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

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

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

[28] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, winter 1988.

[29] J. Purtilo. Polylith: An environment to support management of tool interfaces. In *Proceedings of the ACM SIGPLAN Symposium on Language Issues in Programming Environments*, pages 12–18, July 1985.

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

[31] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot. Sos: An object-oriented operating system—assessment and perspectives. *Computer Systems*, 2(4):287–338, Dec. 1989.

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

[33] A. Tanenbaum. *Computer Networks*. Prentice-Hall, 1981.

[34] D. L. Tennenhouse. Layered multiplexing considered harmful. In *Proceedings of the 1st International Workshop on High-Speed Networks*, Nov. 1989.

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

[36] 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.

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

[38] H. Zimmermann. Osi reference model—the iso model of architecture for open systems interconnection. *IEEE Transactions on Communications*, COM-28(4):425–432, Apr. 1980.

[39] W. Zwaenepoel. Protocols for large data transfers over local area networks. In *Proceedings of the 9th Data Communications Symposium*, pages 22–32, Sept. 1985.

[9] 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.

[10] D. R. Cheriton. Exploiting recursion to simplify RPC communication architectures. In *Proceedings of the SIGCOMM '88 Symposium*, pages 76–87, Aug. 1988.

[11] 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.

[12] 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.

[13] D. D. Clark. The design philosophy of the DARPA Internet protocols. In *Proceedings of the SIGCOMM '88 Symposium*, pages 106–114, Aug. 1988.

[14] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Symposium*, pages 200–208, Sept. 1990.

[15] J. Crowcroft, Z. Wang, I. Wakemane, and D. Sirovica. Layering considered harmful. *IEEE Networks*, Jan. 1992.

[16] D. Hernek and D. Anderson. Efficient automated protocol implementation using RTAG. Technical Report UCB/CSD 89/?, University of California at Berekely, Aug. 1988.

[17] N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[18] V. Jacobson. Congestion avoidance and control. In *Proceedings of the SIGCOMM '88 Symposium*, pages 314–32, Aug. 1988.

[19] S. Lam and A. Shankar. Specifying modules to satisfy interfaces: A state transition system approach. Technical Report TR-88-30, The University of Texas as Austin, Aug. 1989.

[20] M. Makpangou. *Protocols de Communications et Programmation par Objects: L'Example de SOS*. PhD thesis, L'Universite de Paris-VI, Feb. 1989.

[21] S. Mishra, L. L. Peterson, and R. D. Schlichting. Implementing fault-tolerant replicated objects using Psync. In *Eighth Symposium on Reliable Distributed Systems*, pages 42–52, Oct. 1989.

[22] S. Mishra, L. L. Peterson, and R. D. Schlichting. A membership protocol based on partial order. In *Second Working Conference on Dependable Computing for Critical Applications*, Feb. 1990.

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

[24] 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.

monolithic protocol. Even though there exist examples that illustrate how layering can perform poorly—e.g., [15]—these examples say nothing about the *intrinsic* cost of layering; they merely serve to illustrate how poor protocol design can adverely affect performance.

Moreover, modular design has several advantages. The first advantage is that modularity promotes reuse, which in turn makes communication services easier to implement. Although difficult to quantify, our experiences programming the communication services illustrated in Section 4 suggest this advantage is real. The second advantage is that application programmers are able to configure communication services that precisely meet their needs. The alternative, which happens quite frequently in practice, is that programmers settle for the protocol that comes the closest to providing the right level of service. The final advantage is that the architecture makes it easier to both adapt to changing network technologies and dynamically select the right packet delivery vehicle for each message. The alternative is to extend a single protocol to work optimally in all circumstances.

Acknowledgements

Norm Hutchinson is a co-designer of the *x*-kernel, and provided valuable feedback on this work. Mark Abbott implemented Sprite RPC in the *x*-kernel, Chien-Chang Lao implemented the RRS protocol described in Section 4.2, and Shivakant Mishra implemented the fault-tolerant multicast service described in Section 4.3.

References

[1] D. Anderson. *A Grammar-Based Methodology for Protocol Specification and Implementation*. PhD thesis, University of Wisconsin - Madison, Aug. 1985.

[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] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, Feb. 1990.

[4] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, Feb. 1987.

[5] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.

[6] R. T. Braden. Transactional TCP. Request for Comments XXXX, USC-ISI, 1991.

[7] J. B. Carter and W. Zwaenepoel. Optimistic bulk data transfer protocols. In *Sigmetrics 89 and Performance 89*, May 1989.

[8] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, Aug. 1984.

are numerous software engineering environments that attempt to perform the same task. Our approach, however has been to assist with the manual construction of network software. Therefore, we draw most of our inspiration from the more pragmatic side of software engineering, concentrating on program encapsulation, reusability and composibility.

The Unix programming environment is the most successful example of a programming environment that supports program composition. Unix's model is extremely simple and effective—all programs that read from **stdin** and write to **stdout** can be composed linearly.[7] Unix defines what a successful environment for program composition is. There have been many other attempts to promote program reuse through program composition, but none of them has come close to the level of practicality provided by Unix. The ultimate goal of our work it to make the composition of protocols as simple as Unix makes the composition of programs.

Our methodology also attempts to apply the techniques developed for the multi-language programming system POLYLITH [29] to protocol development. The primary goal of POLYLITH is to encapsulate procedures written separately in different languages so that they may be composed together into a single working program. POLYLITH was the primary influence for much of our support for encapsulation and composition. However, POLYLITH suffers from two problems that are unacceptable in our context. First, encapsulation adversely affects the performance of the composite system. In multi-language programming, the goal is to allow one language to call another and the performance of that call is not of great importance. Second, POLYLITH's support for composition is not nearly as successful as that provided by Unix. The interfaces of two arbitrarily designed protocols are unlikely to be compatible. The key insight about multi-language programming environments gained from this work is that communication protocols are much more amenable to the techniques developed for such environments than multi-language programming itself. Encapsulating communications protocols is much simpler than encapsulating arbitrary programs, and protocols also tend to compose more naturally than do arbitrary procedures.

8 Concluding Remarks

This paper proposes a new approach for designing and implementing network software that is based on a dynamic architecture. Our approach exploits micro-protocols that are generated using common software engineering techniques, virtual protocols, and a meta-protocol. The main conclusion that we draw from this work is that conventional wisdom—which argues that one should be able to use layering to describe (think about) communication software, but that one should not make the implementation strictly adhere to layering—is at best overstated, and at worst simply wrong. Protocol layers should be explicitly preserved in the implementation and dynamically selected based on the semantics required by the application and the characteristics of the underlying network. In other words, the answer to the question 'What is the right number of layers?' is 'It depends on the message.'

We have implemented several different communication services using the proposed architecture. Experiments with these services indicate that a dynamic architecture consisting of a large number of micro-protocols and virtual protocols performs competitively with static architectures consisting of a single

[7] Reading from **stdin** and writing to **stdout** is analogous to our meta-protocol.

kernel's support is not optimal is that protocol and session objects are implemented using procedure calls. It seems that significant improvement in the minimum cost of layering will require a compiled protocol implementation that generates code to perform protocol invocation.

7 Related Work

The problem of how best to implement network software has been an issue in the networking community at least since the early 1980s [11]. Most of the work in this area discusses the problems encountered in implementing existing network software using existing operating systems, whereas we consider new protocol designs facilitated by a new operating system. This section briefly outlines some of this work.

Unix System V Streams [30] is the best example of an operating system mechanism that successfully supports composible modules, in this case filters for processing terminal I/O. The stream mechanism was later extended to support network protocols, but no attempt has been made to develop flexibly layered network protocols analogous to the original character stream modules. Experimenting with the architecture described in this paper on System V Streams would be an interesting way to compare the *x*-kernel with Streams, but is orthogonal to this paper.

RTAG [1, 16] is a protocol development system that represents communications protocols as attributed grammars; there are many similar systems that use finite state machines. The grammar defines the content and legal ordering of messages exchanged by a protocol. RTAG decomposes communications protocols into productions, where each production is itself a protocol. RTAG supports a model of protocol decomposition more radical than the one we propose. RTAG represents each protocol (or production) as a lightweight process. This drastically limits the performance potential of RTAG protocols. Unlike our approach, decomposing a protocol into RTAG productions results in a protocol that is compatible with the original. While this is obviously a plus for maintaining backward compatibility, it restricts the potential of the RTAG system. As in System V streams, RTAG research has concentrated upon finding efficient implementations for existing protocols rather than developing new protocols that fully exploit RTAG.

Our use of virtual protocols to bypass unnecessary functionality is similar to the way the Synthesis kernel collapses layers [28]. Unlike the Synthesis kernel, however, we do not generate code at runtime and we have the ability to dynamically select between lower layers on a per-message messages rather than just a per-connection basis.

The SOS operating system developed at INRIA [31] is an object-oriented operating system that includes some work in object-oriented protocols [20]. While SOS was primarily concerned with the development of distributed objects—objects spread over multiple machines—SOS has also developed a new object-oriented protocol suite. SOS protocols bear some surface resemblance to our protocols, but they are much larger. Unlike RTAG or our approach, SOS protocols do not possess the property of self-decomposition—i.e., SOS does not decompose large protocols into smaller protocols.

Much of the novelty of our approach is that it attempts to apply the techniques of software engineering to the problem of designing and implementing network software. What is even more novel is the areas of software engineering that provide the basis of our approach. Most protocol development environments are highly formal systems designed to generate a protocol from an abstract specification [19]. There

packet size. For small records—1k-bytes or under—BLAST is not used, and RRS achieves the same throughput as TCP. For larger records, the BLAST protocol becomes increasingly important.

A key factor in using a sliding window protocol on top of a blast protocol concerns timeouts—RRS deals with logical records, but it takes longer to transmit a 16k-byte record than a 1k-byte record. If RRS times out too soon, it will attempt to retransmit a record at the same time as BLAST is trying to send the original copy. CHANNEL has the same problem when it uses BLAST to send large messages. Clearly, both RRS and CHANNEL must compute a timeout as a function of the record size. This is easy on an ethernet, where the performance of BLAST is very predictable, but it is a difficult problem in an internet, where round trip delays are highly variable. We have successfully experimented with protocols that timeout on top of BLAST in an internet setting, and have been able to set the high-level timer as a function of the low-level timer.

6.3 Incremental Costs

Finally, we measure the cost of individual layers using our architecture. We found that on a Sun 3/75, static conditional protocols are free, hybrid and dynamic conditional protocols cost approximately 50 μsec, trivial micro-protocols (e.g., SELECT) cost approximately 100 μsec, and more complex micro-protocols (e.g., CHANNEL, RRS, and PSYNC) cost from 500 to 1000 μsec. Note that these costs represent the round trip latency of each protocol, which corresponds to the time it takes a message to traverse that layer four times, twice going up and twice going down. However, the full impact of invoking a layer does not show up in the latency numbers. This is because unwinding the call stack happens in parallel with the transmission of the message.

Although we find the cost of layering to be acceptable, it is still the case that layering provides a great opportunity for having disastrous performance; a single mistake repeated at multiple layers can quickly lead to unacceptable performance. In fact, many of the features of the *x*-kernel outlined in Section 2 were motived by early failures with extensive layering. For example, we found that unnecessarily establishing and freeing state information at each level degrades performance. This problem is avoided by caching open sessions at all levels. As another example, the cost of manipulating message headers can be significant. In an earlier version of the *x*-kernel, we used a buffer management scheme that allocated a buffer for each new header added to a message. In contrast, the current version pre-allocates a single buffer that is large enough to hold all the headers and simply adjusts a pointer for each new header. The original approach resulted in a 500 μsec minimum cost for each layer, whereas the current approach has a minimum cost of 100 μsec per layer. Finally, we believe that the most important factor in achieving good performance of layering is associating processes with messages instead of with protocols.

One question that needs to be addressed is how well a highly layered communication service could run given the optimal platform support. A rough lower bound for the cost of a round trip traversal of a protocol layer that simply reads and writes a header is the cost of two demultiplex operations and two null procedure calls that take the header as argument. The null procedure calls represent the addition and removal of the headers from the message. On a Sun 3/75, the cost of these four operations is approximately 40 μsec, while the minimum cost of traversing a trivial protocol is 100 μsec. Therefore, the *x*-kernel is within approximately a factor of two of the optimal performance. The reason the *x*-

Amoeba RPC represents what is probably the limits of what can be done to improve protocol performance by tuning a monolithic protocol to a special purpose implementation [36]. Amoeba RPC takes 1.4 milliseconds in the user-to-user case. One reason that even the monolithic RPC implementation in the *x*-kernel is slower than Amoeba RPC is that the *x*-kernel dispatches a process for each incoming message; costing 270 μsec over the round trip. However, this process provides the machinery necessary to support efficient layering, and as the process creation cost gets amortized over more layers, the per-layer performance improves. Also note that implementing protocols that perform like Amoeba RPC is very difficult, and this level of performance is rarely achieved in practice. We believe that while the potential performance of monolithic protocols running on special purpose platforms may be somewhat better than the potential performance of layered protocols on generic platforms, the actual performance of a more modular communication system is often superior. This is because layered implementations require less skill, and as such, are more likely to be implemented efficiently. This is especially true when well implemented micro-protocols are reused.

We also measured the latency of LRPC—our light-weight RPC protocol that delivers messages between processes in different address spaces on the same machine. Our LRPC protocol was able to achieve 308 μsec latency on a Sun 3/75. By comparison, LRPC developed at the University of Washington and the DEC Systems Research Center (both running on the Dec Firefly, a MicroVax-based multiprocessor) report 157 μsec and 464 μsec latency, respectively. Our LRPC protocol took only a couple of hours to write, and achieves remarkably good performance. This is because of the *x*-kernel's process-per-message architecture: the caller's process enters the kernel, gets demultiplexed to the user's address space, executes the procedure, and returns to the caller's address space without blocking.

6.2 Stream-Oriented Protocols

We next consider the performance of a protocol graph consisting of TCP, IP, and ETH, and the protocol graph illustrated in Figure 10. Both protocol graphs were implemented in the *x*-kernel. We measured the latency of the two graphs to be 3.30 msec and 3.00 msec, respectively. The latter graph is a bit faster because it avoids the cost of IP.

The throughput performance of the two graphs is more interesting. The throughput tests involve sending a large amount of data—ranging from 1M-bytes to 16M-bytes—from one user process to another. In the 1M-byte case, the sending process writes 1024 1k-byte records (blocks of data), while in the 16M-byte case, the sending process writes 1024 16k-byte records (blocks of data). As one would expect, TCP achieves the same throughput rate—approximately 400 kbytes/sec—independent of the size of the record written by the sending process. This is because TCP does not preserve record boundaries. Instead, it fragments all data blocks into the same sized packets, and in this case, always had a backlog of packets to send.

In contrast, the protocol graph depicted in Figure 10 achieves a throughput rate ranging from a little over 400 kbytes/sec to nearly 700 kbytes/sec, depending on the record size. This is because that protocol graph includes two different flow control protocols—RRS uses the sliding window protocol to manage the transmission of complete records, and BLAST (which is hidden in the VADDR subgraph) uses the selective retransmission blast algorithm to transmit individual records that are larger than the network

The latency tests measure the round trip delay for invoking a null procedure with null request and reply messages, and the throughput tests measure the round trip delay for invoking a null procedure with a large request message (16k-bytes) and a null reply message. The performance of the first four RPC protocols are measured kernel-to-kernel and the last two are measured user-to-user. Sprite RPC implemented on the Sprite kernel numbers are taken from the literature; the other numbers were all measured at the University of Arizona.

Protocol	Sprite RPC	Sprite RPC	Figure 6	Figure 8	Sun RPC	Sun RPC
Platform	Sprite Kernel	*x*-kernel	*x*-kernel	*x*-kernel	Unix	*x*-kernel
Kernel vs. User	K-K	K-K	K-K	K-K	U-U	U-U
Layers	2	2	7	11	4	4
Latency (msec)	2.6	1.79	1.78	2.35	12.0	5.2
Throughput (kbytes/sec)	700	860	840	820	350	490

Table 2: RPC Protocol Performance

The most important conclusion to draw from these experiments is that the fully decomposed version performs better than the native implementations of both Sprite RPC and Sun RPC. This is in spite of the fact that each message has to traverse significantly more layers. Protocol modularity, or lack thereof, seems to be only one of many factors that influence the performance of network protocols. In both the Sun RPC case, and to a lesser extent, the Sprite RPC case, these other factors dominated any performance penalty incurred through increased modularity.

When comparing Sprite RPC implemented on the *x*-kernel with the fully decomposition version, it is clear that modularity does inflict a performance penalty, 31% in this case. However, the performance of the moderately decomposed version shows that a less radical decomposition of a monolithic protocol can have comparable performance to the original version. As is the case with traditional programming, the extent that modularity or layering should be sacrificed for performance is up to the implementor.

Modularity seems to have little effect on throughput, although the throughput figures have more to do with the limitations of the Ethernet controller than any software concerns. The numbers are so close to the effective limit of the Ethernet controller we are using that more conclusive tests will require faster networks. We expect, however, that the throughput performance of highly layered protocol graphs will remain competitive with monolithic implementations. This is because each fragment of a large message traverses only a couple of protocols near the bottom of the graph; it does not visit all the protocols in the graph. Also note that while the throughput figures for the first four columns of Table 2 are kernel-to-kernel, padding headers out to a fixed length allows arbitrarily layered protocol graphs to take advantage of optimistic blast algorithms developed to support monolithic protocols [7 , 23]. Using such algorithms, we have achieved user-to-user throughput rates that are within 90% of the kernel-to-kernel throughput. Also note that if we remove the PAD protocol—whose only purpose is to facilitate good user-to-user throughput—then the kernel-to-kernel latency numbers for the fully decomposed RPC is 2.06 msec, which is only a 15% performance penalty.

to VRPC. VRPC would decide to route each message to either the backward compatible protocol or to one of the other branches based on the destination host's address. That is, VRPC would use the backward compatible branch if it did not know the remote host to be running the new protocol graph. Hosts that are running the new graph could inform each other of this fact using a broadcast-based "graph information" protocol. Alternatively, a name server could report what protocols a particular host is running.

Third, one can use virtual protocols to distinguish between protocols that are tuned for different network technologies. Ignoring for a moment the possibility of suite of micro-protocols that implement the functionality of TCP, consider a protocol graph rooted at a virtual protocol VTCP, with branches to three different versions of TCP: one tuned for a low-utilization, low-latency ethernets; one tuned for congestion-prone, high-latency wide-area networks; and one tuned for high-bandwidth, high-latency wide-area networks. In this case, the virtual protocol can select the right implementation based on the destination address, and in fact, the three versions of TCP need not even interoperate (e.g., the high-bandwidth, high-latency version might use a 32-bit window instead of a 16-bit window). It is our contention that this would be more desirable than trying to engineer a single TCP that performs optimally over all network technologies.

6 Performance

This section reports a set of experiments designed to evaluate the performance of protocols implemented using the proposed architecture. The experiments measure the relative performance of two different communication services—the remote procedure call service and the stream-oriented service—using both conventional protocol graphs and protocol graphs constructed using our methodology. We have also implemented the fault-tolerant multicast service, but examining its performance is not germane to the discussion because we do not have a monolithic implementation against which a meaningful comparison can be made.

The results of the experiments come from two sources: tests run locally at the University of Arizona, and performance figures from the literature. All tests run at the University of Arizona used the same experimental technique: messages were exchanged between a pair of Sun 3/75s connected by an isolated 10 megabit/second Ethernet; the protocols and the operating system were all compiled using the SunOS version 4.0 C compiler; and the reported numbers are generated by running the test case 10,000 times and dividing the total elapsed time by 10,000.

6.1 RPC Protocols

Table 2 compares the latency and throughput for six different RPC protocols: two interoperable versions of Sprite RPC (one implemented in the Sprite kernel and one implemented in the *x*-kernel),[6] two layered versions of Sprite RPC (the moderately decomposed version corresponds to Figure 6 and the fully decomposed version corresponds to Figure 8), and two interoperable versions of Sun RPC [32] (one implemented on Unix and one implemented in the *x*-kernel).

[6]Unlike the protocol graph given in Figure 2, these two implementations are directly on top of the ethernet, not IP.

The basic difference between our architecture and existing architectures is that we attempt to know a little about all protocols—including those that have not yet been written—while existing architectures attempt to know everything about a small, fixed set of protocols. Our technique provides all of the information needed to support a coherent network architecture without precluding changes to individual protocols or protocol suites.

5.4 Applicability

We conclude this section by considering the question of how widely our proposed architecture can be applied. The question of applicability has two dimensions: how widely the approach can be applied in principle, and how widely it can be applied in practice. Consider each dimension, in turn.

In principle, we believe our approach can be applied to the entire communication subsystem. As represented by the four examples given in this and the previous section, we have implemented micro-protocols that correspond to the core functions found in almost all protocols, including request/reply, sliding window, demultiplexing, fragmentation/reassembly, routing, stop-and-wait, data encryption, data encoding, and so on. While it is true that any given protocol specification can become arbitrarily complex—and therefore may not seem amenable to our approach—we believe that this complexity is an artifact of the current static approach to building network software, and that it is not inherent. Protocols often become complex either because they are designed to satisfy the needs of too many applications, or because they have been extended while preserving backwards compatibility. In both cases, the reason the additional complexity was put into a new protocol (or added to an existing protocol) is to avoid the difficulty in modifying a static network architecture. In contrast, the fundamental insight of this work is that the functionality found in communication systems is relatively simple, and that a dynamic architecture allows applications to take advantage of this simplicity. Others have recognized and are trying to exploit this same insight [14].

Consider a complex protocol like TCP. There have recently been several proposals to extend TCP, including an optimization to better support request/reply traffic [6], even though such traffic would be more obviously accommodated by an RPC protocol. One argument made in support of such an extension is that TCP already provides a congestion control mechanism; i.e., the TCP slow start algorithm [18]. It is our contention, however, that the slow start algorithm is an ideal candidate for encapsulation in a self-contained protocol. This would allow a variety of transport services to take advantage of a good algorithm without having to redesign it from scratch.

In practice, we envision three ways in which our network architecture can "co-exist" with current architectures. First, the approach can be applied to the design of new protocols that augment today's network architectures, especially at the transport level and above. In fact, because of the limited functionality and static nature of today's network architectures, application programmers typically write their own protocols outside the network architecture proper. These programmers can apply our approach outside of the rest of the architecture.

Second, it is possible for a given distributed system to adopt our approach internally, and still interoperate with rest of the world using existing protocols. For example, consider the protocol graph illustrated in Figure 8. One could augment this graph by attaching an existing (backward compatible) RPC protocol

Decide	Act	Category	Examples
open	open	static	VSYM, VRPC
open	send	hybrid	VADDR, VBP, VSR
send	open	impossible	
send	send	dynamic	VSIZE, VFIFO, VROUND

Table 1: A Taxonomy of Virtual protocols

performed upon VADDR always returns an IP address, even if IP is not being used as the network protocol.

5.3 Meta-Protocol

The dynamic nature of protocols in our architecture presents problems that are not found in traditional static network architectures. In existing architectures, the semantics of the protocols and the structure of the graph is fixed, and all protocols can take advantage of the absolute knowledge of what protocols are above and below them. In contrast, our protocols have no *a priori* knowledge about the protocols with which they will be composed. If a protocol makes an assumption about a neighboring protocol, the assumption must hold for all protocols that might be composed with it. Therefore, we have defined a *meta-protocol* that defines a set of properties that all protocols must possess.

The primary purpose of the meta-protocol is to ensure that incoming messages traverse the protocol graph correctly. In traditional network architectures, each protocol defines its own demultiplexing strategy. For example, IP identifies each protocol that uses it with an 8-bit protocol id and the Ethernet identifies each protocol that uses it with a 16-bit type field. Such relative identification makes arbitrary composition difficult, if not impossible. In contrast, the meta-protocol enforces a single dynamic demultiplexing scheme, which is defined as follows:

- Each protocol is identified by a unique system-wide 32-bit identifier.
- Each protocol's header begins with its unique identifier.
- Each protocol inspects the next protocol's header to get its id, and then demultiplexes to this id.
- All protocols without headers (i.e., virtual protocols) "lie" about their protocol id, and claim to be the protocol that opened them.
- A special protocol id (0) is added to all messages as they cross the user/kernel boundary.
- Protocols that fragment messages must duplicate the high level protocol's identifier at the start of each non-initial fragment.

If all protocols follow these rules, then messages can be demulitplexed through arbitrarily complex protocol graphs with the assurance that they will reach there correct destination. The meta-protocol also requires all protocols to provide a function that returns a formatted version of its header. This rule facilitates protocol-independent debugging tools such as a protocol stack tracer.

bottom, branching out into disjoint (linearly composed) protocol stacks.

5.1.5 Reuse

Our experience using the architecture suggests that it provides an effective platform for protocol reuse. We believe that micro-protocols are especially suitable to reuse because there seem to be a small number of algorithms that appear over and over in various communication systems; e.g., sliding window, blast with selective retransmission, request/reply, and so on. This characteristic seems to be so important that even when we failed to encapsulate or under-specify a protocol correctly, we were still able to enjoy the benefits of *partial reuse*—creating a new protocol through the simple modification of an existing protocol. In our experience, there have been many cases where an existing micro-protocol was modified in a matter of hours to suit the needs of another application. Finally, even in those cases where we had to implement a micro-protocol from scratch, we found micro-protocols significantly easier to write, debug and optimize than traditional protocols. This is because micro-protocols are regular, allowing a collection of implementation techniques to be reused in nearly all protocols. In fact, we have incorporated many such implementation techniques into the operating system infrastructure.

5.2 Virtual protocols

Virtual protocols are the most novel feature of our architecture, and they play critical role in being able to dynamically select the right path through the protocol graph. While micro-protocols can be understood as very small traditional network protocols, virtual protocols are unique to our approach. The basic idea is to think of the protocol as a programming abstraction. Virtual protocols represent this concept taken to is logical extreme—the protocol as a programming language statement. Specifically, virtual protocols remove IF statements from the protocol implementation and place them in the protocol graph. Virtual protocols do not define headers because they do not exchange information with their peers, and therefore are not truly protocols in the traditional sense. Because virtual protocols are used to replace IF statements, they are only practical on a system like the *x*-kernel that has a very low per-protocol overhead. From the software engineering perspective, virtual protocols facilitate non-linear composition.

Virtual protocols are so simple that they can be easily categorized into a partial taxonomy. All virtual protocols perform two functions. The first is to decide which lower level session to direct a message to. The second is to perform the act of directing the message. Both of these operations may take place at either open time or send time. Table 1 partitions virtual protocols presented so far by when these two functions take place. Dynamic virtual protocols, such as VSIZE, make all of their decisions at send time. Static virtual protocols make all of their decisions at open time. Hybrid protocols are protocols that decide at open time which lower level protocol to use, but for various reasons defer the implementation of that decision until send time.

Hybrid virtual protocols simulate the interface of one of their lower level protocols. For example, VADDR makes every attempt to look exactly like IP to all protocols that use it. Therefore, VADDR always opens IP to create an IP session , even if it plans to use ETH to send messages. This IP session is then used to service all control operations performed on VADDR. For example, a MYADDR operation

Another factor that influences the complexity of the interface—and hence the composibility of the protocols—is the amount of functionality encapsulated in each protocol. To take full advantage of this functionality, the interface must often provide "access" to the various functions. For example, VMTP [9] contains a request/reply protocol, a novel broadcast RPC protocol, a highly tuned large message transfer protocol, and a robust forwarding mechanism. Thus, VMTP is only usable in situations requiring (or at least tolerating) all of these functions. Protocols like VMTP get around this problem by allowing the user to selectively enable and disable certain parts. However, this technique simply increases the complexity of the interface and results in no net composibility gain.

One problem we uncovered in the original design of the *x*-kernel is that the single standard interface to all protocols is overly restrictive. Specifically, in synchronous protocols such as CHANNEL, each send operation returns a message, while asynchronous protocols such as IP and UDP do not. More importantly, a protocol that expects to be composed on top of a synchronous protocol cannot instead be composed on top of an asynchronous protocol. We have recently modified the *x*-kernel interface to provide explicit support for both asynchronous and synchronous protocols.

5.1.3 Under-Specification

The over-specification of most existing protocols restricts composition in spite of the use of a uniform protocol interface. An over-specified protocol is one whose interface is more specific than its semantics demand. For example, UDP [26] accepts addresses consisting of a port number and a host IP address, despite the fact that the algorithm used by UDP does need to know the host address. Thus, while the semantics of UDP permit the use of other lower-level protocols, the syntax of UDP requires IP. An under-specified version of UDP would accept addresses that contain a port number and *any* network address. If a protocol merely passes its address, or part of its address, to the protocol below it, that protocol should be polymorphic with respect to addresses.

Under-specified protocols are common in our architecture. The basic observation is that when one decomposes a protocol into many layers, there is not enough address information to go around. In the protocol graph presented in Figure 8, the only protocols with concrete address types are IP, and ETH. The address defined by C_SELECT and S_SELECT is partially polymorphic; it places the procedure number in its header and passes the remainder of the address to the next lower level protocol. C_CHAN, S_CHAN, BLAST, VSYM, VFIFO, and VSIZE are completely polymorphic with respect ot addresses because they pass the addresses passed to them to the next lower level protocol.

5.1.4 Multiple Instantiation

It has been argued that layered multiplexing—i.e., making a multiplexing decision at each of several layers—is inherently bad. This is because merging several data streams into a single data stream causes type of service information to be lost [34]. Observe, however, that the proposed architecture does not imply layered multiplexing. Although not illustrated in the examples, micro-protocols need not multiplex to multiple high level protocols. Instead, a given micro-protocol can be multiply instantiated into each data stream in which it is required, passing messages from a single high-level protocol to a single low-level protocol. In this case, the protocol graph would contain a single multiplexing protocol near the

5 Discussion

The previous two sections illustrate a new approach to designing network software through a series of examples. This section summarizes the techniques used to build a dynamic architecture, and evaluates their utility. It concludes with a discussion of the applicability of the proposed architecture.

5.1 Composibility of Micro-Protocols

A dynamic architecture requires a collection of primitive building blocks (micro-protocols) that can be composed together in various ways. We created a collection of composible micro-protocols by applying common software engineering techniques—encapsulation, interface standardization, and under-specification—to conventional communication services. This section discusses these techniques in more detail, and comments on what success and failures we have had in applying them.

5.1.1 Encapsulation

Protocol composition requires protocol encapsulation. For a protocol to be composible it must make as few assumptions as possible about other protocols in the protocol graph. Encapsulation has syntactic and semantic dimensions. Syntactic encapsulation is ensured through the use of an explicit protocol graph. Each protocol is implemented as a completely self contained object that contains no external references. A protocol accesses the protocols it depends on indirectly through pointers contained in the object, and all inter-protocol interactions take place through operations invoked upon that pointer. Changing a single pointer is all that is required to change which protocol is used.

As one might expect, semantic encapsulation is difficult to ensure; the author of a protocol implicitly makes assumptions about the other protocols in the protocol graph. Our experience suggests two things about semantic encapsulation. First, complete syntactic encapsulation results in improved semantic encapsulation. A programmer is much more likely to make arbitrary demands upon a lower-level protocol when that protocol is hard-coded into the source code. Second, even though we seldom got the encapsulation exactly right the first time, the dynamic nature of the architecture made it easy to change the protocol. That is, protocol *evolution* is a very natural process in a dynamic architecture.

5.1.2 Standard Interface

Arbitrary composition requires that the interfaces to all modules be identical. Most traditional protocols are complex enough to make them de facto non-composible. For example, the Sun RPC protocol supports over twenty protocol-specific interface routines, and any protocol that is designed to use Sun RPC may be composed only with Sun RPC, even if the protocol itself is completely encapsulated. Complex interfaces simply encode the internals of the module in the interface. Our approach avoids this problem by imposing a uniform protocol interface on all protocols and by permitting control operations to be delegated. The use of a common interface greatly reduces the severity of de facto non-composibility by eliminating gratuitous interface incompatibilities. The delegation of control operations results in a significant relaxation of composition—the interface to two adjacent protocols in the protocol graph need not be identical.

an agreement about the failure or recovery of a process among the group members [22]. RECOVER provides three services: it retrieves the state of a recovering process from the last checkpoint, it restores the recovering process' state to the current state of the system, and it incorporates the process in the group. MONITOR is a protocol that provides two basic functions: it keeps tabs on all the processes in the group and it removes a failed process from the group. Finally, P_SELECT is a variation of SELECT that dispatches each incoming message to several high-level protocols in parallel. P_SELECT is required because more than one protocol needs to see each message delivered by PSYNC.

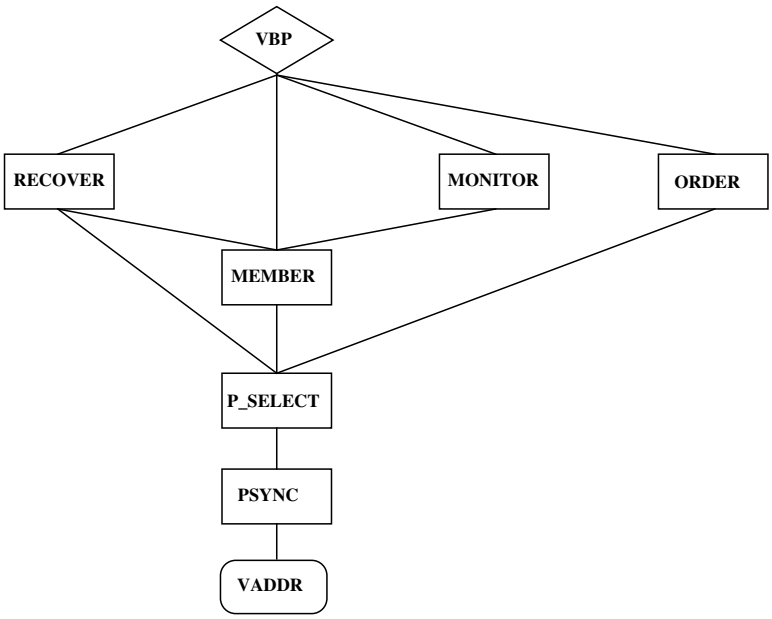

Figure 11: Fault-Tolerant Multicast Protocol Graph

Details about the algorithms used by these protocols are beyond the scope of this paper. The important point is that we have been able to successfully apply our methodology to an extremely complex communication service, and in doing so, have realized several advantages. The first advantage is that the modular design simplified the process of implementing, debugging, and optimizing each piece. The second is that the decomposition aids in finding the inherent dependencies among the pieces, and as a consequence, reduces unnecessary interference among pieces. In this case, the ORDER and MEMBER protocols can actually run in parallel. In all similar systems of which we are aware, the processing of regular messages must be suspended while the membership protocol is running. The final, and most important advantage is that the application is able to configure the protocol graph to meet its needs. For example, it can substitute one ORDER protocol for another and it can configure in (out) different recovery protocols. This flexibility is especially important when designing fault-tolerant systems because being forced to use a protocol that provides unnecessary or overly restrictive functionality can be extremely expensive.

closed. In other words, CMP uses the CHANNEL protocols three times per connection: once to open the connection and twice (once in each direction) to close the connection.

Our approach has three unique advantages. First, we are able to reuse the CHANNEL protocols developed for the RPC service. In contrast, TCP uses the three way handshake algorithm [35] to establish and terminate a connection. It is the case, however, that the request/reply algorithm exchanges exactly the same set of messages as does the three way handshake algorithm: a request from the client to the server, a reply from the server to the client, and an acknowledgment from the client to the server. Second, the state transition diagram used by CMP is significantly simpler than that used by TCP. This is because many of the substates have effectively been pushed down into C_CHAN, S_CHAN, and RRS. Finally, CMP is a general connection management protocol that can be reused by any communication service that explicitly establishes a connection.

4.3 Multicast Service

Recall the BRPC (broadcast) branch of the remote procedure call service illustrated in Figure 6. Many systems implement this service directly on the ethernet, and as a consequence, make very weak guarantees about the protocol's semantics; i.e., how many of the remote procedures will be invoked. It is often desirable to provide stronger guarantees about a broadcast service, such as those provided by atomic and reliable broadcast protocols [4, 8]. Our final example—a fault-tolerant multicast service—further supports our claim that it is useful to be able to implement a communication service using building block pieces.

It is clear from our experience that fault-tolerant multicast services are extremely difficult to implement. One source of complexity is the need to enforce a consistent ordering on the messages exchanged. A second source of complexity is the presence of failure. Mechanisms are needed to detect processor failure and recovery, to remove failed processes and incorporate recovered processes, and to restore the state of failed processes upon recovery. A final factor that adds to the complexity of such systems is that the message ordering mechanisms and failure handling mechanisms are not independent. For example, to correctly order messages, the current set of running processors must be known.

In addition to this complexity, it is often the case that different applications have different requirements. For example, some applications require that messages be totally ordered, while for others, a total order is overly restrictive. As another example, some applications assume at most one failure, and as a consequence, a system that tolerates multiple failures is too expensive. As a final example, it is appropriate for one application to replay the messages when it recovers, while for another, checkpointing local state might be more appropriate. The bottom line is that a single monolithic protocol that locks in one set of design choices will necessarily restrict the set of applications for which it is suitable.

Our methodology suggests the modular design illustrated in Figure 11. At the heart of the protocol graph is PSYNC—an IPC protocol that explicitly preserves the partial order of messages exchanged among a set of processes in the presence of host and network failures [24]. On top of PSYNC we have designed three different protocols that enforce a particular ordering of messages, generically denoted as ORDER in Figure 11 [21]. We have also implemented three protocols that are used to recover from process failure: MEMBER, RECOVER, and MONITOR. MEMBER is a protocol that establishes

sliding window algorithm [33]. RRS is a micro-protocol in the sense that it implements only the sliding window algorithm; connection establishment and termination concerns have been separated out into three other protocols—CMP, C_CHAN, and S_CHAN. In contrast, all these issues are merged in TCP. Also, RRS sends logical records of up to 16k-bytes; it depends on BLAST (hidden in the VADDR subgraph) to fragment large records. In contrast, TCP is responsible for fragmenting large data blocks.

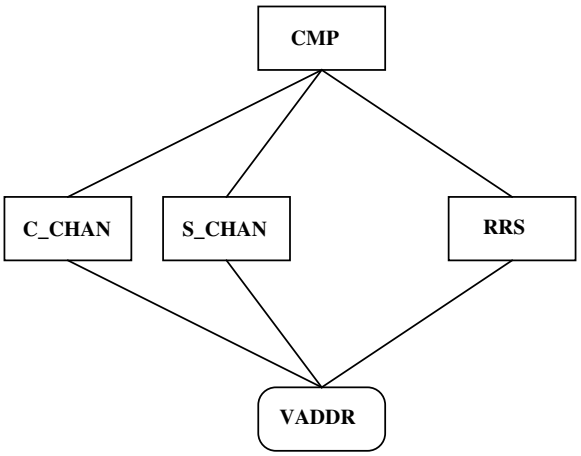

Figure 10: Stream Protocol Graph

CMP (connection manager protocol), maintains a simple state transition diagram that represents whether a logical RRS connection is closed, established, or in transition. CMP uses the two CHANNEL protocols to reach agreement with its peer on another machine about certain transitions. It then passes the appropriate connection parameters to RRS. Specifically, when a high level entity opens CMP to establish a connection, CMP in turn calls an "open connection" procedure provided by its peer, with the local port, the remote port, and the initial sequence number it plans to use as arguments. This procedure returns the initial sequence number that the server plans to use. Once the call returns on the client side, CMP opens RRS with the two port numbers and the two initial sequence numbers, moves to the "connection established" state, and returns a CMP session to the user. Subsequent sends to this CMP session are directly passed onto RRS.

The sequence of events on the server machine is more complicated. When the "open connection" procedure is called, CMP sends a reply message indicating the initial sequence number it plans to use, and it opens the local RRS protocol with the two ports and the two sequence numbers. Instead of moving to the established state, however, the server side goes to a transitional "ok to receive but not send" state until data begins to arrive from RRS. This is because the server does not know that the client has received its reply.[5] The server side of CMP moves to the fully established state once data arrives from RRS.

Closing a connection is similar, with the complication that either side may initiate a close to indicate that they have no more data to send, and both sides must have closed the connection before it is considered

[5]C_CHAN acknowledges the reply message, but this ACK only informs S_CHAN that it can free the reply message; the acknowledgement is not visible to the high-level protocol.

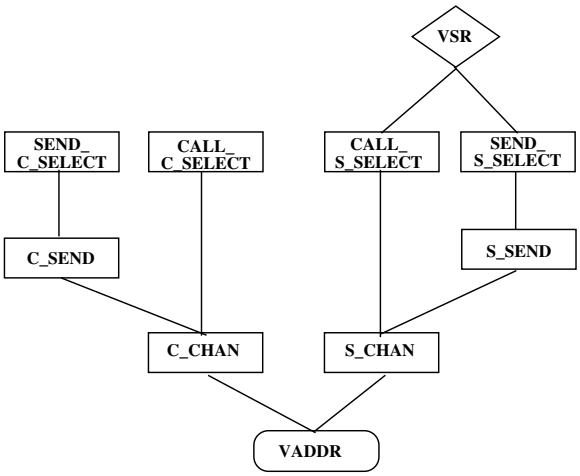

Figure 9: SR Support Protocols

large message support. RPC's asymmetric C_CHAN and S_CHAN protocols provide the right support for the SR **call** statement. The asymmetric protocols C_SEND and S_SEND implement semi-synchronous message passing by having S_SEND reply with an acknowledgement message immediately upon receiving the request message. S_SEND then demultiplexes the request message up the graph and discards any attempted reply. Because C_CHAN and S_CHAN implement a reliable algorithm with *at most once* semantics, the C_SEND and S_SEND protocols need not worry about dropped packets and are therefore trivial protocols. The various SELECT protocols are a simple modification of the asymmetric SELECT protocols used previously—they use SR capabilities rather than procedure numbers.

The SR runtime support system actively opens CALL_C_SELECT for each **call** statement that is invoked, and it actively opens SEND_C_SELECT for each **send** statement that is invoked. So as to enable each **proc** and **in** statement to receive invocations from both **calls** and **sends**, the SR runtime support system passively opens VSR, which in turn passively opens both SEND_S_SELECT and CALL_S_SELECT.

The SR protocol graph can be constructed in a matter of a few hours, and is an improvement over the current implementation—which uses TCP for external communication—in two ways. First, by avoiding TCP and in most cases IP, the performance of SR is improved. This is because TCP connection establishment and IP's reassembely algorithm are avoided. Second, our approach allows code to be migrated out of the SR runtime support system and into the protocol graph, thereby simplifying the implementation of the runtime support system.

4.2 Stream-Oriented Communication

Our second example, depicted by the protocol graph given in Figure 10, provides a stream-oriented communication service. The central protocol in this example is RRS—a "reliable record stream" transport protocol. Unlike a byte-stream protocol such as TCP, in which the blocks of data written by the sender are not necessarily equal to the blocks of data read by the receiver, RRS allows two user processes to exchange discrete records. Like TCP, RRS guarantees the sequential delivery of those records using the

environment in which the path a message follows through the protocol graph corresponds to the flow of control: micro-protocols correspond to simple statments and virtual protocols correspond to conditional statments.

Due to space considerations, none of the protocol graphs presented in this section contain modules to marshall arguments, encrypt data, or implement authentication. It is our experience, however, that our methodology is the ideal way to handle such functionality. For example, Sun RPC supports three different authentication options. Each of these options can be represented by a different micro-protocol, where a virtual protocol selects the appropriate one for a particular application. As another example, we have found it best to treat argument marshalling as completely separate from the request/reply protocol. This is because argument marshalling requires agreement between each caller and remote procedure, whereas the request/reply protocol is common to all calls.

4 Programming with Protocols

Assuming a library of micro-protocols and virtual protocols already exists, the next question is how to create a new communications service by reusing these pieces in different combinations. In some cases, new micro-protocols may need to be created, but quite often these new protocols are easily derived by simple modifications to existing protocols. The process is manual but the existence of the protocol library lowers the effective effort. Only after the creation of such a protocol graph would the implementor consider optimizations; e.g., merging a subgraph into single node.

This section presents three examples that illustrate how one might program with protocols. The first example constructs a protocol graph that implements the communication support needed by the SR programming language, the second example implements a stream-oriented service, and the third example implements a fault-tolerant multicast service.

It is important to keep in mind that we are concentrating on one communication service at a time. In general, a given machine might support a protocol graph that implements several different communication services, such that each micro-protocol and virtual protocol processes messages on behalf of multiple services.

4.1 SR Runtime Support System

As a first example, we construct a protocol graph to support communications for the distributed programming language SR [2]. The salient feature of SR is that it supports two types of remote invocation—the **call** statement and the **send** statement—and two types of remote objects—the **proc** statement and the **in** statement. These statements may be mixed to produce a variety of semantics: a **call** to a **proc** corresponds to a remote procedure call; a **send** to an **in** corresponds to semi-synchronous message passing; **call** to an **in** statement is a rendezvous; and **send** to a **proc** corresponds to a process fork.

The SR runtime support system running on each machine implements its own primitives for local communication, but must use some external protocol for all non-local communication. Figure 9 depicts a protocol graph used by the SR runtime support system. The same VADDR subgraph defined for RPC is used to provide SR with improved performance for messages on the local network and more robust

In the CHANNEL protocol described in Stage 1, each CHANNEL session manages several logical request/reply channels. With the division of CHANNEL into the asymmetric protocols C_CHAN and S_CHAN, all of the request/reply management code is implemented in C_CHAN, which represents these logical channels as protocol-specific objects. These objects are stored internal to the C_CHAN protocol and are not explicitly part of the protocol graph. The new C_CHAN protocol[3] returns a session that is equivalent to a single logical request/reply channel. That is, there is now a one-to-one relationship between C_CHAN sessions and logical channels.

VFIFO is a virtual protocol that implements the same state management algorithm found in the original CHANNEL protocol, but on C_CHAN session objects instead of locally defined state-objects. That is, VFIFO multiplexes an arbitrary number of request messages over a fixed number of C_CHAN session objects. Each VFIFO session object opens a fixed number of C_CHAN session objects and allocates them to incoming messages on a first-come-first-served basis. If a message arrives when all the C_CHAN session objects are busy, VFIFO blocks the message's shepherd process until a channel is free. A C_CHAN session object becomes free when the previous send on that session object returns. Because VFIFO was decomposed from the original C_CHAN, it exists only on the client side of the protocol graph.

A similar decomposition is performed upon the BLAST protocol. The BLAST protocol defined in Stage 1 created a protocol defined state object for each message sent. In the new version of BLAST, each session object handles one blast at a time, and if a new message is sent before the old one has been completed, the old message is discarded. The virtual protocol VROUND manages the BLAST sessions in a round robin fashion. VROUND opens a fixed number of BLAST sessions and sends the next message to the next session without verifying whether it is currently in use. It also automatically frees dangling fragments.

3.7 Final Result

To complete the picture, the two instantiations of protocol PAD are added to the graph. This protocol pads the header out to a pre-determined length; e.g., 128 bytes. Although this has a small adverse effect on latency, doing this ensures that all messages that go out on the network have the same size header, independent of the path they followed through the protocol graph. This makes it possible to predict where the user data begins in each message, much as one would be able to do if the RPC service had been implemented by a single monolithic protocol, thereby facilitating techniques that improve user-to-user throughput [7, 23].

Figure 8 represents the fully decomposed version of the RPC communication service.[4] To use this service, the application opens VRPC with an address that contains a procedure number and the IP address of the target machine. The protocol graph determines the optimal path of any given message without assistance from the user. From another perspective, the architecture can be thought of as a programming

[3]For simplicity, we still call it C_CHAN, and specify the "original" or the "new" version when the distinction is important.

[4]Keep in mind that the decomposition of a protocol (or protocol graph) results in a new protocol graph that is only semantically equivalent to the original. Thus, the graphs given in Figures 2 and 8 are not interoperable.

The virtual protocol VSYM makes a pair of asymmetric protocols look like a single protocol to any higher level protocols, thereby maintaining the composibility of symmetric and asymmetric protocols. When a high-level protocol actively opens VSYM, VSYM in turn opens C_SELECT and returns the resulting lower level session. If a high-level protocol passively opens VSYM, VSYM in turn passively opens S_SELECT. VSYM creates no session objects of its own.

3.6 State Elimination

The sixth and final stage of the decomposition involves elevating all protocol-defined state into the protocol graph. Protocols that define their own internal objects can be decomposed into a new version of the original protocol in which each session object corresponds to a single state-object defined by the old version of the protocol. A new virtual protocol implements the old internal-object management strategy on the new protocol's session objects. Elevating protocol-specific state objects to the protocol graph has two advantages. First, it makes the creation of protocol development and support tools simpler by eliminating protocol dependencies. Second, because each management policy is implemented as a separate virtual protocol, it is easier to substitute one state management policy for another. In traditional protocols, modification to existing protocol code would be required. Figure 8 depicts the result of performing state elimination upon the protocols CHANNEL and BLAST.

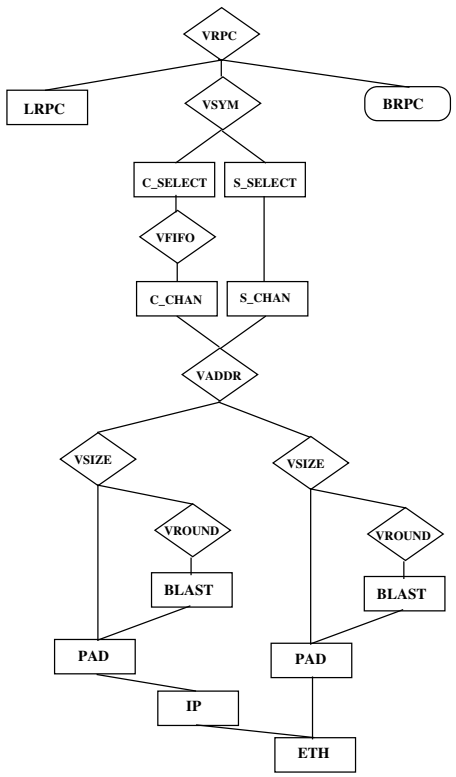

Figure 8: Stage 6—State Elimination

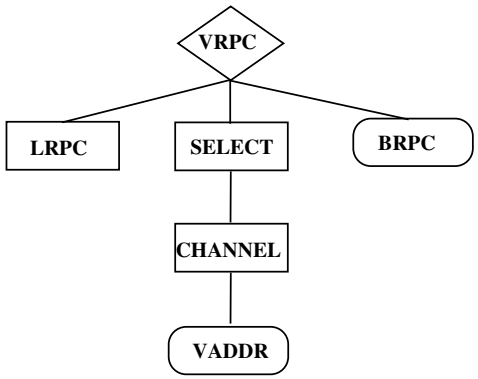

Figure 6: Stage 4—Factoring Options

3.5 Asymmetric Protocols

In request/reply protocols such as CHANNEL, the algorithm running on the client host differs greatly from the algorithm running on the server host. Stage 5 divides micro-protocols which are client/server in structure into a pair of asymmetric protocols. Decomposing CHANNEL into a client-side micro-protocol (C_CHAN) and a server-side micro-protocol (S_CHAN) simplifies the CHANNEL source code by separating the concerns of the client from those of the server. Figure 7 depicts the result of decomposing both CHANNEL and SELECT into a pair of asymmetric protocols.

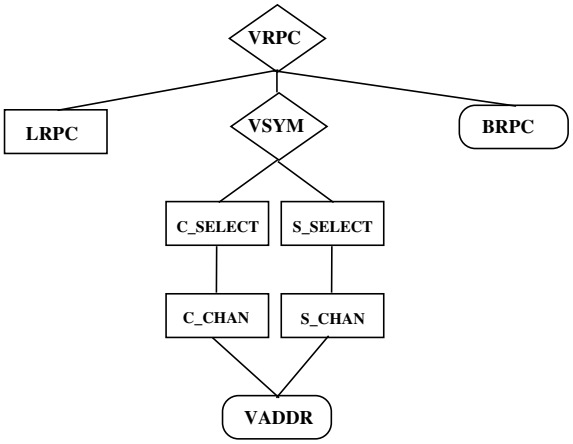

Figure 7: Stage 5—Asymmetric Protocols

C_CHAN implements the entire client side of the request/reply algorithm and does not support passive opens. S_CHAN supports the entire server side of the request/reply protocol and does not support active opens. C_CHAN and S_CHAN share the same header format. The protocol SELECT is also decomposed into two asymmetric protocols: C_SELECT and S_SELECT. While not as severe, this protocol also exhibits asymmetric behavior.

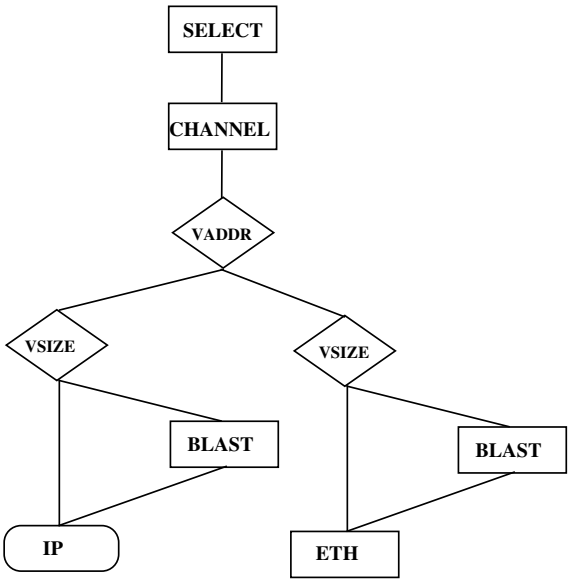

Figure 5: Stage 3—Multiple Instantiation

3.4 Factoring Options

Traditional protocols support a wide variety of options. The fourth stage of the decomposition uses virtual protocols to factor options out of RPC. Each option is implemented by a separate protocol, and a virtual protocol connects all of the options into a unified protocol graph. The key advantage provided by this decomposition is that it simplifies the process of adding new options—options are added by adding a new micro-protocol to the protocol graph.

For example, consider an RPC protocol that uses IP addresses to identify hosts. The protocol should use one of three different algorithms depending upon the host address it is passed at open time. If the address identifies the local machine, then the local IPC mechanism should be used. If the address is the broadcast IP address, then a broadcast RPC mechanism should be used. If the address denotes a single remote host, then the regular algorithm, as described previously, should be used.

As illustrated in Figure 6, each branch is implemented as a separate protocol; the VADDR subgraph has been truncated for brevity. The virtual protocol VRPC is used to direct messages to the appropriate protocol, or more accurately, the appropriate subgraph. Local communication is handled by the micro-protocol LRPC, which implements Bershad's light-weight RPC protocol [3] for efficient local interprocess communication. It accepts messages from a client in one address space and immediately turns them around and demultiplexes them to the correct "remote" procedure in another address space. Broadcast communication is handled by the subgraph labelled BRPC, which supports a simple and unreliable broadcast RPC algorithm that is based upon the broadcast capabilities of the underlying local area network. An example BRPC subgraph is presented in Section 4.3. Finally, the common case is still handled by CHANNEL.

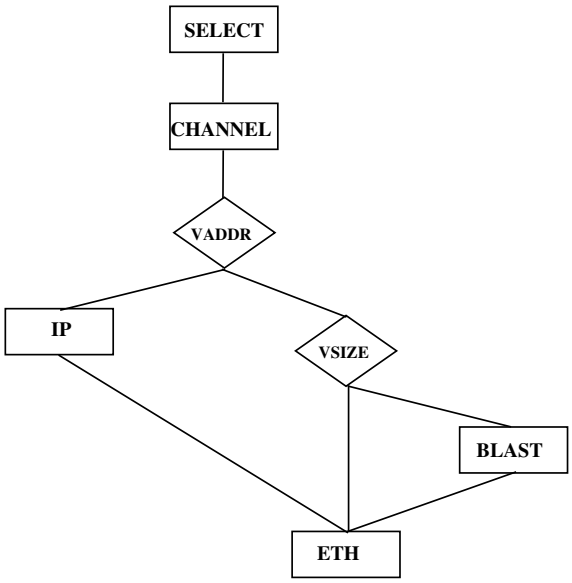

Figure 4: Stage 2—Bypassing IP and BLAST

set of rules that protocols must follow so that messages can traverse the protocol graph.

3.3 Multiple Instantiations

While the protocol presented in Figure 5 avoids the cost of IP in the local case, it incurs the problems associated with IP's fragmentation algorithm in the remote case; it does not selectively retransmit lost fragments. The obvious solution is to also use BLAST in the non-local case. The BLAST protocol, while not the perfect Internet large message protocol, is superior to IP's fragmentation algorithm in that it selectively retransmits lost fragments. Stage 3 of the decomposition adds functionality to the protocol graph through the multiple instantiation of existing protocols. In this case, a second instantiation of BLAST and VSIZE are used on top of IP, as illustrated in Figure 5. The second instantiation of VSIZE directs short non-local messages to IP and long non-local messages to the second instantiation of BLAST. In this figure, the rounded box for IP represents the root of a subgraph that has been omitted for simplicity.

The reader should notice one potential problem with this protocol graph. The instance of VSIZE on top of the ETH protocol is opened with Ethernet addresses, while the instance of VSIZE on top of IP is opened with IP addresses. Thus, to be used in this situation, VSIZE must be polymorphic in the sense that it accepts any address type. Also, for maximum composibility, VSIZE is generalized to direct messages over an arbitrary number of lower level protocols. That is, VSIZE expects to be configured on top of a set of lower level protocols, in decreasing order of performance and increasing order of packet size. VSIZE directs each message to the first protocol with a maximum packet size at least as large as the message length.

CHANNEL session manages a set of *logical channels* between two hosts, where a single logical channel can support only one invocation (request/reply pair) at a time. The CHANNEL session directs each new request message to a free logical channel or blocks the message until a logical channel becomes free. Each logical channel is represented by an internal object defined by the CHANNEL protocol; this object is not represented in the protocol graph. In other words, there is a one-to-many relationship between CHANNEL sessions and logical channels as defined by the algorithm.

SELECT is a trivial protocol that maps a procedure number onto a remote procedure. On the sending side, SELECT adds a header identifying the correct procedure to each message, then sends the message to a CHANNEL session. On the receiving side, SELECT examines the header of each incoming message and dispatches the message to the appropriate procedure. Procedures are identified as $\langle$*procedure number, host*$\rangle$ pair, where the host address is an IP address. As mentioned above, the reason for separating SELECT into a separate protocol rather than embedding it in CHANNEL is that we want to be able to support multiple schemes for addressing procedures. By creating a separate SELECT protocol for each RPC address space, an existing protocol graph can support a new address space simply by replacing its version of SELECT with one that reflects the new address space. For example, a SELECT protocol that implements the Sun RPC [32] address space would use the following four-tuple to identify a remote procedure: $\langle$*program number, version number, procedure number, host*$\rangle$.

3.2 Bypassing Protocols

The protocol graph created in Stage 1 suffers from one problem: it is less efficient than the original simple protocol graph. This is because there is a small penalty for each protocol layer. However, it is not necessary for every message to traverse every layer of the protocol graph. Short messages do not need to be fragmented and therefore do not need to traverse the BLAST layer. Also, local messages—those addressed to hosts on the local Ethernet—do not need the services of IP. The second stage of the decomposition allows messages to bypass certain protocols. We introduce two virtual protocols—VSIZE and VADDR—to bypass IP and BLAST, respectively. Figure 4 depicts the graph that results from this decomposition. We represent virtual protocols with diamonds.

Protocol VSIZE replaces the IF statement found in a typical implementation of the blast algorithm; this IF checks to see if the message requires fragmentation. VSIZE depends on two low-level protocols, directing small messages to the first and large messages to the second. VSIZE makes its decision purely upon the size of the message; it does not communicate with its peers. Note that this decomposition results in a protocol graph that bypasses BLAST in its entirety. In contrast, while typical RPC implementations bypass the code necessary to fragment a message, they must still fill in the header fields associated with the blast algorithm.

VADDR is a virtual protocol that directs messages to either IP or the Ethernet. VADDR attempts to resolve the destination IP address into an Ethernet address using the Address Resolution Protocol (ARP) [25] (not shown in the graph). If ARP can resolve the destination address on the local network, then IP can be bypassed.

Note that virtual protocols make the process of navigating the protocol more complex. In fact, the protocol graph on the sending and the receiving machine need not be identical. Section 5.3 introduces a

This is the case with both the request/reply algorithm and the blast algorithm, as demonstrated in the next section.

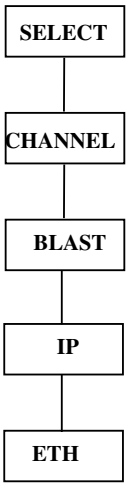

Figure 3: Stage 1—Encapsulating Algorithms as Protocols

Figure 3 depicts the protocol graph that results from the implementation of each of these algorithms as a separate protocol. The protocols are composed linearly on top of IP, and as a consequence, any request message must travel through five protocols before being sent. We now consider the semantics of these three layers from the bottom-up.

BLAST supports the unreliable delivery of large messages between a pair of hosts. Each host is identified by its IP address. BLAST is unreliable in the sense that messages may be delivered out of order, duplicate copies of the same high-level message may be delivered, and a given message may not be delivered at all. Specifically, each message sent through BLAST is assigned a unique sequence number. The message is then fragmented and transmitted, with a copy of the fragments saved in the local state. If the receiving host detects that it is missing one or more fragments, it sends a request for the missing fragments to the sending host.

The algorithm used in BLAST differs from that used in the original RPC protocol in that the receiving host never sends a positive acknowledgment (implicit or explicit) when all the message fragments have been successfully received. Instead, the sending host associates a timer with each message it sends and discards the message when the timer expires. BLAST also differs from most RPC implementations in its handling of retransmission. If a higher level protocol retransmits a message, BLAST treats the second incarnation of the message as an independent message and assigns it a new BLAST-level sequence number. These modifications are necessary because the blast algorithm, as implemented in RPC, depends on the *at most once* semantics of the request/reply algorithm. In our version, BLAST may be used with other higher level protocols, and as a consequence, can make no such assumptions.

CHANNEL supports request/reply transactions with *at most once* semantics. The semantics of CHANNEL are identical to those provided by Sprite RPC. To use CHANNEL, a user program or high-level protocol sends a request message through a CHANNEL session, and the reply message is returned. Each

3 Protocol Decomposition

This section introduces the architecture by walking through a six stage decomposition of a conventional remote procedure call (RPC) service. This decomposition can be thought of as a top-down design of the service—starting with the service, we derive a set of building block micro-protocols and virtual protocols. Once a collection of micro-protocols and virtual protocols have been designed, additional communication services can be built more easily by reusing these pieces. The next section illustrates how additional communication services are constructed using an existing protocol library.

We begin with a typical protocol graph that implements an RPC service. The graph, as illustrated in Figure 2, contains three protocols. The topmost protocol implements the Birrell-Nelson remote procedure call algorithm [5]. To make the example more concrete, we use an actual protocol that implements this algorithm—Sprite RPC [37]. IP is a protocol that routes messages across the internet [27], and ETH is a device driver protocol for the ethernet.[2]

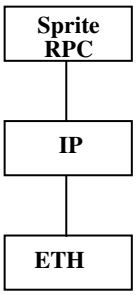

Figure 2: Stage 0—Conventional RPC

3.1 Encapsulating Algorithms as Protocols

Many traditional protocols implement several different algorithms upon each message they process. The first stage of the decomposition involves encapsulating the identifiable algorithms in a large protocol into separate protocols, and creating a new protocol graph from the linear composition of the newly created protocols. In the RPC case, we have identified three distinct algorithms: a demultiplexing algorithm that dispatches messages to the right procedure [26], a request/reply algorithm that matches request messages with reply messages while preserving *at most once* semantics [5], and a blast algorithm with selective retransmission that fragments large messages into packets that can be transmitted on the underlying network [39].

There are many reasons to decide that an algorithm should become a protocol. One is that one can imagine alternative algorithms being better suited for different situations. For example, we have used different demultiplexing algorithms, depending on the remote procedure address space. Another reason is that there exists a recognized need to reuse a particular algorithm in a different communication system.

[2]We have taken one liberty with our implementation of Sprite RPC. Instead of placing it directly on top of the ethernet, we place it on top of IP and we use IP addresses in place of Sprite's 32-bit host identifiers.

Second, the *x*-kernel provides a facility for caching open session objects. Thus, even if the low-level protocol does not support explicit connections, the high-level protocol is able to cache an open session and reuse it for multiple messages. One consequence of this design is that protocol implementations do as much work as possible at open time—e.g., initializing header templates—thereby minimizing the overhead involved in sending and receiving messages.

Third, in addition to sending and receiving messages, session objects also support *control operations* that allow high-level entities to get and set various parameters. For example, a session object might support control operations called MYADDR and PEERADDR that return the local and remote address, respectively. If the session on which the control is invoked is not able to service the operation, then the *x*-kernel does a depth first search of the subgraph rooted at that session for an object that can service the control. This feature, called *delegated controls*, allows a high-level entity to invoke a service without knowing which object actually implements the service. An example of how delegated controls are used is given in [23].

Fourth, the *x*-kernel associates processes with messages, and these processes are allowed to migrate across protection boundaries. 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 the protocol graph; this process begins by invoking the lowest-level protocol. Should the message eventually reach the user/kernel boundary, the shepherd process does an upcall and continues executing as a user process [12]. The kernel process is returned to a pool and made available for reuse whenever the initial protocol returns. 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. In both cases, the message is passed from one protocol to another by the former protocol invoking a procedure call on the latter protocol; no context switches are involved to pass a message between two protocols. Thus, when the message does not encounter contention for resources, it is possible to send or receive a message with no context switches. The obvious alternative is to associate a process with each protocol and use an IPC mechanism to pass messages between protocols. This has the disadvantage of requiring multiple context switches to process each message.

Fifth, messages are implemented in a two-part data structure that separates the header from the user data. The first part—a stack—holds all the headers that are attached to a message. This part is optimized for adding and deleting protocol headers by doing simple pointer arithmetic and structure copies. The second part—a directed acyclic graph of data buffers—implements the user data. This part is optimized to avoid unnecessary data copying during fragmentation and reassembly.

Sixth, the *x*-kernel supports an efficient id mapping facility that is used for demultiplexing. A given protocol extracts some collection of fields from a message header (e.g., source and destination port numbers) and constructs a composite id. The protocol then looks up this id in the map to retrieve a pointer to (a capability for) the next protocol that is to process the message. The map is implemented as a hash table, but it caches the last binding it looked up. Thus, in the common case where back-to-back messages traverse the same path through the protocol graph, demultiplexing costs only a single level of indirection.

In support of the thesis that network architectures should be dynamic, this paper makes the following four points:

- A dynamic network architecture based on a large number of micro-protocols and virtual protocols can be made to perform competitively with conventional static architectures consisting of a small number of monolithic protocols.
- Micro-protocols, because they implement a single function, promote reuse, and as a consequence, have the advantage of making new communication systems easier to program.
- A dynamic architecture has the advantage of allowing application programmers to configure exactly the right combination of protocols for their application.
- A dynamic architecture has the advantage of adapting more quickly to changes in the underlying network technology.

This paper is organized as follows. Section 2 briefly describes the *x*-kernel. The next two sections illustrate the ideas behind the architecture through a series of examples: Section 3 shows how a common communication service can be decomposed into a collection of micro-protocols and virtual protocols and Section 4 demonstrates how an existing collection of micro-protocols and virtual protocols can be used to "program" new communication systems. Section 5 then summarizes the techniques introduced in the previous two sections, and reports our experience using them. Finally, Section 6 describes the performance of the protocol graphs derived using our approach, Section 7 discusses related work, and Section 8 offers some conclusions.

2 OS Platform

We use the *x*-kernel as our operating system platform. Since the *x*-kernel has already been described in the literature, this section simply highlights the important features that we were able to exploit. Note that the approach introduced in this paper is not unique to the *x*-kernel; it could be implemented in any operating system that supports late binding between layers and a low per-layer overhead.

First, the *x*-kernel explicitly implements the protocol graph. That is, the kernel supports a set of *protocol objects* that possess capabilities for each other. The kernel programmer configures a given instance of the *x*-kernel by specifying the protocol graph using a graphical editor. In addition to the protocol graph—which represents the static relationship between protocols—the *x*-kernel also defines *session objects* to represent the dynamics aspects of communication. Each session object corresponds to the local endpoint of an open connection. Operationally, a high-level protocol *opens* a low-level protocol, and the low-level protocol returns a session object. This open can be either *active* or *passive*: the former involves the high-level protocol giving the low-level protocol the address for the peer it wants to communication with, and the later involves the high-level protocol informing the low-level protocol that it is willing to accept messages from any of its peers. In either case, the high-level protocol *sends* and *receives* messages using this session object. Although the *x*-kernel defines a uniform interface for protocol and session objects, for the purposes of this paper it will be sufficient to refer to what happens at "open time" and "send time".

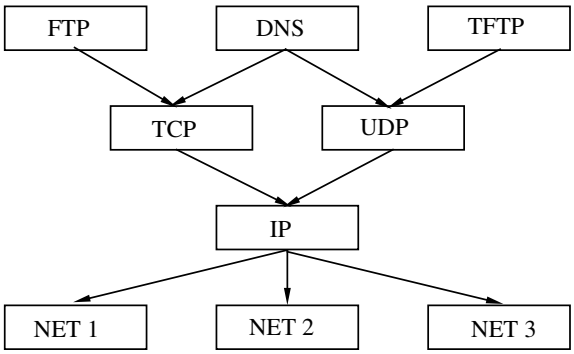

Figure 1: Protocol Graph of the TCP/IP Network Architecture

This paper argues that the right number of layers cannot cannot be defined statically, and introduces a dynamic network architecture in which the number of layers visited by each message is variable.

Two ingredients are necessary for a dynamic network architecture to achieve good performance. First, the operating system in which the architecture is implemented must support a late binding between layers and a low overhead for each layer. We have already described such an operating system in our earlier work with the *x*-kernel [17]. Second, the protocols that make up the network architecture must be designed to take advantage of these two features. This paper uses the *x*-kernel as a foundation for addressing this second factor. The importance of this second factor is not obvious—if one has an operating system like the *x*-kernel that supports efficient layering, then it would seem that existing protocols could be effectively layered. Experience indicates that this is not necessarily true. In fact, we believe that the reason many people equate layering with poor performance is that they have attempted to layer existing protocols; i.e., layering Sun RPC on top of TCP will not result in efficient network code no matter how good the operating system [15].

In way of an introduction to this new architecture, consider that conventional architectures share three key features: their protocol graphs are simple, the protocols are complex, and both the set of protocols and the topology of the protocol graph are fixed by the architecture. The architecture presented in this paper shares none of these features: the protocol graph is complex, the protocols are simple, and both the set of protocols and the topology of the protocol graph are selected by application programs. Moreover, our approach supports dynamic protocol graph topologies by providing a protocol development environment. In contrast, conventional architectures provide no support for protocol development, which is consistent with their static nature.

We represent network software with a graph of two types of protocols: *micro-protocols* and *virtual protocols*. Micro-protocols are like conventional protocols in that they communicate with their peers, add headers to messages, and demultiplex messages to any number of higher level protocols. Micro-protocols differ from traditional protocols in that they implement a single function and do not support options. Virtual protocols, on the other hand, are quite different from conventional protocols. They do not have headers, do not necessarily have a peer on the other machine, and serve only to direct messages through the protocol graph.

A Dynamic Network Architecture

Sean W. O'Malley and Larry L. Peterson[1]
University of Arizona

Network software is a critical component of any distributed system. Because of its complexity, network software is commonly layered into a hierarchy of protocols, or more generally, into a *protocol graph*. Typical protocol graphs—including those standardized in the ISO and TCP/IP network architectures—share three important properties: the protocol graph is simple, the nodes of the graph (protocols) encapsulate complex functionality, and the topology of the graph is relatively static. This paper describes a new way to organizing network software that differs from conventional architectures in all three of these properties. In our approach, the protocol graph is complex, individual protocols encapsulate a single function, and the topology of the graph is dynamic. The main contribution of this paper is to describe the ideas behind our new architecture, illustrate the advantages of using the architecture, and demonstrate that the architecture results in efficient network software.

Categories and Subject Descriptors: C.2.1 [**Computer Communication Networks**]: Network Architecture and Design—*network communications*; C.2.2 [**Computer Communication Networks**]: Network Protocols—*protocol architecture*; D.2.10 [**Software Engineering**]: Design—*methodologies*.

General Terms: Design, Performance

Additional Key Words and Phrases: Composibility, Reuse, Dynamic Configuration

1 Introduction

Network software, because of its complexity, is commonly layered into a hierarchy of protocols. Each protocol exchanges messages with its peers on other machines to implement some abstract communication service. Except at the hardware level, peer-to-peer communication is indirect—the protocol passes messages to some lower level protocol, which in turn delivers the message to *its* peer. We abstractly represent the protocol layers that make up a communication system with a directed acyclic graph, called a *protocol graph*. The nodes of the graph correspond to protocols and the edges represent a *depends on* relation; e.g., if protocol A sends messages to its peers using protocol B, then there is an edge from node A to node B. Standardization bodies, such as the International Standards Organization (ISO) and the Internet Activities Board (IAB), typically establish policies about the form and content of the protocol graph for a particular network architecture. For example, Figure 1 illustrates a portion of the protocol graph that represents the TCP/IP network architecture [13].

What this protocol graph should look like—or more precisely, the question of 'What is the right number of layers?'—has been debated in the networking community for many years [38, 11, 14, 34, 15, 10]: the ISO community claims that the right number is seven, the TCP/IP community argues for four, and the operating systems research community often implements just one (i.e., a single RPC protocol).

[1]This work was supported in part by National Science Foundation Grant CCR-8811423. Authors' current address: Department of Computer Science, University of Arizona, Tucson, AZ 85721.