EDGE

Project report

Project on Implementing Network protocols at user level

Course title; Basic Computer Networking
Course Batch; Networking-002

Supervised by

Md. Rashid Al Asif

Assistant Professor

Department of cse

University of Barishal

Submitted by

Md.Hasibul Islam

Student ID; 02-002-12

Batch: Basic Networking -002

Date of Submission: 15 November,2024

Abstract

Traditionally, network software has been structured in a monolithic fashion with all protocol stacks executing either within the kernel or in a single trusted user-level server. This organization is motivated by performance and security concerns. However, considerations of code maintenance, ease of debugging, customization, and the simultaneous existence of multiple protocols argue for separating the implementations into more manageable user-level libraries of protocols. This paper describes the design and implementation of transport protocols as user-level libraries.

We begin by motivating the need for protocol implementations as user-level libraries and placing our approach in the context of previ- ous work. We then describe our alternative to monolithic protocol organization, which has been implemented on Mach workstations connected not only to traditional Ethernet, but also to a more mod- ern network, the DEC SRC AN1. Based on our experience, we discuss the implications for host-network interface design and for overall system structure to support efficient user-level implemen- tations of network protocols.

1.Introduction

1.1Motivation

typically, network protocols have been implemented inside the kernel or in a trusted, user-level server [10, 12]. Security and/or performance are the primary reasons that favor such an organi- zation. We refer to this organization as monolithic because all protocol stacks supported by the system are implemented within a single address space.

The goal of this paper is to explore alternatives to a monolithic structure. There are several factors that motivate protocol imple- mentations that are not monolithic and are outside the most obvious of these are ease of prototyping, debugging, and maintenance. T\vo more interesting factors are:

1. The co-existence of multiple protocols that provide materi- ally differing services, and the clear advantages of easy ad- dition and extensibility by separating their implementations into self-contained units.

2. The ability to exploit application-specific knowledge for im- proving the performance of a particular communication pro- tocol.

We expand on these two aspects in greater detail below.

Multiplicity of Protocols

Over the years, there has been a proliferation of protocols driven primarily by application needs.

For example, the need for an efficient transport for distributed systems was a factor in the development of request/response pro- tocols in lieu of existing byte-stream protocols such as TCP [2]. Experience with specialized protocols shows that they achieve re- markably low latencies. However these protocols do not always deliver the highest throughput [3]. In systems that need to sup- port both throughput-intensive and latency-critical applications, it is realistic to expect both types of protocols to co-exist.

We expect the trend towards multiple protocols to continue in the future due to at least three factors.

Emerging communication modes such as graphics and video, and access patterns such as request-response. bulk transfer, and real-time, will require transport services which may have differing characteristics. Further, the needs of integration require that these transports co-exist on one system.

Future uses of workstation clusters as message passing multi- computers will undoubtedly influence protocol design: efficient implementations of this and other programming paradigms will drive the development of new transport protocols.

As newer networks with different speed and error characteristics are deployed, protocol requirements will change. For example, higher speed, low error links may favor forward error correcfion and rate-based flow control over more traditional protocols (7]. Once again, if different network l inks exist at a single site, multiple protocols may need to co-exist.

Exploiting Application Knowledge

In addition to using special purpose protocols for different ap- plication areas, further performance advantages may be gained by exploiting application-specific knowledge to fine tune a partic- ular instance of a protocol. Watson and Mamrak have observed that conflicts between application-level and transport-level abstrac- tions lead to performance compromises [26]. One solution to this is to “partially evaluate” a general purpose protocol with respect to a particular application. In this approach, based on applica- tion requirements, a specialized variant of a standard protocol is used rather than the standard protocol itself. A different applica- tion would use a slightly different variant of the same protocol. Language-based protocol implementations such as Morpheus [1] as well as protocol compilers [9) are two recent attempts at exploit- ing user specified constraints to generate efficient implementations of communication protocols.

The general idea of using partial evaluation to gain better UO performance in systems has been used elsewhere as well [15]. In particular, the notion of specializing a transport protocol to the needs of a particular application has been the motivation behind many recent system designs [11, 20, 24].

1.2 Alternative Protocol Structures

The discussion above argues for alternatives to monolithic protocol implementations since they are deficient in at least two ways. First, having all protocol variants executing in a single address space (especially if it is in-kernel) complicates code maintenance, de- bugging, and development. Second, monolithic solutions limit the ability of a user (or a mechanised program) to perform application- specific opfimizations.

In contrast, given the appropriate mechanisms in the kernel, it is feasible to support high performance and secure implementa- tions of relatively complex communication protocols as user-level libraries.

Figure 1 shows different alternatives for stnicturing communi- cation protocols.

Surprisingly, traditional operating systems like UNIX and mod- em microkernels such as Mach 3.0 have similar monolithic protocol organizations. For instance, the Mach 3.0 microkernel implements protocols outside the kernel within a trusted user-level server The code for all system-supported protocols runs in the single, trusted, UX server's address space. There are at least three vari- ations to this basic organization depending on the location of the network device management code, and the way in which the data is moved between the device and the protocol server. In one variant of the system, the Mach/UX server maps network devices into its address space, has direct access to them, and is functionally similar to a monolithic in-kernel implementation. In the second variant device management is located in the kernel. The in-kernel device driver and the UX server communicate through a message based interface. The performance of this variant is lower than the one with the mapped device [I 0). Some of the performance lost due to the message based interface can potentially be recovered by using a third variant that uses shared memory to pass data between the device and the protocol code as described in [19].

One alternative to a monolithic implementation is to dedicate a separate user-level server for each protocol stack, and sepa- rate server(s) for network device management. This arrange-ment has the potential for performance problems since the critical send/receive path for an applicafion could incur excessive domain- switching overheads because of address space crossings between the user, the protocol server, and the device manager. That is, given identical implementations of the protocol stack and support func- tions liKe buffering, layering and synchronization, inter-domain crossings come at a price. Further, and perhaps more importanfly, this arrangement, like the monolithic version, does not permit easy exploitation of application-level information.

Perhaps the best known example of this organization was done in the context of the Packet Filter [18]. This system implemented

packet demultiplexing and device management within the kernel and supported implementations of standard protocols such as TCP and VMTP outside the kernel. It did not rely on any special-purpose hardware or on extensive operating system support. Several pro-tocols including the PUP suite and VMTP were implemented. A similar organization for implementing UDP is described in [13].

Another alternative, the one we develop in this paper, is to orga- nize protocol functions as a user linkable library. In the common case of sends and receives, the library talks to the device manager without involving a dedicated protocol server as an intermediary. (Issues such as security need to be addressed in this approach and are considered in greater detail in Section 3.)

An earlier example of this approach is found in the Topaz im- plementation of UDP on the DEC SRC Firefly [23). Here the UDP library exists in each user address space. However, this design has some limitations. First, UDP is an unreliable datagram service, and is easier to implement than a protocol like TCP. Second, the design of Topaz trades off strict protection for increased performance and ease of implementation of protocols. A more recent example of encapsulating protocols in user-level libraries is the ongoing work at CMU [14]. This work shares many of the same objectives as ours but, like the Topaz design, does not enforce strict protection between communicating endpoints.

1.3 Paper Goals and Organization

The primary goal of this paper is to explore high-performance implementations of relatively complex protocols as user libraries. We believe that efficient protocol implementation is a matter of policy and mechanism. That is, with the right mechanisms in the kernel and support from the host-network interface, protocol im- plementation is a matter of policy that can be performed within user libraries. Given suitable mechanisms, it is feasible for li- brary implementations of protocols to be as efficient and secure as traditional monolithic implementations.

We have tested our hypothesis by ñnplementing a user-level library for TCP on workstation hosts running the Mach kernel con- nected to Ethernet and to the DEC SRC AN1 network [21]. We chose TCP for several reasons. First, it is a real protocol whose level of detail and functionality match that of other communication protocols; choosing a simpler protocol like UDP would be less con- vincing in this regard. Second, we could expeditiously reuse code from one of the many existing implementations of the protocol. Since these implementations are mature and stable, performance comparisons with monolithic implementations on similar hard- ware are straightforward and unlikely to be affected by artifacts of bad or incorrect implementation. Finally, our experience with a connection-oriented protocol is likely to be relevant in networks like ATM that appear to be biased towards connection-oriented approaches.

The rest of the paper is organized as follows. Section 2 describes the necessary kernel and host-network interface mechanisms that aid efficient user-level protocol implementations. Section 3 details the structure, design and implementation of our system. Section 4 analyzes the performance of our TCP/IP implementation. Section 5 offers conclusions based on our experience and suggests avenues for future work.

2.Mechanisms for User-Level Protocol Implementation

In this section, we discuss some of the fundamental system mecha- nisms that c; i help in efficient user-level protocol implementation. The underpinnings of efficient communication protocols are one or more of:

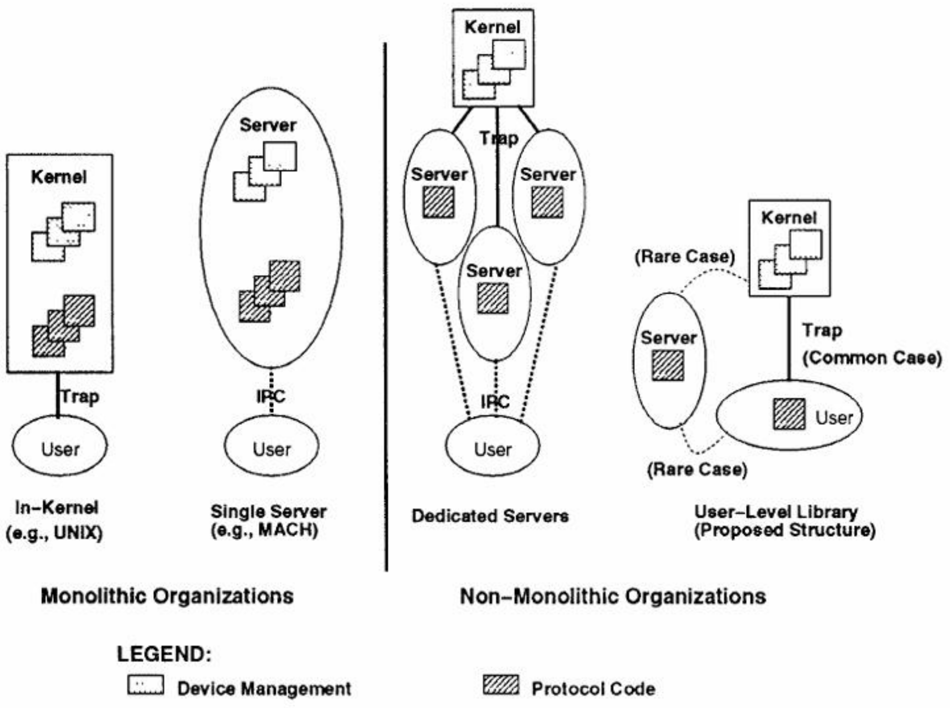


Figure 1: Alternative Organizations of Protocols

1. Lightweight implementation of context switches and timer

events.

2. Combining (or eliminating) multiple protocol layers.

3. Improved buffering between the network, the kernel, and the user, and elimination of unnecessary copies.

The first two items — lightweight context switching, layering, and timer implementafions — have already been studied in earlier systems and are largely independent of whether the protocols are located in the kernel or in user libraries. We therefore briefly summarize the impact of these factors in Section 2.1, and then concentrate for the most part on the buffering and packet delivery mechanisms, where innovation is needed.

2.1 Layering, Lightweight Threads, and Fast Timer Operations

Transport protocol implementations can benefit from being mul- tithreaded if inter-thread switching and synchronization costs are kept low. Older operafing systems such as UNIX do not pro- vide the same level of support for multiple threads of control and synchronization in user space as they do inside the kernel. Conse- quently, user-level implementations of protocols are more difficult and awkward to implement than they need to be. With more modern operating systems, which support lightweight threads and synchro- nization at user-level, protocol implementation at user-level enjoys the facilities that more traditional implementations exploited within the kernel.

Issues of layering, lightweight contextswitching and timers have been extensively studied in the literature. Examples include Clark's Swift system [4], the x-kernel [11], and the work by Watson and Mamrak (26]. It is well known that switching between processes that implement each layer of the protocol is expensive, as is the data copying overhead. Proposed solutions to the problem are generally variations of Clark's multitask modules, where context

switches are avoided in moving data between the various transport layers. Additionally, there are many well understood mechanisms for fast context switches, such as continuafions [8] and others. Timer implementations also have a profound impact on transport performance, because practically every message arrival and depar- ture involves timer operations. Once again, fast implementations of timer events are well known, e.g., using hierarchical timing wheels [25].

2.2 Efficient Buffering and Input Packet Demul- tiplexing

The buffer layer in a communication system manages data buffers between the user space, the kernel and the host-network interface. The security requirements of the kernel transport protocols, and the support provided by the host-network interface, all contribute to the complexity of the buffer layer.

A key requirement for user-level protocols is that the buffer layer be able to deliver network packets to the end user as efficiently as possible. This involves two aspects — (1) efficient demultiplexing of input packets based on protocol headers, and (2) minimizing unnecessary data copies. Demultiplexing functions can be located in two places: either in hardware in the host-network interface, or in software, in the kernel or as a separate user-level demultiplexer. In any case, demultiplexing has to be done in a secure fashion to prevent unauthorized packet recepfion. We describe below two approaches to support input packet delivery that can benefit user-level protocol implementations.

son are Support for Packet Delivery

Typically, there are multiple headers appended to an incom- ing packe(for example, a link-level header, followed by one. or more higher-level protocol headers. Ideally, address demultiplex- ing should be done as low in the protocol stack as possible, but should dispatch to the highest protocol layer [22]. This is usually not done in hardware because the host-network interface is typi- cally designed for link-level protocols and has no knowledge of higher level protocols. As a specific example, a TCPfP packet on an Ethernet link has three headers. The lirik-level Ethernet header only identihes the station address and the packet type — in this case, IP. This is not sufficient information to determine the final user of the data, which requires examining the protocol control block maintained by the TCP module.

In the absence of hardware support for address demultiplexing, the only realistic choice is to implement this in software inside the kernel. The alternative of using a dedicated user-level process to demultiplex packets can be very expensive because multiple context switches are required to deliver network data to the final destination. In the past, software implementations of address de- multiplexing have offered flexibility at the expense of performance and have ignored the issues of multiple data copies.

For example, the original UNIX implementation of the Packet Filter [18) features a stack-based language where "filter programs" composed of stack operations and operators are interpreted by a kernel-resident program at packet reception time. While the interpretation process offers flexibility, it is not likely to scale with CPU speeds because it is memory intensive. Performance is more important than flexibility because slow packet demultiplexing tends to confine user-level protocol implementations to debugging and development rather than production use. The recent Berkeley Packet Filter implementation recognizes these issues and provides higher performance suited for modern RISC processors [17].

In the absence of hardware support, effective input demultiplex- ing requires two mechanisms:

1. Support for direct execution of demultiplexing code within the kernel.
2. Support for protected packet buffer sharing between user space and the kernel.

Neither of thèse facilities is very difficult to implement. The logic required for address demulüplexing is simple and can be incor- porated into the kemel either via run time code synthesis or via compilation when new protocols are added [16]. Based on our experience, the demultiplexing logic requises only a few instruc- tions. In addition, virtual memory operations can be exploited so that the user-level library and the kemel can securely share a buffer area. Section 3 describes how thèse mechanisms are exploited in our design to achieve good performance without compromising security.

Hardware Support for Demultiplexing

In general, older Ethernet host-network interfaces do not pro- vide support for packet demultiplexing because it is not possible to accurately determine the final destination of a packet based on link-level fields alone. Intelligent host-network interfaces that offload protocol processing from the host are capable of packet demultiplexing, but their utility is limited to a single protocol at a time. Newer networks such as AN1 and ATM have fields in their link-level headers that may be used to provide support for packet demultiplexing.

Host-network interfaces can be built to exploit these link-level fields to provide address demultiplexing in a protocol-independent manner. As an example, the host-network interface that we use on the ANl network has hardware that delivers network packets to the final destination process. In the AN1 controller a single field (called the buffer queue index, BQI) in the link-level packet header provides a level of indirection into a table kept in the controller. The table contains a set of host memory address descriptors, which specify the buffers to which data is transferred. Strict access con- trol to the index is maintained through memory protection. In

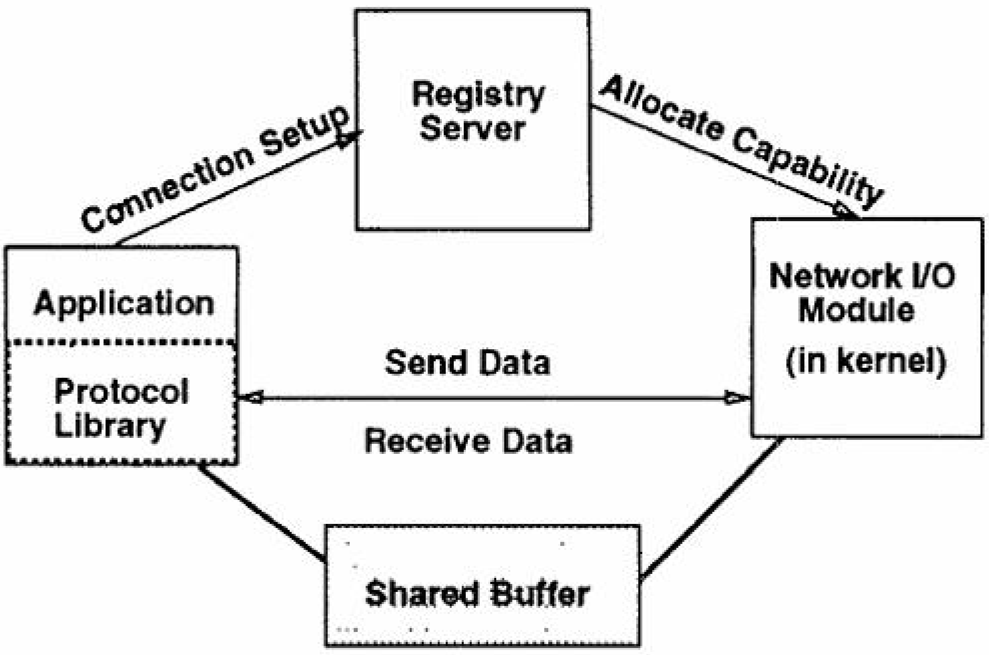


Figure 2: Structure of the Protocol Implementation

a connection-based protocol such as TCP, the index value can be agreed upon by communicating entities as part of connection setup. Connectionless protocols can also use this facility by “discovering” the index value of their peer by examining the link-level headers of incoming messages. Section 3.4 discusses this mechanism in the context of our implementation.

In considering mechanisms for packet delivery, two overall com- ments are in order. First, hardware support for packet demultiplex- ing is applicable only as long as the link level supports it. In the cases where a packet has to traverse one or more networks without a suitable link header field, demultiplexing has to be done in soft- ware. Second, details of the packet demultiplexing and delivery scheme are shielded from the application writer by the protocol library that is linked into the application. The application sees whatever abstraction the protocollibrary chooses to provide. Thus, programmer convenience is not an issue with either a software or hardware packet delivery scheme.

3. Design and Implementation of User-Level Protocols

3.1 Design Overview

This section describes our design at a high level. In our design, pro- tocol functionality is provided to an application by three interacting components — a protocol library that is linked into the application, a registry server that runs as a privileged process, and a network UO module that is co-located with the network device driver. Figure 2 shows an overall view of our design and the interaction between the componeqts.

The library contains the code that implements the communi- cation protocol. For instance, typical protocol functions such as retransmission, flow control, checksumming, etc., are located in the library. Given the timeout and retransmission mechanisms of reliable transport protocols, the library typically would be multi- threaded. Applications may link to more than one protocol library at a fime. For example, an application using TCP will typically link to the TCP, IP, and ARP libraries.

The regiswy server handles the details of allocafing and Real- locating communication end-points on behalf of the applications. Before applications can communicate with each other, they have to be named in a mutually secure and non-conflicting manner. The registry server is a trusted piece of software that runs as a privileged process and performs many of the functions that are usually im- plemented within the kernel in standard protocol implementations.

There is a dedicated registry server for each protocol.

The third module implements network access by providing effi- cient and secure input packet delivery, and outbound packet trans- mission. There is one network I/O module for each host-network interface on the host. Depending on the support provided by the host-network interface, some of the functionality of this module may be in hardware.

Given the library, the server, and the network I/O module, ap- plications can communicate over the network in a straightforward fashion. Applications call into the library using a suitable interface to the transport protocol (e.g., the BSD socket or the AT&T TLI in- terface). The library contacts the registry server to negotiate names for the communication entities. In connection-oriented protocols this might require the server to complete a connection establish- ment protocol with a remote entity. Before returning to the library, the registry server contacts the network I/O module on behalf of the application to set up secure and efficient packet delivery and transmission channels. The server then returns to the application library with unforgeable tickets or capabilities for these channels. Subsequent network communication is handled completely by the user-level library and the network I/O module using the capabil- ities that the server returned. Thus, the server is bypassed in the common path of data transmission and recepfion.

Our organization has some tangible benefits over the alternative approaches of a monolithic implementation, or having a dedicated server per protocol stack. Our approach has software engineering arguments to recommend it over the monolithic approach. More importantly, our structure is likely to yield better performance than a system that uses a single dedicated server per protocol stack for two reasons. First, by eliminating the server from the common- case send and receive paths, we reduce the number of address space transitions on the critical path. Second, we open the possibility of additional performance gains by generating applicafion-specific protocols.

Our approach is not without its disadvantages, however. Each application links to a communication library that might be of sub- stantial size. This could lead to code bloat which might stress the VM system. This problem can be solved with shared libraries and therefore is not a serious concern.

A more serious problem is that a malicious (or buggy) applica- tion library could jam the network with data, or exceed pre-arranged rate requirements, or exhibit other anti-social behavior. Since in our design, the device management is still in the kernel, we could conceivably augment its functions to safeguard against malicious or buggy behavior. Even traditional in-kernel and trusted server implementafions only alleviate the problem of incorrect behavior but do not solve it as long as the network can be tapped by intrud- ers. We believe that administrative measures are appropriate for handling these types of problems.

To test the viability of our design, we built and analyzed the performance of a complete and non-trivial communication proto- col. We chose TCP primarily because it is a realistic connection- oriented protocol. We used Mach as the base operating system for our implementation. In Mach, a small kernel provides fundamen- tal operating system mechanisms such as process management, virtual memory, and IPC. Traditional higher level operating sys- tem services are implemented by a user-level server. We chose Mach because it provides user-level threads and synchronization, virtual memory operations to simplify buffer management, and un- forgeable capabilities in the form of Mach “port” abstractions, all of which are helpful in user-level protocol implementations. Of particular benefit are Mach’s “ports”, which form the basis for se- cure and trusted communication channels between the library, the server, and the network l/O module. We describe below the details of our implementation.

3.2 Protocol Library

When an application initiates a connection, the library contacts the registry server to allocate connection end-points (in our case, TCP ports). After the registry server finishes the connection establish- ment with the remote peer, the registry server returns a set of Mach ports to the library.

The Mach ports returned to the application contain a send ca- pability. In addition, a virtual memory region in the library is mapped shared with the particular I/O module for the network de- vice that the connection is using. This shared memory region is used to convey data between the protocol and the network device. Application requests to write (or read) data over a connection are translated into protocol actions that eventually cause packets to be sent (or received) over the network via the shared memory.

On transmissions, the library uses the send capability to identify itself to the network module. The network I/O module associates with the capability a template that constrains the header fields of packets sent using that capability. The network UO module verifies this against the library packet before network transmission. On receives, packet demultiplexing code within the network I/O module delivers packets to the correct and authorized end points. Additional details of this mechanism are described in Section 3.4. Once a connection is established, it can be passed by the appli- cation to other applications without involving the registry server oz the network I/O module. The port abstractions provided by the Mach kernel are sufficient for this. A typical instance of this occurs in UNIX-based systems where the Internet daemon (inetd) hands off connection end-points to specific servers such as the TELNET

or FTP daemons.

The protocol library is the heart of the overall protocol im- plementation. It contains the code that implements the various functions of the protocol dealing with data transmission and recep- tion. The protocol code is borrowed entirely from the UX server which in turn is based on a 4.3 BSD implementation. As mentioned earlier, to use TCP, support from other protocol libraries such as IP and ARP are needed. Our implementation of the IP and ARP libraries makes some simplifications. In particular, our IP library does not implement the functions required for handling gateway traffic.

Though the bulk of the code in our library is identical to a BSD kernel implementation, the structure of the library is slightly different. First the protocol library is not driven by interrupts from the network or traps from the user. Instead, network packet arrival notification is done via a lightweight semaphore that a li- brary thread is waiting on, and user applications invoke protocol functions through procedure calls. Second, multiple threads of control and synchronization are provided by user-level C Thread primitives [5] rather than kernel primitives. In addition, protocol control block lookups are eliminated by having separate threads per connection that are upcalled. Finally, user data transfer between the application and the network device exploits shared memory to avoid copy costs where possible. We describe the details of data transfer in Section 3.3.

While it is usually the case that transport protocols are standard- ized, the application interface to the protocol is not. This leads to multiple ad hoc mechanisms which are typically mandated by fa- cilities of the underlying operating system. For instance, the BSD socket interface and the AT&T TLI interface are typically found in UNIX-based systems. Non-UNIX systems have their own inter- faces as well. In our implementations, we provide some but not all the functionality of the BSD socket layer. The use of Mach ports allows many of the socket operations like sharing connections, waiting on multiple connections, and others to be implemented conveniently. Though a BSD-compliant socket interface was not a goal of our research, our functionality is close enough to run BSD applications. For instance, users of the protocol library continue to create sockets with socket, call bind to bind to sockets, and use connect, 1is Men, and accept to establish connections over sockets. Data transfer on connected sockets and regular files is done as usual with read and wri te calls. The library handles all the bookkeeping details. Our current implementation does not correctly handle the notions of inheriting connections via fork, or the semantics of select.

3.3 Network UO Module

The network I/O module is located with the in-kernel network device driver. There is a separate module for each network device. The primary function of the network UO module is to provide efficient and protected access to the network by the libraries.

All access to the network I/O module is through capabilities. Ini- tially, only the privileged registry server has access to the network module. At the end of connection establishment, the registry server and the network FO module collaborate in creating capabilities that are returned to the application. A region of memory is created by the network I/O module and the registry server for holding network packets. This memory is kept pinned for the duration of the con- nection and shared with the application. Incoming packets from the network are moved into the shared region and a notification is sent to the application library via a lightweight semaphore. Our implementation attempts, where possible, to batch multiple net- work packets per semaphore notification in order to amortize the cost of signaling.

The exact mechanism for transferring the data from the network to shared memory varies with the host-network interface. The DECstation hosts connect to the Ethernet using the DEC PMADD- AA host-network interface [6]. This interface does not have DMA capabilities to and from the host memory. Instead, there are special packet buffers on board the controller that serve as a staging area for data. The host transfers data between these buffers and host memory using programmed I/O. On receives, the entire packet, complete with network headers, is made available to the protocol code.

In contrast, the AN1 host-network interface is capable of per- forming DMA to and from host memory. Host software writes descriptors into on-board registers that describe buffers in host shared memory that will hold incoming packets. The controller allows a set of host buNers to be aggregated into a ring that can be named by an index called the buffer queue index (BQI). Incoming network packets contain a BQI field that is used by the controller in determining which ring to use. The controller initiates DMA into the next buffer in this ring and hands the buffer to the protocol library. When the library is done with the buffer it hands it back to the network module which adds it tn the BQI ring. As with the Ethernet controller, complete packets, including network headers, are transferred to shared memory.

On outbound packet transmissions, the library makes a system call into the network module. The system call arguments describe a packet in shared memory as well as supplying a send capability. The capability identifies the template, including the BQI in the case of the AN1, against which the packet header is checked.

In our design, the network I/O module and the library are both involved in managing the shared buffer memory. However, the end user application need not be aware of this memory management because the protocol library handles all the details. For the library, bookkeeping of shared memory is a relatively modest task com- pared to the buffer management that must be performed to handle segmentation, reassembly, and retransmission.

3.4 Registry Server

The registry server runs as a trusted, privileged process manag- ing the allocation and deallocation of communication end-points.

There are several reasons that a central, trusted agent is required to mediate the allocation of these end-points. First, connecfion end- points act as names of the communicating entities and are therefore unique across a machine for a particular protocol. Thus, having untrusted user libraries allocate these names is a security and ad- ministrative concern. Second, in many protocols (including TCP), connection state needs to be maintained after a connection is shut-down. A transient user linkable library is clearly not appropriate for this.

In connection-oriented protocols like TCP, connection estab- lishment and communication end-point allocation are often inter- twined. For example, the registry server for TCP executes the three- way handshake as part of the connection establishment. Thus, our organization can be logically thought of as the protocol library providing a set of functions to both the application and the reg- istry server. Each executes a different subset of the functionality provided in the library. The registry server, as part of allocating communication end-points, also transfers necessary state about the communication. Under normal operation, connection shutdown is done by the protocol library. However, when the application exits, the registry server inherits the connections and ensures that the pro- tocol specified delay period is maintained before the connection is reused. Resources allocated to the application and registered with the network I/O module are now reclaimed. To guard against an abnormal application termination, the protocol server issues a reset message to the remote peer.

While it is the case that the privileged server performs cer- tain necessary operations on behalf of the user application, better performance may be achieved by avoiding the server on all net- work transmission and reception. With this rationale, we explored organizations that were different from earlier user-level protocol implementations that used a server as an intermediary.

Protection Issues

With trusted applications, a simple structure is possible: the network device module exports read and write RPC interfaces that the application libraries invoke to transfer packets to and from the network. One might argue that since networks are easily tappable, trusting applications in this manner is not a cause for undue con- cern. However. this scheme provides markedly lower security than what conventional operating systems provide and what users have come to expect. In contrast, our scheme provides good security (no scheme can be completely secure without suitable encryption on the network) without sacrificing performance.

There are two aspects to protection. First, only entities that are authorized to communicate with each other should be able to communicate. Second, entities should not be able to impersonate others. Our scheme achieves the first objective by ensuring that applications negotiate connection setup through the trusted registry server. Without going through this process, libraries have no send (or receive) capability for the network. Impersonation is prevented by associating a header template with a send capability. When the network I/O module receives packets to be transmitted, it matches fields in the template against the packet header. Similarly, unautho- rized access to incoming packets is prevented because the registry server activates the address demultiplexing mechanism as part of the connection establishment phase.

The checks required for header matching on outgoing packets are similar to those needed for address demultiplexing on incoming network packets. Since our host-network controllers do not provide any hardware support for this, the logio required for this needs to be synthesized (or compiled) into the network I/O module. Usually, this code segment is quite short. Our scheme has the defect that it violates strict layering — the lower level network layer manipulates higher level protocol layers. We regard this as an acceptable cost for the benefit it provides.

In a typical local area environment, network eavesdropping and tapping are usually possible. Our scheme, like other schemes that do not use some form of encryption, does not provide absolute guarantees on unauthorized accesses or impersonation. However, our scheme can be augmented with encryption in the network I/O module if additional security is required.

Packet Demultiplexlng Issues

We described earlier the notion of the BQI that is provided by the host-network controller for demultiplexing incoming data. To summarize, the AN1 link header contains an index into a table that describes the eventual destination of the packet in a (higher-level) protocol independent way. BQI zero is the default used by the controller and refers to protected memory within the kernel. To use the hardware packet demultiplexing facility for user-level data transfer, non-zero BQIs have to be exchanged between the two parties. In our case, the server performs this function as part of the TCP three-way handshake.

Before inifiating connection the server requests the network I/O module for a BQI that the remote node can use. It then inserts the BQI into an unused field in the AN I link header which is extracted by the remote server. The remote server, as part of setting the template with the network I/O module, specifies the BQI to be used on outgoing packets. Subsequent packets have the BQI field set correctly in their link-level header. Since the handshake is three- way, both sides have a chance to receive and send BQIs before starting data exchanges. After BQIs have been exchanged at call setup time, all packets for that connection are transferred to host buffers in the ring for that BQI.

4. Performance

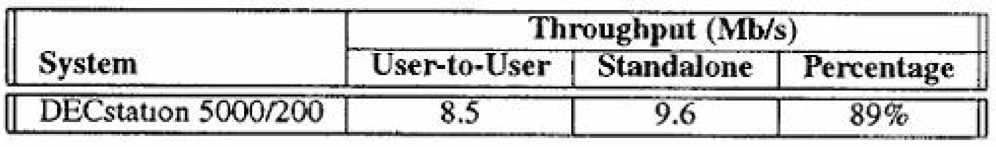
This section compares the performance of our design with mono- lithic (in-kernel and single-server) implementations. Our goal was to ensure that our design is competitive with kernel-level imple- mentations or the Mach single-server implementation, and there- fore superior to a user-level implementation that uses intermediary servers.

Our hardware environment consists of two DECstafion 5000/200 (25 MHz R30O0 CPUs) workstations connected to a 10 Mb/sec Ethernet, as well as to a switchless, private segment of a 100 Mb/sec ANl network.

In order to generate accurate measurements of elapsed time, we used a real-time clock that is part of the AN1 controller. This clock ticks at the rate of 40 ns and can be read by user processes by mapping and accessing a device memory location.

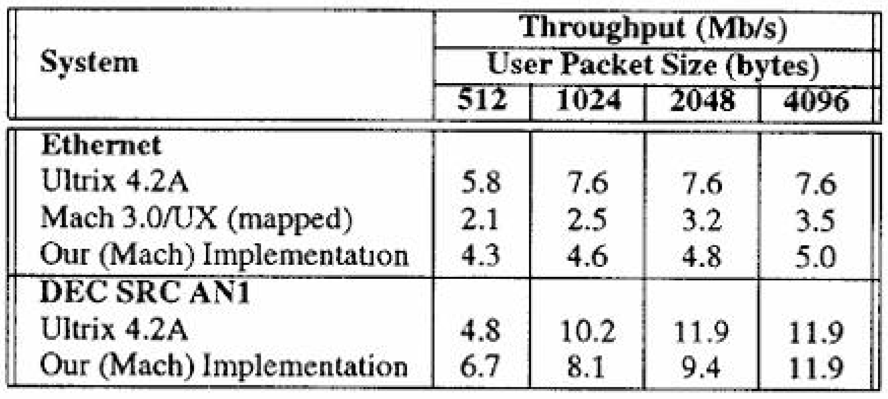
Impact of Mechanisms

First, we wanted to estimate the cost imposed by our mecha- nisms (shared memory, library-device signaling, protecfion check- ing in the kernel, software template matching, etc.) on the overall throughput of data transfer. To estimate this overhead, we ran a micro-benchmark that used two applications to exchange data over the 10 Mb/sec Ethernet, without using any higher-level protocols. All the standard mechanisms that we provide (including the library-kernel signaling) are exercised in this experiment. (However, this test does not exercise any of Mach's thread or synchronization primitives that a real protocol implementation would. Thus, a real- istic protocol implementation in our design is likely to have lower throughput than our benchmark. This can be attributed to two factors — inherent protocol implementation inefficiency, and the overheads introduced by using multiple threads, context switching, synchronization, and timers.



System	Throughput (Mb/s)		
	User-to-User	Standalone	Percentage
DECstation 5000/200	8.5	9.6	89%

Table 1: Impact of Our Mechanisms on Throughput



System	Throughput (Mb/s)			
	User Packet Size (bytes)			
	512	1024	2048	4096
Ethernet				
Ultrix 4.2A	5.8	7.6	7.6	7.6
Mach 3.0/UX (mapped)	2.1	2.5	3.2	3.5
Our (Mach) Implementation	4.3	4.6	4.8	5.0
DEC SRC AN1				
Ultrix 4.2A	4.8	10.2	11.9	11.9
Our (Mach) Implementation	6.7	8.1	9.4	11.9

Table 2: Throughput Measurements (in megabits/second)

Table 1 gives the measured absolute throughputs using maximum-sized Ethernet packets. For comparison, it also shows throughput as a percentage of the maximum achievable using the raw hardware with a standalone program and no operating system. (Note that the standalone system measurement represents link sat- uration when the Ethernet frame format and inter-packet gaps are accounted for.) Our measurements show that our mechanisms in- troduce only very modest overhead in return for their considerable benefits.

Throughput

Next, we compare the performance of our library with two mono- lithic protocol implementations. The systems we use for compar- ison are Ultrix 4.2A, and Mach (version MK74) with the UNIX server (version UX36). We did not alter the Ultrix 4.2A kernel in any way except to add the AN1 driver. This driver does not currently implement the non-zero BQI functions that we described earlier and uses only BQI zero to transfer data from the network to protected kernel buffers. We did not alter either the stock Mach kernel or the UX server significantly. The main changes we made were restricted to adding a driver for our AN1 network device and appropriate memory and signaling support for the buffer layer.

The hardware platforms for the three systems are identical — DECstation 5000/200s connected to Ethernet and DEC SRC AN1. Our implementation of the protocol stack has not exploited any special techniques for speeding up TCP such as integrating the checksum with a data copy. The implementations we compare our design with also do not exploit any of these techniques. In fact, the protocol stack that is executed is nearly identical in all three systems. Thus, this is an “apples to apples” comparison: any performance difference is due to the structure and mechanisms provided in the three systems.

The primary performance metric for a byte-stream protocol like TCP is throughput. Table 2 indicates the relative performance of the implementations. Throughput was measured between user-level programs running on otherwise idle workstations and unloaded networks. In each case the user-level programs were running on identical systems. The user-level program itself is identical except for the libraries that it was linked against. We report the perfor- mance for several different user-level packet sizes. User packet size has an impact on the throughput in two ways. First network efficiency improves with increased packet size up to the maximum allowable on the link, and thus we see increasing throughput with packet size. Second, user packet sizes beyond the link-imposed maximum will require multiple network packet transmissions for each packet. This effect influences overall performance depending on the relative locations of the application, the protocol implemen- tation, and the device driver, and the relative costs of switching among these locations.

Table 2 has two interesting aspects to it. First, the user-level library implementation outperforms the monolithic Mach/UX im- plementation. Our implementation is 42% faster than the Mach/UX implementation for the 4K packet case (and even faster for smaller packet sizes). The protocol stack and the base operating system's support for threads and synchronization are the same in the two systems, indicating that our structure has clear performance advan- tages. For instance, crossing between application and the protocol code can be made cheaper, because the sanity checks involved in a trap can be simplified. Similarly, a kernel crossing to access the network device can be made fast because it is a specialized entry point.

Another interesting point in Table 2 is the performance differ- ence between the Ultrix-based version and the two Mach-based versions. For example, Ultrix on Ethernet is 35—65% faster than our implementation. However, on ANl, the difference is far less pronounced. We instrumented the Ultrix kernel and our Mach- based implementation to better understand the differences between the two systems.

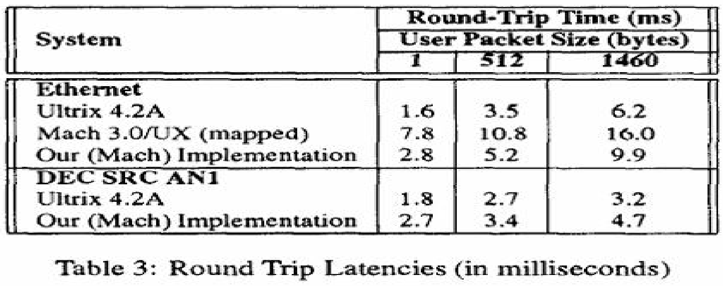
Our measurements indicate that, under load, there is consid- erable difference in the execution time of the code that delivers packets from the network to the protocol layer in the two imple- mentations. The code path consists primarily of low-level, intemipt

driven, device management code in both systems. Our implemen- tation also contains code to signal the user thread as well as special packet demultiplexing code for the Ethernet that is not present in Ultrix.

To summarize our measurements, the times to deliver ANl pack- ets to the protocol code in Ultrix and in our implementation are comparable. This is not very surprising because the device driver code is basically the same in the two systems and there is no spe- cial packet filter code to be invoked for input packetdemultiplexing since it is done in hardware. The only difference between the de- vice drivers is that our implementation uses non-zero BQIs while Ultrix uses BQI zero. The user level signaling code does not add significantly to the overall time because network packet batching is very effective. The TCPfIP protocol code in Ultrix and our imple- mentation are nearly identical and hence the overall performance is comparable in the two systems.

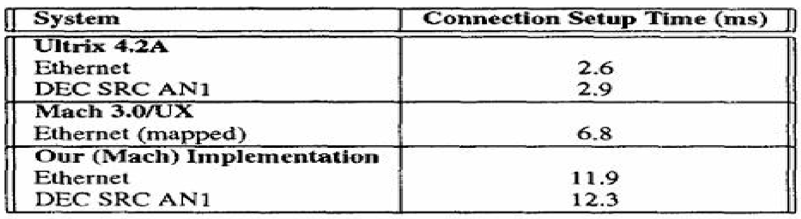
In contrast, the time to deliver maximum-sized Ethernet pack- ets to our user-level protocol code is about 0.8 ms greater than in Ultrix. Under load, this time difference increases due to increased queueing delays as packets arrive at the device and await service. In addition to the increased queueing delay, fewer network pack- ets are batched to the user per semaphore notification. However, we don't view this as an insurmountable problem with user-level library implementations of protocols. Some of this performance can be won back by a better implementation of synchronization primitives, user level threads, and protocol stacks. (For instance, the implementation in [14], which uses a later version of the Mach kernel, an improved user-level threads package, and a different TCP implementation reportedly achieves higher throughput than the Ultrix version.)

The observed throughput on AN1 is lower than the maximum the network can support. The primary reason for this is that the AN1 driver does not currently use maximum sized AN1 packets which can be as large as 64K bytes: it encapsulates data into an Ethernet datagram and restricts network transmissions to 1500-byte packets. We achieve better performance than Ultrix with 512-byte user packets because our implementation uses a buffer organization that eliminates byte copying. Ultrix uses an identical mechanism, but it is invoked only when the user packet size is 1024 bytes or larger.



System	Round-Trip Time (ms)		
	User Packet Size (bytes)		
	1	512	1460
Ethernet			
Ultrix 4.2A	1.6	3.5	6.2
Mach 3.0/UX (mapped)	7.8	10.8	16.0
Our (Mach) Implementation	2.8	5.2	9.9
DEC SRC AN1			
Ultrix 4.2A	1.8	2.7	3.2
Our (Mach) Implementation	2.7	3.4	4.7

Table 3: Round Trip Latencies (in milliseconds)



System	Connection Setup Time (ms)
Ultrix 4.2A	
Ethernet	2.6
DEC SRC AN1	2.9
Mach 3.0/UX	
Ethernet (mapped)	6.8
Our (Mach) Implementation	
Ethernet	11.9
DEC SRC AN1	12.3

Table 4: Connection Setup Cost (in milliseconds)

Unliite the mapped Ethernet device, standard Mach does not cur- rently support a mapped AN1 driver. Measuring native Mach/UX TCP performance using our unmapped, in-kernel AN1 driver is likely to be an unfair indicator of Mach/UX performance. We therefore do not report Mach/UX performance on AN1.

Latency

We compared the latency characteristics of our implementation with the monolithic versions. The latency is measured by doing a simple ping-pong test between two applications. The first ap- plication sends data to the second, which in turn, sends the same amount of data back. The average round-trip time for the exchange with various data sizes is shown in Table 3. This does not include connection setup time, which is separately accounted for below. As the table indicates, latencies on the Ethernet are significantly reduced from the Mach/UX monolithic implementation and is on average about 61% higher than the Ultrix implementation. On the AN1, the difference between Ultrix and our implementation is about 40%.

Connection Setup Cost

In addition to througbput and latency measurements, another useful measure of performance is the connection setup time. Con- nection setup time is important for applications that periodically open connections to peers and send small amounts of data before closing the connection. In a kernel implementation of TCP, con- nection setup time is primarily the time to complete the three-way handshake. However, in our design, the time to set up a con- nection is likely to be greater because of the additional actions that the registry server must perform. Anticipating this effect, our implementation overiaps much of this with packet transmission.

In measuring TCP connection setup time, we assumed that the passive peer was already listening for connections when the active connection was initiated.

Table 4 indicates the connection setup time of the different sys- tems. The speed of the network is not a factor in the total time because the amount of data exchanged during connection setup is insignificant. As the table indicates, our design introduces a no- ticeable cost for connection setup but it is a reasonable overhead if it can be amortized over multiple subsequent data exchanges. The connection setup time is slightly higher for the AN1 because the machinery involved to set up the BQI has to be exercised.

The 11.9 ms overhead in our Ethernet implementation can be roughly broken down as follows.

1. The time to get to the remote peer and back is the bulk of tire cost (4.6 ms). Network transmission time is not a factor because it is on the order of 100 ps oz so. Most of the overhead is local and includes the server's cost of accessing the network device. Unlike the protocol library, the registry server does not access the network device using shared memory, but instead uses standard Mach IPCs.

2. There is a part of the outbound processing that cannot be overlapped with data transmission. This includes allocating connection identifiers, executing the start of connection set up phase, etc., and accounts for about 1.5 ms.

3. Nearly 3.4 ms are spent in setting up user channels to the net- work device when the connection set up is being completed.

4. The time to go from the application to the server and back is about 900 ps, and is relatively modest.

5. Finally, it takes about 1.4 ms to transfer and set up TCP state to user level.

There are obvious ways of reducing the overhead that we did not pursue. For example, having a more efficient path between the registry server and the device and using shared memory to transfer the protocol state between the server and the protocol library is likely to reduce overhead. Nonetheless, it is unlikely to be as low as the Ultrix implementation.

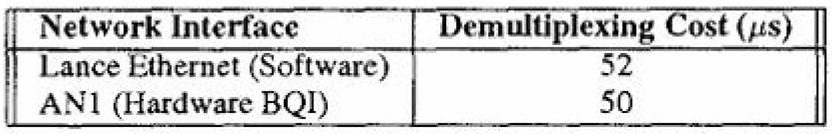
Packet Demultiplexing Tradeoffs

Finally, we quantify the cost/benefit tradeoff of hardware sup- port for demultiplexing incoming packets. Table 5 indicates the execution time for demultiplexing an incoming packet with and without hardware support. For the Ethernet programmed I/O is used to transfer the packet to host memory from the controller, and input packet demultiplexing is done entirely in software. On the ANl, DMA is used to transfer the data and the BQI acts as the demultiplexing field.

Table 5 represents only the cost of software/hardware packet de- mulfiplexing; copy and DMA costs are not included. The cost of device management code inherent to pacicet demultiplexing in the case of the AN1 is included. As the table indicates, there is no sig- nificant difference in the timing. The AN1 host-network interface has more complex machinery to handle multiplexing. Part of the cost of programming this machinery and bookkeeping accounts for the observed times. As packet size increases, the tradeoff between the two schemes becomes more complex depending on the details of the memory system (e.g., the presence of snooping caches), and specifics of the protocols (e.g., can the checksum be done in hard- ware). For example, if hardware checksum alone is sufficien(and the cache system supports efficient DMA by I/O devices, we expect the BQI scheme to have a significant perfomance advantage over one that uses only software.

5. Summary

In summary, our performance data suggests that it is possible to structure protocols as libraries without sacrificing throughput relative to monolithic organizations. Given the right mechanisms



Network Interface	Demultiplexing Cost (μs)
Lance Ethernet (Software)	52
AN1 (Hardware BQI)	50

Table 5: Hardware/Software Demultiplexing Tradeoffs

in the base operating system, user-level implementations can be competitive with monolithic implementations of identical proto- cols. Further, techniques that exploit application-specific knowl- edge that are difficult to apply in dedicated server and in-kernel organizations now become easier to apply. A relatively expen- sive connection setup is needed, but in practice a single setup is amortized across many data transfer operations.

6. Conclusions and Future Work

We have described a new organization for structuring protocol im- plementations at user-level. The feature of this organization that distinguishes it from earlier work is that it avoids a centralized server, achieving good performance without compromising secu- rity. The motivation for choosing a user-level library implementa- tion over an in-kernel implementation is that it is easier to maintain and debug, and can potentially exploit application-specific knowl- edge for performance. Software maintenance and other software engineering issues are likely to be increasing concerns in the future when diverse protocols are developed for special purpose needs.

Based on our experience with implementing protocols on Mach, we believe that complex, connection-oriented, reliable protocols can be implemented outside the kernel using the facilities provided by contemporary operating systems in addition to simple support for input demultiplexing. In -kernel techniques to simplify lay- ering overheads and context switching overheads continue to be applicable even at user-level.

Our organization is demonstrably beneficial for connection- oriented protocols. for connectionless protocols, the answer is less clear. Typical request-response protocols do not require an initial connection setup, yet require authorized connection identifiers to be used. However, these protocols are often used in an overall con- text that has a connection setup (or address binding) phase, e.g., in an RPC system. In these cases, after the address binding phase, the dedicated server can be bypassed, reducing overall latency which is the important performance factor in such protocols.

A similar observation applies to hardware packet demultiplex- ing mechanisms as well. To fully exploit the benefits of the BQI scheme, indexes have to be exchanged between the peers. This is easy if connection setup (as in TCP) or binding (as in RPC) is per- formed prior to normal data transfer. In other cases, the hardware packet demultiplexing mechanism is difficult to exploit because there is no separate connection set up phase that can negotiate the BQIs.

Another area that we have not explored is the manner and ex- tent to which application-level knowledge can be exploited by the library. Simple approaches include providing a set of canned op- tions that determine certain characteristics of a protocol. A more ambitious approach would be for an external agent like a stub compiler to examine the application code and a generic protocol library and to generate a protocol variant suitable for that particular application.

References

[1] Mark B. Abbot and Larry L. Peterson. A language-based approach to protocol implementation. In Pmceedings of the 1992 SIGCOMM Symposium on Communications Architec- tures and Protocols, pages 27—38, August 1992.

[2] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. ACM Transactions on Computer\Systems, 2(1):39—59, February 1984.

[3] David R. Cheriton and Carey L. Williamson. VMTP as the transport layer for high-performance distributed systems. IEEE Communications Magazine, 27(6):37—44, June 1989.

[4] David Clark. The structuring of systems with upcalls. In Pro- ceedings of the 10th ACM Symposium on Operating Systems Principles, pages 171—180, December 1985.

[5] Eric C. Cooper and Richard P. Draves. C threads. Technical Report CMU-CS-88-154, Carnegie Mellon University, June 1988.

[6] Digital Equipment Corporation, Workstation Systems Engi- neering. PMADD-AA TurboChannel Ethernet Module Func- tioned Specification, Rev L2., August 1990.

[7] Willibald A. Doeringer, Doug Dykeman, Matthias Kaiser- werth, Bemd Werner Meister, Harry Rudin, and Robin Williamson. A survey of light-weight transport protocols for high-speed networks. IEEE Transactions on Communi- cations, 38(11):20—31, November 1990.

[8] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In Proceedings of the 13th ACM Symposium on Operating S ys- rems Principles, pages 122—136, October 1991.

[9] Edward W. Felten. The case for application-specific commu- nication protocols. In Pmceedings of Intel Supercomputer Systems Division Technolog y Focus Conference, pages 171— 181, 1992.

[10] Alessandro Fiorin, David B. Golub, and Brian N. Bers had. An l/O system for Mach 3.0. In Pmceedings of the Second Usenix Mach Workshop, pages 163—176, November 1991.

[11] Norman C. Hutchinson anti Larry L. Peterson. The x-kernel: An architecture for implementing network protocols. IEEE Transactions on Software Eng ineerin8. 17(1):64-76, I anuary 1991.

[12] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quwterman. The Design and Imple- mentation of the 4.3BSD UNIX Operating System. Addison- Wesley Publishing Company, Inc., 1989.

[13] Chris Maeda and Brian N. Bershad. Networking performance for microkernels. In Proceedings of the Ttiird Workshop on Workstation Operati*8 Systems, pages 154—159, April 1992.

[14] Chris Maeda and Brian N. Bershad. Protocol service decom- position for high performance intemetworking. Unpublished Carnegie Mellon University Technical Report, March 1993.

[15] Henry Massalin. S ynthesis.' An Efficient Implementation of Fundamental Operating System Services. Ph.D. thesis, Columbia University, 1992.

[16] Henry Massalin and Calton Pu. Threads and input/output in the Synthesis kernel. In Proceedings ofl2th ACM S ympostumon Operating Systems Principles, pages 191—201, December 1989.

[17] Steven McCanne and Van Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In Proceed-ings of the 1993 Wnter USENIX *Conerence*, pages 259—269, January 1993.

[18] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The Packet Filter: An efficient mechanism for user-level network code. In Proceedi•8S Of the 11ih ACM Symposiumon Operating Systems Principles, pages 39—51, November 1987.

[19] Franklin Reynolds and Jeffrey Heller. Kernel support for net- work protocol servers. In Proceedings of the Second Usenix Mach Workshop, pages 149-162, November 1991.

[20] Douglas C. Schmidt, Donald F. Box, and Tatsuya Suda. ADAPTIVE: A flexible and adaptive transport system ar- chitecture to support lightweight protocols for multimedia applications on high-speed networks. In Proceedings of the Symposium on High Performance Distributed Computing, pages 174—186, Syracuse, New York, September 1992. IEEE.

[21] Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Ed- win H. Satterthwaite, and Charles P. Thacker. Autonet: A high-speed, self-configuring local area network using point- to-point links. IEEE Journal on Selected Areas in Communi- cations, 9(8):1318—1335, October 1991.

[22] David L. Tennenhouse. Layered multiplexing considered harmful. In Proceedin8s of the 1st International Workshop on High-Speed Networks, pages 143—148, November 1989.

[23] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Sat- terthwaite, Jr. Firefly: A multiprocessor workstafion. /F6F Transactions on Computers, 37(8):909—920, August 1988.

[24] Christian Tschudin. Flexible protocol stacks. In Pmceedin8• of the 1991 SIGCOMM Symposium on Communications Ar- chitectures and Protocols, pages 197—205, September 1991.

[25] George Varghese and Tony Lauck. Hashed and hierarchical timing wheels: Data structures for the efficient implemen- tation of a timer facility. In Proceedings of the 11th ACM Symposium on Operattng Systems Principles, pages 25—38, November 1987.

[26] Richard W. Watson and Sandy A. Mamrak. Gaining effi- ciency in transport services by appropriate design and imple- mentation choices. ACM Transactions on Computer Sysiems, 5(2):97—120, May 1987.