Networking Performance for Microkernels

Chris Maeda
Brian N. Bershad

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

March 17, 1992

Abstract

Performance measurements of network protocols in microkernel systems have been discouraging; typically 2 to 5 times slower than comparable macrokernel systems. This disparity has led many to conclude that the microkernel approach, where protocols reside at user-level, is inherently flawed and that protocols must be placed in the kernel to get reasonable performance. We show that user-level network protocols have performed poorly because they rely on code designed to run in a kernel environment. As a result, they make assumptions about the costs of primitive protocol operations such as scheduling, preemption, and data transfer which can require substantial overhead to satisfy at user level. Good user-level protocol performance can be achieved by restructuring protocol servers to take advantage of microkernel facilities, rather than ignore them.

1 Introduction

Microkernel operating systems, such as Mach 3.0 [Accetta et al. 86], provide support for scheduling, virtual memory, and cross-address space IPC. Higher level services, such as Unix emulation and network communication, are implemented in user-level servers. In contrast, "macrokernel" systems, such as Sprite [Ousterhout et al. 88] and Berkeley's 4.3BSD [Leffler et al. 89], provide complete operating system functionality within the kernel.

Network protocols can run in kernel space or user space. Macrokernel systems put protocols in the kernel for performance reasons; input processing can be done at interrupt level, which reduces latency, and packets need only be transferred from the kernel to the receiver's address space, which reduces data copying. In contrast, microkernel systems are more likely to put the protocols in user space for flexibility; user-level code is easier to debug and modify. Moreover, microkernels generally relegate the protocols to a specific user address space and require protocol clients to indirect through that address space.

Critics of microkernel operating systems argue that important system services, such as network communication, have higher latency or lower throughput than in monolithic kernels. For example, in UX, CMU's single-task Unix server for Mach 3.0 [Golub et al. 90], the round trip latency for UDP datagrams is about three times greater than for the Mach 2.5 system, where all Unix functionality is in the kernel.

In this paper we show that the *structure* of a protocol implementation and not the *location* is the primary determinant of performance. We support our position by measuring the performance of a simple protocol, UDP, in several different versions of the Mach operating system. Although UDP performance for Mach's single server Unix system is poor, the performance problems lie in the Unix server, not the microkernel. The goal of the Unix server implementation was to get the code working quickly and then, where necessary, to get quickly working code. This approach optimizes programmer time, since it is possible to build a Unix server by taking an in-kernel version of Unix and applying "wrapper" code so it runs at user level.

To justify our argument, we show that a carefully implemented user-level protocol server can perform as well as an in-kernel version. Our "proof" consists of a reimplementation of UDP which does not assume that it is running in the "Unix kernel" but is instead optimized for the microkernel environment of Mach 3.0.

2 Network Performance Measurements

Based on the performance of network protocols in Mach 3.0, it would be easy to conclude that putting protocols at user-level is a bad idea. Table 1 shows the average round trip time for small UDP packets (1 data byte plus headers) using an in-kernel and out-of-kernel protocol implementation. All times were collected by sending 1000 packets back and forth between two user-level processes running on DECstation 2100's connected by a 10Mb/s ethernet. The data in the tables represent the mean values of multiple 1000 packet trials. The variance in all cases was between 5 and 10 percent of the mean. The DECstation 2100 uses a MIPS R2000 processor running at 12Mhz. Our machines each had 16MB of RAM.

In Mach 2.5, UDP is in the kernel. In Mach 3.0, UDP is in the Unix server but uses nearly the same code as in Mach 2.5. The Unix server described in Table 1 uses Mach IPC to send outgoing packets to an in-kernel ethernet device driver. Incoming packets are read by the device driver and then demultiplexed to user tasks (normally just the Unix server) by the *packet filter* [Mogul et al. 87]. Once they are demultiplexed, these incoming packets are forwarded via Mach IPC to a Unix server thread which does subsequent protocol processing.

The advantage of the IPC interface is that it allows network-transparent device access; clients of a device may be located on nodes to which the device is not physically attached (as might be the case on a multicomputer like the Touchstone). The disadvantage of the IPC interface is that outgoing data must be copied into a message buffer (or manipulated through VM operations), and that there is additional IPC latency between the protocol server and the device. The Unix server sends each outbound packet to the ethernet device driver as an IPC message with "out-of-band" data which allows the IPC system to map the data into the receiver's (here, the kernel's) address space rather than copy it. In Mach, however, it can be more efficient to copy, rather than remap, small amounts of data between address spaces.

UDP Implementation	Time (ms)
Mach 2.5 kernel	4.8
Unix server (IPC,VM)	19.5

Table 1: UDP Round Trip Times for Various Systems. This is what you would observe if you ran the Mach 2.5 macrokernel (first line), and Mach 3.0 version MK65 with the Unix single server version UX29 (second line). The Unix server uses Mach IPC and VM operations to communicate with an in-kernel device driver.

2.1 Some Easy Improvements

We can reduce the latency of a round trip message by eliminating the VM and IPC operations on the outgoing path. VM overhead can be eliminated by sending the data "in-band" in an IPC message to the device driver. IPC overhead can be eliminated by using a system call to trap directly into the kernel device driver instead of using Mach IPC.[1] The first and second rows of Table 2 show the improvement in latency which comes by first eliminating the VM operations, and then the VM operations in conjunction with the IPC.

A more aggressive approach is to *map* the device directly into the protocol server's address space. This technique, described in [Forin et al. 91], allows the protocol server to also act as the network device driver, controlling the network interface directly, and avoiding the copy and the

[1] A user-to-kernel RPC for Mach 3.0 on the DECstation 2100 takes between 100 and 300 μsecs, depending on cache effects [Bershad et al. 91], whereas a system call takes only about 30 μsecs. Little effort has been spent trying to reduce the latency of user-to-kernel RPC, whereas significant effort has gone towards reducing user-to-user RPC latency [Draves et al. 91].

UDP Implementation	Time (ms)
Unix server (IPC, no VM)	14.3
Unix server (syscall, no IPC, no VM)	13.6
Unix server (mapped driver)	13.1

Table 2: Second Cut UDP Round Trip Times for Various Systems. This is what you would observe if you ran Mach 3.0 with an experimental version of UX29 (first row), or if you extend the kernel system call interface to include a direct `device_write` system call (second row), or if you enable mapped ethernet support in a standard system (third row).

IPC to the in-kernel device driver. The round trip UDP latency for this approach is shown in the third row of Table 2. The 6 ms improvement is because a user-to-kernel IPC and several VM operations have been avoided.

2.2 What Do The Numbers Mean?

Although the times in Table 2 are better than the Mach 3.0 times in Table 1, they are still worse than Mach 2.5. It is tempting to conclude that the network performance in Mach 3.0 is slower because the protocols are out of kernel. Since the actual protocol code which executes in the two systems is the same, the slowdown must be caused by the extra interprocess communication costs associated with the microkernel architecture. However, a few observations about the cost of crossing address space boundaries between the Unix server and the kernel, and between the Unix server and the protocol clients show that this conclusion can't be right.

On a DECstation 2100, it takes about 490 μsecs to access the Unix server's socket layer from a user program. This path includes one pass through the system call emulation library (about 280 μsecs), which translates system calls into IPC messages, and one IPC round trip between the user task and the Unix server (about 180 μsecs). The IPC time can be further broken down into 140 μsecs for the actual RPC and 40 μsecs for the stub code. In contrast, it takes about 60 μsecs to access the in-kernel socket layer using a system call in Mach 2.5.

Each UDP round trip includes four socket accesses (a read and write on each host). If we assume the worst case where there is no overlap of reads and writes on the two hosts, having the sockets in the Unix server should add no more than 2 ms (4 * (490 - 60) μsecs) to the total round trip time.

It takes about 200 μsecs to write a packet to the ethernet device driver using a kernel syscall. Receiving a message from the packet filter with Mach IPC takes about 300 μsecs. (UX29 with the mapped ethernet driver does not currently support the packet filter.) Thus, crossing the address-space boundary between the device driver in the kernel and the protocol stack in the Unix server should add about 1 ms (2 * 500 μsecs) to the total round trip time.

Assuming the protocol stack runs as fast in the Unix server as it does in the kernel, a UDP round trip in Mach 3.0 should then be about 3 ms slower than in Mach 2.5 (2 ms to talk to the client and 1 ms to talk to the device driver). The indirection through the Unix server should thus have a relatively small effect on network performance.

2.3 Where Does The Time Go?

Little attention has been paid to the implementation of the protocols themselves within the Unix server, so even small measures can still go a long way. For example, we were able to gain an easy 30% performance improvement in network round trip times simply by changing the way in which the lowest level of the Unix server interacts with the device driver. In the rest of this section we describe two other problems with the Unix server implementation.

The wrong kind of synchronization

The Mach 2.5 and BSD kernels use processor priorities (`splx`) to synchronize internally. The `spl` machinery was designed for a uniprocessor architecture and for a kernel split into a non-preemptive top half and an interrupt-driven bottom half. However, the Unix server is multithreaded and relies on fine-grain locks for concurrency control. Instead of converting the approximately 1000 `spl` calls in the server to use lock-based synchronization, we simulate the `spl` calls at user-level with locks and software interrupts. Each UDP round trip has about 50 `spls` on the critical path, with each requiring about 12 μsecs on a DECstation 2100. We observed a speedup of about 150 μsecs per round trip when we switched to a cheaper locking primitive [Bershad 91].

Too many levels of scheduling

The Unix server uses Mach's C-Threads library to multiplex several user-level threads on top of a few kernel threads [Cooper & Draves 88]. Synchronization between user-level threads is generally cheap, since it is little more than a coroutine switch. However, the Unix server's network input threads are *wired* to dedicated kernel threads. This means that scheduling operations between the network interrupt thread and other threads in the Unix server (such as those waiting on incoming packets or locks) involves two kernel threads. Mach kernel threads can only synchronize with Mach IPC, so interthread synchronization within the Unix server involves additional passes through the kernel.

3 A New User-Level UDP Server

To better show that the *implementation*, rather than the address space, is responsible for poor network performance, we built a standalone UDP protocol server which exports an RPC interface analogous to Unix's `sendto` and `recvfrom` interface. UDP clients communicate with their local UDP server using Mach RPC without memory remapping to move data between address spaces.

The UDP server's lowest layer is the same as the Unix server; incoming packets are received from the packet filter and outgoing packets are written to the device driver with a `device_write` system call. The server takes care of checksumming, packet formatting, port dispatching, etc.

The main differences between our UDP server and the Unix server are:

- The incoming packet path is optimized so that packets are only copied once from a buffer on the network input thread's stack to an IPC message buffer that is sent to the destination task. This path only requires one primitive locking operation.
- None of the server's C-threads are wired to kernel threads so interthread context switching is cheap.
- Protocol clients call the server via Mach IPC instead of Unix syscalls, thus avoiding the syscall-to-IPC translation step in the system call emulator.

UDP Implementation	Time (ms)
New UDP Server (user-to-user)	4.2
New UDP Server (server-to-server)	4.0
Mach 2.5 kernel	4.8
Unix server (IPC,VM)	19.5

Table 3: UDP Round Trip Times for Various Systems. The first line is for UDP access through a special UDP server running on each machine. The second line is for server-to-server communication. This eliminates the Mach IPC between the end users and the protocol server. The third and fourth lines are repeated from earlier tables.

The round trip times for two user-level applications to communicate over the network via the new UDP server are shown on the first lines of Table 3. The table also repeats the times for the unoptimized Unix server and for the in-kernel protocol implementation for comparison. Table 3 shows that it is possible to achieve round trip latencies comparable to a macrokernel with a user-level protocol implementation. Furthermore, since the top-level (user) and bottom-level (kernel) interfaces to the protocol stack are the same, it is clear that the speedup over the Unix server's UDP is due to the difference in implementation. The difference between the user-to-user and server-to-server times reflects the cost of the additional RPC's through the Mach 3.0 microkernel.

4 Absolute Performance Is What Counts

There is one compelling observation which weakens our claim that protocols can be effectively implemented at user level: *our UDP server is still slow.* Even a 4 ms UDP round trip time, in-kernel or out-of-kernel, is slow. We agree. Mach 3.0 has an in-kernel network RPC that is about twice as fast [Barrera 91].[2] Researchers at DEC SRC [Schroeder & Burrows 90] have shown that the way to reduce latency to the bare minimum is to put pieces of the protocol in the kernel's interrupt handler, thereby eliminating an additional dispatch and RPC

There are three responses to the "sorry, not fast enough" argument. The first is that we could make it faster if we tried — we just haven't. Our goal was to show that a user-level implementation could match an in-kernel implementation, and we've done that. We could, for example, improve the implementation of the packet filter, or use it more aggressively by having each receiving thread register its own filter. We could also use shared memory to pass data between the UDP server and its clients, as is done now in some cases by the Unix server[3], or between the kernel and the Unix server [Reynolds & Heller 91].

The second response is that, for low latency request-response protocols, the important parts of the protocol (that is, the common cases) can be implemented directly within the sending and receiving address spaces. This enables the protocol server to be bypassed altogether. With the exception of using a general packet filter, this is the approach taken in DEC SRC's RPC implementation for the Firefly.

The third response is that not all protocols are expected to have low latency. A protocol such as TCP, where throughput and not latency is crucial, can run efficiently at user level [Forin et al. 91].

5 Conclusions

We have shown that at least one user-level protocol is slow when running on top of a microkernel and that it can be made fast by "de-Unixifying" it. We expect that other protocols will be amenable to this general approach as well.

The UDP server described in this paper is not part of the standard Mach distribution from CMU. It was done within the context of an experimental kernel and user environment so that we could demonstrate what was and what was not responsible for microkernel networking performance.

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 Summer 1986 USENIX Conference*, pages 93–113, July 1986.

[Barrera 91] Barrera, J. S. A Fast Mach Network IPC Implementation. In *Proceedings of the Second Usenix Mach Workshop*, November 1991.

[Bershad 91] Bershad, B. N. Mutual Exclusion for Uniprocessors. Technical Report CMU-CS-91-116, School of Computer Science, Carnegie Mellon University, April 1991.

[Bershad et al. 91] Bershad, B. N., Draves, R. P., and Forin, A. Using microbenchmarks to evaluate system performance. In *Submitted to WWOS 92*, December 1991.

[Cooper & Draves 88] Cooper, E. C. and Draves, R. P. C-threads. Technical Report CMU-CS-88-54, School of Computer Science, Carnegie Mellon University, February 1988.

[Draves et al. 91] Draves, R. P., Bershad, B. N., Rashid, R. F., and Dean, R. W. Using Continuation to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 122–136, October 1991.

[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*, November 1991.

[Golub et al. 90] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an Application Program. In *Proceedings of the Summer*

[2] Unlike UDP, Mach RPC has kernel-oriented semantics, such as the ability to pass out-of-line data and port rights, which also motivate an in-kernel implementation.

[3] Work is underway at CMU to pass network messages between the Unix server and protocol clients via shared memory buffers.

1990 USENIX Conference, pages 87–95, June 1990.

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

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

[Ousterhout et al. 88] Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M. N., and Welch, B. B. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, February 1988.

[Reynolds & Heller 91] Reynolds, F. and Heller, J. Kernel Support for Network Protocol Servers. In *Proceedings of the Second Usenix Mach Workshop*, November 1991.

[Schroeder & Burrows 90] Schroeder, M. D. and Burrows, M. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990. Also appeared in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.