next stage submit* (NSS) function to submit a packet to the respective next stage. Final stages are those that include IP and higher-layer protocols necessary to give a final disposition to each packet. The end-application stage, for example, includes IP, TCP, and UDP, and enqueues the packet in the corresponding socket receive queue.

Only the `ether_input` stage runs at interrupt level. The end-application stage runs in the context of the respective receiving application. All other stages run in the context of system processes with CPU guarantees from Eclipse/BSD. In the current SRP implementation, all protocol processing occurs inside the kernel. With minor SRP modifications, however, user-level processing would also be possible.

5.2 Packet demultiplexing and buffering

This subsection discusses how SRP demultiplexes and buffers incoming packets.

NSS is the central component in SRP's demultiplexing. NSS uses SRP's *multi-stage early demultiplex* (MED) function. MED returns a pointer to the PCB of the next stage to which a packet should be submitted, based on current stage and packet header. MED caches the PCB pointer in a field in the packet's first buffer, so that later, for example, TCP and UDP do not have to again look up the packet's PCB. Each PCB points to a socket, which in turn points to an *unprocessed input queue* (UIQ), to an *input function*, and to a list of *owner processes*, which are the processes that have the socket open. In order to reduce demultiplexing latency, MED optimistically assumes the common case where the packet header has appropriate length and version, is contiguous in the first buffer, and has a correct checksum. Stages can invoke an early demultiplex verifier function, EDV, to verify MED's assumptions. EDV caches the veri-

[2] The initial LRP prototype [14] used a system-wide TCP process. However, its reported results for TCP show only immunity to receive livelock, not correct resource accounting (the RPC results use UDP, which is always synchronous, and demonstrate the performance benefits of improving memory locality) [14]. Note that LRP's later implementation in the Resource Container prototype used an additional kernel thread per process for TCP processing, instead of a system-wide TCP process [1].

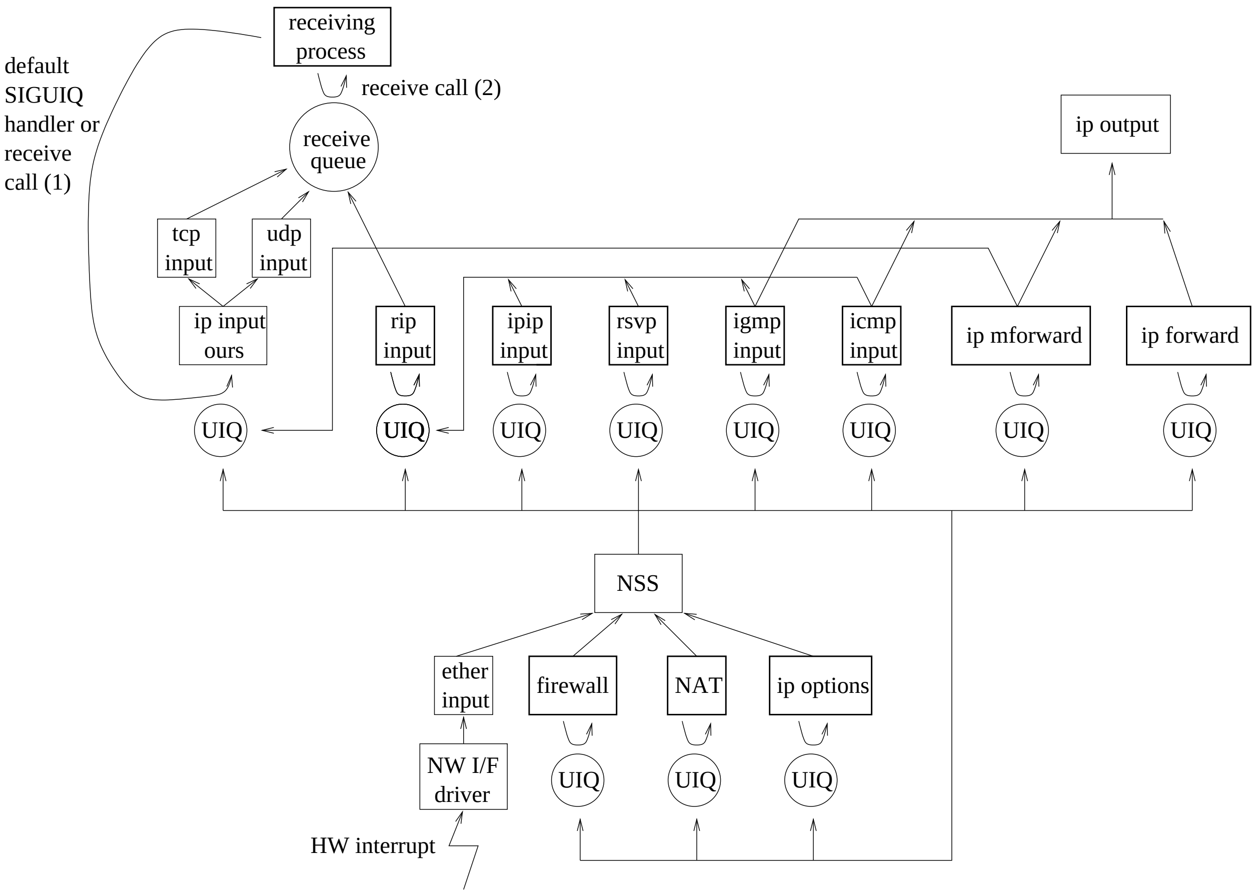

Figure 3: SRP processes received packets in the context of the receiving process, either in the default SIGUIQ signal handler or in receive calls. System processes with CPU reservations implement gateway protocol functionality.

fication in a flag in the packet's first buffer.

SRP buffers packets as follows. NSS invokes MED and determines the socket and UIQ pointed by the returned PCB. If the total amount of buffering used for the packet plus the socket's UIQ and receive queue exceeds the socket's receive buffer limit, NSS drops the packet; otherwise, NSS enqueues the packet in UIQ.[3] By demultiplexing and, in overload conditions, dropping packets *before* further processing them, SRP avoids receive livelock.

5.3 Packet notification and delivery

This subsection describes how protocols and applications receive notification and delivery of packets from SRP.

When SRP's NSS demultiplexes and enqueues a packet in a certain socket's UIQ, if there are processes waiting for that UIQ to be non-empty, NSS wakes up those processes; otherwise, NSS signals SIGUIQ to the socket's owner processes.[4]

SIGUIQ is a new signal whose default action is to dequeue packets from UIQ of the process's sockets and submit each packet to the respective socket's input function. However, any process can catch, block, or ignore SIGUIQ signals and, for example, defer protocol processing of received packets until a subsequent receive call.

On receive calls, the receiving process first dequeues

[3] Receive buffer limits can be set using the `setsockopt` system call.

[4] If all owner processes are sleeping non-interruptibly, signal delivery will be delayed until the first owner process is woken up. However, because processes only sleep non-interruptibly while waiting for a short-term (e.g., disk) event, the resulting SIGUIQ delay should not exceed other scheduling delays also present in a multitasking environment.

unprocessed packets from the socket's UIQ and submits those packets to the socket's input function. In the case of TCP or UDP sockets, the input function is a modified version of `ip_input`, which calls a modified version of `tcp_input` or `udp_input`, which finally enqueues the packet in the socket's receive queue. (The modifications in `ip_input`, `tcp_input`, and `udp_input` replace the original demultiplexing operations by cheaper verifications of MED's optimistic demultiplexing.) The receiving process then checks the socket's receive queue. If the queue is empty, the process sleeps; otherwise, the process dequeues the data and copies it out to application buffers.

5.4 Protocol scheduling options

This subsection discusses SRP's scheduling alternatives for protocol processing.

SRP's default SIGUIQ signal handler makes protocol processing asynchronous, accommodating those TCP applications for which synchronous protocol processing would be inappropriate.

Any application can, however, catch, block, or ignore SIGUIQ. To catch SIGUIQ, an application specifies one of the application's functions, f, to be called asynchronously by the system when a packet destined to the application arrives. f has the option of (1) issuing a receive call immediately, making protocol processing asynchronous with respect to the application's main control flow, or (2) deferring protocol processing to a synchronous receive call in the application's main control flow.

Another option is to block SIGUIQ. If an application blocks SIGUIQ, delivery of this signal is delayed until the application unblocks SIGUIQ. A final alternative is to ignore SIGUIQ. If an application ignores SIGUIQ, the system generates no signal when a packet destined to the application arrives. In such case, protocol processing is synchronous, occurring only when the application issues a receive call.

Applications may exploit the flexibility of catching, blocking, or ignoring SIGUIQ, for example: to control how much CPU time they spend on protocol processing; to prevent being interrupted while they are processing some critical event; or to perform protocol processing only immediately before they need the received data, which may improve memory locality.

If SIGUIQ signals are ignored or blocked, the number of context switches is the same as in the conventional BSD approach (no context switching when packets arrive), but memory locality may improve (packet data first accessed immediately before the application needs the data). LRP realizes the same benefits for UDP, but not for TCP.

If SIGUIQ signals are processed by the default handler or caught, the number of context switches may be higher than that of BSD, depending on the scheduling policy. However, the number of context switches will usually be less than one per packet. The default handler processes multiple packets in a process's UIQs when the process is scheduled. Additionally, if scheduling is time-sharing or real-time priority-based, a receiving process will preempt the currently running process only if the receiving process has higher priority (otherwise, context switching cannot happen). But if the receiving process has higher priority and is blocked on the socket, it would usually preempt the running process also on BSD. Similar observations apply to LRP's TCP handling. However, compared to SRP, LRP incurs additional context switching between the asynchronous TCP kernel threads or process and the receiving processes. Memory locality is similar for BSD, LRP's TCP, and SRP with default or caught SIGUIQ: The packet data is accessed asynchronously during protocol processing, possibly disturbing the cache.

5.5 Advantages

Because SRP checks buffering limits before protocol processing and processes protocols in the context of the receiving processes, SRP avoids receive livelock, charges protocol processing to the correct processes, and allows Eclipse/BSD to enforce and honor CPU guarantees.

SRP also solves LRP's problems mentioned in Section 4. Because SRP requires only a signaling facility, it can be easily ported to most existing systems, including those that support neither kernel threads nor Resource Containers. SRP's multistage demultiplexing function supports gateways, including NAT and other gateway functionality that may require packets to be demultiplexed multiple times. SRP's signals allow all applications, including those that use TCP, to opt for synchronous or

Protocol processing	Throughput (Mbps)		Utilization (%)	
	ave	std dev	ave	std dev
FreeBSD	69.35	0.70	36.3	0.6
SRP/default SIGUIQ	67.88	0.77	36.2	0.4
SRP/ignored SIGUIQ	68.30	1.00	36.4	0.6

Table 1: SRP does not significantly alter FreeBSD's TCP throughput and CPU utilization.

asynchronous protocol processing, thereby possibly improving performance. Finally, SRP's interaction with time-sharing, real-time, and flat or hierarchical proportional-share CPU schedulers is straightforward. SRP processes protocols always in the context of and under the control of the receiving process, regardless of how that process is scheduled. SRP thus avoids the difficult assignment of real-time priorities or proportional shares to separate TCP threads or system-wide TCP processes.

6 Experimental results

This section presents an experimental evaluation of SRP.

The first two experiments show that SRP does not significantly hurt FreeBSD's networking performance (throughput, CPU utilization, and latency). For these experiments, we connected two PCs S and R to the same Fast Ethernet hub (100 Mbps). Host S has a 300 MHz Pentium II CPU and 32 MB RAM and runs FreeBSD. Host R has a 266 MHz Pentium II CPU and 64 MB RAM and runs FreeBSD, Eclipse/BSD with default SIGUIQ, or Eclipse/BSD with ignored SIGUIQ. Both hosts use Intel EtherExpress PRO 100 Ethernet cards. In the first experiment, we ran on S a sender application that sends 10 MB data to a receiver application on R, using TCP with 64 KB send and receive socket windows and no delayed acknowledgments. A compute-bound background application also ran on R, but the hosts and network were otherwise idle. We measured the TCP throughput and R's CPU utilization (estimated by the background application's rate of progress) during the data transfer, repeating the experiment ten times. Table 1 presents the averages and standard deviations of our measurements.

In the second experiment, we ran on S and R applications that send to each other packets of increasing length, using UDP with 64 KB send and receive

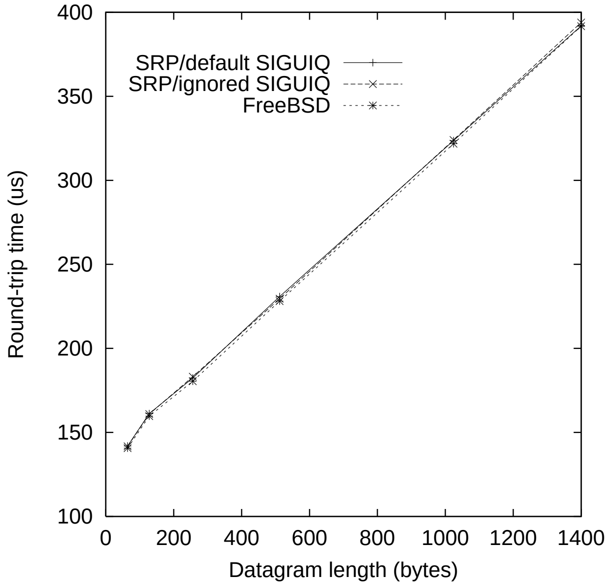

Figure 4: SRP does not significantly alter FreeBSD's UDP round-trip times.

socket windows. No other application ran on R. We measured the round-trip times and report on Figure 4 the averages of ten tries.

The results in Table 1 and Figure 4 suggest that SRP's performance penalties on FreeBSD are quite small or not statistically significant.

The third experiment demonstrates that SRP avoids receive livelock. In this experiment, host S is a Pentium Pro PC running FreeBSD, while host R is a PC running either FreeBSD or Eclipse/BSD on a 266 MHz Pentium Pro CPU with 64 MB RAM. The hosts were connected by Fast Ethernet at 100 Mbps. There was no other load on the hosts or network. A sender application on host S sent 10-byte UDP packets at a fixed rate to a receiver application on host R. When running on Eclipse/BSD, the receiver application used SRP's default SIGUIQ handler. We measured the application-level reception rate while varying the transmission rate, and report the av-

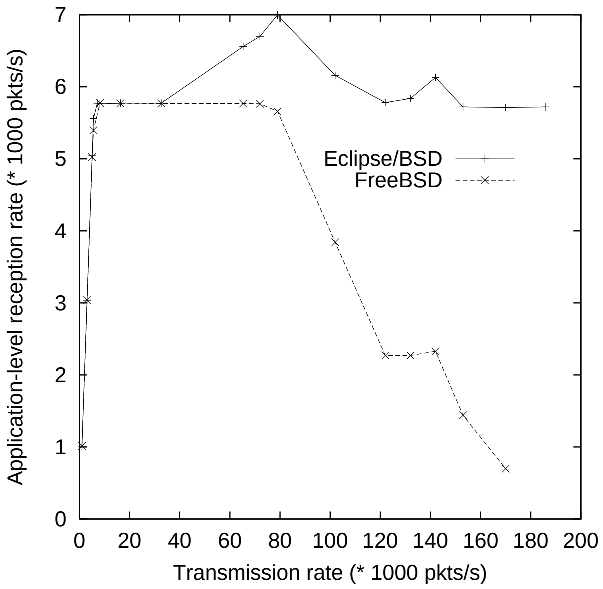

Figure 5: SRP prevents receive livelock in Eclipse/BSD.

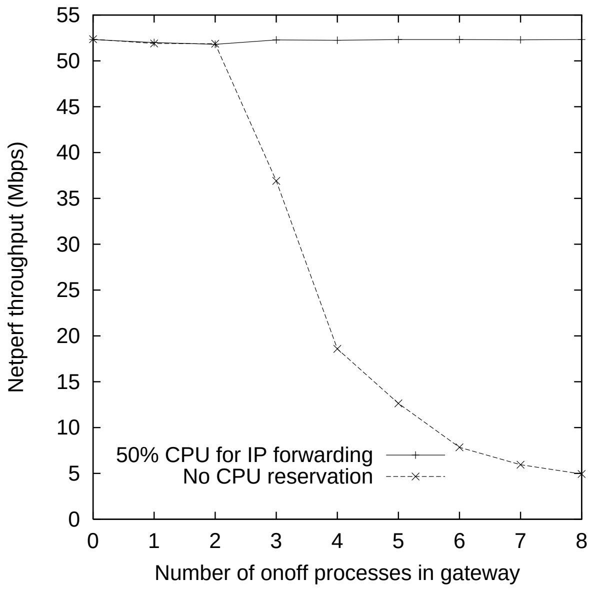

Figure 6: IP forwarding needs a CPU reservation to avoid losing performance due to other applications on the gateway.

erages of five runs. Figure 5 shows that, both on FreeBSD and on Eclipse/BSD, essentially all packets were received up to a transmission rate of about 5600 packets per second. Above a certain transmission rate, however, FreeBSD's reception rate collapses because of receive livelock. On the contrary, Eclipse/BSD's reception rate reaches a plateau and remains substantially at that level as the transmission rate increases. Eclipse/BSD's stability is due to SRP, which avoids receive livelock.

The fourth experiment shows that proportional-share scheduling is desirable in gateways that process application-specific code, in addition to forwarding packets. We used the `netperf` utility to measure TCP throughput between hosts A and B on two separate Fast Ethernets connected via a gateway G. Gateway G is a 266 MHz Pentium II PC with 64 MB RAM and running Eclipse/BSD, while host A is a 400 MHz Pentium II PC with 64 MB and running Linux, and host B is a 133 MHz Pentium PC with 32 MB RAM and running FreeBSD. In addition to IP forwarding, the gateway ran a variable number of instances of an application called `onoff`. After each time an `onoff` process runs for 11 ms, it sleeps for 5 ms. The IP forwarding process ran either with a 50% CPU reservation or with no reservation. The `onoff` processes ran with no CPU reservation. There was no other load on the hosts or network. Figure 6 demonstrates that, without a CPU reservation for IP forwarding, other application load on the gateway can cause TCP throughput to collapse. On the contrary, an appropriate CPU reservation isolates IP forwarding performance from other load on the gateway.

The final experiment demonstrates that some applications can benefit from SRP's ability to defer protocol processing. In this experiment, a client application on host C continuously executed transactions each consisting of sending requests with 512 KB of random integer data to a server application on host S, and then receiving the server's reply of a few bytes. Host C is a 300 MHz Pentium II PC with 64 MB RAM and running FreeBSD, while host S is a 266 MHz Pentium II PC with 32 MB RAM and running Eclipse/BSD. The hosts were connected by a Fast Ethernet at 100 Mbps. Client and server applications communicated over a TCP connection between sockets with 512 KB send and receive buffers. Hosts and network were otherwise unloaded. The server application processed requests using one of three algorithms: compute five averages of the request data; view the request data as two sets of four 64 KB vectors and compute their 16 internal products; or select the nth largest number among the request data (using `partition` [13]). While processing these algorithms, the server application either used the default SIGUIQ handler or ignored the SIGUIQ

Application	Elapsed time (ms) with SIGUIQ default ave	default std dev	ignored ave	ignored std dev	Improvement %
Averages	63.1	0.8	54.8	0.2	13.1
Internal products	50.7	0.2	46.5	0.4	8.3
Select nth	61.0	0.3	55.7	0.2	8.6

Table 2: On Eclipse/BSD, some applications can improve performance by catching, blocking, or ignoring SIGUIQ signals.

signal. We used the CPU's internal cycle counter to measure, in the server application, the time interval necessary for sending the reply to the previous request, computing the current request, and receiving the next request. We report the averages and standard deviations of ten runs. Table 2 shows that this server application runs up to 13% faster when it ignores SIGUIQ, making protocol processing synchronous. This improvement is due to better memory locality when protocol processing is performed only immediately before the data is needed.

7 Related and future work

We are not aware of previous reports about experiences in porting LRP or Resource Containers to other systems, especially systems that offer QoS guarantees via proportional-share scheduling, such as Eclipse/BSD.

This paper describes only how Eclipse/BSD processes packets received from a network. Other papers describe Eclipse/BSD's overall architecture and application programming interface (`/reserv` [7]), CPU scheduler (MTR-LS [5]), disk scheduler (YFQ [6]), and network output link scheduler (Bennet and Zhang's WF^{2}Q [2, 3]). Eclipse/BSD is easy to use: Even unmodified legacy Unix applications can automatically run with appropriate QoS guarantees under Eclipse/BSD [8].

Nemesis [18] is an operating system built from scratch according to a radical new architecture designed to prevent *QoS cross-talk*, that is, one application's interference in another application's performance. Most Nemesis services, including TCP/IP, are implemented as libraries that are linked with applications. Therefore, services are performed in the context of and charged to the respective applications, similarly to what is achieved by SRP's SIGUIQ signals. However, SRP's signals and kernel-mode signal handler may be easier to port to today's mainstream systems, which typically have a monolithic architecture quite unlike that of Nemesis.

Because LRP is not available on FreeBSD, we were unable to compare SRP and LRP directly. Such comparison would be interesting future work.

8 Conclusions

We proposed a new mechanism, SRP, whereby packet arrival sends a signal to the receiving process. The default handler of this signal performs protocol processing on the packet, but the receiving process may catch, block, or ignore the signal and defer protocol processing until a subsequent receive call. In any case, protocol processing occurs in the context of the receiving process and is correctly charged. Our experiments show that, like LRP, SRP avoids BSD's receive livelock. However, SRP has the advantages of being easily portable to systems that support neither kernel threads nor Resource Containers, such as FreeBSD; giving applications control over protocol scheduling; using a multi-stage demultiplexing strategy that supports gateway functionality; and easily enabling real-time or proportional-share scheduling.

Acknowledgments

We thank John Bruno and Banu Özden for valuable discussions during the design phase of this work. We also thank the anonymous referees and our paper's shepherd, Liviu Iftode, for their useful comments.

References

[1] G. Banga, P. Druschel and J. Mogul. "Resource containers: A new facility for resource management in server systems," in *Proc. OSDI'99*, USENIX, Feb. 1999.

[2] J. Bennet and H. Zhang. "WF^2Q: Worst-Case Fair Weighted Fair Queueing," in *Proc. INFOCOM'96*, IEEE, Mar. 1996, pp. 120-128.

[3] J. Bennet and H. Zhang. "Hierarchical Packet Fair Queueing Algorithms," in *Proc. SIGCOMM'96*, ACM, Aug. 1996.

[4] R. Braden, L. Zhang, S. Berson, S. Herzog and S. Jamin. "Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification." IETF, RFC 2205, Sept. 1997.

[5] J. Bruno, E. Gabber, B. Özden and A. Silberschatz. "The Eclipse Operating System: Providing Quality of Service via Reservation Domains," in *Proc. Annual Tech. Conf.*, USENIX, June 1998, pp. 235-246.

[6] J. Bruno, J. Brustoloni, E. Gabber, B. Özden and A. Silberschatz. "Disk Scheduling with Quality of Service Guarantees," in *Proc. ICMCS'99*, IEEE, June 1999, vol. II.

[7] J. Bruno, J. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. "Retrofitting Quality of Service into a Time-Sharing Operating System," in *Proc. Annual Tech. Conf.*, USENIX, June 1999, pp. 15-26.

[8] J. Brustoloni, E. Gabber, A. Silberschatz, and A. Singh. "Quality of Service Support for Legacy Applications," in *Proc. NOSSDAV'99*, June 1999, pp. 3-11. Available at `http://www.nossdav.org/`.

[9] K. Calvert, S. Bhattacharjee, E. Zegura and J. Sterbenz. "Directions in Active Networks," in *Communications Magazine*, IEEE, 1998.

[10] J. Case, M. Fedor, M. Schoffstall and J. Davin. "A Simple Network Management Protocol (SNMP)," IETF, RFC 1157, May 1990.

[11] K. Cho. "A Framework for Alternate Queueing: Towards Traffic Management by PC-UNIX Based Routers," in *Proc. Annual Tech. Conf.*, USENIX, June 1998.

[12] Cisco. "FlowCollector Overview," at `http://-www.cisco.com/univercd/cc/td/doc/-product/rtrmgmt/nfc/nfc_3_0 /nfc_ug/-nfcover.htm`

[13] T. Cormen, C. Leiserson and R. Rivest. "Introduction to Algorithms." MIT Press, Cambridge, MA, 1990.

[14] P. Druschel and G. Banga. "Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems," in *Proc. OSDI'96*, USENIX, Oct. 1996, pp. 261-275.

[15] K. Egevang and P. Francis. "The IP Network Address Translator (NAT)," IETF, RFC 1631, May 1994.

[16] Flux Research Group. "The OSKit," at `http://www.cs.utah.edu/projects/flux/-oskit/`.

[17] P. Goyal, X. Guo and H. Vin. "A Hierarchical CPU Scheduler for Multimedia Operating Systems," in *Proc. OSDI'96*, USENIX, Oct. 1996, pp. 107-121.

[18] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns and E. Hyden. "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications," in *JSAC*, 14(7), IEEE, Sept. 1996, pp. 1280-1297.

[19] M. McKusick, K. Bostic, M. Karels and J. Quarterman. "The Design and Implementation of the 4.4 BSD Operating System," Addison-Wesley Pub. Co., Reading, MA, 1996.

[20] J. Mogul and K. K. Ramakrishnan. "Eliminating Receive Livelock in an Interrupt-Driven Kernel," in *Proc. Annual Tech. Conf.*, USENIX, 1996, pp. 99-111.

[21] L. Peterson, S. Karlin and K. Li. "OS Support for General-Purpose Routers," in *Proc. HotOS'99*, IEEE, Mar. 1999.

[22] P. Srisuresh and D. Gan. "Load Sharing using IP Network Address Translation (LSNAT)." IETF, RFC 2391, Aug. 1998.

[23] W. R. Stevens. "TCP/IP Illustrated," vol. 1, Addison-Wesley Pub. Co., Reading, MA, 1994.