Protocol Service Decomposition for High-Performance Networking

Chris Maeda Brian N. Bershad

March 1993
CMU-CS-93-131

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in the 14th ACM Symposium on Operating Systems Principles. December 5-8, 1993.

This research was sponsored in part by the Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by ARPA/CMO under Contract MDA972-90-C-0035, the Xerox Corporation, and Digital Equipment Corporation. Bershad was partially supported by a National Science Foundation Presidential Young Investigator Award. Maeda was partially supported by a National Science Foundation Graduate Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, Xerox Corporation, Digital Equipment Corporation, the National Science Foundation, or the U.S. Government.

Keywords: Operating Systems, Protocol architecture, Mach, Network communication, Measurements

Abstract

In this paper we describe a new approach to implementing network protocols that enables them to have high performance and high flexibility, while retaining complete conformity to existing application programming interfaces. The key insight behind our work is that an application's interface to the network is distinct and separable from its interface to the operating system. We have separated these interfaces for two protocol implementations, TCP/IP and UDP/IP, running on the Mach 3.0 operating system and UNIX server. Specifically, library code in the application's address space implements the network protocols and transfers data to and from the network, while an operating system server manages the heavyweight abstractions that applications use when manipulating the network through operations other than send and receive. On DECstation 5000/200 systems connected by 10Mb/sec Ethernet, this approach to protocol decomposition achieves TCP/IP throughput of 1088 KB/second, which is comparable to that of a high-quality in-kernel TCP/IP implementation, and substantially better than a server-based one. Our approach achieves small-packet UDP/IP round trip latencies of 1.23 ms, again comparable to a kernel-based implementation and more than twice as fast as a server-based one.

1. Introduction

In this paper we describe a new approach for implementing network protocols that enables them to have high performance and high flexibility. The key insight behind our work is that an application's interface to the network is distinct and separable from its interface to the operating system. By separating the interfaces, we can provide a fast path between the application and the network while maintaining the semantics of operating system abstractions specified by standard application programming interfaces. Specifically, code in the application address space implements the network protocols and transfers data to and from the network, while an operating system server implements the machinery required when applications manipulate a network session through operations other than send and receive. By placing the critical paths of the protocol in the application's address space, we avoid protection boundary crossings, data copying, and unnecessary software layers in the important common case of send and receive. We provide flexibility because the user-level networking software may be developed, configured, and specialized independently from the rest of the operating system.

We have implemented our protocol architecture in the context of the Mach 3.0 operating system [Accetta et al. 86] and CMU's UNIX server [Golub et al. 90] on MIPS R3000 [Kane 88] and Intel i486 [Intel 90] processors running on 10Mb/sec Ethernet-based networks. Our system includes a complete TCP/IP and UDP/IP stack implemented as a code library linked with each application program, and a set of operating system facilities that emulate completely the BSD socket programming interface [Stevens 90].

Our approach of separating the protocol implementation into two pieces, one fast that resides in the application's address space providing network connectivity, and one complete that resides in an operating system server providing full interface compatibility, has resulted in substantial performance improvements relative to a server-based implementation. More importantly, our user-level protocol libraries achieve performance (both throughput and latency) that is comparable to, and in some cases better than, well-tuned kernel-based implementations.

The rest of this paper

In the next section we detail the motivation and goals for application-level protocols. In Section 3 we present an overview of our design. In Section 4 we describe the system's performance, and an application-specific protocol optimization that demonstrates the flexibility of our approach. In Section 5 we discuss related work. Finally, in Section 6 we present our conclusions.

2. Motivation and goals

Our work is motivated by the desire to have network protocol software execute at user-level with the same or better performance than when it executes in the kernel. Protocol software is generally implemented as part of an operating system kernel [Leffler et al. 89], or as part of a dedicated server process [Rashid & Robertson 81, Golub et al. 90]. The main advantage of a server-based protocol implementation is flexibility because the protocol code is decoupled from the kernel [Mogul et al. 87], allowing it to be more easily modified [Jacobson et al. 92, Clark et al. 91, Clark et al. 92] and optimized [Clark & Tennenhouse 90, Forin et al. 91], especially on an application-specific basis [Felten 92].

Protocols implemented in user-level servers, although flexible, have tended towards worse performance than when implemented in the kernel. In a server-based protocol, control and data cross twice as many protection boundaries when travelling between the network and the application. In one case, the extra overhead resulted in performance that was two to four times worse than an in-kernel implementation [Maeda & Bershad 92].

Clearly, neither a server-based nor a kernel-based strategy is ideal because each demands a tradeoff be-

tween efficiency and flexibility. The remaining strategy, and the one described in this paper, is to implement network protocols as a library linked into the address space of each application. This approach can retain the performance advantages of an in-kernel implementation and the flexibility advantages of a user-level implementation. Good performance is achieved because the number of boundary crossings on the send and receive paths is the same as the in-kernel case. Flexibility is achieved because the application, not the operating system, can define the behavior of the network protocol.

The difficulty with application level protocols

The key challenge with application-level protocols, which control the format of data on the wire, is their integration with the rest of the operating system, which provides abstractions to manage process state, I/O channels, and other machine resources. We address this challenge by identifying and providing a set of key interfaces between the application, the operating system, and the network. The protocol library provides for services such as rapid data movement between hosts. This interface must be efficient, but it does not need to be particularly complex. The operating system provides for important abstractions such as the network as a first-class I/O channel. This interface must be complete in that it supports all operations that may be applied to a network connection layered beneath a file abstraction, but it does not need to be particularly efficient.

2.1. Other goals

In addition to flexibility and good performance, we have the following goals in our design:

- *Reuse of existing protocol code.* Our interfaces allow the use of existing network protocol code as protocol libraries. This allows us to leverage the protocol construction work of others [Jacobson 88, Hutchinson & Peterson 91], and to more easily compare the performance of a given protocol implementation running in the kernel, in a protocol server, and in an application.
- *Source-level compatibility with existing protocol clients.* We are willing to recompile or relink existing protocol clients against our new implementation, but we (as we expect most others) are unwilling to modify these clients. Consequently, our operating system and protocol interface is syntactically and semantically compatible with existing interfaces.
- *Security.* A protocol implementation must not degrade the security of the network. Our design offers the same level of network security as is found in the protocol implementation that it supplants.
- *A portable architecture.* We intend to use our protocol architecture on uniprocessors, shared memory multiprocessors, and multicomputers in which processors share a high-speed dedicated mesh. The application protocol library is structured as a component of a distributed system in which protocol state is maintained by both the operating system server and the application.

While this work has been performed in the context of a specific microkernel-based operating system, it is neither specific to it, nor to microkernel-based operating systems in general.

3. Design overview

In this section we present the design of our protocol architecture in the context of a reliable, byte-stream protocol (TCP) and an unreliable datagram protocol (UDP) accessed through the BSD UNIX socket interface. We first describe the responsibility and relationships of each major component. Next, we discuss the system's behavior during the establishment of connections and the transfer of data. Throughout, we

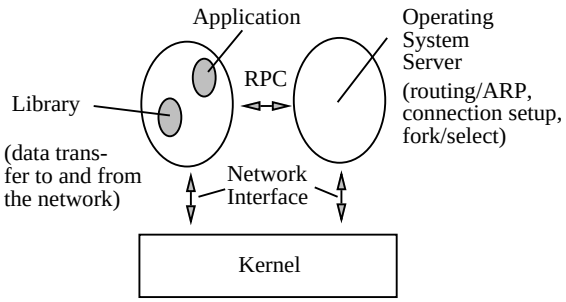

Figure 1: *In our protocol architecture, critical-path functionality is implemented by libraries in the application's address space. The operating system server manages shared protocol databases, handles connection set up, and implements high-level abstractions. The kernel exports a packet send and receive interface.*

highlight techniques that we use to handle many of the complex cases that arise during application-level protocol management.

3.1. The major components

Our application-level protocol architecture includes three software components as shown in Figure 1.

1. The *operating system server* is responsible for network operations that have non-critical performance requirements, such as connection establishment, teardown, and the handling of exceptional network packets like ARP queries. In addition, the operating system maintains long-lived, and shared, protocol state such as routing information and TCP port namespaces. The operating system also provides applications with network service in cases where application-level networking becomes difficult due to a conflict between a library-based protocol and operating system semantics. Finally, the operating system provides an interface that allows the application to integrate its own protocol management with the operating system's file abstraction.

2. A multithreaded *library* within each application implements a protocol stack, in particular the send and receive components.

3. The *network interface* provides a thin software layer on top of the raw network hardware. It is used to both send and receive packets. As its performance limits that of applications, it is required to have low latency and high bandwidth.

The operating system server and protocol libraries cooperate to manage network sessions. A network session is specified by a 3-tuple consisting of a protocol, a local endpoint, and a remote endpoint. The state of a network session consists of a set of protocol-specific state variables. For example, a TCP session has state variables for the send and receive window sizes, the send and receive sequence numbers, and any unacknowledged or undelivered data on the send and receive queues [Postel 81]. UDP, as a connectionless and stateless protocol, has no session state variables.

Sessions are created and terminated by the operating system. Once established, a network session is migrated into the address space of the application for which it was created. The application then manages the session until it executes an exceptional operation that makes application management difficult, or until the session is terminated. For example, the BSD UNIX `fork` system call makes a copy (the "child") of the process doing the fork (the "parent"), and the file descriptors in both the parent and child must refer to the same I/O stream. These semantics are difficult to emulate if the session is maintained in either

the parent's or child's address space, instead of the operating system's.[1] In such cases, the active protocol session migrates back from the application to the operating system, and all subsequent network operations are routed through the server.

The protocol library relies on the kernel's network interface to send and receive packets. Applications send packets directly to the network interface using a low-latency system call. For security reasons, packets are received through the *packet filter* [Mogul et al. 87, McCanne & Jacobson 93, Yuhara et al. 94]. The operating system creates and installs a new packet filter for each network session.

3.2. Emulating an existing interface

We emulate the BSD UNIX socket interface through a simple proxy structure that distributes protocol state across the operating system and the application's address space. A proxy is a small body of code that resides in the application's address space. It exports a procedure call interface that is identical to the socket system call interface otherwise exported by the operating system. An application's system call involving sockets is first routed through its proxy where the call is either handled locally, forwarded untouched to the operating system server, or translated into an alternate sequence of calls on the operating system server. Table 1 lists the calls implemented by the proxy module in the library, and the corresponding calls exported by the operating system server to assist the proxy in its implementation.

Proxy exports	Server exports	Action
`socket`	`proxy_socket`	Create a network session that is managed by the operating system.
`bind`	`proxy_bind`	Set local address of session. UDP sessions migrate to the application.
`connect`	`proxy_connect`	Set remote address of session. UDP and TCP sessions migrate to the application.
`listen`	`proxy_listen`	Open session passively. The operating system awaits new connections.
`accept`	`proxy_accept`	Migrate passively opened session from the operating system to the application when connection is established.
all send and receive variants	N/A	Transfer data to or from the network.The operating system is not involved.
`fork`	`proxy_return`	Return session to operating system server. All sessions should be returned to the operating system before `fork` is called.
`select`	`proxy_status`	Notify operating system of change in proxy session state.

Table 1: *The proxy exports the standard socket interface, which it implements through a combination of indirect calls onto the operating system, and direct calls onto the network. The operating system manages session establishment and teardown, while the operating system handles session data transfer.*

Creating network sessions

The `socket` call creates a file descriptor to represent a network session. In the case of a connection-oriented protocol such as TCP, all three components of a session's 3-tuple (protocol, local endpoint, remote endpoint) must be specified before the session is established. In the case of a connectionless protocol such as UDP, only

[1] If two address spaces were to comanage directly a network connection, then they could each corrupt one another's protocol state, violating the separation and protection semantics normally associated with address space boundaries.

the protocol and local endpoint are needed to establish the session, as the remote endpoint is supplied with each outgoing and incoming packet.[2] For the `socket` call, the library performs a `proxy_socket` call to the operating system which returns a new file descriptor. Both the library and the operating system associate the new descriptor with a data structure that represents an unconnected session for the specific protocol.

The `bind` system call specifies the local endpoint of a network session and is applied to a socket descriptor. The library maps the call into a `proxy_bind` call, causing the operating system to associate the socket with the specified address. Once the protocol and local endpoint have been specified for a UDP session with a `proxy_bind` call, the session may be used for sending and receiving packets. Consequently, the operating system returns the (null) network session state along with a local endpoint and a packet filter port. The library binds its local socket to the endpoint returned by the operating system and awaits incoming packets on the packet filter port. For TCP, only the local endpoint is returned when a proxy socket is bound because the remote endpoint is not yet known.

Establishing connections

Connection establishment is managed entirely by the operating system. There are four reasons for this. First, as with UDP, the creation of a new endpoint requires that the operating system construct and install a new packet filter to receive data on that endpoint, so there is necessarily at least one interaction between the operating system and the application. Second, it is necessary to interact with a local IP port manager to ensure that the endpoint is uniquely named; the operating system is a convenient place to implement this manager. Third, the operating system must track all connected sessions so that they are cleanly terminated in the event that the address space holding the local endpoint itself terminates. Finally, unlike send and receive, connection establishment is not a performance-critical operation. The additional IPC overhead required to contact the operating system server is negligible compared to the latency of a multi-phase network handshake.

Sessions in connection-oriented protocols such as TCP may be opened *actively* or *passively*. In a passive open, the local protocol waits for connection requests on a local endpoint. In an active open, the local protocol contacts a remote protocol and requests that a connection be established on a specified pair of endpoints. Using the socket interface, connections are passively opened with the `listen` and `accept` system calls, and actively opened with the `connect` call.

On the passive side, the protocol library maps the `listen` system call into a `proxy_listen` call to the operating system. The operating system is primed for incoming connection requests addressed to the passively opened connection. When contacted by a remote peer, it negotiates the establishment of the connection.

Once the connection is established, the operating system places it on a queue of passively opened sockets until the local listener performs an `accept` operation. The library implements the `accept` system call with a `proxy_accept` call to the operating system, which returns a new file descriptor to represent the passively opened connection. The call also returns a local endpoint, a remote endpoint, the connection state variables, and a packet filter port. The protocol library records the existence of the new connection, sets the initial connection state to that returned by the operating system, and begins reading packets from the new packet filter.

On the active side, the library transforms `connect` calls into `proxy_connect` calls to the operating system. A `proxy_connect` causes the operating system to initiate connection establishment, and to then return the local and remote endpoints of the session, the protocol session state, and a packet filter port. As with a passive open, the library then waits for packets to arrive on the packet filter port. If the socket is unbound (i.e. has no local endpoint) at the time of the `connect` call, the socket is also given a local endpoint.

[2] The BSD UDP (and our) implementation also permits "connection-oriented" sessions where the remote endpoint is implicit to the session.

Sending and receiving data

The BSD socket interface has ten different ways to move data through a session (`recv`, `recvfrom`, `recvmsg`, `read`, `readv`, and `send`, `sendto`, `sendmsg`, `write`, and `writev`). For sockets, these calls are implemented entirely within the application's protocol library. Once a network session has been established, data can be sent and received over the network without operating system intervention.

Outgoing unreliable data (UDP) is sent immediately and then discarded. Data is sent on a reliable connection (TCP) by placing it on the socket send queue and calling the protocol's network output routine which may or may not send a segment immediately, depending on the current state of the connection. Reliable protocols keep the data on the send queue until it has been acknowledged by the remote endpoint.

The receive operations block until data is available on the socket receive queue, and then copy the data out to a user buffer specified in the receive call. The socket interface, which has the receiver specify the destination address of an incoming message, incurs an unnecessary copy at this point. A better integration between the application and the protocol stack, described in Section 4.2, avoids this copy.

Terminating session state

Some protocols have sophisticated tear-down requirements. For example, properly closing a TCP connection requires a four-way handshake (a two-way handshake in each direction) followed by a waiting period to ensure that any segments delayed in the network have time to die [Postel 81]. For a clean shutdown, which occurs when the application explicitly requests a close on session, we migrate the session state back to the operating system and follow the shutdown protocol there. For an unexpected shutdown, for example, when a process terminates in error, the connection can be left hanging in an undefined state. The operating system, though, can detect the death of processes that are managing network connections, abort outstanding connections by sending reset messages to remote peers, and delay the reopening of any aborted connections.

Cooperative interfaces

Some operations on network sessions interact only with the operating system's scheduling and process management interfaces, but do not move data. For example, the `select` call is used by applications to check the status of a set of file descriptors. Because these descriptors may not all be managed by the application (some may be actual files, for example) it is not possible to implement `select` entirely within the application. Similarly, because some of the descriptors may be managed by the application, the call cannot be implemented entirely within the operating system; the operating system has no direct way of knowing when these sessions change status.

We bridge this "information gap" through a cooperative interface that is jointly implemented by the application and the operating system. The library implements its side of `select` by examining the argument file descriptor sets to determine which of the sockets managed by the application are ready. For each of these sockets, the library records that the socket is being `select`'ed upon, and notifies the operating system of the socket status. The library then calls through to the operating system's `select` system call. When the application discovers data on one of the selected sockets, it signals the operating system of a status change (`proxy_status`), forcing any relevant outstanding `select`s to return. In cases where all descriptors are managed by the application, the operating system is not involved.

3.3. Caching protocol metastate

A good deal of a protocol implementation is responsible for managing state that is independent of any particular session. We maintain this state in the operating system server to preserve its long-livedness, and

to protect it from damage by applications. For example, route table entries and ARP mappings represent long-term state that is used by all sessions, but owned by none. When sending data, application protocol code must read this state when constructing outgoing packets. In the same way that the operating system caches these entries from network queries, applications cache them to avoid communication with the operating system on the packet send path. The operating system maintains *callbacks* into applications for these cached entries and invalidates them as they expire or are updated.

3.4. Security considerations

The kernel's packet filter ensures that an application can only receive packets that are destined for it. We, however, do not prevent applications from sending arbitrary data packets over the network. We expect that a packet limiting mechanism, if desired, could be implemented by checking each outgoing packet using a service similar to the packet filter [Thekkath et al. 93]. Because network security is already quite fragile in the presence of physically vulnerable connections [Garfinkel & Spafford 91], though, the basic problem of inter-machine security is better addressed through the use of authentication mechanisms and encryption [Voydock & Kent 83].

Application-level protocols can be used with session-level encryption software, provided that session keys are confined to the application's address space. A small amount of additional operating system support is required to ensure that session keys are cleared before a process' image is stored to disk (for example, as a result of a core dump). Host-to-host, or metasession, encryption, will require an additional level of packet addressing indirection on top of, and encryption below, the network send and receive interface. Specifically, a process would send packets to a logical secure host, rather than an IP (or lower-level) address. The kernel's network interface would be responsible for encrypting the packet, and routing it to the corresponding physical host. Presently, we have no experience with a secure implementation of our protocols, though, so cannot comment on their use.

4. Performance

In this section we discuss the performance of our application-level protocol architecture, which we have implemented on top of the Mach 3.0 microkernel. We first describe a number of microbenchmarks that reveal the throughput and latency of our implementation in the context of several different user/kernel network interfaces. We then demonstrate the benefit of a flexible user-level implementation by changing the socket interface to eliminate data copies between the application and the protocol stack. Finally, we present a detailed latency breakdown for TCP and UDP processing.

Platforms

We have run our experiments across 10 Mb/s Ethernet using DECstation 5000/200 workstations and Gateway personal computers. The DECstation uses a 25Mhz R3000 MIPS processor [Kane 88] with a Lance Ethernet interface. The Gateway uses a 33 Mhz i486 processor [Intel 90] with a 3Com 3C503 Ethernet interface.

On the DECstations, we compare the performance of our protocol library with DEC's Ultrix 4.2A, the Mach 2.5 integrated kernel, and UX, CMU's single-server UNIX operating system. On the i486-based machines, we compare the performance of our protocol library with the Mach 2.5 kernel, the 386BSD kernel [Jolitz 92], the BNR2SS UNIX single-server [Dean 92], and CMU's UX.[3]. In the comparison systems, protocols are implemented in the server for the single-server based systems (UX and BNR2SS), and in

[3] BNR2SS and 386BSD are not available for the DECstation. Ultrix 4.2A is not available for the Gateway.

the kernel otherwise (Mach 2.5, Ultrix 4.2A, and 386BSD). Our protocol library, the 386BSD kernel, and BNR2SS all rely on protocol code derived from the Berkeley Networking Release Tape II (BNR2). Mach 2.5, Ultrix 4.2A, and UX use the 4.3BSD protocol implementation. Both implementations, though, are comparable and of high-quality as each is capable of nearly saturating a 10Mb/sec Ethernet between a pair of DECstation 5000/200s [Thekkath et al. 93].

We have compiled and run a large collection of network-intensive applications against our protocol library, including `telnet`, `ftp`, and the X11 libraries and clients [Gettys et al. 90]. For this discussion, though, we focus on two microbenchmark programs: `ttcp`, a memory-to-memory throughput benchmark for TCP that transfers 16 MB of data from one host to another, and `protolat`, a program that measures protocol round trip latency for UDP and TCP. The programs are measured on a private network while the machines are in single-user mode.

	TCP							**UDP**				
	Throughput		Latency (ms)					Latency (ms)				
		Receive Buffer Size	Message size (bytes)					Message size (bytes)				
	(KB/sec)	(KB)	1	100	512	1024	1460	1	100	512	1024	1472
DECstation 5000/200												
Mach 2.5 In-Kernel	1070	24	1.40	1.73	3.05	4.56	6.04	1.45	1.74	3.05	4.56	5.88
Ultrix 4.2A In-Kernel	996	16	1.52	1.89	3.50	4.78	6.13	1.52	1.81	3.29	4.69	6.05
Mach 3.0+UX Server	740	24	3.64	4.20	5.90	7.82	9.73	3.61	4.04	5.89	7.99	9.84
Mach 3.0+UX Library-IPC	910	24	1.69	2.09	3.43	5.09	6.63	1.40	1.77	3.08	4.71	6.14
Mach 3.0+UX Library-SHM	1076	120	1.82	2.29	3.66	5.32	6.73	1.34	1.68	2.95	4.59	5.95
Mach 3.0+UX Library-SHM-IPF	1088	120	1.72	2.11	3.44	5.09	6.56	1.23	1.57	2.83	4.41	5.79
Gateway 486												
Mach 2.5 In-Kernel	457	8	2.08	2.69	5.45	8.78	12.05	1.83	2.44	5.19	8.51	11.41
386BSD In-Kernel	320	8	2.71	3.64	6.24	NA	NA	2.63	3.49	6.04	9.54	12.50
Mach 3.0+UX Server	415	16	4.09	4.88	7.76	11.30	14.29	3.96	4.67	7.86	11.65	15.00
Mach 3.0+BNR2SS Server	382	12	3.88	4.70	8.00	NA	NA	4.64	5.37	8.95	13.23	16.84
Mach 3.0+UX Library-IPC	469	24	2.49	3.10	5.84	9.25	14.09	2.12	2.68	5.41	8.74	11.66
Mach 3.0+UX Library-SHM	503	24	2.38	3.07	5.79	9.15	12.58	2.02	2.59	5.30	8.64	11.62

Table 2: *This table shows TCP throughput and latency, and UDP latency for various system configurations and message sizes. Throughput for UDP is not given as this is tied to windowing and acknowledgement strategies as much as to latency. The entries labeled NA are because 386BSD and BNR2SS have a bug that prevents them from sending large TCP packets. The performance of the library-based implementations is comparable to the native in-kernel implementations. Although the i486 processor is comparable in performance to the R3000, the Gateway's low-performance Ethernet card (transfers are done 8 bits at a time) severely limits its throughput. Both the library and the server-based implementations on the Gateway have lower latency than the in-kernel version because of inefficiencies in the way that the 386BSD kernel handles network interrupts and scheduling. The library-based implementations labeled IPC, SHM, and SHM-IPF reflect runs using successively modified versions of the kernel's network packet filter interface.*

4.1. Throughput and latency

We have implemented several different versions of the user/kernel network interface. In our baseline version, the packet filter uses Mach IPC to deliver each incoming packet to the protocol in a separate message. The second version uses a modified packet filter that permits applications to receive multiple packets with a single wakeup from the kernel. The third version uses a modified packet filter that eliminates a copy on the critical receive path by integrating the packet filter with the underlying device driver.

Table 2 shows throughput and round trip latency for TCP, and round trip latency for UDP under

different protocol configurations and software network interfaces. Latencies for both protocols are shown for a range of packet sizes. We did not measure throughput for UDP, as it depends more on the windowing and acknowledgement strategies than on the datagram transport machinery. For each system, we ran the throughput benchmarks with the best possible receive buffer size for each implementation. We determined the best size by running the throughput benchmarks with increasing buffer size until further increases did not improve throughput. For the server and library-based protocols, the receive buffers are kept in virtual memory and can be reallocated on demand for busy sessions.

For the first library-based configuration *(Library-IPC)*, the network interface uses Mach's packet filter and IPC mechanisms to dispatch incoming network packets to the appropriate address space. Packet trains are not coalesced into contiguous messages, requiring that the protocol library collect and process an IPC message for every incoming packet. Because each IPC crosses the user/kernel boundary and is on the critical path of the receiver, we achieve only about 85% of the in-kernel throughput.

We have implemented an alternate packet filter mechanism (*Library-SHM*) that transfers data in memory shared between the kernel and the application. On receiving a packet, the packet filter transfers data into the shared buffer, and uses a lightweight condition variable to signal a protocol library that new data has arrived. The use of shared memory in this case does not reduce the number of packet copies, as an incoming packet is first copied from the Ethernet driver to an internal kernel buffer before it is run through the packet filter. Consequently, the change has little effect on single-packet latencies. The change is more effective for throughput, since the scheduling overhead of packet delivery is amortized over multiple packets. The shared memory interface delivers 1076 KB/sec on the DECstation configuration, which is an 18% improvement in throughput relative to the IPC-based implementation, and slightly better than the in-kernel implementation.

We can eliminate the extra copy into the kernel buffer by more closely integrating the device driver and the packet filter (*Library-SHM-IPF*). A packet filter program for Internet protocols typically only examines the packet header to determine the receiving endpoint. We can defer copying the rest of the packet until the final destination has been determined. By deferring, the packet filter can copy a packet's data directly from the device interface into the receiver's address space. The shared memory interface combined with the integrated packet filter delivers 1088 KB/sec on the DECstations, which is about 2% better than the in-kernel protocol. This modification has a more dramatic effect on latency since the number of data copies on the critical path is the same for the kernel-based and library-based protocol implementations. The integrated packet filter is device and machine-dependent, and we have not implemented it on the Gateway.

4.2. Changing the socket interface

A simple, but effective application-specific optimization can improve throughput and latency. As mentioned in Section 3.2, when an application program sends and receives data using the socket interface, it specifies a buffer from which outgoing or into which incoming data should be placed (copied). By changing the send and receive interface to allow the protocol and the application to share buffers, this copy can be eliminated.

Table 3 compares the TCP and UDP round trip latencies for a kernel-based implementation of the conventional socket interface with the library-based implementation using the modified interface. The modified interface outperforms the kernel-based implementation for large packets where copying costs become significant. For TCP throughput, the change is less effective since bandwidth is generally controlled by the speed with which the receiver can process and acknowledge segments. The copies eliminated by the interface change occur after the segment has been processed by TCP, and are not on the critical path for throughput. User-user throughput increases by 5% from 910 KB/sec to 959 KB/sec with the IPC-based packet filter interface. When used with the more efficient integrated packet filter, user-user throughput increases from 1088 KB/sec to 1099 KB/sec.

	TCP							UDP				
	Throughput	Receive Buffer Size	Latency (ms)					Latency (ms)				
			Message size (bytes)					Message size (bytes)				
	(KB/sec)	(KB)	1	100	512	1024	1460	1	100	512	1024	1472
DECstation 5000/200												
Mach 2.5 In-Kernel	1070	24	1.40	1.73	3.05	4.56	6.04	1.45	1.74	3.05	4.56	5.88
Ultrix 4.2A In-Kernel	996	16	1.52	1.89	3.50	4.78	6.13	1.52	1.81	3.29	4.69	6.05
Mach 3.0+UX Library-NEWAPI-IPC	959	24	1.67	2.02	3.35	4.96	6.45	1.42	1.75	3.05	4.69	6.09
Mach 3.0+UX Library-NEWAPI-SHM	1083	120	1.70	2.07	3.33	4.94	6.38	1.34	1.66	2.93	4.54	5.95
Mach 3.0+UX Library-NEWAPI-SHM-IPF	1099	120	1.63	1.98	3.24	4.80	6.26	1.25	1.57	2.83	4.38	5.76

Table 3: *This table shows the effect that a modified socket interface has on throughput and latency. The library uses a new application programming interface (NEWAPI) that eliminates a redundant copy between the protocol stack and the application.*

4.3. Latency breakdown

On the DECstation 5000/200s, we have determined the time spent in the various protocol layers using a high-resolution timer. Table 4 compares the average time spent in each layer of the TCP and UDP protocol stacks for our library (*SHM-IPF*), the Mach 2.5 kernel, and CMU's UNIX server. Each column corresponds to a single trial of 50000 round trips run in single-user mode on a private network. Since TCP sends extra acknowledgement segments in addition to the data segments, the numbers for TCP only approximate the critical path latency.

Send path

The first four lines define the send path. The first line, `Entry/copyin`, is the time required to enter the socket layer code and convert the send buffer into a linked list of `mbuf` data structures (the internal unit of memory allocation for the protocols). Entry is a procedure call for the library-based protocol, a trap for the kernel-based protocol, and a trap followed by an RPC for the server-based protocol. For the library with TCP, and the kernel for either TCP or UDP, the send buffer must be copied into an `mbuf`. For the library-based UDP implementation, the user data can be referenced instead of copied. For the server with both protocols, copyin requires sending an IPC message to the operating system server which then executes the socket layer code that constructs the `mbuf` chain. This component is large because the data is copied four times as part of an RPC: from the user buffer to the IPC message, from the IPC message into the kernel, from the kernel into an IPC message buffer in the protocol server's address space, and again from the IPC message buffer to the `mbuf` chain.

The remaining three components on the send path are for the actual protocol stack. The top layer constructs the protocol header and checksum (header and data). The IP layer constructs the IP header and determines the route to the destination. The Ethernet layer maps the destination IP address to an Ethernet address, constructs the Ethernet header, and transmits the packet over the network.

The UDP and Ethernet layers have different latencies in each implementation. The `ether_output` component is larger in the library-based and server-based implementations because the protocol code traps into the kernel and copies the packet from user space into a wired kernel buffer before copying it to device memory. In contrast, the in-kernel version copies outgoing data directly from the `mbuf` chain (which is already wired) to the device.

The `tcp_output` and `udp_output` components are faster in the library than in the server. This discrepancy

Layer	TCP						UDP					
	Library		Kernel		Server		Library		Kernel		Server	
	1	1460	1	1460	1	1460	1	1472	1	1472	1	1472
Send Path												
entry/copyin	19	203	*50	*153	*254	*579	6	7	*65	*104	*293	*628
tcp,udp_output	82	328	65	307	224	447	18	239	70	273	229	398
ip_output	26	26	24	20	31	25	17	18	22	25	24	27
ether_output	*98	*274	75	105	*166	*331	*105	*280	74	163	*188	*367
Send Path Total	225	831	214	585	675	1382	146	544	231	565	734	1420
Receive Path												
device intr/read	42	43	77	469	101	496	39	40	74	481	99	497
netisr/packet filter	82	95	79	73	53	52	58	70	83	84	76	61
kernel copyout	*123	*534	0	0	*113	*148	*107	*517	0	0	*124	*207
mbuf/queue	22	21	0	0	79	58	20	20	0	0	68	64
ipintr	37	35	30	37	127	95	35	33	30	54	121	91
tcp,udp_input	214	445	76	270	249	365	103	318	67	279	61	273
wakeup user thread	92	95	54	54	194	213	73	80	70	69	262	274
copyout/exit	46	261	*32	*220	*222	*1028	21	63	*27	*75	*208	*619
Receive Path Total	658	1529	348	1123	1138	2455	456	1141	351	1042	1019	2086
Network Transit Time	51	1214	51	1214	51	1214	51	1214	51	1214	51	1214
Total	934	3574	613	2922	1864	5051	653	2899	633	2821	1804	4720

Table 4: *For a library-based (*SHM-IPF*), kernel-based (Mach 2.5), and server-based (UX) protocol implementation on the DECstation 5000/200, this table shows the average TCP and UDP latencies on Ethernet by component for the sender and receiver. The minimum (1 byte) and maximum (1460 bytes for TCP, 1472 bytes for UDP) unfragmented message sizes were used. Times are in microseconds. Entries marked with asterisks denote protection boundary crossings. Times reported in this table are from an instrumented version of the protocols, and reflect a small percentage error relative to an uninstrumented version.*

is due to the different synchronization primitives used in each implementation. The server's synchronization mechanisms are based on scheduling priority levels and locks. The priority levels, which are artifacts of the code's kernel origins, are retained in the server because protocol processing must be synchronized with other services, such as process management and filing, that also rely on priority levels. The priority level machinery simulates hardware interrupt priorities using locks and condition variables, resulting in expensive priority manipulation, and high contention for specific priority levels among independent services. In contrast, our protocol library does not synchronize with other operating system services, and internally synchronizes using less expensive locks. [4]

Receive path

The next eight lines describe the receive path. The `device intr/read` component is the time to field an interrupt from the network device. For the kernel and the server, the entire packet is also copied out of the device into a wired kernel buffer. The `netisr/packet filter` component reflects the time to demultiplex the packet to the appropriate protocol stack. The `kernel copyout` component measures the time required to deliver the packet to the destination protocol stack. This component does not apply to Mach 2.5 and is shown as zero. For the library, the packet is copied from the network device into the protocol stack's address space. For the server, the copy is from kernel memory, which has lower read latency than network device memory [DEC 90].

The remaining components execute in user space for both the server-based and library-based implemen-

[4] The server's synchronization mechanisms have been replaced with lighter-weight versions in later releases of CMU's UNIX server.

tations. The `mbuf/queue` component measures the time required to package the incoming packet as an `mbuf` chain and to queue the chain on the protocol stack's input queue. For Mach 2.5, this work occurs as part of the `netisr/packet filter` processing. Although both are implemented at user-level, the overhead for `mbuf/queue` manipulation is higher than in the library. Again, as with `udp_output` and `tcp_output`, this is because the server uses a heavyweight synchronization mechanism.

The IP layer (`ipintr`) dequeues incoming IP packets, processes the IP header, and passes each packet up to TCP or UDP. These layers then checksum the protocol header and data, queue the data on the destination socket, and awaken any thread waiting for data to arrive on the socket.

The `wakeup user thread` component is the time required to pass control from the network protocol thread to an application thread awaiting data. Again, synchronization overheads account for the difference in times between the server and library.

Finally, the `copyout/exit` component reflects the time required to copy data from the *mbuf* chain into the destination buffer specified by the caller, and leave the protocol. TCP's receive queue management, and its support for urgent data [Postel 81] make this component larger than for UDP. For the server-based implementation, this component involves sending an IPC reply message to the application and includes the same number of redundant copies as the *entry/copyin* component.

5. Related work

Although there has been substantial work in the area of improving protocol performance, and in moving pieces of protocol processing into user space, we are aware of no previous work that has attempted fully to integrate application-level processing with other operating system services. An experimental protocol library built on top of Mach 3.0 at the University of Washington [Thekkath et al. 93] implements a subset of the socket interface for TCP, and provides for protected transmission between hosts that are on the same physical Ethernet. As their system is intended to address the needs of application-specific protocols, they are not faithful to the operating system interface.

The *x*-Kernel is an object-oriented protocol implementation environment that facilitates the construction of new protocols. The *x*-Kernel currently runs as a dedicated protocol server and provides an RPC stub library that implements the socket interface. Our system is complementary to the *x*-Kernel's, as ours facilitates the integration of protocols into a complete operating system environment. Tschudin [Tschudin 91] advocates a protocol server into which protocol implementations can be dynamically loaded and unloaded. We are not aware of an implementation of these ideas. Clark and Tennenhouse [Clark & Tennenhouse 90] assert that protocol layering is a desirable design but undesirable implementation technique. They advocate two new techniques to improve performance: application-level framing, where higher-level protocols determine the granularity of lower-level protocol processing, and integrated layer processing, where the processing for all protocol layers is performed in one pass over the data. Our protocol decomposition strategy facilitates the application of these techniques.

6. Conclusions

It is possible to achieve both good performance and high flexibility in the networking domain. Careful protocol decomposition places the responsibility for defining network abstractions with the operating system, and of implementing the performance-critical components of those abstractions with the application. Our work can be interpreted as part of a "RISC movement" in operating systems [Wilkes 92] where programming interfaces are decoupled from the operating system implementation. This movement will make it possible to experiment with newer and better programming implementations and interfaces while at the same time retaining support for existing ones.

Acknowledgments

David Eckhardt, David Keppel, Gregor Kiczales, John Lamping, Ed Lazowska, Sue Lee, Keith Marzullo, Dylan McNamee, Gail Murphy, Larry Peterson, and Chandu Thekkath provided valuable feedback on earlier drafts of this paper. Jose Brustoloni helped us implement shared buffers correctly in the context of volatile protocol sessions. Wayne Sawdon and Matt Zekauskas were early users of the system and suffered through our mistakes with us. Masanobu Yuhara assisted with the integration of the packet filter. Mary Thompson and Alessandro Forin helped with the integration of our libraries into Mach 3.0.

References

[Accetta et al. 86] Accetta, M. J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid, R. F., Tevanian, Jr., A., and Young, M. W. Mach: A New Kernel Foundation for Unix Development. In *Proceedings of the 1986 Summer USENIX Conference*, pages 93–113, July 1986.

[Clark & Tennenhouse 90] Clark, D. D. and Tennenhouse, D. L. Architectural Considerations for a New Generation of Protocols. In *Proceedings of the SIGCOMM '90 Symposium*, pages 200–208, September 1990.

[Clark et al. 91] Clark, D., Chapin, L., Cerf, V., Braden, R., and Hobby, R. Towards the Future Internet Architecture. Request for Comments 1287, December 1991.

[Clark et al. 92] Clark, D. D., Shenker, S., and Zhang, L. Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism. In *SIGCOMM '92 Conference Proceedings*, pages 14–26, August 1992.

[Dean 92] Dean, R. W. A License-Free BSD 4.4 Single Server. In *Open Software Foundation Symposium '92*, Cambridge, MA, February 1992.

[DEC 90] DEC Workstation System Engineering. *DECstation 5000/200 KN02 System Module Functional Specification (Revision 1.3)*, August 1990.

[Felten 92] Felten, E. The Case for Application-Specific Communication Protocols. In *Proceedings of Intel Supercomputer Systems Division Technology Focus Conference*, pages 171–181, 1992.

[Forin et al. 91] Forin, A., Golub, D. B., and Bershad, B. N. An I/O System for Mach 3.0. In *Proceedings of the Second Usenix Mach Workshop*, pages 163–176, November 1991.

[Garfinkel & Spafford 91] Garfinkel, S. and Spafford, G. *Practical Unix Security*. O'Reilly and Associates, Inc., Sebastopol, CA, 1991.

[Gettys et al. 90] Gettys, J., Karlton, P., and McGregor, S. The X Window System, version 11. *Software – Practice and Experience*, 20(S2):35–67, October 1990.

[Golub et al. 90] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an Application Program. In *Proceedings of the 1990 Summer USENIX Conference*, pages 87–95, June 1990.

[Hutchinson & Peterson 91] Hutchinson, N. C. and Peterson, L. L. The *x*-kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[Intel 90] Intel. *386 Programmer's Reference Manual*. Intel, Mt. Prospect, IL, 1990.

[Jacobson 88] Jacobson, V. Congestion Avoidance and Control. In *Proceedings of the SIGCOMM '88 Symposium on Communications Architectures and Protocols*, pages 314–329. ACM, August 1988.

[Jacobson et al. 92] Jacobson, V., Braden, R., and Borman, D. TCP Extensions for High-Performance. Request for Comments 1323, May 1992.

[Jolitz 92] Jolitz, W. F. Porting UNIX to the 386. *Dr. Dobbs' Journal*, January 1991 through July 1992.

[Kane 88] Kane, G. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Leffler et al. 89] Leffler, S. J., McKusick, M., Karels, M., and Quarterman, J. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[Maeda & Bershad 92] Maeda, C. and Bershad, B. N. Networking Performance for Microkernels. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 154–159, April 1992.

[McCanne & Jacobson 93] McCanne, S. and Jacobson, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the 1993 Winter USENIX Conference*, pages 259–269, January 1993.

[Mogul et al. 87] Mogul, J. C., Rashid, R. F., and Accetta, M. J. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 39–51. ACM, November 1987.

[Postel 81] Postel, J. Transmission Control Protocol. Request for Comments 793, USC Information Sciences Institute, September 1981.

[Rashid & Robertson 81] Rashid, R. F. and Robertson, G. G. Accent: A Communication Oriented Network Operating System Kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 64–75, December 1981.

[Stevens 90] Stevens, R. *Unix Network Programming*. Prentice-Hall, 1990.

[Thekkath et al. 93] Thekkath, C. A., Nguyen, T. D., Moy, E., and Lazowska, E. D. Implementing Network Protocols at User Level. In *Proceedings of SIGCOMM '93*, September 1993.

[Tschudin 91] Tschudin, C. Flexible Protocol Stacks. In *Proceedings of the SIGCOMM '91 Symposium*, pages 197–204, September 1991.

[Voydock & Kent 83] Voydock, V. L. and Kent, S. T. Security Mechanisms in High-Level Network Protocols. *ACM Computing Surveys*, 15(2):135–171, June 1983.

[Wilkes 92] Wilkes, M. The Case for a New Approach to Operating Systems for Personal Workstations. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 164–167, April 1992.

[Yuhara et al. 94] Yuhara, M., Bershad, B. N., Maeda, C., and Moss, J. E. B. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proceedings of the 1994 Winter USENIX Conference*, January 1994.