# Network Subsystems Reloaded: A High-Performance, Defensible Network Subsystem

Anshumal Sinha Sandeep Sarat Jonathan S. Shapiro
anshumal@cs.jhu.edu sarat@cs.jhu.edu shap@cs.jhu.edu
*Systems Research Laboratory*
*Department of Computer Science*
*Johns Hopkins University*

**Abstract**

Traditionally, operating systems have used monolithic network stack implementations: implementations where the whole network stack executes in the kernel or (in microkernels) in a single, trusted, user level server. Code maintenance issues, ease of debugging, need for simultaneous existence of multiple protocols, and security benefit have argued for removing the networking implementation from kernel and dividing it into multiple user level protection domains. Previous attempts to do so have failed to deliver adequate performance. Given the advances made in both hardware (CPU, Memory, NIC) and micro-kernel design over the last decade, it is now appropriate to re-evaluate how these re-factored implementations perform, and to examine the reasons for earlier failures in greater detail.

Building on the primitives of the EROS microkernel, we have implemented two network subsystems: one a conventional, user mode, monolithic design and the other a domain-factored user level networking stack that restructures the network subsystem into several protection domains. We show that the restructuring maintains performance very close to that of the monolithic design, and that *both* designs compare favorably to a conventional in-kernel implementation. We discuss the issues faced in engineering the domain-factored implementation to achieve high performance, and present the quantitative evaluation of the resulting network subsystems.

## 1 Introduction

Traditionally, network systems have been implemented as monolithic subsystems that execute in the kernel or (in microkernels) in a single, trusted, user level server. To achieve performance, monolithic implementations sacrifice flexibility, maintainability, and security. Application specific knowledge cannot easily be incorporated into the network subsystem, debugging is more difficult, and the network stack itself becomes either a single point of failure for the entire system (in-kernel) or a single point of failure for the application (library approaches). Since network stacks are large, complicated software systems, they are intrinsically vulnerable to attack. It is therefore desirable to isolate *both* the operating system and the client application from the security vulnerabilities of the network subsystem.

Previous attempts to do so – most notably by Thekkath *et al.* [TNML93] – have generated disappointing results, suggesting that this design approach may be impractical. Unfortunately, Thekkath's analysis does not evaluate in detail the breakdown of time spent in various components. As a result, it is difficult to separate the impacts of three effects: user-level implementation, domain factoring, and the poor performance of the Mach Microkernel. Given the advances made in both hardware (CPU, Memory, NIC) and micro-kernel designs over the last decade, it is now appropriate to re-evaluate how such re-factored implementations perform, and to examine the reasons for earlier failures in greater detail.

Since in-kernel network stacks are the most common implementation approach, we use Linux [BC00] as a reference baseline for performance comparison. When a domain structured implementation is compared directly to an in-kernel implementation, the resulting comparison can be difficult to understand. In particular, it is difficult to know how to separate the performance consequences of user level implementation from the performance consequences of domain factoring. In order to support such evaluation, we have implemented *two* network stacks derived from a common code base: one monolithic, and the other factored into multiple protection domains.

Our monolithic implementation is a conventional microkernel network stack implemented as a user mode application. The network stack (based on lwIP [Dun02]) includes the ethernet drivers. Client application(s) and the network stack execute in separate protection domains. This implementation is roughly comparable to the conventional Linux implementation. Compromising the network stack compromises all network client applications, and also the entire kernel: most modern network interface

cards (NICs) implement physical DMA for performance reasons which implies that the network driver can overwrite arbitrary kernel memory.

The domain factored implementation places the ethernet driver in a separate protection domain, using a packet filter [MRA87] to demultiplex packets to the appropriate network stack. In this implementation, the network stack itself has no privileged access to the hardware. The impact of a compromised protocol stack is limited to a single application. The complexity of the packet demultiplexer (the ethernet driver) is primarily driven by the hardware interface, and can be validated independent of the protocol stack. The primary added cost of this implementation is the introduction of additional protection domain crossing delays.

Thekkath *et al.* [TNML93] measured a conceptually similar design, showing performance degradations of 39% and 20% for 10 Mbit and 100 Mbit ethernet implementations (see Table 1; we consider here only those results using packet sizes that conform to the ethernet specification). We show in the present work that this overhead can be reduced to 13% using an unoptimized kernel implementation. We believe that an optimized kernel would bring this performance to within 5%.

In this paper we present the design, implementation and performance of our restructured network subsystem on a modern, high-performance microkernel.

# 2 Objectives

Based on the above discussion regarding the limitations faced by an in-kernel network stack and the constraints needed to implement the domain factored design, we arrived at a list of goals for the domain factored network subsystem. In this section we discuss these goals and issues involved.

Ideally a network subsystem should meet several simultaneous goals:

- **Flexibility:** It should support the co-existence of multiple protocols that may be fine-tuned to exploit application-specific knowledge.
- **Resource Accountability:** Clients should be responsible for providing all the resources necessary to support their network activities. Buffers used to store network data must therefore be supplied by the client. This immunizes the stack from the potential denial of resource attacks.
- **Isolation:** QoS Crosstalk should be avoided to prevent clients from interfering with each other.
- **Resilience:** The network subsystem should be resilient to faults, both intentional or unintentional, which might have crept into the implementation of the network subsystem.
- **Performance:** In spite of isolating the network subsystem in its own protection domain, the network subsystem should deliver throughput and latency comparable to a conventional implementation.

Meeting all of these goals simultaneously is challenging. Monolithic network subsystems combine all resource management into a single protection domain, which compromises resource accountability and isolation. In-kernel protocol stacks are a single point of failure that may impact the entire kernel. User-mode monolithic stacks, as have been built for several microkernels, remain a single point of failure impacting set of applications that are using the network. While this single point of failure cannot be entirely eliminated (the packet filter is necessarily shared), its size can be substantially reduced. This allows quality assurance efforts to be focused more effectively.

When previous networking stacks have been split into multiple protection domains, performance has suffered. Protection domain boundaries usually imply data copies from one address space to another. Both the copies themselves and the cross-domain control transfer operations (IPCs) become a source of performance degradation. Because of these overheads, it is frequently asserted that protection carries intrinsic overhead.

Of the various user level network subsystems that have been created by researchers, Thekkath's work come closest to our design. Thekkath proposed a user level implementation using an in-kernel packet demultiplexer and transport protocols as user level libraries [TNML93]. The performance results from this work (Table 1) were disappointing. We believe that this is primarily due to faults of the Mach microkernel [GDFR90] that was used in Thekkath's experiments. The Mach microkernel interface was not flexible enough to provide full resource accountability, its cache performance was inadequate to support Thekkath's design [CB93], and its interprocess communication performance was significantly lower than current designs such as L4 [HHL+97] or EROS [SSF99].

Unfortunately, Thekkath's analysis does not evaluate in detail the time spent in various components. As a result, it is difficult to separate the impacts of three effects: user-level implementation, domain factoring, and the poor per-

| System | Throughput(Mb/s) User Packet Size(bytes) | | | |
|---|---|---|---|---|
| | 512 | 1024 | 2048 | 4096 |
| **Ethernet** | | | | |
| Ultrix 4.2A | 5.8 | 7.6 | 7.6 | 7.6 |
| Mach 3.0/UX(mapped) | 2.1 | 2.5 | 3.2 | 3.5 |
| Thekkaths | 4.3 | 4.6 | 4.8 | 5.0 |
| **DEC SRC AN1** | | | | |
| Ultrix 4.2A | 4.8 | 10.2 | 11.9 | 11.9 |
| Thekkaths | 6.7 | 8.1 | 9.4 | 11.9 |

Table 1: Performance results reported by Thekkath *et al.* Table reproduced with permission of the author.

formance of the Mach Microkernel. In order to provide a better understanding of these contributions, we have implemented two protocol stacks: one is a monolithic user mode implementation and the other is factored into multiple protection domains.

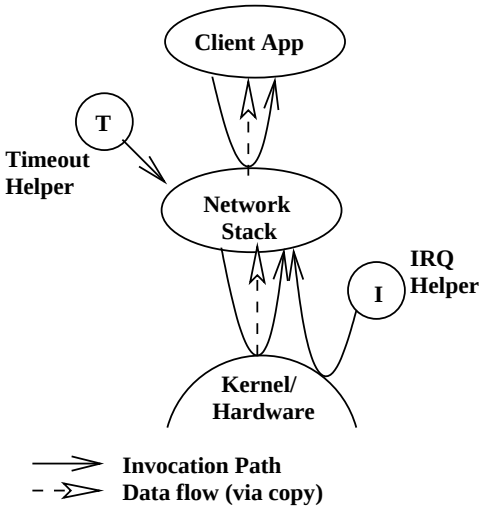


Figure 1: The EROS monolithic network subsystem

# 3 Monolithic Network Subsystem

Our monolithic network subsystem is a conventional microkernel network stack implemented at user level. The protocol suite is based on lwIP, which is a lightweight implementation of IP, UDP and TCP designed for low memory embedded systems. We chose lwIP as our initial protocol stack because it is simple and highly portable. Our ethernet drivers were created by adapting existing Linux ethernet drivers to operate at user level. Client applications execute in a protection domain separate from the network stack.

Our initial objective in creating the monolithic stack was ease of implementation. Since the stack is monolithic, simultaneous existence of multiple protocol stacks is impossible, and it is a single point of failure in the system. Resources cannot be accounted for as network buffers are allocated from a private stack pool instead of the clients being charged for them. Lack of resource accountability and multiplexing at a high level (Session layer) [Ten01] imply that there exists crosstalk between the clients using the stack. However, this implementation serves as a useful reference for comparison against other monolithic implementations. It also shares most of its code with the domain structured implementation (Section 4), allowing an "apples to apples" comparison between the two approaches.

The monolithic stack has two "helper" processes. The IRQ helper notifies the stack of newly arrived interrupts. A timeout helper is used to notify the stack that a timeout has occurred (e.g. TCP timeouts). The helpers are highly specialized and therefore small (<128 KB). The EROS kernel transparently allocates such small processes to small address spaces [Lie95] to reduce context switch overheads. As shown in Fig 1, the stack copies network data to/from the hardware DMA buffers to/from its buffer pool during network processing. Data flow across the client-stack process boundary is accomplished by a copy.

The performance of this implementation is comparable to the native Linux network stack, and is evaluated in Section 5.

# 4 Factored Network Subsystem

Like the monolithic implementation, the factored EROS network implementation is based on lwIP and executes in user mode. We present the design and implementation of the factored network subsystem and show how the designed goals are met.

## 4.1 Design

The factored version of the network subsystem has been structured to achieve all the design goals described earlier. We start by dividing the network subsystem into three independent protection domains namely, *enet* - consists of the ethernet drivers and a packet demultiplexer, *network stack* - consists of the various protocol implementations like IP, TCP, UDP, ICMP and ARP, and the network communication enabled *client application*. The division of the

network subsystem in this fashion is mostly at points of data multiplexing. We have effectively pushed the point of multiplexing down, adjacent to the network interface (into the enet layer). This is an accepted practice to minimize QoS crosstalk [Ten01].

Factoring makes it feasible to isolate attacks on the network subsystem and restrict the damage to the domain of attack, thus compartmentalizing the vulnerabilities of each domain. The result is a fault resilient, layered defense mechanism. Separating the enet layer from the network stack adds to the modularity of the design. This enables a client application to spawn a new stack in the event of a failure of the running stack. This also helps achieve flexibility as factoring supports simultaneous existence of multiple protocols.

#### 4.1.1 Client-provided shared memory

Monolithic network subsystems use a centralized buffer pool. Network data meant for a client is buffered using buffers from this pool. So it is possible for a client to throttle the network bandwidth of another client using the stack without explicit resource management policies [DB96].

To accomplish proper resource accountability, it is necessary for the client to provide the store for the network data it requests/transmits. In the factored network subsystem, the client provides shared memory regions which are mapped three-way into the enet, the network stack and the client itself. This shared memory is used as a network buffer store by the stack for that particular client.

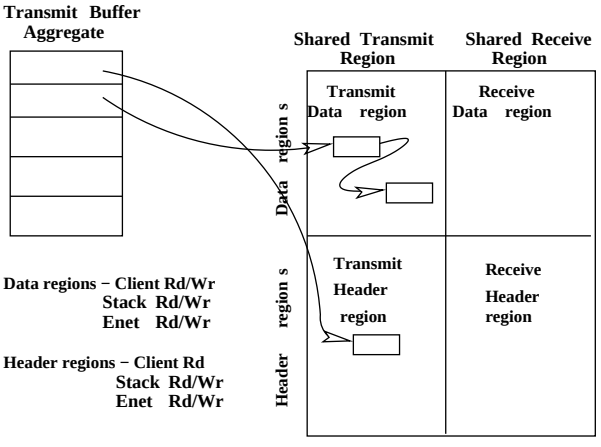


Figure 2: Client provided shared memory regions

The client is responsible for supplying four shared buffer regions. They are:

- Transmit data region: Client specific transmit data is stored in buffers allocated from this region.
- Transmit header region: Client specific transmit protocol header is stored in buffers allocated from this region.
- Receive data region: Client specific received data is stored in buffers allocated from this region.
- Receive header region: Client specific received protocol header is stored in buffers allocated from this region.

The client has (read,write) permissions to only two of these regions - the transmit data region and the receive data region. The other two regions - the receive header region and the transmit header region are read only for the client. All regions are mapped into the enet and the network stack, with (read, write) permissions. (Figure 2).

The data regions are used as a store to allocate network buffers which contain client-specific data. The header regions are used as a store to allocate network buffers which contain protocol headers. A malicious client cannot manipulate or corrupt protocol headers as it has only read permissions to the buffers containing the header. Hence, it cannot influence the protocol state machine of the stack.

#### 4.1.2 Scatter-Gather

Additional process boundaries add an extra cost to the data movement across these boundaries. We use shared memory to avoid copying across protection domains. Scatter-gather is used to enable copy avoidance. We describe this mechanism in detail for inbound and outbound network packets.

For outbound network traffic, the client acquires an unused network buffer from the transmit data region and places the data to be transmitted in it. The stack uses a free buffer in transmit header region to write the protocol headers and then chains these two buffers together into a single buffer chain. The enet uses the transmit descriptor to this buffer chain to transmit data.

The case of inbound network traffic is exactly inverse. The enet "scatters" the network packet into client specific data, storing it into buffers from the receive data region, and into protocol headers storing it into buffers from receive header region.

Since the buffers used for storing data come from the client, eager demultiplexing [DB96] is necessary at the enet level. As mentioned earlier, enet has a packet filter, which is used to identify the client to which the incoming data belongs. If the enet is unable to allocate network buffers from either of the two receive regions due to exhaustion, we simply drop the packet as is done in lazy receiver processing. The justification behind this policy is that ethernet does not provide guaranteed packet delivery in any case. If the client cannot keep up with the pace of incoming data, the ethernet driver is free to discard packets.

Resource accountability using the shared memory design presented can avert a potential denial of resource attack in which a rogue application can request network data and then refuse to dequeue its packets. Refusal to do so deprives other needy applications of precious network buffers. This attack is no longer feasible in the factored network subsystem as the buffers for the packets are allocated from the client's store. The client's refusal to dequeue the packets will only lead to client's buffer pool exhaustion. Once exhausted, the packets meant for that particular client are dropped by the enet. No other client can be affected due to the mis-behavior of the rogue client.

One advantage of this resource accounted shared memory design besides security is the *scatter-gather* mechanism which ensures a zero-copy, and hence helps in performance enhancement. Banga *et al.* [BDM99] only account for the execution time spent in the network stack on behalf of the client. With the factored network subsystem design, it is possible to extend the notion of network resource to include the buffers used for packet transfer also. This ensures that a client application is not able to deprive other clients of network buffer resources as in a centralized buffer pool design. This, in combination with the layered structure of the factored network subsystem, prevents the QoS crosstalk as the multiplexing is pushed further down in the network subsystem and the effect of one client on another is minimized.

## 4.2 Implementation

We have modified lwIP keeping in mind the design goals discussed earlier in Section 2. lwIP uses a stack-centralized pool of network buffers. We modified lwIP to use shared memory buffers. Enet includes a packet filter (adapted from the LRP implementation [DB96] which is used to eagerly demultiplex packets to the appropriate receiving network stack.

We now describe the components which are handled differently when compared to existing user level network implementations.

### 4.2.1 Shared Memory

The increased number of process boundary crossings in a factored design results in an increase in latency incurred during the processing. The main source of this latency is the cost of cross-space control transfers and cross-space data copies. To avoid these expensive data copies, shared memory is employed. In most existing implementations, the shared memory for the network buffers is a global entity allocated by some component of the network stack or is a memory pinned resource. In our factored design, the client provides a source of storage that is used to allocate a shared memory region that is used exclusively on behalf of that client. We list the steps involved in the creation of a shared buffer:

1. The client grants a storage allocator to the stack. The client can rescind this storage, but has no access to pages that are allocated using this allocator.
2. The stack allocates the four shared buffer regions that we described earlier using the storage allocator.
3. The buffers in the transmit regions are reserved for transmission exclusively and the buffers in the receive region for reception exclusively. The stack 'formats' these pages as ring buffers (described in the next section).
4. The stack requests the enet to map read/write version of all these pages into its address space.
5. The stack requests the client to map read/write version of the pages of the transmit, receive data region into its address space. Note that the client has only read access to the buffers in the header regions.

The mapping of a client-specific shared memory region into the various domains is now complete. This mapping is done during setup time when the client registers with the stack and the enet domains. The header region buffers (Figure 2) are used for buffering protocol headers and the data region buffers for client specific data. In the case of transmission, the client places data into the transmit data region buffers. The stack prepares protocol headers in transmit header region buffers and links the respective buffers together. This buffer chain is passed off to enet for transmission. The scenario of reception is exactly the inverse. A rogue client can at most mangle the buffers in the data region but has no access to the buffers in the header region. Hence, in no way can affect the stack or the enet domains.

### 4.2.2 Ring Buffers

We now describe the data structure used to hold the network buffers themselves. For ease of presentation, we only discuss the scenario during reception. The case of transmission is exactly the inverse.

As already described, buffers for client data are allocated from the data regions, while the protocol header is allocated from header regions. We initially format (Step 2 in the creation of the shared memory region) the four shared buffer regions into uniformly sized ring buffers. Each buffer consists of a buffer header (not to be confused with the packet header) and space which stores the buffer payload (network headers and network data reside here). The buffer header stores meta-data, including the status of the particular buffer, the size of payload stored in the buffer and the pointer to the next buffer (See figure 3).

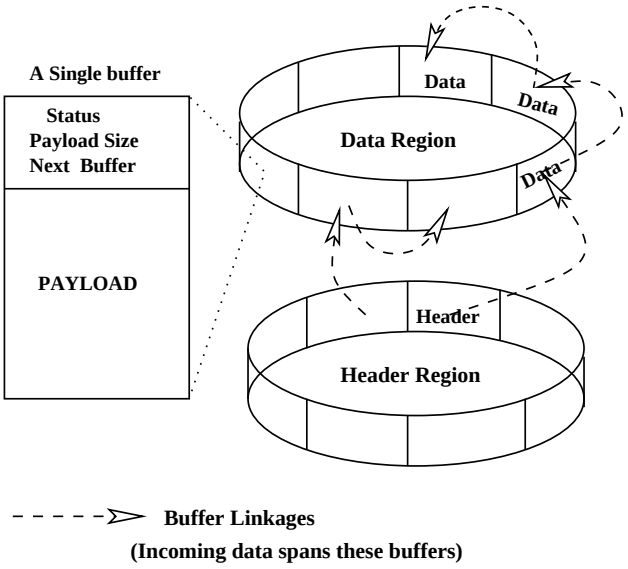


Figure 3: Receive ring buffers. The incoming data packet is scattered into header and client data and stored into appropriate ring buffers by the enet packet filter.

When a packet arrives for a particular client, the enet packet filter splits the packet into buffers allocated from two different ring buffers. Buffers for the packet data are allocated from the receive data region ring buffer. Buffers for the protocol headers are allocated from the receive header region ring buffers. Traditionally, the functionality of the packet filters in the ethernet layer is to demultiplex the incoming packet to the recipient stack [EK92]. We have extended the packet filter to include placement of the inbound packet into the appropriate client and stack data and header ring buffers. Packets can appear on the network interface out of order and it is up to the higher layer protocols to re-order them. Buffer pointers can be appropriately redirected to make the data appear "contiguous".

The ring buffer mechanism employed is similar to the hardware DMA ring buffer mechanism used by most network interfaces. A direct consequence of using ring buffers for the network packet is that allocation and deallocation occur in strictly increasing ring sequence and hence their time complexity is **O(1)**. A network buffer list implementation also achieves **O(1)** allocation and deallocation. But it is slower, as it needs explicit mutexes to read/write the buffers.

### 4.2.3 Inter Domain Communication

We have extensively used the EROS IPC mechanisms, specifically *call*, *return* and *retry*. Equivalents to CALL and RETURN are commonly available in most micro kernels. The RETRY operation is specific to EROS. Clients queued using RETRY do not block other requests, and their reactivation honors the scheduling policy of the operating system.

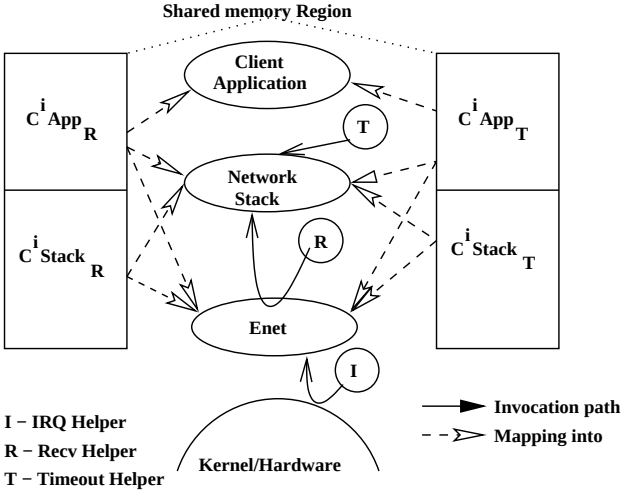


Figure 4: The three domains and Helpers. The client specific shared memory regions are shown too

EROS uses a synchronous IPC mechanism. Since calls are blocking, upcalls are prohibited as they can lead to denial of service attacks. If the enet makes an upcall to

the stack and the stack fails to return, the enet remains waiting indefinitely. Further upcalls can also lead to a potential deadlock where the stack and the enet are attempting to call each other simultaneously. So we introduce a few domains (Helpers) which have highly specific roles (Figure 4). They are:

- IRQ helper: Notifies the enet domain of newly arrived interrupts from the network interface. This process runs in an infinite loop waiting for an IRQ and on an IRQ signal turns around calls the enet layer.
- Rx helper: Notifies the stack that the enet has received data. This process sits in an infinite loop calling the stack and the enet in turns. The enet issues a RETRY to block Rx until data is available. The Rx helper relieves the network stack of having to block for I/O data so that client applications can avail the stack's services promptly.
- Timeout Agent: Notifies the stack of timeouts (e.g TCP timeouts). This process runs in an infinite loop calling the stack after regular time intervals (say 100 msec).

Separating the protocol processing, packet demultiplexing and applications into different threads has a performance disadvantage. By doing so, we have effectively replaced what would have been function calls in a monolithic design with more expensive IPC calls. Adding the helpers could potentially degrade our performance further.

Decreasing the latency in processing can be pursued by reducing the number of IPC invocations and the context switch time. We carry out a number of optimizations to this effect.

- The Helpers are specialized and small (<128KB). The EROS kernel transparently allocates such small process to small address spaces. The advantage is that unlike typical address space switches we can avoid TLB flush while switching to/from a small address space.
- "Amortizing" the IPC invocations that are needed so that we can reduce the per packet address space switches. We describe this in the next subsection.

#### 4.2.4 Ping-Pong design

In the receiving data path we typically incur the following address space switches triggered by IPC : IRQ helper → enet (CALL enet IRQ notification), enet → IRQ helper, enet → Rx Helper (RETRY Rx Helper to notify the stack), Rx Helper → stack (CALL stack packet reception Notification), stack → Rx Helper (RETURN), Rx Helper → enet (CALL) (Figure 5). This is quite a high overload for a single packet. High speed NICs can generate thousands of interrupts per second. The CPU cannot keep up with this rate of interrupt generation. This can lead to *live-lock*. 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.

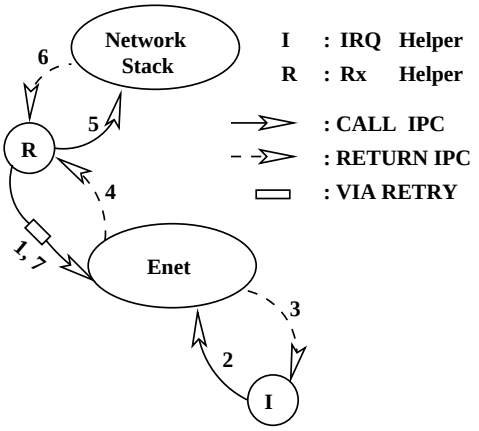


Figure 5: The various IPC invocations in the Rx data path. The numbers show the sequence of events.

Linux uses the NAPI poll approach [SO01] where interrupts are disabled and polling is scheduled for a certain time interval before re-enabling the interrupts . We use a similar flow-aware solution to tackle this. The rationale is that if the enet, stack, Rx Helper are already doing their respective tasks no IPC invocations are necessary for notification. Accordingly, the enet and stack processes ping-pong between receiving and transmitting. The enet alternates between reading a network packet from the hardware Rx ring buffer and writing a packet into the hardware Tx ring buffer. The stack alternates between reading data off the shared memory Rx ring buffers and writing data into the shared memory Tx ring buffers. The state of maximum throughput is achieved when both the stack and the enet are ping-ponging simultaneously between transmitting and receiving. In this state no IPC invocations are issued and the stack processes whatever the enet receives and the enet transmits the data on the transmit ring buffer named by the transmit descriptors. The only overhead in the data path is the context switch time between the stack and the enet(the IPC overhead is absent). The network interrupts are disabled until the enet stops ping-ponging.

# 5 Evaluation

This section presents both a quantitative evaluation of the various implementations and a qualitative evaluation of the resilience achieved in the factored implementation.

## 5.1 Performance

Performance measurements were carried out using two 900 MHz Pentium IIIs with 256 MB RAM and 33 MHz PCI bus, each equipped with a 3com 3C905C-TX (100Mbps) ethernet card and a NetGear 602T - BCM5701 (1000Mbps) gigabit card, interconnected by standard Cat-5e patch cables. The linux drivers for these NICs were ported to EROS. To fit our design we removed the NAPI poll interrupt mitigation approach to mitigate interrupts that linux uses in its gigabit driver.

Due to unrelated research that is ongoing in our laboratory, the version of EROS reported here does not provide an optimized IPC implementation. On the kernel used, a typical EROS round trip IPC takes 2268 cycles ( 2.40 $\mu$s on our test machine). For comparison, the optimized implementation takes approximately 500 cycles per round trip ( 0.53$\mu$s). Transfers using smaller packet sizes are significantly influenced by IPC performance in all implementations, and we expect they would improve when using an optimized kernel.

We used the standard network tests `ping` and `ttcp` for benchmarking, considering the linux 2.4 network stack as a touchstone. We rate the performance of the refactored network subsystem and a monolithic version of the network subsystem (running lwIP) on EROS. lwIP running on linux is not listed here because it is severely crippled, as it uses a tuntap interface over linux.

### 5.1.1 Ping

The ping time between two hosts is a good measure of the latency. We measured the round trip latency time of ICMP echo requests with size varying from 64 bytes to 9000 bytes (fig 6).

Linux and the EROS-monolithic stack have comparable ping latencies throughout. EROS-monolithic is slightly slower because it uses the services of an IRQ notifier between the monolithic subsystem and the kernel. This is primarily an IPC-related delay. The EROS-factored implementation has a higher latency due to the increased number of address space switches as a result of the higher number of processes (primarily the receive helper).

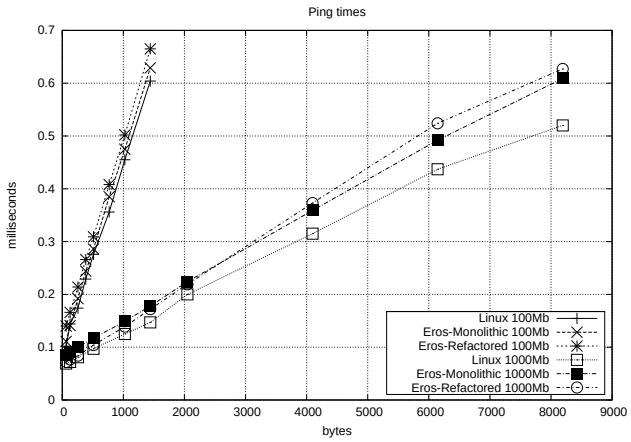


Figure 6: Ping times for the various stacks

### 5.1.2 ttcp

Ttcp is a benchmark that measures the time taken to transmit and receive data between two systems using the UDP or TCP protocols. We ran ttcp-r for different sizes of socket buffers. We set up a ttcp transmitter on a redhat 2.4 linux machine and the receiver was on the candidate to be benchmarked. The maximum wire capacity on a 100 Mbps ethernet is 12.5 MBps and on a 1000Mbps Gigabit is 125MBps.

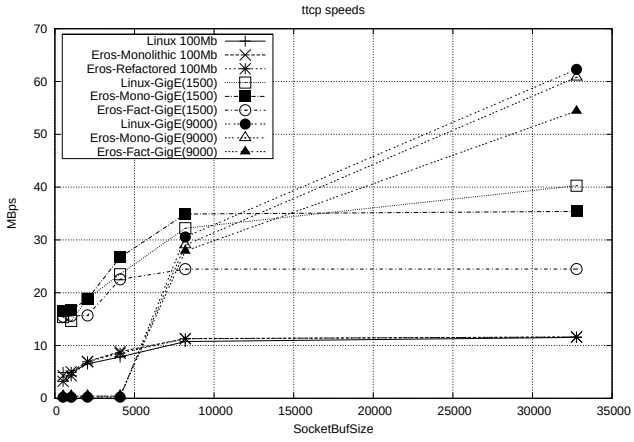


Figure 7: Observed ttcp speeds

Linux and the EROS-monolithic stack perform comparably in this test too. The Linux gigabit stack is slightly faster because of the NAPI interrupt mitigation in their driver. The EROS gigabit driver does not use this technique. Also the EROS-monolithic uses the services of an IRQ notifier between the monolithic subsystem and the kernel. As the socket buffer size decreases linux pays an increasing performance penalty due to the higher number of data copies from kernel to user space and back required for the same size of data to be transferred. EROS-monolithic incurs a similar cost copying the data from the user-mode network stack to the application using IPC.

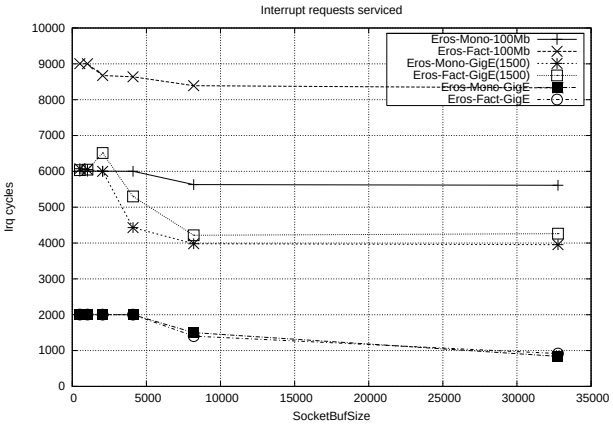


Figure 8: IRQs per 10,000 packets serviced during ttcp.

Figure 8 and Figure 9 show the IRQs serviced and the user cycles consumed respectively during a typical ttcp run. As seen from Figure 8, the factored network subsytem takes more number of interrupts than the monolithic version. This is due to the fact that in the monolithic stack, the protocol processing occurs in the context of the interrupt service routine (ISR). During protocol processing, packets that arrive do not flag interrupts and are serviced in the same context subsequently. In the factored network subsystem, the protocol processing and ISR occur in different processes viz. the stack and the enet. On packet arrival, the enet signals the stack to carry out packet processing and itself becomes available to receive new interrupts. The higher interrupt rate of the factored subsystem results in higher processor usage as seen from the user cycles figure (Figure 9).

The factored network subsystem achieves 87% performance of the monolithic linux stack (worst case) and up to 89% performance of the monolithic EROS stack. While the EROS-factored implementation avoids data copy overhead using shared memory, it incurs a larger number

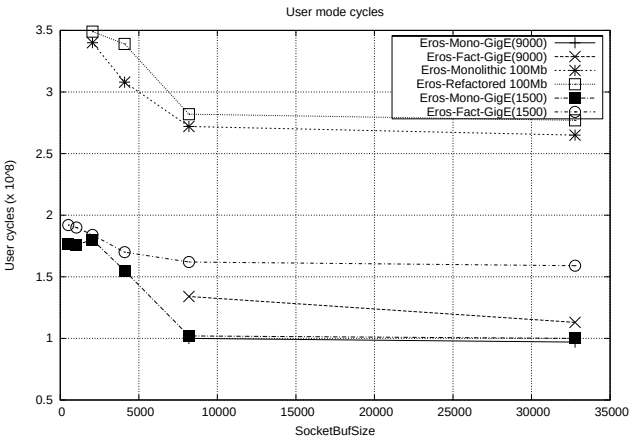


Figure 9: User cycles per 10,000 packets consumed during ttcp. Data for GigE(9000) for socketbuf lengths less than 8192 have been omitted to avoid graph compression. Measurements for smaller socket buffer sizes for the Gigabit (9000 mtu) case are limited by Linux transmit performance.

of context switches for control transfer purposes. The RETRY operation in particular is excessively expensive.

Chen *et al.* [CB93] have argued that cache performance is an insurmountable obstacle to microkernel performance in general. While this claim was convincingly rebutted by Liedtke [Lie96], Blackwell has pointed out that instruction cache locality plays a significant role in small packet processing [Bla96], and Druschel's work on Fbufs [DP93] was motivated in part by cache-based domain crossing costs in the *x*-kernel. Mossberger *et al.* identify a number of important cache-related optimizations for reducing network processing latency [MPBO96].

Figure 10 compares instruction and data cache misses in the monolithic and factored EROS implementations. The results suggest that there *is* a relationship between instruction cache traffic and performance, but it is not a simple relationship. In some cases, the instruction cache costs are noticeably *higher* on the monolithic network stack, even though the monolithic implementation has slightly better performance in those cases. Note also that instruction cache effects dominate data cache effects overall by three decimal orders of magnitude in conventional ethernet frames. This is a pleasing result, both because instruction cache tuning is relatively easier to accomplish and because we know that the lwIP implementation objective was simplicity rather than performance. More detailed in-

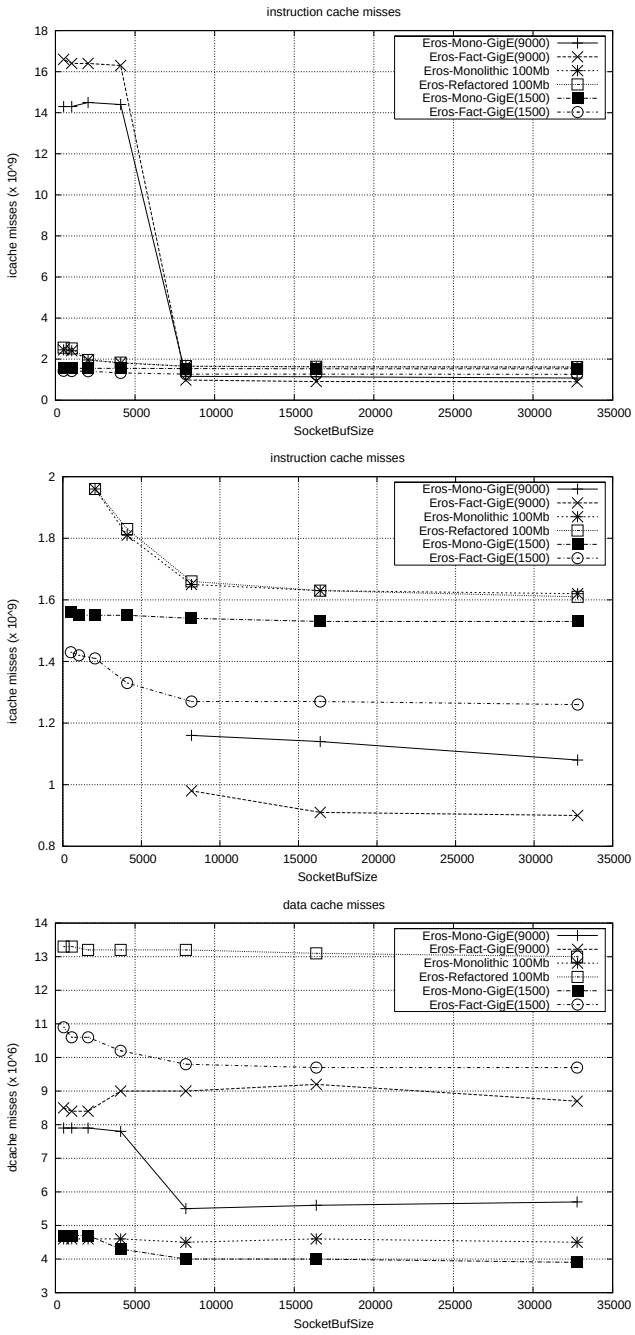


Figure 10: User instruction and data cache misses per 10,000 packets during ttcp. The second graph is identical to the first with selected points removed to eliminate compressed display.

vestigation of these results is indicated.

We believe that higher performance would be achieved using the optimized IPC. Further, we note that lwIP was developed for low memory usage rather than for high performance, which suggests that further performance enhancements are possible in the protocol stack itself. However, the key lesson in these figures is that the performance throughout is directly proportional to the interrupt service rate. Our current implementation does not poll the network at high packet rate. Performance is therefore limited by the interrupt latency.

Microbenchmarks provide reliable measurement of low-level performance, but often fail to accurately predict the behavior of the overall system. End to end performance is heavily influenced by issues of locality, and domain factoring raises the risk of defeating these performance gains. Given this, the microbenchmark results reported here must be interpreted with caution, and may not prove to be definitive.

## 5.2 Resilience

A monolithic network stack suffers from several shortcomings that impact resilience:

1. Because it is monolithic, the network stack is a single point of failure for all applications using the network. In an in-kernel implementation, bugs in the network stack manifest as failures of the system as a whole.
2. Achieving precise per-client resource accountability is exceedingly difficult.
3. The amount of robustness-critical code (in lines) is large, which increases the intrinsic vulnerability of the network stack.
4. There are no well-defined internal interface boundaries at which fail-fast and recover-fast disciplines can naturally be applied.

Improvements on the first three criteria can be measured directly using straightforward metrics. The last requires performance evaluation.

The factored network stack presented in Section 4 is both a user-mode stack and a per-client stack. Being user-mode, it is unable to crash the system as a whole. Being per-client, failures in one stack do not impact the behavior of other client applications.

Several previous network stacks have provided CPU resource accountability and used eager demultiplexing for purposes of maintaining quality of service [Ten01, DB96], The per-client network stack described here runs from a client-supplied scheduling class, ensuring that CPU allocation for network processing purposes observes the same scheduling policy as all other processing that occurs on behalf of an application. Novel to the factored design presented here is that storage resources are *also* allocated using per-client resources: each client provides a storage source from which its buffers are allocated.

The monolithic EROS stack described in Section 3 includes a total of 10,803 critical lines of code. Of these, 5,217 are the ethernet driver, and 5586 are the lwIP implementation. Any failure at any point in this code potentially compromises all networking applications. In contrast, the factored implementation of Section 4 places the lwIP implementation outside of the trusted code base, reducing the amount of critical code to 5,612 lines (the ethernet driver), leaving an untrusted lwIP implementation of 6,297 lines. All of these lines are in addition to the EROS kernel itself, which is approximately 15,985 lines of code. In contrast, the Mach kernel used in Thekkath's measurements was over 100,000 lines of code *excluding the network stack and drivers*. While precise sizes for Thekkath's stack are not available, the total critical component size of the stack described here are less than 16% of the earlier effort.

The fault recovery strategy for the factored network stack is first to fail eagerly. Failure may be signalled by various sorts of anomoly and intrusion detectors; choosing a specific mechanism for detection is beyond the scope of this paper. The interested reader may wish to examine [CPM+98] and its citations. Once a failure has been recognized, the relevant network subsystem is involuntarily halted and a new network subsystem is instantiated to replace it. Running on a 931Mhz Pentium-III, instantiation of a new network stack is accomplished in 387ms. This number is severely impacted by the currently unoptimized IPC implementation. Based on earlier measurements, a factor of two improvement would be expected in a production-suitable kernel.

### 5.3 Lessons for Microkernel Design

One lesson to draw from the factored network stack is the need for non-blocking notification mechanisms in microkernel architectures. The RETRY system call was originally introduced as a way to work around the absence of such a mechanism in the EROS design. It provides a means to enqueue a client in such a way that the client will re-execute its last CALL when unblocked. This, coupled with the use of worker threads, can be used to simulate non-blocking notification. The RETRY operation design attempted to satisfy a philosophy, shared with Liedtke [Lie93] and Ford [FL94], that IPC operations should be thread migrating, blocking, and usually non-preemptive.

However, the end result is cumbersome to use, difficult for the programmer to understand, and introduces unnecessary context switches. Future derivatives of the EROS kernel will incorporate a non-blocking notification mechanism whose behavior is similar to that of hardware interrupts. In contrast to UNIX signals, delivery of notifications will be deferred until the recipient process next enters a "ready to receive" state, and their semantics and prioritization will be entirely determined by the recipient.

## 6 Related Work

A great deal of of research has gone into building user level network subsystems. Several of them have relied on specialized hardware support in which network buffer pools on the card can be reserved by applications. HP developed such a mechanism for the Jetstream LAN [EWL+94] where applications could reserve buffer pools on the AfterBurner [DWB+93] card. The data on arrival was demultiplexed to the correct application pool. Unet [vEBBV95] implementation for SBA 200 modified the firmware to add a new Unet compatible interface.

The factoring of the subsystem into layers is an extension of the design proposed by Thekkath,U-net and Mogul [MRA87] who have all argued for separating the interface driver from the protocol stack. They have also extensively used shared memory between these layers to mitigate copy costs.

U-net follows a similar approach for a parallel and distributed computing architecture. Although Unet showed promising results performance-wise, it failed to address certain vulnerability issues. The shared memory buffers allocated to buffer network data were typically pinned to physical memory. This made them a scarce resource, thus making it vulnerable to possible denial of resource attacks. Using shared memory in network subsystem has been widely used concept. Druschel's work on Fbufs [DP93] uses the notion of transferrable buffers to reduce domain crossing overheads in the *x*-kernel. IOLite [PDZ99] uses shared memory and *ACLs* to ensure a protected unified buffering scheme. Exokernel [EKO95], which supports application level resource management scheme also refers to the design of sharing the network

buffers similar to the one suggested by Druschel [PDP94]. But none of the above mentioned works talk about the issues of making the applications accountable for the buffer they use. The EROS factored implementation requires that clients supply the resource for all buffers used on their behalf. It is this concept of resource accountability that makes EROS-factored design different from the above mentioned works.

We use eager demultiplexing and lazy processing in our design. Demultiplexing immediately at the network interface is necessary for purposes of QoS [Ten01, DB96] and for user-level implementation of network subsystems. This also accomplishes proper time resource accountability. In our scheme, space resource accounting is also accomplished. If the client is devoid of space for the network data, we simply drop the packet i.e. the stack does not waste time in protocol processing. The reason for this policy is that if the client is not able to keep up with the pace of incoming data, there is no need for the stack and driver to do that on behalf of the client.

We introduce a novel invocation pattern between the various domains in the network to improve the speed of the stack, by amortizing the cost in the invocation and interrupt servicing. Previous mechanisms [KMY01, SO01] use interrupt mitigation or interrupt coalescing. At high speeds, the processor cannot keep up with 1 interrupt per packet. The interrupt mitigation schemes work by disabling interrupts when there is work and re-enabling them when there is none. These schemes are application inconsiderate and need changes to the driver. Soft timers [AD00, ST93] allow efficient scheduling of software events. These can avoid interrupts and reduce context switches associated with network processing. But these are operating system facilities and hence need changes to the kernel. We use a scheme similar to the mitigation scheme where the driver ping-pongs between transmission and reception. Interrupts are turned off during this period of activity. When activity ceases, interrupts are re-enabled. We thus amortize the cost of interrupts and domain switches incurred in the data path.

The "fail fast" concept has recently been rediscovered by Candea *et al.* [CF03], but was well known to earlier practitioners [Gra86, SS83]. Fail-fast (or "crash-only," as Candea *et al.* call it) techniques were used in production in the KeyKOS operating system (the predecessor to EROS) by 1983 [Har85], and in the KeyTXF transaction processing system by 1987. The notion has been an underlying design pattern in EROS since 1990.

# 7 Acknowledgments

The network stack in both the monolithic and the factored designs is based on the lwIP TCP/IP protocol suite. Although lwIP was designed primarily for embedded system, we adapted it to EROS because of its small simple code base. The packet demultiplexer used in the factored version of the network subsystem is a simplified version of the soft LRP demultiplexer.

Chandramohan Thekkath graciously permitted us to reproduce the performance results shown in Table 1. Our shepherd, Vivek Pai, was exceptionally patient with us, which allowed us to present our results more accurately.

We would like to thank John Vanderburgh and Eric Northup of the Systems Research Laboratory, Johns Hopkins University for (respectively) helping in the implementation of shared memory in EROS and assisting in debugging adapted the lwIP stacks.

# 8 Conclusion

We have presented the design and implementation of a domain factored Network Subsystem which provides a defensible multi-layered network subsystem with microbenchmark performance comparable to the existing in-kernel monolithic networking subsystem.

Although a great deal of research has gone into improving the network subsystem, we are aware of no prior work attempting to address both performance and security together, nor work that has successfully factored a network stack into multiple protection domains. In this work, we have demonstrated that such factoring is practical. Our results show that speed need not be sacrificed in any substantial measure to attain security. This suggests that the much-promoted "cost of protection" have been greatly overestimated in previous work. The cache performance results presented here suggest that the cache effects resulting from factoring are more complex than had previously been assumed.

One key to the performance we have achieved is integrating the packet filter concept with scatter-gather technology. The EROS-factored packet filter simultaneously demultiplexes the packets to the appropriate recipient and divides the packets into component parts that are delivered into the appropriate memory region.

When combined with appropriate use of CPU scheduling, the EROS-factored networking stack satisfies all of

the goals identified in Section 2. Fault resilience, protocol flexibility, resource accountability, and performance isolation are achieved without unduly compromising performance. This is possible in part because the EROS kernel interface exposes low-level resources using a protection model that allows us to correctly align the protection mechanisms with the interests of the various parties. In particular, the ability to separate authority to deallocate storage from authority to read that storage is essential to maintaining the trust and resource relationships between the different components.

# References


[AD00] Mohit Aron and Peter Druschel. Soft timers: efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, 18(3):197–228, 2000.

[BC00] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Press, October 2000.

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Operating Systems Design and Implementation*, pages 45–58, 1999.

[Bla96] Trevor Blackwell. Speeding up protocols for small messages. In *Proc. ACM SIGCOMM '96*, pages 85–95, September 1996.

[CB93] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proc. 14th Symposium on Operating Systems Principles*, December 1993.

[CF03] George Candea and Armando Fox. Crash-only software. In *Proc. HotOS-IX*, Lihue, Hawaii, USA, May 2003.

[CPM+98] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[DB96] Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Operating Systems Design and Implementation*, pages 261–275, 1996.

[DP93] Peter Druschel and Larry Peterson. Fbufs: A high-bandwidth cross-domain transfer utility. In *Proc. 14th Symposium on Operating Systems Principles*, December 1993.

[Dun02] Adam Dunkels. lwip - a lightweight tcp/ip stack, October 2002. http://www.sics.se/ adam/lwip/.

[DWB+93] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner: Architectural support for high-performace protocols, 1993.

[EK92] D. Engler and M. F. Kaashoek. Dpf: Fast, flexible message demultiplexing using dynamic code generation. In *Proc. SIGCOMM '96 Conference*, pages 53–59, Stanford, CA, USA, August 1992.

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. 15th Symposium on Operating Systems Principles*, pages 251–266, December 1995.

[EWL+94] Aled Edwards, Greg Watson, John Lumley, David Banks, Costas Calamvokis, and C. Dalton. User-space protocols deliver high performance to applications on a low-cost gb/s lan. *SIGCOMM Comput. Commun. Rev.*, 24(4):14–23, 1994.

[FL94] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a migrating threads model. In *Proc. Winter USENIX Conference*, pages 97–114, January 1994.

[GDFR90] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. UNIX as an application program. In *Proc. USENIX Summer Conference*, pages 87–96, June 1990.

[Gra86] Jim Gray. Why do computers stop and what can be done about it? In *Proc. 5th Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, CA, January 1986.

[Har85] Norman Hardy. The KeyKOS architecture. *Operating Systems Review*, 19(4):8–25, October 1985.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian. Schönberg, and Jean Wolter. The performance of $\mu$-kernel-based systems. In *Proc. 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France., October 1997.

[KMY01] Ilhwan Kim, Jungwhan Moon, and Heon Y. Yeom. Timer-based interrupt mitigation for high performance packet processing. In *Proc. 5th International Conference on High-Performance Computing in the Asia-Pacific Region*, Gold Coast, Australia, September 2001.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proc. 14th ACM Symposium on Operating System Principles*, pages 175–188. ACM, 1993.

[Lie95] Jochen Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Arbeitspapiere der GMD No. 933, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1995.

[Lie96] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, 1996.

[MPBO96] David Mosberger, Larry L. Peterson, Patrick G. Bridges, and Sean W. O'Malley. Analysis of techniques to improve protocol processing latency. In *SIGCOMM*, pages 73–84, 1996.

[MRA87] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proc. Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, Austin, TX, USA, November 1987.

[PDP94] Bruce S. Davie P Druschel and Larry L. Peterson. Experiences with a high-speed network adaptor: A software perspective. In *Proc. of ACM SIGCOMM 94*, 1994.

[PDZ99] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Io-lite: A unified i/o buffering and caching system. In *Proc. Third USENIX Symposium on Operating Systems Design and Implementation*, pages 22–25, New Orleans, Louisiana, USA, February 1999. USENIX Association.

[SO01] Jamal Hadi Salim and Robert Olsson. Beyond softnet, 2001.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *Computer Systems*, 1(3):222–238, 1983.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawah Island Resort, near Charleston, SC, USA, December 1999. ACM.

[ST93] Jonathan M. Smith and C. Brendan S. Traw. Giving applications access to gb/s networking. *IEEE Network*, 7(4):44–52, 1993.

[Ten01] D.L Tennenhouse. Layered multiplexing considered harmful, 2001.

[TNML93] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, 1993.

[vEBBV95] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, 1995.