[31] Sun Microsystems, Inc., Mountain view, Calif. *Network File System*, Feb. 1986.

[32] Sun Microsystems, Inc., Mountain view, Calif. *Remote Procedure Call Programming Guide*, Feb. 1986.

[33] C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, Aug. 1988.

[34] USC. Transmission control protocol. Request For Comments 793, USC Information Sciences Institute, Marina del Ray, Calif., Sept. 1981.

[35] R. van Renesse, H. van Staveren, and A. S. Tanenbaum. Performance of the world's fastest distributed operating system. *Operating Systems Review*, 22(4):25–34, Oct. 1988.

[36] R. W. Watson and S. A. Mamrak. Gaining efficiency in transport services by appropriate design and implementation choices. *ACM Transactions on Computer Systems*, 5(2):97–120, May 1987.

[37] B. B. Welch. The Sprite remote procedure call system. Technical Report UCB/CSD 86/302, University of California Berkeley, Berkeley, Calif., June 1988.

[15] N. C. Hutchinson, L. L. Peterson, M. Abbott, and S. O'Malley. RPC in the *x*-Kernel: Evaluating new design techniques. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, pages 91–101, Dec. 1989.

[16] K. A. Lantz, W. I. Nowicki, and M. M. Theimer. An empirical study of distributed application performance. *IEEE Transactions on Software Engineering*, SE-11(10):1162–1174, Oct. 1985.

[17] S. J. Leffler, W. N. Joy, and R. S. Fabry. 4.2BSD networking implementation notes. In *Unix Programmer's Manual, Volume 2C*. University of California at Berkeley, July 1983.

[18] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, 1989.

[19] P. Mockapetris. Domain names—implementation and specification. Request For Comments 1035, USC Information Sciences Institute, Marina del Ray, Calif., Nov. 1987.

[20] S. W. OMalley, M. B. Abbott, N. C. Hutchinson, and L. L. Peterson. A transparent blast facility. *Journal of Internetworking*, 1(2), Dec. 1990.

[21] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *IEEE Computer*, pages 23–36, Feb. 1988.

[22] L. L. Peterson, N. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, Aug. 1989.

[23] L. L. Peterson, N. C. Hutchinson, S. W. O'Malley, and H. C. Rao. The *x*-Kernel: A platform for accessing internet resources. *IEEE Computer*, 23(5):23–33, May 1990.

[24] D. Plummer. An ethernet address resolution protocol. Request For Comments 826, USC Information Sciences Institute, Marina del Ray, Calif., Nov. 1982.

[25] J. Postel. User datagram protocol. Request For Comments 768, USC Information Sciences Institute, Marina del Ray, Calif., Aug. 1980.

[26] J. Postel. Internet message control protocol. Request For Comments 792, USC Information Sciences Institute, Marina del Ray, Calif., Sept. 1981.

[27] J. Postel. Internet protocol. Request For Comments 791, USC Information Sciences Institute, Marina del Ray, Calif., Sept. 1981.

[28] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):311–324, Oct. 1984.

[29] M. D. Schroeder and M. Burrows. Performance of Firefly RPC. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, pages 83–90, Dec. 1989.

[30] K. Sollins. The TFTP protocol (revision 2). Request For Comments 783, USC Information Sciences Institute, Marina del Ray, Calif., June 1981.

Acknowledgments

Sean O'Malley, Mark Abbott, Clinton Jeffery, Shivakant Mishra, Herman Rao, and Vic Thomas have contributed to the implementation of protocols in the *x*-kernel. Also, Greg Andrews, Rick Schlichting, Pete Downey and the referees made valuable comments on earlier versions of this paper.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of Summer Usenix*, July 1986.

[2] G. R. Andrews and R. A. Olsson. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, Jan. 1988.

[3] M. S. Atkins. Experiments in SR with different upcall program structures. *ACM Transactions on Computer Systems*, 6(4):365–392, Nov. 1988.

[4] A. Black, N. C. Hutchinson, E. Jul, H. M. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, Jan. 1987.

[5] L.-F. Cabrera, E. Hunter, M. Karels, and D. Mosher. User-process communication performance in networks of computers. *IEEE Transactions on Software Engineering*, SE-14(1):38–53, Jan. 1988.

[6] D. R. Cheriton. VMTP: A transport protocol for the next generation of communications systems. In *Proceedings of the SIGCOMM '86 Symposium*, pages 406–415, Aug. 1987.

[7] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, Mar. 1988.

[8] D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[9] D. D. Clark. Modularity and efficiency in protocol implementation. Request for Comments 817, MIT Laboratory for Computer Science, Computer Systems and Communications Group, July 1982.

[10] D. D. Clark. The structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 171–180, Dec. 1985.

[11] E. W. Dijkstra. Hierachical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1968.

[12] A. Goldberg and D. Robson. *Smalltalk80: The Language and its Implementation*. Addison-Wesley, May 1983.

[13] A. Habermann, L. Flon, and L. Cooprider. Modularization and hierarchy in a family of operating systems. *Communications of the ACM*, 19(5):266–272, May 1976.

[14] N. C. Hutchinson, S. Mishra, L. L. Peterson, and V. T. Thomas. Tools for implementing network protocols. *Software—Practice and Experience*, 19(9):895–916, Sept. 1989.

One could also argue that user code is easier to debug than is kernel code, and this would be true. To get around this problem, we have built an *x*-kernel simulator that runs on top of Unix. Protocol implementors are able to code and debug their protocols in the simulator, and then move them, unchanged, to the stand-alone kernel.

The key shortcoming of our approach is that because all protocols run in the kernel address space, there is no protection between protocols. It is not our intent, however, that arbitrary protocols be allowed to execute inside the kernel. There must be a policy by which protocols are tested and approved for inclusion in the kernel. It is also the case that the *x*-kernel was designed for use on personal workstations. In such an environment, it is not unreasonable to let users run new protocols in the kernel at their own risk.

6 Conclusions

The major conclusion of this work is that it is possible to build an operating system architecture for implementing protocols that is both general and efficient. Specifically, we have demonstrated that the communication abstractions provided by the *x*-kernel are general enough to accommodate a wide range of protocols, yet protocols implemented using those abstractions perform as well as, and sometimes better than, their counterpart implementations in less structured environments.

Our experience suggests that the explicit structure provided by the *x*-kernel has the following advantages. First, the architecture simplifies the process of implementing protocols in the kernel. This makes it easier to build and test new protocols. It also makes it possible to implement a variety of RPC protocols in the kernel, thereby providing users with efficient access to a wider collection of resources. Second, the uniformity of the interface between protocols avoids the significant cost of changing abstractions and makes protocol performance predictable. This feature makes the *x*-kernel conducive to experimenting with new protocols. It also makes it possible to build complex protocols by composing a collection of single-function protocols. Third, it is possible to write efficient protocols by tuning the underlying architecture rather than heavily optimizing protocols themselves. Again, this facilitates the implementation of both experimental and established protocols.

In addition to using the *x*-kernel as a vehicle for doing protocol research and as a foundation for building distributed systems, we plan to extend the architecture to accommodate alternative interfaces to objects. Specifically, we have observed a large class of objects that appear to be protocols "on the bottom" but provide a completely different interface to their clients. For example, a network file system such as Sun's NFS uses the services of multiple protocols such as RPC and UDP, but provides the traditional file system interface—e.g., **open**, **close**, **read**, **write**, **seek**—to its clients. Our extensions take the form of a type system for objects that may be configured into the kernel and additional tools to support entities other than communication protocols. Finally, we are in the process of porting the *x*-kernel to new hardware platforms.

First, consider a synchronous send in which the sender is blocked until an acknowledgment or reply message is received from the receiver. The process per message paradigm facilitates synchronous sends in a straightforward manner: after calling a session object's **push** operation, the sender's thread of control blocks on a semaphore. When the session object eventually receives an acknowledgment or reply message via its **pop** operation, the process associated with the reply message signals the sender's process, thereby allowing it to return. In contrast, the process per protocol paradigm does not directly support synchronous sends. Instead, the sending process asynchronously sends the message and then explicitly blocks itself. While a user process can afford to block itself, a process that implements a protocol cannot; it has to process the other messages sent to the protocol. The protocol process must spawn another process to wait for the reply, but this means a synchronous protocol must simulate the process per message paradigm. Thus, the programmer is forced to "step outside" the stream model to implement an RPC protocol.

Second, consider the extent to which the two paradigms restrict the receipt of messages; that is, how easily can various disciplines for managing the order in which messages are received be represented in the two paradigms. To illustrate the point, consider the guarantees each of the following three protocols make about the order in which messages are delivered: UDP makes no guarantees, TCP ensures a total (linear) ordering of messages, and Psync preserves only a partial ordering among messages. In the case of the process per protocol paradigm, the queue of messages from which each protocol retrieves its next message implicitly enforces a linear ordering on messages. It is therefore well suited for TCP, but overly restrictive for protocols like UDP and Psync. In contrast, because arbitrarily many processes (messages) might call a protocol object's **demux** operation or a session object's **pop** operation, the process per message paradigm enforces no order on the receipt of messages, and as a consequence, does not restrict the behavior of protocols like UDP.

It is, of course, possible to enforce any ordering poliy by using other synchronization mechanisms such as semaphores. For example, the *x*-kernel implementation of TCP treats the adjacent high-level protocol object as a critical section; i.e., it protects the **demux** operation with a mutual exclusion semaphore. This enforces a total ordering on the messages it passes to the adjacent high-level protocol object. As a second example, the implementation of Psync in the *x*-kernel permits multiple—but not arbitrarily many—outstanding calls to the adjacent high-level protocol's **demux** operation. In general, our experience suggests that the process per message paradigm permits more parallelism, and as a consequence, is better suited for a multiprocessor architecture.

5.4 Kernel Implementation

An important design choice of the *x*-kernel is that the entire communication system is embedded in the kernel. In contrast, operating systems with minimal kernels—e.g., Mach [1] and Taos [33]—put the communication system in a user address space. One argument often made in favor of minimalist kernels is that they lead to a clean modular design, thereby making it easier to modify and change subsystems. Clearly, the *x*-kernel is able to gain the same advantage. In effect, the *x*-kernel's object-oriented infrastructure forms the "kernel" of the system, with individual protocols configured in as needed. The entire system just happens to run in privileged mode, with the very significant advantage of being more efficient.

of TCP in the *x*-kernel would be as efficient as the Unix implementation.

5.2 Protocol Composition

A second issue involves how the kernel's communication objects are composed to form paths through the kernel. Protocol and session objects are composed at three different times. Initially, protocol objects are statically composed when the kernel is configured. For example, TCP is given a capability for IP at configuration time. At the time the kernel is booted, each protocol runs some initialization code that invokes **open_enable** on each low-level protocol from which it is willing to receive messages. For example, IP and ARP invoke the ethernet's **open_enable**, TCP and UDP invoke IP's **open_enable**, and so on. Finally, at runtime, an active entity such as an application program invokes the **open** operation on some protocol object. The **open** operation, in turn, uses the given participant set to determine which lower-level protocol it should open. For example, when an application process opens a TCP session, it is the TCP protocol object that decides to open an IP session. In other words, protocol and session objects are *recursively* composed at run time in the *x*-kernel.

This scheme has two advantages. First, a kernel can be configured with only those protocols needed by the application. For example, we have built an "Emerald-kernel" that contains only those protocols needed to support Emerald programs. In contrast, many kernels are implemented in a way that makes it very difficult to configure in (out) individual protocols. Such systems often implement "optional" protocols outside the kernel, and as illustrated by Sun RPC, these protocols are less efficient than if they had been implemented in the kernel. Second, protocols are not statically bound to each other at configuration time. As demonstrated elsewhere, the architecture makes it possible to dynamically compose protocols [15].

5.3 Process per Message

A key aspect of the *x*-kernel's design is that processes are associated with messages rather than protocols. The process per message paradigm has the advantage of making it possible to deliver a message from a user process to a network device (and vise versa) with no context switches. This paradigm seems well suited for a multiprocessor architecture. In contrast, the process per protocol paradigm inserts a message queue between each protocol and requires a context switch at each level. While it has been demonstrated by System V Unix that it is possible to implement the process per protocol paradigm efficiently on a uniprocessor,[7] it seems likely that a multiprocessor would have to implement a real context switch between each protocol level.

Furthermore, there is the issue of whether the process per message or process per protocol paradigm is more convenient for programming protocols [3]. While the two paradigms are duals of each other—each can be simulated in the other—our experience illustrates two, somewhat subtle, advantages with the process per message paradigm.

[7] System V multiplexes a single Unix process over a set of stream modules. Sending a message from one module to another via a message queue requires two procedure calls in the best case: one to see if the message queue is empty and one to invoke the next module.

5.1 Explicit Structure

One of the most important features of the *x*-kernel is that it defines an explicit structure for protocols. Consider two specific aspects of this structure. First, the *x*-kernel partitions each network protocol into two disjoint components: the protocol object switches messages to the right session and the session object implements the protocol's interpreter. While this separation implicitly exists in less structured systems, explicitly embedding the partition into the system makes protocol code easier to write and understand. This is because it forces the protocol implementor to distinguish between protocol-wide issues and connection-dependent issues. Second, the *x*-kernel supports buffer, map, and event managers that are used by all protocols. Similar support in other systems is often ad hoc. For example, each protocol is responsible for providing its own mechanism for managing ids in Unix.

Our experience is that the explicit structure provided by the *x*-kernel has two advantages. First, efficient protocol implementations can be achieved without a significant optimization effort. For example, the performance numbers presented in Section 4 correspond to protocol implementations that have not been heavily optimized. It was not necessary to do fine-grained optimizations of each protocol because the architecture itself is so highly tuned. Instead, one only applies a small collection of high-level "efficiency rules", such as always to cache open sessions, not touch the header any more than necessary, pre-allocate headers, optimize for the common case, and never copy data. Our experience is that these rules apply uniformly across all protocols. Of course, no amount of operating system optimizations can compensate for poor implementation strategies; e.g., a poorly tuned timeout strategy.

Second, by making the structure explicit, we have been able to make the interface to protocols uniform. This has the desirable effect of making performance predictable, which is a necessary feature when one is designing new protocols. For example, by knowing the cost of individual protocol layers, one is able to predict the cost of composing those protocols. As illustrated by Berkeley Unix, "predictability" is not a universal characteristic.

The potential down side of this additional structure is that it degrades protocol performance. Our experience is that the most critical factor is the extent to which the *x*-kernel's architecture limits a given protocol's ability to access information about other protocols that it needs to make decisions. For example, in order to avoid re-fragmentation of the packets it sends via IP, TCP needs to know the maximum transmission of the underlying network protocol, in our case, the ethernet. Whereas an ad hoc implementation of TCP might learn this information by looking in some global data structure, the *x*-kernel implementation is able to learn the same information by invoking a control operation on IP. While one might guess that an unweildy number of different control operations would be necessary to access all the information protocols need, our experience is that a relatively small number of control operations is sufficient; i.e., on the order of a dozen. By using these control operations, protocols implemented in the *x*-kernel are able to gain the same advantage available to ad hoc implementations.

Also, it is worth noting that TCP is the worst protocol we've encountered at depending on information from other protocols—not only does it depend on information from other protocols, but it also depends on their headers—and even it suffers at most a 10% performance penalty for using control operations rather than directly reading shared memory. Futhermore, this 10% penalty is inflated by the fact that we retrofitted the Unix implementation of TCP into the *x*-kernel. We believe a "from scratch" implementation

4.3 Comparisons with Sprite

The third set of experiments involve comparing the *x*-kernel to the Sprite operating system. Comparing the *x*-kernel to Sprite is interesting because, like other recent experimental systems [7, 35], Sprite is optimized to support a particular RPC protocol. Specifically, Sprite implements an RPC protocol that supports *at most once* semantics [37]. We compared an implementation of Sprite RPC in the *x*-kernel with a native implementation whose performance was also measured on a Sun 3/75.[6] Both versions were compiled using the standard Sun C compiler. The latency and throughput results are presented in Table VI.

	x-Kernel	Sprite
Latency (msec)	2.19	2.60
Throughput (k-bytes/sec)	858	700
Incremental (msec/1k-bytes)	1.20	1.20

Table VI: Latency and Throughput

The key observation is that Sprite RPC performs just as well in the *x*-kernel as it does in the Sprite kernel, and this performance is competitive with other fast RPC implementations [29]. Being able to implement a protocol in the *x*-kernel that is as efficient as an implementation in a kernel that was designed around the protocol further substantiates our claim that the *x*-kernel's architecture does not negatively impact protocol performance.

5 Experience

To date, we have implemented a large body of protocols in the *x*-kernel, including:

- Application-level protocols: Sun NFS [31], TFTP [30], DNS [19], the run time support system for the Emerald programming language, the run time support system for the SR programming language [2].
- Interprocess communication protocols: UDP, TCP, Psync, VMTP [6], Sun RPC, Sprite RPC;
- Network protocols: IP;
- Auxiliary protocols: ARP [24], ICMP [26];
- Device protocols: ethernet drivers, display drivers, serial line drivers.

Generally speaking, our experience implementing protocols in the *x*-kernel has been very positive. By taking over much of the "bookkeeping" responsibility, the kernel frees the programmer to concentrate on the communication function being provided by the protocol. This section reports our experience implementing protocols in the *x*-kernel in more detail. Note that because System V streams are similar to the *x*-kernel in many ways, this section also draws direct comparisons between the two systems.

[6]The 2.6 msec latency reported for Sprite is computed by subtracting 0.2 msec from the reported time of 2.8 msec. The reported time included a crash/recover monitor not implemented in the *x*-kernel version.

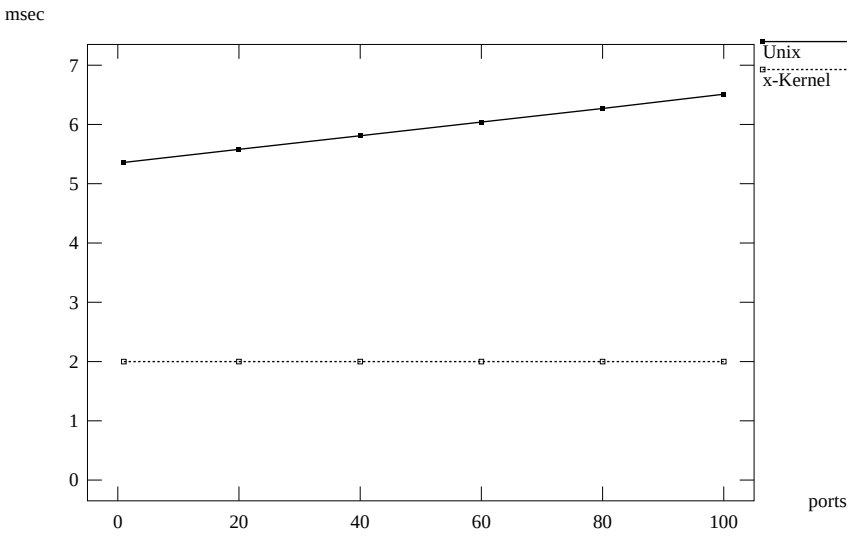

Figure 6: Map Manager

connections. Up to this point all of the experiments have been run with a single UDP port open at each host. (The same is true for the TCP experiments.) Thus, UDP's **demux** operation (and the corresponding code in Unix) had a trivial decision to make: there was only one session (socket) to pass the message to. As illustrated by the graph, the *x*-kernel exhibits constant performance as the number of ports increases, while Unix exhibits linear performance. The reason for this is simple: the *x*-kernel's map manager implements a hash table, while Unix UDP maintains a linear list of open ports. The important point is that under typical loads (approximately 40 open ports) Unix incurs a 10% performance penalty for using the "wrong" mechanism for managing identifiers.

4.2.4 Summary

In summary, this set of experiments support the following conclusions. First, the *x*-kernel is significantly faster than Unix when measured at a course-grain level. Second, the cost of the Unix socket interface is the leading reason why user-to-user performance is significantly worse in Unix than it is in the *x*-kernel. Third, the performance of individual protocols in the two systems—when controlling for differences in implementation techniques—are comparable. This supports our claim that the *x*-kernels' architecture does not negatively impact protocol performance. Fourth, the additional structure provided by the *x*-kernel has the potential to drastically improve protocol performance, as illustrated by our implementation of Sun RPC. Finally, the *x*-kernel's underlying support routines perform better than their Unix counterparts under increased load.

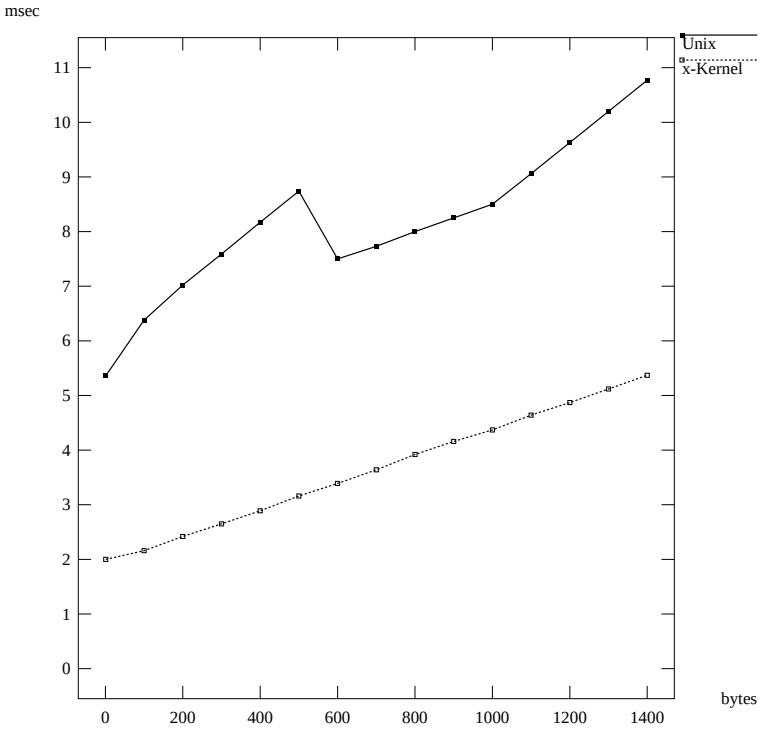

Figure 5: Message Manager

cost in Unix varies significantly. In particular, the Unix curve can be divided into four distinct parts: (1) the incremental cost of going from 200 bytes to 500 bytes is 0.57 msec per 100 bytes; (2) the cost of sending 600 bytes is over 1 msec less than the cost of sending 500 bytes; (3) the incremental cost of sending 600 to 1000 bytes is 0.25 msec per 100 bytes (same as the *x*-kernel); and (4) the incremental cost of sending between 1100 and 1400 bytes is again 0.57 msec per 100 bytes.

The reason for this wide difference of behavior is that Unix does not provide a uniform message/buffer management system. Instead, it is the responsibility of each protocol to represent a message as a linked list of two different storage units: mbufs which hold up to 118 bytes of data and pages which hold up to 1024 bytes of data [18]. Thus, the difference between the four parts of the Unix curve can be explained as follows: (1) a new mbuf is allocated for each 118 bytes (i.e., 0.57msec/2 = 0.28msec is the cost of using an mbuf); (2) a page is allocated when the message size reaches 512 bytes (half a page); (3) the rest of the page is filled without the need to allocate additional memory; and (4) additional mbufs are used. Thus, the difference in the cost of sending 500 bytes and 600 bytes in Unix concretely demonstrates the performance penalty involved in using the "wrong" buffer management strategy; in this case, the penalty is 14%. Perhaps just as important as this quantitative impact is the "qualitative" difference between the buffer management scheme offered by the two systems: someone had to think about and write the data buffering code that results in the Unix performance curve.

Second, Figure 6 gives the performance of the *x*-kernel and Unix as a function of the number of open

time can be attributed to to the overhead of sockets themselves. In contrast, 0.55 msec of the 0.61 msec of the interface cost for the *x*-kernel is associated with the cost of crossing the user/kernel boundary; it costs only 50 μsec to create an initial message buffer that holds the message. Note that this 50 μsec cost is not repeated between protocols within the kernel.

4.2.2 User-to-User Throughput

We also measured user-to-user throughput of UDP and TCP. Table V summarizes the results. In the case of UDP, we sent a 16k-byte message from the source process to the destination process and a 1-byte message in reply. This test was run using the UDP-IP-ETH protocol stack. To make the experiment fair, the *x*-kernel adopts the Unix IP fragmentation strategy of breaking large messages into 1k-byte datagrams; e.g., sending a 16k-byte user message involves transmitting sixteen ethernet packets. By sending the maximum number of bytes (1500) in each ethernet packet, we are able to improve the *x*-kernel user-to-user throughput to 604 kilobytes/sec.

Protocol	*x*-Kernel (k-bytes/sec)	Unix (k-bytes/sec)
UDP	528	391
TCP	417	303

Table V: Throughput

In the case of TCP, we measured the time necessary to send 1M-byte from a source process to a destination process. The source process sends the 1M-byte by writing 1024 1k-byte messages to TCP. Similarly, the destination process reads 1k-byte blocks. In both cases TCP was configured with a 4k-byte sending and receiving window size, effectively resulting in stop-and-wait behavior. This explains why the TCP throughput for both systems is less than the UDP throughput, which effectively uses the blast algorithm. Finally, as in the UDP experiment, the data is actually transmitted in 1k-byte IP packets.

In the case of both Unix and the *x*-kernel, the user data is copied across the user/kernel boundary twice—once on the sending host and once on the receiving host. We have also experimented with an implementation of the *x*-kernel that uses page remapping instead of data copying. Remaping is fairly simple on the sending host, but difficult on the receiving host because the data contained in the incoming fragments must be caught in consecutive buffers that begin on a page boundary. A companion paper describes how this is done in the *x*-kernel [20].

4.2.3 Support Routines

In addition to evaluating the performance of the architecture as a whole, we also quantify the impact the underlying support routines have on protocol performance. Specifically, we are interested in seeing the relative difference between the way messages and identifiers are managed in the *x*-kernel and Unix.

First, Figure 5 shows the performance of UDP the *x*-kernel and Unix for message sizes ranging from 1 byte to 1400 bytes; i.e., the UDP message fits in a single IP datagram. It is interesting to note that the incremental cost of sending 100 bytes in the *x*-kernel is consistently 0.25 msec, while the incremental

messages to the appropriate port. There is no room for variation in the implementation strategy adopted by the *x*-kernel and Unix implementations. As a consequence, the incremental cost of UDP is a fair representation of the minimal cost of a base protocol in the two systems.[5] While it is possible that the 140 μsec difference between the two implementations can be attributed to coding techniques, the protocol is simple enough and the Unix implementation mature enough that we attribute the difference to the underlying architecture.

TCP is a complex protocol whose implementation can vary significantly from system to system. To control for this, we directly ported the Unix implementation to the *x*-kernel. Thus, the difference between the incremental cost of TCP in the two systems quantifies the penalty for coercing a complex protocol into the *x*-kernel's architecture. Our experiments quantify this penalty to be 110 μsec, or less than 10%. We attribute this difference to TCP's dependency on the IP header. Specifically, TCP uses IP's length field and it computes a checksum over both the TCP message and the IP header. Whereas the *x*-kernel maintains a strong separation between protocols by forcing TCP to query IP for the necessary information using a control operation, the Unix implementation gains some efficiency by directly accessing the IP header; i.e., it violates the boundary between the two protocols. While one could argue that the rigid separation of protocols enforced by the *x*-kernel's architecture is overly restrictive, we believe the more accurate conclusion to draw from this experiment is that protocol specifications should eliminate unnecessary dependencies on other protocols. In this particular case, having TCP depend on information in the IP header does not contribute to the efficiency of TCP; it is only an artifact of TCP and IP being designed in conjunction with each other.

RPC is also a complex protocol, but instead of porting the Unix implementation into the *x*-kernel, we implemented RPC in the *x*-kernel from scratch. Thus, comparing the incremental cost of RPC in the two systems provides a handle on the potential advantages of implementing a complex protocol in a highly structured system. Because this experiment was much less controlled than the other two, we are only able to draw the following weaker conclusions. First, because the *x*-kernel implementation is in the kernel rather than user space, it is able to take advantage of kernel support not available to user-based implementations. For example, the kernel-based implementation is able to avoid unnecessary data copying by using the kernel's buffer manager. While this makes the comparison somewhat unfair, it is important to note it is the structure provided by the *x*-kernel that makes it possible to implement a new protocol like RPC to the kernel. In contrast, implementing RPC in the Unix kernel would be a much more difficult task. Second, the *x*-kernel implementation is dramatically cleaner than the Unix implementation. Although difficult to quantify, our experience suggests that the additional structure provided by the *x*-kernel led to this more efficient implementation. We do not believe, however, that our experience with RPC is universally applicable. For example, it is doubtful that a "from scratch" implementation of TCP in the *x*-kernel would be significantly more efficient than the Unix implementation.

Finally, it is clear that the Unix socket abstraction is both expensive and non-uniform. Sockets were initially designed as an interface to TCP; coercing UDP and IP into the socket abstraction involves additional cost. Furthermore, independent measurements of the time it takes to enter and exit the Unix kernel and the time it takes to do a context switch in Unix indicate that roughly 2/3 of the Unix interface

[5] By "base" protocol we mean the simplest protocol that does any real work. One could imagine a simpler "null" protocol that passes messages through unchanged.

Furthermore, the measurements highlight an interesting anomaly in Unix: it costs more to send a message using the IP-ETH protocol stack than it does using the UDP-IP-ETH protocol stack. We refer to this as the cost of *changing abstractions*. That is, the socket abstraction is tailored to provide an interface to transport protocols like UDP and TCP; there is an added cost for using sockets as an interface to a network protocol like IP.[4] Our experience with Unix also suggests that the 4.87 msec round trip delay for ETH is inflated by a significant abstraction changing penalty. In particular, because we had to use the System V streams mechanism available in SunOS to directly access the ethernet—but SunOS uses the Berkeley Unix representation for messages internally—each message had to be translated between its Berkeley Unix representation and its System V Unix representation. Note that 4.87 msec is not a fair measure of ETH when it is incorporated in the other protocol stacks we measured—e.g., UDP-IP-ETH—because the stream abstraction is not used in those cases.

The limitation of the preceding experiments is that they are not fine-grained enough to indicate where each kernel is spending its time. To correct for this, we measured the incremental cost for the three end-to-end protocols: UDP, TCP, and RPC. The results are presented in Table IV.

In the case of the *x*-kernel, the incremental cost for each protocol is computed by subtracting the measured latency for appropriate pairs of protocol stacks; e.g., TCP latency in the *x*-kernel is given by 3.30 msec - 1.89 msec = 1.41 msec. That is, crossing the TCP protocol four times—twice outgoing and twice incoming—takes 1.41 msec. In the case of Unix, we modified the Unix kernel so that each protocol would "reflect" incoming messages back to their source rather than pass them up to the appropriate higher level protocol. In doing this, we effectively eliminate the overhead for entering and exiting the kernel and using the socket abstraction; i.e., we measured kernel-to-kernel latency rather than user-to-user latency. The kernel-to-kernel latency is 2.90 msec for IP, 3.15 msec for UDP, and 4.20 msec for TCP. These numbers are in turn used to compute the incremental cost of UDP and TCP; e.g., TCP latency in Unix is given by 4.20 msec - 2.90 msec = 1.30 msec. We compute the incremental cost of RPC in Unix by subtracting the UDP user-to-user latency from the user-to-user RPC latency.

Finally, by subtracting the kernel-to-kernel latency from the user-to-user latency for IP, UDP, and TCP, we are able to determine the cost of the network interface to each of these protocols. In the case of Unix, the cost of the socket interface to TCP is given by 6.10 msec - 4.20 msec = 1 .90 msec. This time includes the cost of crossing the user/kernel boundary and the overhead imposed by socket themselves. In the case of the *x*-kernel, the difference between kernel-to-kernel latency and user-to-user latency yields a uniform 0.61 msec overhead for all protocol stacks.

Component	*x*-Kernel (msec)	Unix (msec)
UDP	0.11	0.25
TCP	1.41	1.30
RPC	2.00	3.84
Interface	0.61	1.90/2.25/2.75

Table IV: Incremental Costs

UDP is a trivial protocol: it only adds and strips an 8-byte header and demultiplexes incoming

[4]It is possible that this cost is not intrinsic, but that the IP/socket interface is less optimized because it is used less often.

dominate performance.

4.2 Comparisons with Unix

The second set of experiments involve comparing the *x*-kernel to Berkeley Unix. For the purpose of this comparison, we measured the performance of the DARPA Internet protocol suite—IP, UDP, and TCP—along with Sun RPC [32]. We used SunOS Release 4.0.3, a variant of 4.3 Berkeley Unix that has been tuned for Sun Microsystem workstations. SunOS release 4.0.3 also includes System V streams, but streams are not used by the implementation of IP, UDP, or TCP. They do, however, provide an interface for directly accessing the ethernet protocol. The Unix timings were all made while executing in single user mode; i.e., no other jobs were running.

Our objective in comparing the *x*-kernel with Unix is to quantify the impact the *x*-kernel architecture has on protocol performance. Toward this end, it is important to keep two things in mind. First, although the Berkeley Unix *socket* abstraction provides a uniform interface to a variety of protocols, this interface is only used between user processes and kernel protocols: (1) protocols within the kernel are not rigidly structured, and (2) the socket interface is easily separated from the underlying protocols. Thus, once the cost of the socket abstraction is accounted for, comparing the implementation of a protocol in Unix and the same protocol in the *x*-kernel provides a fair measure of the penalty imposed on protocols by the *x*-kernel's architecture. Second, so as to eliminate the peculiarities of any given protocol, we consider three different end-to-end protocols: UDP, TCP, and RPC. We believe these protocols provide a representative sample because they range from the extremely trivial UDP protocol to the extremely complex TCP protocol.

4.2.1 User-to-User Latency

Initially, we measured latency between a pair of user processes exchanging 1-byte messages using different protocol stacks. The results are presented in Table III. Each row in the table is labelled with the protocol stack being measured. In the *x*-kernel, all the protocols are implemented in the kernel; only the user processes execute in user space. In Unix, all the protocols are implemented in the kernel except RPC. Thus, in the case of the RPC-UDP-IP-ETH protocol stack, the user/kernel boundary is crossed between the user and RPC in the *x*-kernel and between RPC and UDP in Unix.

Protocol Stack	*x*-Kernel (msec)	Unix (msec)
ETH	1.68	4.87
IP-ETH	1.89	5.62
UDP-IP-ETH	2.00	5.36
TCP-IP-ETH	3.30	6.10
RPC-UDP-IP-ETH	4.00	9.20

Table III: User-to-User Latency

Although these measurements provide only a course-grain comparison of the two systems, they are meaningful in as much as we are interested in evaluating each system's overall, integrated architecture.

configured with the ethernet controller in promiscuous mode. Second, crossing the user/kernel boundary costs 20μsec in the user-to-kernel direction and 254μsec in the kernel-to-user direction. The latter is an order of magnitude more expensive because there is no hardware analog of a system trap that can be used to implement upcalls. Thus, for a pair of user processes to exchange two messages involves crossing the user/kernel boundary twice in the downward direction and twice in the upward direction, for a total boundary penalty of $2 \times 20\mu sec + 2 \times 254\mu sec = 548\mu sec$. Third, the cost of coercing protocols into our connection-based model is nominal: it costs 260μsec to open and close a null session. Moreover, this cost can usually be avoided by caching open sessions.

We next quantify the relative time spent in each part of the kernel by collecting profile data on a test run that involved sending and receiving 10,000 1-byte messages using the UDP-IP-ETH protocol stack. Table II summarizes the percentage of time spent in each component. The percentages were computed by dividing the estimated time spent in the procedures that make up each component by the difference between the total elapsed time and the measured idle time. That is, the 21.8% associated with the buffer manager means that 21.8% of the time the kernel was doing real work—i.e., not running the idle process—it was executing one of the buffer manager routines. The times reported for each of the protocols—ethernet, IP, and UDP— do not include time spent in the buffer or id managers on behalf of those protocols.

Component	Percentage
Buffer Manager	21.8
Id Manager	1.8
Ethernet	43.7
IP	9.8
UDP	2.8
Interface Overhead	5.3
Boundary Crossing	5.9
Process Management	8.6
Other	0.3

Table II: Percentage of Time Spent in each Component

As one would expect, given the simplicity of IP and UDP, the performance is dominated by the time spent in the ethernet driver manipulating the controller. In addition, we make the following four observations. First, the time spent in the buffer manager is significant: over one fifth of the total. Because 1-byte messages were being exchanged, this percentage is independent of the cost of copying messages across the user/kernel boundary; it includes only those operations necessary to add and strip headers. Second, the 5.3% reported for the interface overhead corresponds to the object infrastructure imposed on top of the C procedures that implement protocols and sessions. This percentage is both small and greatly over stated. In practice, this infrastructure is implemented by macros, but for the sake of profiling, this functionality had to be temporarily elevated to procedure status. Third, the percentage of time spent crossing the user/kernel boundary is also fairly insignificant, but as the message size increases, so does this penalty. Finally, the process management component (8.6%) indicates that while the cost of dispatching a process to shepherd each message through the kernel is not negligible, neither does it

s_{ip}^{tcp} are held there for reassembly into IP datagrams. Complete datagrams are then passed up to p_{tcp}'s **demux**, which in turn pops the message into the appropriate TCP session based on the source/destination ports contained in the message. Finally, when s_{tcp}^{user1} is eventually closed, it in turn closes s_{ip}^{tcp}, and so on.

4 Performance

This section reports on three sets of experiments designed to evaluate the performance of the *x*-kernel. The first measures the overhead of various components of the *x*-kernel, the second compares the performance of the *x*-kernel to that of a production operating system (Berkeley Unix) [17], and the third compares the *x*-kernel to an experimental network operating system (Sprite) [21]. The purpose of the latter two experiments is to quantify the impact the architecture has on protocol performance by comparing protocols implemented in the *x*-kernel with protocols implemented in two less structured environments.

The experiments were conducted on a pair of Sun 3/75s connected by an isolated 10Mbs ethernet. The numbers presented were derived through two levels of aggregation. Each experiment involved executing some mechanism 10,000 times, and recording the elapsed time every 1,000 executions. The average of these ten elapsed times is reported. Although we do not report the standard deviation of the various experiments, they were observed to be on the order of the clock granulatiry.

When interpreting the results presented in this section, it is important to keep in mind that we are interested in quantifying how operating systems influence protocol performance, not in raw protocol performance, per se. There are many implementation strategies that can be employed independent of the operating system in which a given protocol is implemented. It is also possible that coding techniques differ from one implementation to another. We attempt to control for these strategies and techniques, commenting on variations in protocol implementations when appropriate.

4.1 Overhead

An initial set of experiments measure the overhead in performing several performance-critical operations; the results are presented in Table I.

Component	Time (μsec)
Context Switch	38
Dispatch Process	135
Open/Close Session	260
Enter/Exit Kernel	20
Enter/Exit User	254
Copy 1024 Bytes	250

Table I: Cost of Various System Components

Three factors are of particular importance. First, the overhead to dispatch a light-weight process to shepherd a message through the *x*-kernel is 135μsecs. Dispatching a shepherd process costs two context switches and about 50μsec in additional overhead. This overhead is small enough, relative to the rate at which packets are delivered by the ethernet, that the *x*-kernel drops less than 1 in a million packets when

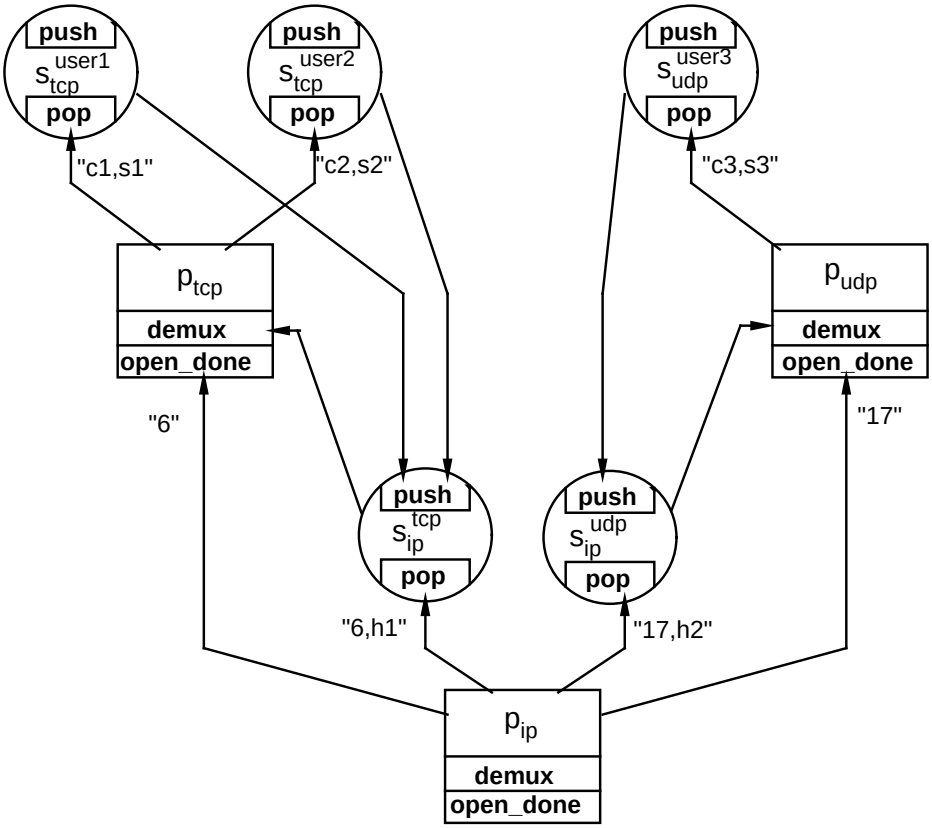

Figure 4: Example Protocol and Session Objects

set consisting of the invoking object's protocol number. In practice, such invocations occur at boot time as a result of initialization code within each protocol object.

Next, consider how the sessions were created. Suppose s_{tcp}^{user1} is a TCP session created by a client process that initiates communication. Session s_{tcp}^{user1} directly opens IP session s_{ip}^{tcp} by invoking p_{ip}'s **open** operation, specifying both the source and destination IP addresses. Session s_{ip}^{tcp}, in turn, creates a template header for all messages sent via that session and saves it in its internal state; it may also open one or more lower-level sessions. In contrast, suppose s_{udp}^{user3} is a UDP session indirectly created by a server process that waits for communication. In this case, s_{ip}^{udp} is created by p_{ip} invoking p_{udp}'s **open_done** operation when a message arrives at p_{ip}'s **demux** operation. Session s_{ip}^{udp} then invokes p_{udp}'s **demux** operation, which in turn creates s_{udp}^{user3}.

Once established, either by a client as in the case of s_{ip}^{tcp} and s_{tcp}^{user1}, or by a server and a message arriving from the network as in the case of s_{ip}^{udp} and s_{udp}^{user3}, the flow of messages through the protocols and sessions is identical. For example, whenever s_{tcp}^{user1} has a message to send, it invokes s_{ip}^{tcp}'s **push** operation, which attaches an IP header and pushes the message to some lower-level session. Messages that arrive from the network are eventually passed up to p_{ip}, which examines the header and pops the message to s_{ip}^{tcp} if the protocol number is "6" and the source host address matches *h1*. Messages popped to IP session

3 Examples

A composition of protocol and session objects form a path through the kernel that messages follow. For example, consider an *x*-kernel configured with the suite of protocol objects given in Figure 3, where Emerald_RTS is a protocol object that implements the run time support system for the Emerald programming language [4] and ETH is a protocol object that corresponds to the ethernet driver. In this scenario, one high-level participant—an Emerald object—sends a message to another Emerald object identified with Emerald_Id *eid*. This identifier is only meaningful in the context of protocol Emerald_RTS. Likewise, Emerald_RTS is known as *port2005* in the context of UDP, which is in turn known as *protocol17* in the context of IP, and so on. A set of protocols on a particular host are known in the context of that host.

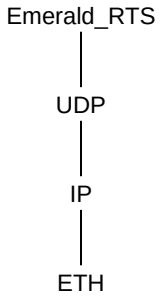

Figure 3: Example Suite of Protocols

As a session at one level opens a session at the next lower level, it identifies itself and the peer(s) with which it wants to communicate. Emerald_RTS, for example, opens a UDP session with a participant set identifying itself with the relative address *port2005*, and its peer with the absolute address $\langle$*port2005, host192.12.69.5*$\rangle$. Thus, a message sent to a peer participant is pushed down through several session objects on the source host, each of which attaches header information that facilitates the message being popped up through the appropriate set of sessions and protocols on the destination host. In other words, the headers attached to the outgoing message specify the path the message should follow when it reaches the destination node. In this example, the path would be denoted by the "path name"

eid@port2005@protocol17@...

Intuitively, the session's **push** operation constructs this path name by pushing participant identifiers onto the message, and the session's **pop** operation consumes pieces of the path name by popping participant identifiers off the message.

As a second example, consider the collection of protocols and sessions depicted in Figure 4. The example consists of three protocols, denoted p_{tcp}, p_{udp}, and p_{ip}; IP sessions s_{ip}^{tcp} and s_{ip}^{udp}; TCP sessions s_{tcp}^{user1} and s_{tcp}^{user2}; and UDP session s_{udp}^{user3}. Each edge in the figure denotes a capability one object possess for another, where the edge labels denote participant identifiers that have been bound to the capability.

Initially, note that p_{ip} possesses a capability (labelled "6") for p_{tcp} and a capability (labelled "17") for p_{udp}, each the result of the higher-level protocol invoking p_{ip}'s **open_enable** operation with the participant

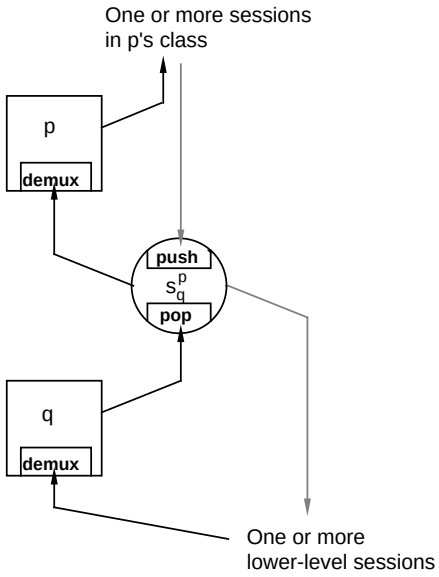

Figure 2: Relationship Between Protocols and Sessions

generates them) and flow downward to a device. While flowing downward, a message visits a series of sessions via their **push** operations. While flowing upward, a message alternatively visits a protocol via its **demux** operation and then a session in that protocol's class via its **pop** operation. As a message visits a session on its way down, headers are added, the message may fragment into multiple message objects, or the message may suspend itself while waiting for a reply message. As a message visits a session on the way up, headers are stripped, the message may suspend itself while waiting to reassemble into a larger message, or the message may serialize itself with sibling messages. The data portion of a message is manipulated—e.g., headers attached or stripped, fragments created or reassembled—using the buffer management routines mentioned in Section 2.1.

When an incoming message arrives at the network/kernel boundary (i.e., the network device interrupts), a kernel process is dispatched to *shepherd* it through a series of protocol and session objects; this process begins by invoking the lowest-level protocol's **demux** operation. Should the message eventually reach the user/kernel boundary, the shepherd process does an upcall and continues executing as a user process. The kernel process is returned to a pool and made available for re-use whenever the initial protocol's **demux** operation returns or the message suspends itself in some session object. In the case of outgoing messages, the user process does a system call and becomes a kernel process. This process then shepherds the message through the kernel. Thus, when the message does not encounter contention for resources, it is possible to send or receive a message with no process switches. Finally, messages that are suspended within some session object can later be re-activated by a process created as the result of a timer event.

In addition to creating sessions, each protocol also "switches" messages received from the network to one of its sessions with a

demux(protocol, message)

operation. **demux** takes a message as an argument, and either passes the message to one of its sessions, or creates a new session—using the **open_done** operation—and then passes the message to it. In the case of a protocol like IP, **demux** might also "route" the message to some other lower-level session. Each protocol object's **demux** operation makes the decision as to which session should receive the message by first extracting the appropriate external id(s) from the message's header. It then uses a map routine to translate the external id(s) into either an internal id for one of its sessions (in which case **demux** passes the message to that session) or into an internal id for some high-level protocol (in which case **demux** invokes that protocol's **open_done** operation and passes the message to the resulting session).

For example, given the invocations of **open** and **open_enable** outlined above, q's **demux** operation would first extract the *local_port* and *remote_port* fields from the message header and attempt to map the pair $\langle local\_port, remote\_port \rangle$ into some session object s. If successful, it would pass the message on to session s. If unsuccessful, q's **demux** would next try to map *local_port* into some protocol object p. If the map manager supports such a binding, q's **demux** would then invoke p's **open_done** operation with the participant set $\{local\_port, remote\_port\}$—yielding some session s'—and then pass the message on to s'.

2.3 Session Objects

A session is an instance of a protocol created at runtime as a result of an **open** or an **open_done** operation. Intuitively, a session corresponds to the end-point of a network connection; i.e., it interprets messages and maintains state information associated with a connection. For example, TCP session objects implement the sliding window algorithm and associated message buffers, IP session objects fragment and reassemble datagrams, and Psync session objects maintain context graphs. UDP session objects are trivial; they only add and strip UDP headers.

Sessions support two primary operations:

push(session, message)
pop(session, message)

The first is invoked by a high-level session to pass a message down to some low-level session. The second is invoked by the **demux** operation of a protocol to pass a message up to one of its sessions. Figure 2 schematically depicts a session, denoted s_q^p, that is in protocol q's class and was created—either directly via **open** or indirectly via **open_enable** and **open_done**—by protocol p. Dotted edges mark the path a message travels from a user process down to a network device and solid edges mark the path a message travels from a network device up to a user process.

2.4 Message Objects

Conceptually, messages are active objects. They either arrive at the bottom of the kernel (i.e., at a device) and flow upward to a user process, or they arrive at the top of the kernel (i.e., a user process

name as an index to find a pointer to the procedure that implements the **op** for that object, and calls that procedure with the arguments $\mathbf{arg}_2, \ldots, \mathbf{arg}_n$. In certain cases, $\mathbf{arg}_1$ is also passed to the procedure. This is necessary when the procedure needs to know what object it is currently operating on.

2.2 Protocol Objects

Protocol objects serve two major functions: they create session objects and they demultiplex messages received from the network to one of their session objects. A protocol object supports three operations for creating session objects:

session = open(protocol, invoking_protocol, participant_set)
open_enable(protocol, invoking_protocol, participant_set)
session = open_done(protocol, invoking_protocol, participant_set)

Intuitively, a high-level protocol invokes a low-level protocol's **open** operation to create a session; that session is said to be in the low-level protocol's class and created on behalf of the high-level protocol.[2] Each protocol object is given a capability for the low-level protocols upon which it depends at configuration time. The capability for the invoking protocol passed to the **open** operation serves as the newly created session's handle on that protocol. In the case of **open_enable**, the high-level protocol passes a capability for itself to a low-level protocol. At some future time, the latter protocol invokes the former protocol's **open_done** operation to inform the high-level protocol that it has created a session on its behalf. Thus, the first operation supports session creation triggered by a user process (an *active* open), while the second and third operations, taken together, support session creation triggered by a message arriving from the network (a *passive* open).

The **participant_set** argument to all three operations identifies the set of participants that are to communicate via the created session. By convention, the first element of that set is the local participant. In the case of **open** and **open_done**, all members of the participant set must be given. In contrast, not all the participants need be specified when **open_enable** is invoked, although an identifier for the local participant must be present. Participants identify themselves and their peers with host addresses, port numbers, protocol numbers, and so on; these identifiers are called *external ids*. Each protocol object's **open** and **open_enable** operations use the map routines to save bindings of these external ids to capabilities for session objects (in the case of **open**) and protocol objects (in the case of **open_enable**). Such capabilities for operating system objects are known as *internal ids*.

Consider, for example, a high-level protocol object p that depends on a low-level protocol object q. Suppose p invokes q's **open** operation with the participant set $\{local\_port, remote\_port\}$. p might do this because some higher-level protocol had invoked its **open** operation. The implementation of q's **open** would initialize a new session s and save the binding $\langle local\_port, remote\_port \rangle \longrightarrow s$ in a map. Similarly, should p invoke q's **open_enable** operation with the singleton participant set $\{local\_port\}$, q's implementation of **open_enable** would save the binding $local\_port \longrightarrow p$ in a map.[3]

[2] We use the Smalltalk notion of classes: a protocol corresponds to a class and a session corresponds to an instance of that class [12].

[3] For simplicity, we use p both to refer to a particular protocol and to denote a capability for that protocol. Similarly, we use s both to refer to a particular session and to denote a capability for that session.

protocols. These routines are described in more detail in a companion paper [14]. First, a set of *buffer manager* routines provide for allocating buffer space, concatenating two buffers, breaking a buffer into two separate buffers, and truncating the left or right end of a buffer. The buffer routines use the heap storage area and are implemented in way that allows multiple references to arbitrary pieces of a given buffer without incurring any data copying. The buffer routines are used to manipulate messages; i.e., add and strip headers, fragment and reassemble messages. Second, a set of *map manager* routines provide a facility for maintaining a set of bindings of one identifier to another. The map routines support adding new bindings to the set, removing bindings from the set, and mapping one identifier into another relative to a set of bindings. Protocol implementations use these routines to translate identifiers extracted from messages headers—e.g., addresses, port numbers—into capabilities for kernel objects. Third, a set of *event manager* routines provides an alarm clock facility. The event manager lets a protocol specify a timer event as a procedure that is to be called at some future time. By registering a procedure with the event manager, protocols are able to *timeout* and act on messages that have not been acknowledged.

Finally, the *x*-kernel provides an infrastructure that supports communication objects. Although the *x*-kernel is written in C, the infrastructure enforces a minimal object-oriented style on protocol and session objects, that is, each object supports a uniform set of operatoins. The relationships between communication objects—i.e., which protocols depend on which others—are defined using either a simple textual graph description language or a X-windows based graph editor. A composition tool reads this graph and generates C code that creates and initializes the protocols in a "bottom-up" order.

Each *x*-kernel protocol is implemented as a collection of C source files. These files implement both the operations on the protocol object that represents the protocol, and the operations on its associated session objects. Each operation is implemented as a C function.

Both protocols and sessions are represented using heap allocated data structures that contain state (data) specific to the object and an array of pointers to the functions that implement the operations on the object. Protocol objects are created and initialized at kernel boot time. When a protocol is initialized, it is given a capability for each protocol on which it depends, as defined by the graph. Data global to the protocol—e.g., unused port numbers, the local host address, capabilities for other protocols on which this one depends—is contained in the protocol state. Because sessions represent connections, they are created and destroyed when connections are established and terminated. The session-specific state includes capabilities for other session and protocol objects as well as whatever state is necessary to implement the state machine associated with a connection.

So that the top-most kernel protocols need not be aware that they lie adjacent to the user/kernel boundary, user processes are required to masquerade as protocol objects; i.e., the user must export those operations that a protocol or session may invoke on a protocol located above it in the graph. A **create_protocol** operation allows a user process to create a protocol object representing the user process; the function pointers in this protocol object refer to procedures implemented in the user's address space. The user process uses the protocol object returned by **create_protocol** to identify itself in subsequent calls to the kernel protocols.

The *x*-kernel infrastructure also provides interface operations that simplify the invocation of operations on protocol and session objects. Specifically, for each operation invocation **op(arg**$_1$**, arg**$_2$**, . . . ,arg**$_n$**)** defined in the rest of this section, the infrastructure uses **arg**$_1$ as a capability for an object, uses the operation

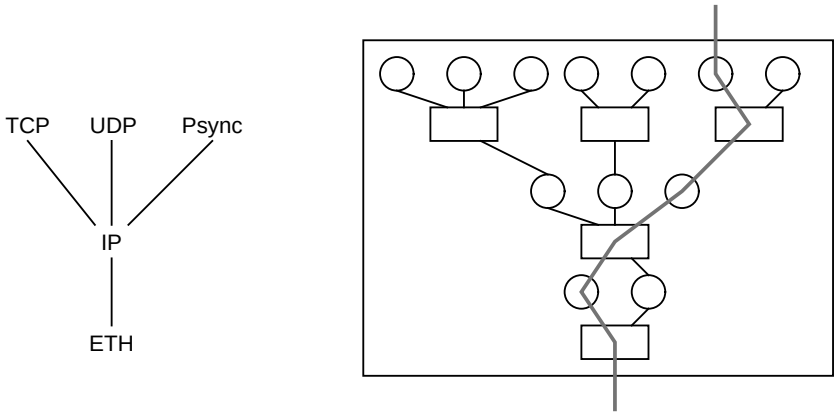

Figure 1: Example *x*-Kernel Configuration

processes running in a given address space share the same user and kernel areas; each process has a private stack in the stack area. All address spaces share the same kernel area. The user and kernel areas of each address space contain code, static data, and dynamic data (heap). Each process' private stack is divided into a user stack and a kernel stack. Processes within an address space synchronize using kernel semaphores. Processes in different address spaces communicate in the same way as do processes on different machines—by exchanging messages through one or more of the kernel's protocol objects.

A process may execute in either user or kernel mode. When a process executes in user mode, it is called a user process and it uses the code and data in the user area and its user stack. Likewise, when a process executes in kernel mode, it is called a kernel process and it uses the code and data in the kernel area and its kernel stack. A kernel process has access to both the kernel and user areas, while a user process has access to only the user area; the kernel area is protected by the memory management hardware.

The *x*-kernel is symmetric in the sense that a process executing in user mode is allowed to change to kernel mode (this corresponds to a system call) and a process executing in kernel mode is allowed to invoke a user-level procedure (this is an *upcall* [10]). When a user process invokes a kernel system call, a hardware trap occurs, causing the process to start executing in the kernel area and using its kernel stack. When a kernel process invokes a user-level procedure, it first executes a preamble routine that sets up an initial activation record in the user stack, pushes the arguments to the procedure onto the user stack, and starts using the user stack; i.e., changes its stack pointer. Because there is a danger of the user procedure not returning (e.g., an infinite loop), the kernel limits the number of outstanding upcalls to each user address space.

On top of this foundation, the *x*-kernel provides a set of support routines that are used to implement

common protocol tasks. In doing so, the architecture is able to accommodate a wide variety of protocols while performing competitively with ad hoc implementations in less structured environments. This paper describes the *x*-kernel's architecture, evaluates its performance, and reports our experiences using the *x*-kernel to implement a large body of protocols.

2 Architecture

The *x*-kernel views a protocol as a specification of a communication *abstraction* through which a collection of *participants* exchange a set of *messages*. Beyond this simple model, the *x*-kernel makes few assumptions about the semantics of protocols. In particular, a given instance of a communication abstraction may be implicitly or explicitly established; the communication abstraction may or may not make guarantees about the reliable delivery of messages; the exchange of messages through the communication abstraction may be synchronous or asynchronous; an arbitrary number of participants may be involved in the communication; and messages may range from fixed sized data blocks to streams of bytes.

The *x*-kernel provides three primitive communication objects to support this model: *protocols*, *sessions*, and *messages*. We classify these objects as to whether they are static or dynamic, and passive or active. Protocol objects are both static and passive. Each protocol object corresponds to a conventional network protocol—e.g., IP [27], UDP [25], TCP [34], Psync [22]—where the relationships between protocols are defined at the time a kernel is configured.[1] Session objects are also passive, but they are dynamically created. Intuitively, a session object is an instance of a protocol object that contains a "protocol interpreter" and the data structures that represent the local state of some "network connection". Messages are active objects that move through the session and protocol objects in the kernel. The data contained in a message object corresponds to one or more protocol headers and user data.

Figure 1(a) illustrates a suite of protocols that might be configured into a given instance of the *x*-kernel. Figure 1(b) gives a schematic overview of the *x*-kernel objects corresponding to the suite of protocols in (a); protocol objects are depicted as rectangles, the session objects associated with each protocol object are depicted as circles, and a message is depicted as a "thread" that visits a sequence of protocol and session objects as it moves through the kernel.

The rest of this section sketches the *x*-kernel's underlying process and memory management facilities and describes protocol, session, and message objects in more detail. We describe the *x*-kernel's architecture only in sufficient detail to understand how the abstractions and the implementation influence each other. Toward this end, we only define the key operations on protocols, sessions and messages. Also, we give both an intuitive specification for the objects and an overview of how the objects are implemented in the underlying system.

2.1 Underlying Facilities

At the lowest level, the *x*-kernel supports multiple address spaces, each of which contains one or more light-weight processes. An address space contains a user area, a kernel area, and a stack area. All the

[1] When the distinction is important to the discussion, we use the term "protocol object" to refer to the specific *x*-kernel entity and we use the term "network protocol" to refer to the general concept.

to connection-less, synchronous to asynchronous, reliable to unreliable, stream-oriented to message-oriented, and so on. For example, to accommodate differences in different protocol layers, Berkeley Unix defines three different interfaces: driver/protocol, protocol/socket, and socket/application. As another example, System V added multiplexors to streams to accommodate the complexity of network protocols.

Not all operating systems provide explicit support for implementing network protocols. At one extreme, systems like the V-kernel [8] mandate a particular protocol or protocol suite. Because such operating systems support only a fixed set of protocols that are known *a priori*, the protocols can be embedded in the kernel without being encapsulated within a general protocol abstraction. At the other extreme, systems such as Mach [1] move responsibility for implementing protocols out of the kernel. Such systems view each protocol as an application that is implemented on top of the kernel; i.e., they provide no protocol-specific infrastructure.

In addition to defining the abstract objects that make up an operating system, one must organize those objects into a coherent system that supports the necessary interaction between objects. More concretely, one must map the abstractions onto processes and procedures. One well-established design technique is to arrange the objects in a functional hierarchy [11]. Such a structure extends nicely to communication objects because protocols are already defined in terms of multiple layers. It has been observed, however, that the cost of communication between levels in the hierarchy strongly influences the performance of the system [13]. It is therefore argued that while the design of an operating system may be hierarchical, performance concerns often dictate that the implementation is not.

More recent studies have focused on how the structure of operating systems influences the implementation of protocols [9, 10]. These studies point out that encapsulating each protocol layer in a process leads to an inefficient implementation because of the large overhead involved in communication and synchronization between layers. They also suggest an organization that groups modules into vertical and horizontal tasks, where modules within a vertical task interact by procedure call and modules within a horizontal task interact using some process synchronization mechanism. Moreover, allowing modules within a vertical task to call both lower-level and higher-level modules is well-suited to the bi-directional nature of network communication.

While the way protocol modules are mapped onto procedures and processes clearly impacts the performance of protocol implementations, studies also show that protocol performance is influenced by several additional factors, including the size of each protocol's packet, the flow control algorithm used by the protocol, the underlying buffer management scheme, the overhead involved in parsing headers, and various hardware limitations [36, 16, 5]. In addition to suggesting guidelines for designing new protocols and proposing hardware designs that support efficient implementations, these studies also make the point that providing the right primitives in the operating system plays a major role in being able to implement protocols efficiently. An important example of such operating system support is a buffer management scheme that allows protocol implementations to avoid unnecessary data copying. In general, it is desirable to recognize tasks common to many protocols, and to provide efficient support routines that can be applied to those tasks.

The novel aspect of the *x*-kernel is that it fully integrates these three ingredients: it defines a uniform set of abstractions for encapsulating protocols, it structures the abstractions in a way that makes the most common patterns of interaction efficient, and it supports primitive routines that are applied to

The *x*-Kernel: An Architecture for Implementing Network Protocols

Norman C. Hutchinson and Larry L. Peterson*

Abstract

This paper describes a new operating system kernel, called the *x*-kernel, that provides an explicit architecture for constructing and composing network protocols. Our experience implementing and evaluating several protocols in the *x*-kernel shows that this architecture is both general enough to accommodate a wide range of protocols, yet efficient enough to perform competitively with less structured operating systems.

1 Introduction

Network software is at the heart of any distributed system. It manages the communication hardware that connects the processors in the system and it defines abstractions through which processes running on those processors exchange messages. Network software is extremely complex: it must hide the details of the underlying hardware, recover from transmission failures, ensure that messages are delivered to the application processes in the appropriate order, and manage the encoding and decoding of data. To help manage this complexity, network software is divided into multiple layers—commonly called protocols—each of which is responsible for some aspect of the communication. Typically, a system's protocols are implemented in the operating system kernel on each processor.

This paper describes a new operating system kernel—called the *x*-kernel—that is designed to facilitate the implementation of efficient communication protocols. The *x*-kernel runs on Sun3 workstations, is configurable, supports multiple address spaces and light-weight processes, and provides an architecture for implementing and composing network protocols. We have used the *x*-kernel as a vehicle for experimenting with the decomposition of large protocols into primitive building block pieces [15], as a workbench for designing and evaluating new protocols [22], and as a platform for accessing heterogeneous collections of network services [23].

Many operating systems support abstractions for encapsulating protocols; examples include Berkeley Unix *sockets* [17] and System V Unix *streams* [28]. Such abstractions are useful because they provide a common interface to a collection of dissimilar protocols, thereby simplifying the task of composing protocols. Defining truly general abstractions is difficult because protocols range from connection-oriented

*Authors' Address: Department of Computer Science, University of Arizona, Tucson, AZ 85721. This work supported in part by National Science Foundation Grant CCR-8811423.