OS Support for High-Speed Networking

Larry L. Peterson

Department of Computer Science
University of Arizona
Tucson, AZ 85721
(llp@cs.arizona.edu)

1 Introduction

We are designing and implementing an operating system (OS) that runs on the end hosts connected to a gigabit network. It is through this OS that application programs access the network and other resources available on the host. Our goal is to provide general OS support for gigabit applications without sacrificing performance.

We believe our emphasis on a general-purpose OS is an important one. While it is tempting to optimize for a specific application, protocol, or network hardware, ultimately this is not a cost-effective approach. It is also the case that the high-speed network will be one of many physical resources connected to the end host, and the network application will be one of many applications that users want to run on the end host. This means that the operating system must be designed to offer a wide range of services that run in multiple protection domains.

On the other hand, operating systems in use today are not adequate for this task because they were designed to function in a world of low-bandwidth I/O devices such as terminals, disks, and ethernets. Thus, a key aspect of this work is to provide new abstractions and mechanisms that allow application programs to take advantage of the performance characteristics of the underlying hardware. In other words, although we are focusing on a general purpose operating system, we are not willing (nor do we believe it is necessary) to sacrifice performance.

In this context, we are focusing on the software architecture for the I/O (networking) subsystem of the operating system. This software architecture provides an environment for implementing the network protocols necessary to support a variety of applications. Our approach is to "open up" to application programmers what is traditionally thought of as the communications subsystem, and to integrate this subsystem in the rest of the OS in a way that results in performance that is competitively with communication subsystems that are hand crafted by OS experts.

Our effort has two components: (1) building a demonstration operating system that is able to support gigabit applications running on gigabit networks, and (2) using this system as a platform for addressing research issues related to system support high-speed networks. These two components are synergistic: the research results in new abstractions and mechanisms that enhance the demonstration system.

2 Demonstration System

We are building a demonstration system that is portable across different hardware platforms. We expect to use this system as a vehicle for exploring issues in system support for high-speed networking. We also plan to make this system available to the high-speed networking community. This section describes the demonstration system; the following section identifies several research problems we are addressing in order to realize this system.

2.1 Software Base

Our software base consists of an integration of the Mach operating system from Carnegie-Mellon University [Acce86, Golu90], and the *x*-kernel from the University of Arizona [Hutc91, O'Ma92]. (We sometimes call the resulting system NP-Mach, for "Network Performant" Mach.)

Mach is a microkernel-based operating system. The philosophy of Mach is to provide a minimal communications kernel, upon which a variety of operating system services can be built. Mach offers three key abstractions: *tasks* are the unit of resource allocation (they roughly correspond to address spaces), *threads* are the unit of CPU utilization (multiple threads may execute concurrently within each task), and *ports* are communication channels through which threads send and receive messages (they are accessed by possessing a send or receive capability).

Mach, which is the *de facto* operating system of the high-performance computing system component of HPCC, has three characteristics that make it suitable for this work. First, Mach is a minimalist kernel that places very few restrictions on how it is used. That is, Mach is explicitly designed to be molded for a particular use. Second, Mach is designed to be portable across a wide number of machines, ranging from RISC workstations to massively parallel processors. Third, Mach is widely used, and several realistic applications/services have been built on top of it.

The *x*-kernel was originally designed to function as a full-fledged operating system kernel. The design philosophy was that an operating system kernel is primarily a communication multiplexor, and as such, the one thing it had better do well is support communication protocols. Towards this end, the *x*-kernel was explicitly designed to support the rapid implementation of efficient network protocols. Because it was a complete OS kernel, the initial implementation of the *x*-kernel on Sun3 workstations included support for lightweight processes and multiple virtual address spaces.

During the last several months, we have refocused our effort by divorcing those aspects of the *x*-kernel that are specifically related to the implementation of network protocols from the other aspects of the operating system, and we are now embedding this *communication-core* of the *x*-kernel in different existing operating systems, most notably Mach. This communication-core captures the abstract view of the underlying system that a protocol needs in order to do its job, and hides the details about how those abstractions are implemented in a particular system. That is, just as the Mach microkernel defines an abstract interface on which operating system services can be implemented, the *x*-kernel defines an abstract interface to which protocols can be written. Specifically, the *x*-kernel consists of four components:

Uniform Protocol Interface: All *x*-kernel protocols are completely encapsulated modules that export a uniform set of operations. A given instance of the *x*-kernel is configured by specifying a desired protocol graph. The protocol graph is explicitly represented within the *x*-kernel.

Protocol Support Library: The *x*-kernel provides a set of support library that includes routines to manipulate message buffers, schedule events, and map identifiers. Not only is the support library highly optimized, but it also completely isolates the protocol implementation from the rest of the OS, as well as the underlying hardware.

Virtual Protocols: Virtual protocols are a technique used to dynamically bypass unnecessary (or less desirable) protocols. They allow a one to configure a protocol graph with a "standard" protocols for backwards compatibility, and a "streamlined" protocol that is optimized for a particular site or application. The virtual protocol chooses the right protocol at runtime.

Application Interfaces: The *x*-kernel explicitly separates the application interface (API) seen by user programs from the protocol(s) that implement that API. This makes it possible for application programs to access *x*-kernel protocols through a variety of API's; e.g., BSD Sockets, SunOS RPC, Mach IPC, and System V TLI.

The *x*-kernel offers three major advantages that make it suitable for use as the communications subsystem of an OS for high-speed networks. First, because it is configurable and API-independent, the *x*-kernel is flexible enough to support a wide range of applications. Second, it is easy to write protocols in the *x*-kernel. This is because the *x*-kernel defines high-level, protocol-centric abstractions that protocols are written to (e.g., connections, messages and events), rather than forcing protocol implementors to become experts at low-level mechanisms (e.g., context switches, buffer management, and clock interrupts). Third, protocols implemented in the *x*-kernel run as fast as (and often faster than) hand crafted implementations in other operating systems. This is accomplished by optimizing the mapping from protocol-centric abstractions onto the underlying architecture.

As mentioned above, we are integrating the *x*-kernel into Mach; it is a combination of these two systems that provides OS support for high-speed networking. Generally speaking, Mach itself provides very little support for networking. One problem is that network software is currently distributed haphazardly throughout Mach—each Unix server implements its own copy of the TCP/IP protocol suite, the NetMsgSvr implements the basic protocols that support non-local Mach IPC invocations, the microkernel implements a faster version of non-local Mach IPC (NORMA IPC) to support multicomputers, a DCE server implements the communications protocols needed to support DCE, and so on. What is worse, each protocol is implemented in a different environment by different programmers with different goals. Hence, it is difficult, if not impossible, to either move a protocol from one location to another, or to optimize the critical path.

Our plan is to completely re-engineer networking in Mach. This will involve "logically centralizing" all the protocols in a single *x*-kernel protocol graph, and then "physically distributing" this graph throughout the operating system. That is, portions of the graph will be implemented in various application tasks, in special network server tasks, and in the kernel. The decision as to where particular protocols belong can then be delayed until configuration time (rather than design time) and can be made based on how the system configurer is willing to trade performance for trust.

2.2 Hardware Base

Initially, we have two target hardware platforms. The hosts were selected both because they represent state-of-the-art RISC processors, and because we were able to aquire interesting (high-speed) network controllers for them.

- DecStation 5000/200 workstations equipped with Aurora SAR boards [Davi91]. The Aurora SAR board is interesting because it supports 622Mbps data rates (which will stress the throughput capabilities of the software), and because the board's programmability will allow us some latitude in defining the controller/OS boundary.
- HP 720 workstations equipped with Afterburner boards [Luml92]. The Afterburner board is interesting because it is designed for low-latency communications (which will stress the latency of our software), because it supports a novel host interface that includes 1MByte of on-board VRAM, and because it can be configured with different link adaptors (e.g., FDDI, ATM, HIPPI).

3 Advanced Research

Generally speaking, our research revolves around designing the abstractions, mechanisms, and implementation techniques needed to achieve high performance in the demonstration system described in the previous section. More precisely, we are looking at fundamental OS mechanisms, the relationship between the OS and network controllers, compiler-based optimizations, the software engineering aspects of network software, and alternative network protocols. This section outlines the individual research problems we are investigating.

3.1 Decoupling Modularity and Protection

The benefits of a modular operating system design are well-known: the system is more easily configured, scaled, extended, ported, maintained, verified, and distributed across multiple processors. Growing support for modular OS design has recently popularized the idea of microkernel-based systems such as Mach. While microkernel-based systems are an improvement over kernelized, monolithic operating systems like Unix, they fall short of satisfying a number of demands placed on modern operating systems.

The problem with microkernel-based systems is that they tightly couple modularity and protection—servers are implemented in separate protection domains, and consequently, the communication mechanisms are designed for the cross-domain case. A number of limitations arise as a result of this coupling. First, the modularity supported by these systems is very coarse-grained. Since each module is implemented in a separate domain, concern for cross-domain communication costs prevents a fine-grained decomposition of the system. For the same reason, it is difficult to extend the system vertically—i.e., through stackable services—because each layer adds communication overhead that degrades performance. Second, the partitioning of functionality into servers—and in particular determining what functionality should be provided by the kernel—is static and part of an early design decision. Consequently, it is difficult to reconfigure functionality among servers and between kernel and servers. Such reconfigurations are desirable to satisfy the needs of applications, to match the characteristics of a variety of hardware architectures, and to integrate new technology as the system evolves.

The solution we are investigating is to provide architectural support for modules that is independent of protection domains [Drus92b]. The key is a location transparent invocation mechanism that (1) has semantics similar to a local procedure call, (2) delivers performance close to that of an ordinary procedure call when the invocation is between modules in the same domain, and (3) allows the use of the most appropriate IPC mechanism in the cases of cross-domain and cross-machine invocations. Given such support, the decomposition of a system into modules can be guided by sound software engineering principles, with the modules distributed across protection domains and machines at configuration time based on criteria such as trust, security, fault-tolerance and performance. Put another way, protection and modularity are decoupled.

In the context of network protocols, this means that protocol modules can easily be moved between the kernel and user tasks at configuration time rather than design time. What is more, the decision as to where a protocol belongs is made by the user of the system based on how he or she is willing to trade performance for trust: trusted protocols can be loaded into the kernel for performance, untrusted protocols can be placed in user tasks at some performance cost, and protocols can be moved between protection domains over time.

3.2 Cross-Domain Data Transfer

One of the most important challenges faced by the communication subsystem is moving data across multiple protection domains efficiently enough so that applications can take advantage of the bandwidth offered by the network. In other words, we are interested in turning good host-to-host performance into good application-to-application performance. This task is difficult because of the limitations of the memory architecture. Existing techniques for cross-domain data transfer—copying, copy-on-write, page remapping, and static shared buffers—all suffer performance problems and/or other limitations on modern RISC workstations.

We are developing a new facility for cross-domain data transfer, called *fast buffers* (fbufs), that is designed to take advantage of the performance characteristics of RISC architectures [Drus92a]. The fbuf facility requires no special hardware other than support for large, sparse virtual address spaces. Fbufs are largely independent of the IPC mechanism used for control transfer, and because they do not require the execution of any privileged instructions, they can be used with user-level IPC mechanisms.

An fbuf is a contiguous memory buffer allocated from a large virtual address space window. This window, called the *fbuf window*, is mapped into every domain of a single shared memory machine. Each domain has read and write permissions to a distinct subwindow, called the domain's *writable subwindow*, and read-only access to the remaining portions of the fbuf window. That is, any page within the fbuf window is mapped for read/write access in exactly one domain, and read-only in all other domains.

Each domain allocates individual fbufs exclusively within its writable subwindow. Once allocated, an fbuf is "transferred" to another domain by sending its address to the recipient domain. When a domain accesses a newly allocated fbuf for the first time, a fault occurs and the system allocates physical memory to the buffer. The access fault following the allocation of a new fbuf can frequently be avoided by exploiting spatial locality in interprocess communication. When an fbuf is freed, it is put on a list of free fbufs associated with the set of domains that have knowledge of its address (i.e., hold its capability). When a program requests an fbuf, it specifies the set of domains that will be allowed access to the buffer. These domains are either trusted, or will eventually receive the fbuf. The allocator can then return any free fbuf that is known to any subset of the specified set of domains. Fbufs reused in this way generally already have physical memory associated with them, and can thus be used without generating a fault. Allocation of a new fbuf is necessary only when all the appropriate free-lists are empty.

In other words, fbufs act like group-wise shared buffers, except that the group of sharing domains can be extended simply by passing the buffer's address to a new domain. One can therefore view fbufs as *dynamically shared buffers*. Virtual-to-physical memory map manipulations occur only when (1) a new fbuf is created, which has as yet no physical memory mapped, and (2) when the kernel decides to reclaim physical memory associated with an unused fbuf. Note that (1) occurs only when no fbuf can be reused, and (2) never happens in the critical path of a data transfer. Thus, allocating, transferring, and deallocating an fbuf requires no kernel involvement in the common case. A page fault and subsequent allocation of physical memory is required only when a new fbuf allocation is necessary.

Preliminary experiments show that a buffer facility based on this approach performs significantly better than traditional copy-on-write solutions. Our analysis suggests that the reason for this better performance is that unlike existing techniques, our facility does not depend on (1) kernel involvement, or (2) fast execution of virtual memory mapping changes.

3.3 Integrated Layer Processing

Experience with network protocols point to *data manipulation*—reading and writing the data contained in a message—as the costliest aspect of data transfer. Data manipulations include such protocol functions as encryption, presentation formatting, and error detection via checksums. While other elements of data transfer have per-message costs, data manipulations involve processing proportionate to the amount of data in the message; i.e., a per-byte cost.

The reason manipulating the data in a message is expensive is that it involves loading each data word from the memory hierarchy and sometimes storing modified data back. The central problem is that the memory loads and stores required to inspect/modify the data are much slower than other instructions on RISC architectures, and this trend is expected to get much worse.

Integrated Layer Processing (ILP) has been suggested [Clar90] as a strategy for reducing the number of memory accesses performed by data manipulations. Normally, data is passed from one protocol to the next in the form of a complete message. Any protocol that manipulates (reads or writes) the data in the message must access the memory in which the data resides. ILP involves combining data manipulations from multiple protocols so that the data need only be read (and possible written) once by that series of protocols.

A straightforward approach to combining data manipulations would be to program a customized implementation for each combination of protocols. Thus, each combination of protocols would require a new implementation, even if the individual protocols or other combinations of them had been previously implemented. In contrast, our approach emphasizes flexible composition of protocols. It supports the independent implementation of protocols, which may be later composed with data manipulations integrated, without any reimplementation.

Towards this end, we are applying ILP in a way that preserves the advantages of flexibly configured protocol stacks composed of independently implemented protocols. Under our design, each protocol has an implementation independent of other protocols with which it might be composed. Previously implemented protocols are composed in accordance with later-specified configurations. The resulting protocol graph incorporates ILP for high performance.

Specifically, protocols are written in C and conform to a standard interface that makes it possible to combine them in arbitrary ways. Send and receive operations are written not as integral functions, but as collections of blocks of code. At system configuration time—i.e., when the protocol graph is specified—the integrated version of the particular protocol suite is constructed by interleaving the send and receive code of the protocols with each other and with system-provided code. At runtime, message data is passed between layers one word at a time. In the common case, a data word is passed from one layer to the next by simple sequencing of instructions. The word is held in a register made available to all the integrated layers.

In addition to the data manipulation phase, each protocol's send or receive operation includes two other execution phases. Tasks such as reading and writing headers and updating protocol state take place in the initial and final phases, which precede and follow the data manipulation phase, respectively. Unlike the data manipulation phase, the initial and final phases execute serially with respect to other protocols. That is, one layer's initial or final phase completes before the next layer's begins. If a series of protocols is integrated, then their initial phases are executed in sequence, then a combined data manipulation loop is executed, and then their final phases are executed in sequence.

3.4 Enchanced Network Controllers

An important issue in the design of host-network interfaces is how much protocol processing should be done in hardware (i.e., on the network device) and how much should be done in software (i.e., on the host). This issue is becoming increasingly important as improving network performance puts more and more stress on the end host. In contrast to other research efforts that are moving one or more protocols onto the network controller, we are designing a new network interface, called Chimera, that implements a collection of *protocol-independent functions* [Bail92]. Chimera does not replace the software implementations of protocols, it just provides hardware assist to these protocols.

Chimera is an outgrowth of our work on the *x*-kernel. One of the design philosophies of the *x*-kernel has been to identify tasks that are common to all protocols, and to provide general tools that support these tasks. Chimera takes this philosophy one step further by moving certain aspects of the support tools into hardware, with the ultimate goal of vertically integrating the network device and the network software infrastructure.

Our decision as to what functionality to move into the network device revolves around three factors: processor cycles, memory cycles, and resource scheduling. In the case of processor cycles, the goal is to minimize the overhead involved in servicing network interrupts. In the case of memory cycles, the goal is to minimize the number of times a network packet is moved from one buffer to another. Finally, we recognize the need to distinguish among different classes of packets, so that policy decisions can be made with regard to allocating various resources (e.g., processor, display, outgoing network connections) to those packets.

We have identified four key functions that we believe should be provided in the network device: demultiplexing, reassembly, header/data separation, and simple data manipulations. Intuitively, moving these functions into hardware takes advantage of the fact that network packets arriving at a particular host exhibit *temporal locality* (a sequence of packets addressed to the same process arrive in a brief period of time) and *spatial locality* (a small, slowly changing set of processes are receiving packets at any given time) [Mogu91].

Chimera is composed of two parts—a custom VLSI matcher chip and a set of FPGA router chips. The matcher chip determines the type of the incoming data stream by comparing specific fields in the packet with packet-type patterns the user has stored in the chip. The router chips separate the header and data information, compute the actual destination memory addresses, and optionally apply some filter to the data (e.g., checksum). Additionally, the router is responsible for interrupting the processor when reassembly is "complete". By having the router put the incoming message into a particular buffer based on the output of the matcher, the hope is that the message will not need to be copied to other buffers at a later time.

Packets enter Chimera as a byte stream. As this byte stream passes through the matcher chip, it determines whether the current packet matches any of the packet-type patterns which have been stored on the chip. Packet types are identified by comparing a set of byte stream offset/value/mask triples which uniquely identify the packet type. If there is a match, then the packet is partitioned according to the "scatter" information stored in the router for this packet type. These byte stream partitions are then output together with the appropriate memory addresses for placement in primary memory. The router provides two methods to generate interrupts, providing interrupts on specific packet types or after the completion of a group of packets.

3.5 Cluster-Based Multicomputer

The use of a cluster of standard workstations connected via a shared media network as a platform for high-performance computing has long been appealing. Such a system has obvious hardware cost advantages over special-purpose hardware. In addition, if such a cluster can use standard operating system software, the potential software cost advantages are also quite large—the development of application and system software for novel architectures is extremely expensive. The problem with such clusters is that the large latency found in most off-the-shelf networks (or more accurately most off-the-shelf network host interfaces) limits the usefulness of such system to very course-grained applications. The goal of this project is to demonstrate a prototype cluster that increases the class of problems that can be addressed by such systems while preserving both the hardware and software cost advantages of the architecture.

The most critical part of the prototype cluster is a low-latency communications subsystem. Increasing the range of problems that can be solved using a cluster architecture is dependent upon achieving a communications latency that is significantly lower than that commonly found in workstations clusters. We are taking a joint hardware/software approach to the design of this subsystem based upon WITLESS interface technology available on the Afterburner board, and the *x*-kernel network software technology.

WITLESS is an approach that involves the careful coupling of minimal hardware support with advanced host interface software design to produce very low latencies using traditional network protocols; it is analogous to the RISC philosophy of moving complexity out of the processor and into the compiler. The interface boards are simple, and they lower both hardware and software costs. Hardware costs are reduced due to the low chip count. Software costs are reduced because no board level programming is required.

While the WITLESS interface was designed to improve the performance of a specific protocol (in this case TCP), we believe that it is possible to provide improved performance to a much wider class of protocols.

What is more, we believe that we can do so using existing protocol source code with little or no modification. This can be achieved by carefully integrating support for the WITLESS interface into the *x*-kernel rather than into any specific protocol or protocols. Specifically, we will extend the *x*-kernel in such a way that *all* protocols implemented to the *x*-kernel interface can enjoy the performance benefits offered by the WITLESS interface. The key to doing this is extending the *x*-kernel message abstraction so it can take advantage of VRAM on the board. This is in contrast to previous efforts to optimize a single protocol suite (e.g., TCP/IP) to exploit the WITLESS interface.

3.6 Image Transfer

In the near future, members of the scientific community will require the ability to transfer extremely large amounts of digital imagery—still images typically ranging in size from 1MByte to 25MBytes. Earth observing platforms will be collecting data continuously at a rate of several hundred gigabytes per day, with researchers around the world performing interdisciplinary, multi-sensor experiments using data from distributed archives. At this collection rate, these users could become one of the largest consumers of wide-area network bandwidth. In order to use this volume of data effectively, it is necessary to give researchers the fastest possible transfer methods.

This project addresses the problem of effectively transferring digital images across high-speed, wide-area networks. In contrast to the way images are currently transferred using general purpose bulk transfer protocols, our approach is based on an end-to-end design that considers all aspects of the problem, including user requirements, characteristics of the data, data compression, and network performance [Turn92]. In other words, rather than use protocols designed to reliably transport arbitrary data, our design is tailored to the transport of digital images.

One important property of satellite images is that almost every aspect of remote sensing involves high degrees of uncertainty. Atmospheric turbulence, variations in solar illumination, and viewing geometry all influence the image formation process. Furthermore, errors are introduced by sensor optics and detectors, as well as during down-link transmission. At typical image resolutions, there is also a high degree of surface brightness variation due to differences in material reflectance, topography, lighting, climate, and so on. The net result is that the remote sensing community has designed applications that are able to deal with systematic uncertainty in the input data. For example, browsing large databases for candidate tests images, quicklook viewing prior to sensor activation, and large scale thematic mapping are applications that are particularly tolerant of imperfect data. Because the data already contains substantial errors and the applications can tolerate these errors, our image transfer protocol is able to relax the reliability guarantees of conventional bulk transfer protocols.

The second significant property of satellite images is spatial redundancy, or highly correlated brightnesses within local neighborhoods. When observed from space, surface brightness changes gradually in most scenes. Additionally, many of the same processes that contribute to image uncertainty do so in such a way as to smooth out irregularities at the pixel and local neighborhood levels—individual pixels actually represent an average brightness of the underlying substances. Because the images contain redundant information, our image transfer protocol is able to recover from errors (i.e., dropped packets) introduced by the network. This is very much like forward error correction, except we take advantage of the spatial redundancy already present in the images instead of explicitly adding redundant information to the data prior to transmission.

Our approach is to not retransmit lost packets, but instead to use the spatial redundancy in the data to recover the missing information. Although intuitively simple, our design is complicated by three factors. First, it is necessary to minimize the impact of each lost packet on image quality. This is accomplished by encoding images in such a way as to insure that the pixels contained in a single packet are scattered throughout the image. Then, when a packet is lost, redundancy in the remaining image data can be used to reconstruct lost information. Second, in order to support a wide range of network bandwidths, it is sometimes necessary to compress the data. Therefore, our encoding algorithm also preserves those qualities of the image that support data compression. Third, we have designed our protocol in such a way that all packets can be processed independently and in any order. This has benefits with respect to parallel protocol design and overall system simplification.

The key characteristic of our design is that by properly encoding images in network packets, we are able to avoid the retransmission of lost packets. Avoiding retransmission has two important advantages. First, retransmission-free protocols can improve end-to-end response time. Although raw network bandwidth is improving, the effective throughput rate of a retransmit-based protocol will not improve proportionally if the network suffers from either: (1) high latency, or (2) significant packet loss. For example, a 1Gbps network is able to transport a 10Mbyte image in 100ms. If even one packet needs to be retransmitted, then the response time doubles on a wide-area network with 100ms round-trip latency. Should a significant number of packets be dropped, then multiple round-trip times may be necessary to successfully transmit the data, further degrading performance.

Second, retransmission-free protocols drastically reduce the buffer space required on the sending machine. Protocols that retransmit lost packets must buffer all transmitted data until it has been acknowledged. For example, on a 1Gbps network with 100ms round trip latency, a reliable protocol like TCP would either need to maintain 12.5 Mbytes of buffer space for each image it is transmitting, or reduce its window size, and consequently, its effective throughput rate.

In support of image transfer, we are designing and implementing a new protocol called LTP—Lossy Transport Protocol. With LTP, the receiver measures the loss rate it is experiencing, and uses feedback to instruct the sender to increase or decrease its transmission rate accordingly. The idea is that LTP send packets at an aggressive rate with the expectation that some percentage of the packets will be dropped. LTP is lossy in the sense that it does not retransmit dropped packets.

3.7 Protocol Logistics

While the performance and size of computer networks has increased exponentially over the last decade, the set of network protocols used in those networks has remained essentially static. Moreover, although the existing Internet protocols have actually performed quite well until recently, the need to replace or enhance them is now critical. For example, the current version of TCP may not work correctly and limits performance on very high speed networks, while the current version of IP is predicted to run out of routing table space in 6-12 months, Class B addresses in two years, and all IP addresses in seven years. Unfortunately the same network growth that drives protocol change makes it difficult to actually distribute new protocols or modifications to existing protocols. Distributing a new protocol or modifying an existing protocol on a globally distributed network is both quantitatively and qualitatively different from making such changes in the small scale networks that existed in the early 1980s. Making changes on this scale is both enormously expensive and extremely slow.

We believe that both the cost and time required to distribute network software must be reduced and that doing so requires a new *logistical system* for network software. Network logistics concerns the creation, maintenance, and distribution of network software. The logistical system currently used by the Internet is ad-hoc, manual, and depends heavily upon surface mail to deliver network software updates. The key is to design a logistical infrastructure that can support exponential network growth. This will require decoupling network software from the operating system and the reliable distribution of network software over computer networks, with the goal of reducing the time required to distribute a new or modify an existing protocol to days rather than years [O'Ma91].

As part of this logistical system, we are investigating *protocol evolution* as an approach to protocol change that escapes the limitations of backward compatibility without incurring all of the costs of creating new protocols. The basic idea is for the protocol designer to take an existing protocol that requires modification and make the desired changes without maintaining backward compatibility. This drastically simplifies the job of the protocol designer. For example, the limited TCP window size could be fixed by changing the definition of the window size in the header from 16-bits to 32-bits, and re-compiling the protocol. The effect of backward compatibility would be ensured by simply keeping both the new and old version of the protocol running until most machines use the new version. Since the change is small and invisible to the user interface, it is a trivial problem to dynamically select the correct TCP version at runtime.

The effective use of protocol evolution implies that each protocol be considered a vector of implementations that share the same top level interface, and perhaps not much else. For example, TCP(0) is the current implementation of TCP and exists to provide backward compatibility with all existing machines. TCP(1) is a version of TCP that is optimized for high-speed networks. TCP(0) is always present; TCP(1) may or may not be. A version control protocol, called VTCP, is used to select the appropriate version of TCP for a particular connection based on cached knowledge of the highest version shared in common with the destination host. VTCP is an example of an *x*-kernel virtual protocol.

4 Acknowledgements

This work is supported in part by DARPA Contract DABT63-91-C-0030, NSF Grants CCR-9102040 and IRI-15407, and the Hewlett-Packard Company.

The Networking Research Group at the University of Arizona includes Mark Abbott, Mary Baily, Larry Brakmo, Peter Druschel, Ed Menze, David Mosberger, Sean O'Malley, Hilarie Orman, Michael Pagels, Richard Schroeppel, and Charlie Turner.

References

[Acce86] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A new kernel foundation for Unix development. In *Proceedings of the USENIX Summer '86 Conference*, July 1986.

[Bail92] Baily, M., Pagels, M., and Peterson, L. The *x*-chip: An experiment in hardware demultiplexing. In *Workshop on High-Performance Communications Subsystems*, February 1992.

[Clar90] Clark, D. D. and Tennenhouse, D. L. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Symposium*, pages 200–208, September 1990.

[Davi91] Davie, B. S. A host-network interface architecture for ATM. In *Proceedings SIGCOMM '91 Conference on Communications Architectures and Protocols*, 1991.

[Drus92a] Druschel, P. and Peterson, L. L. High-performance cross-domain data transfers. Technical Report 92-11, Department of Computer Science, University of Arizona, March 1992.

[Drus92b] Druschel, P., Peterson, L. L., and Hutchinson, N. C. Beyond micro-kernel design: Decoupling modularity and protection in Lipto. In *Proceedings of the Twelfth International Conference on Distributed Computing Systems*, pages 512–520, Yokohama, Japan., June 1992.

[Golu90] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an application program. In *Proceedings of the USENIX Summer '90 Conference*, pages 87–95, June 1990.

[Hutc91] Hutchinson, N. C. and Peterson, L. L. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[Luml92] Lumley, J. The afterburner network interface. In *Workshop on High-Performance Communications Subsystems*, February 1992.

[Mogu91] Mogul, J. C. Network locality at the scale of processes. In *Proceedings SIGCOMM '91 Conference on Communications Architectures and Protocols*, 1991.

[O'Ma91] O'Malley, S. W. and Peterson, L. L. TCP extensions considered harmful. Request for Comments 1263, University of Arizona, October 1991.

[O'Ma92] O'Malley, S. W. and Peterson, L. L. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2), May 1992.

[Turn92] Turner, C. J. and Peterson, L. L. Image transfer: An end-to-end design. In *Proceedings of the SIGCOMM '92 Symposium*, Baltimore, Maryland, August 1992.