A Framework for the Non-Monolithic Implementation of Protocols in the x-kernel

Parag K. Jain, Norman C. Hutchinson, and Samuel T. Chanson*
Department of Computer Science
University of British Columbia
Vancouver, B.C., Canada.
Email : { jain, hutchinson, chanson }@cs.ubc.ca

Abstract

This paper describes a framework for splitting the implementation of network protocols in the x-kernel into a user-level library, which implements the majority of the protocol functionality, and a kernel component, which provides a *fast track* to demultiplex packets into the appropriate recipient address space. The design also provides the user with flexibility to decide which aspects of the protocol implementation should be in user space and which should be in the kernel. We implement an example TCP/UDP-IP-Ethernet protocol stack and report on its performance which is competitive with monolithic implementations. The kernel level demultiplexing of network packets is well structured, modular, and scalable. The cost of demultiplexing is shown to be insensitive to the number of user-level instances of protocols. The performance of the approach on a multiprocessor system shows that non-monolithic protocol implementations are potentially more parallelizable than traditional monolithic ones.

1 Introduction

This paper describes a framework for service decomposition of communication protocols in the context of the x-kernel. It defines a flexible way to divide protocol functionality between the kernel and user level applications such that the kernel provides a *fast track* to demultiplex the packets received from the network and route them to the appropriate recipient address space. At the same time, it allows the user to decide what additional protocol services would be better implemented in the kernel and move them to the kernel.

Traditionally, network protocols have been implemented as part of the kernel. With the advent of the microkernel architecture, there have been attempts to implement protocols in user space. For example, in the Mach 3.0 environment, network protocols are implemented as trusted user-level servers [3]. More recent efforts, however, have been focused on implementing protocols as user level libraries [8, 21]. In these implementations, a *packet filter(PF)* installed in the kernel demultiplexes the packets to the correct recipient address space. Examples include the CMU/Stanford Packet Filter (CSPF) [11], the BSD Packet Filter (BPF) [9], and the Mach Packet Filter (MPF) [22]. Advancements in packet filter technology have improved both the functionality and performance of packet filters to the point that user level protocol implementations can now be constructed that achieve better performance than previous kernel-level implementations [8].

While recent advances in packet filter technology have addressed their performance problems, they have not provided any additional structure or architectural support for protocol implementors.

*On leave at the Hong Kong University of Science and Technology.

What was once an unstructured kernel level protocol implementation becomes, with the addition of a packet filter, an unstructured user level protocol implementation. This paper takes a different approach to the problem of kernel level demultiplexing of network packets. It is based on the x-kernel, a widely used protocol framework for single address space protocol implementations, extending it to allow the implementation of each protocol to be separated into a kernel resident component and a user library component. The architecture requires that the kernel component provide at least the efficient demultiplexing of incoming network packets. Additional functionality can be incorporated in the kernel component as necessary on a protocol by protocol basis. Our framework provides the following functionality:

- Protocols are implemented as user-level libraries.
- There is a fast track in the kernel to route packets to the appropriate user space. The overhead remains constant as the number of protocols in user space increases.
- The framework deals with idiosyncrasies of different protocols in a flexible way. The user can decide what protocol functionality to configure in user space and what in the kernel. However, the design goal would be to place as much functionality as possible in the user's address space.
- In the extreme, it is possible to configure the same protocol in the kernel for one application, and in user space for another application.

2 Motivation and Previous work

Two previous efforts to split the implementation of communication protocols between the kernel and user space are described in [8] and [21]. They point out the main advantages and motivation for the user level implementation of protocols. These advantages can be summarized as follows:

- The protocol code is decoupled from the kernel and hence can be easily modified and customized on an application specific basis.
- During the protocol development phase, it becomes easier to debug and experiment with new protocols.

We observe the following additional advantages:

- Reduced contention because each user has its own protocol stack. This implies that multiple instances of the protocol need not share global state.
- Increased parallelism since user level implementations can exploit the parallelism provided by the new generation of multithreaded environments and parallel machines due to the smaller number of synchronization points in the shared protocol state.
- Performing only demultiplexing operation in the kernel integrates the demultiplexing operation of different protocol layers into a single "one-shot" operation, while maintaining compatibility with existing protocols. This is in agreement with the recent literature [2, 20] which argues that performing multiplexing/demultiplexing at multiple layers in the protocol stack significantly reduces throughput.

Previous user-level protocol implementations [8, 11] are based on having a packet filter in the kernel which determines, for each packet, which user level address space should receive the packet. We believe that packet filter technology [9, 11, 22] has a number of limitations:

- Packet filters are based on interpreted languages. The interpreted-language approach provides excellent support for the run-time installation of the packet filter. However, demultiplexing of network packets (i.e. packet filtering) is also interpreted. The experience of the programming language community argues that interpreted languages are often an order of magnitude slower than compiled ones [16].

- Interpretation of some headers fields is done twice: once during packet filtering and once again in the application to locate the correct session (e.g. protocol control block).
- Packet filters suffer from a lack of modularity in that each packet filter must mention every protocol in the path between the network device and the user. For example, while the transport protocol (UDP or TCP) installs the packet filter, that packet filter depends on the network protocol that will eventually deliver the data, thus losing the modularity advantage of IP which is supposed to shield high level protocols from network protocol details.
- Packet filters are not general enough to deal elegantly with the idiosyncrasies of current communication protocols. As an example, consider the IP [13] protocol. It is difficult for a packet filter to handle the reassembly of IP fragments, IP forwarding, and IP tunneling.[1] Of all the packet filter implementations reported in the literature, only the Mach Packet Filter [22] provides a solution to IP reassembly, and we believe that not even packet filter proponents can argue with conviction that the solution is elegant. With the packet filter approach, IP forwarding and IP tunneling are currently supported by running user-level servers which incur additional performance costs in the form of extra context switches and additional packet data copying. We believe that these operations can be performed much more efficiently in the kernel and that a general protocol architecture should support such an implementation.
- It is difficult for a packet filter to act on protocol specific real-time information that may be either encoded in network packets or negotiated for particular network connections.

Proponents of packet filter technology argue that the performance problems of packet filters can be addressed by good engineering [22], and that making the packet filter do more (like handling IP fragments) is as easy as adding a few new instructions [22]. In many ways, the recent changes to the packet filter amount to a way of creating a kernel resident protocol entity complete with support for multiple connections and even connection specific state. Our solution takes these trends to their logical conclusion: what is really wanted is the ability to embed in the kernel a small fragment of the functionality of the communication protocol. This paper describes our efforts to do exactly that.

Organization of this Paper

The rest of this paper is organized as follows: section 3 presents an overview of the x-kernel which is essential in understanding the rest of this paper. Section 4 describes the design and architecture of our protocol framework and section 5 describes the implementation of an example TCP/UDP-IP-Eth protocol stack. Section 6 analyzes the performance of the example protocol stack. Finally section 7 offers some conclusions.

3 The x-kernel Overview

The x-kernel [6, 7] is a protocol implementation architecture that defines protocol independent abstractions which facilitate the efficient implementation of communication protocols. It provides three primitive communication objects: *protocols*, *sessions*, and *messages*. Each protocol object corresponds to a communication protocol such as IP [13], TCP [15], or UDP [12]. A session object represents the local state of a network connection within a protocol and is dynamically created by the protocol object. Messages contain the user level data and protocol headers and visit a sequence of protocol and session objects as they move through the x-kernel.

Each protocol implements a set of methods which provide services to both upper layer and lower layer protocols via a set of generic function calls provided by the *Uniform Protocol Interface(UPI)*. With the UPI interface, protocols do not need to be aware of the details of protocols above or below them in the protocol graph. By using a dynamic *protocol graph*, the binding of protocols into

[1] Tunneling implies that a complex network with its own protocols is treated like any other hardware delivery system.

hierarchies is deferred until link time. The protocol graph represents the static relationship between protocols. In the x-kernel, user processes are also treated as protocols. Hence, the x-kernel provides *anchor* protocols at the top and bottom of the protocol hierarchy which defines the application interface and device driver interface respectively. The anchor protocols allow easy integration of the x-kernel into any host environment.

The x-kernel provides a set of support tools [5, 7] to implement protocols efficiently and quickly. These tools consist of a set of routines that are commonly used to implement protocols. The major tools include a map manager, message manager, process manager and event manager.

3.1 Operations on Protocol Objects

The two main functions of protocol objects are management of session objects, and demultiplexing received messages to the appropriate session objects.

The protocol object supports three operations to create session objects: *open, open_enable,* and *open_done.* A high level protocol creates a session object in a lower layer protocol by invoking a lower protocol's *open* routine. Alternatively, a high level protocol can pass a capability to a lower level protocol for passive creation of session at a future time by invoking the *open_enable* routine of the lower protocol. The *open_enable* operation creates an *enable* object to store the capability passed by the upper layer protocol. On receipt of a message from the network, the lower layer protocol creates a session and signals the creation of a session by invoking the upper layer protocol's *open_done* routine.

The above three routines take a *participant_list* as one of their arguments. The *participant_list* is a name for some participant in a communication, and consists of a sequence of protocol specific *external ids*[2] such as port numbers, protocol numbers, or type fields used by the protocols to identify message recipients. The mapping between external ids and *internal ids*, which are capabilities for session objects or enable objects, is maintained in the *active* and *passive* maps maintained by each protocol. The maps are implemented as hash tables which cache the last key looked up.

In addition to managing connections, the primary role of the protocol object is to demultiplex incoming data to the appropriate session. This is accomplished by a routine called *demux* which extracts certain fields from the header part of the message (e.g. source and destination port numbers) and constructs an external id. If a binding to a session is found in the active map, then the message is delivered to that session. If no session is found, and searching in the passive map yields an enable object, then a new session is created as described previously. Otherwise, the message is dropped.

3.2 Operations on Session Objects

Sessions support two primary operations - *push* and *pop*. The protocol's *push* routine appends the protocol header to the message and passes the message to a lower layer session. The protocol *pop* routine updates the state of the connection represented by the session and passes the message to an upper layer protocol.

4 Design Overview

Before describing our design, we shall first explain the terms *kernel, user* and *IPC. Kernel* refers to the operating system kernel or the user level trusted server that implements the kernel component of protocol functionality. By *user*, we mean the user level application that links to the user level protocol libraries. The means of communication between kernel and user is referred to as *IPC.* In the case when the kernel is an operating system kernel, then the IPC will mean a system call or upcall. If it is a trusted server, the IPC will mean inter-process communication.

The main idea is to decompose protocol implementations into two parts: one part in the kernel and the other linked with the application as a user level protocol library. The kernel level part

[2] Also referred to as *key* in this paper.

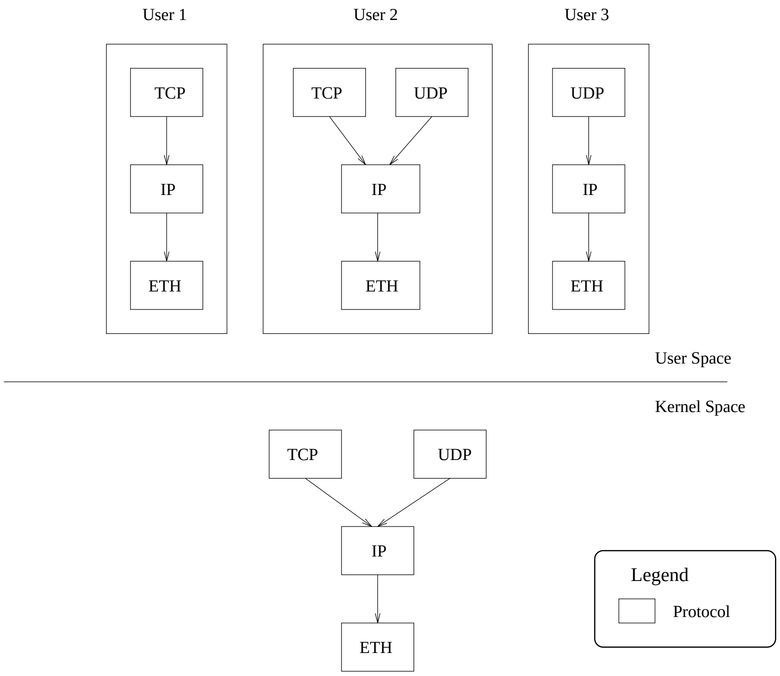

Figure 1: User and Kernel Protocol Graphs

implements, at a minimum, the demultiplexing of packets received by the protocol. In an efficient implementation, the cost of demultiplexing should not increase with the number of user level instances of protocols. In earlier implementations based on the packet filter [11], the cost of demultiplexing increased linearly with the number of connections. The Mach Packet Filter achieves performance that is not sensitive to the number of open connections by collapsing the common parts of different packet filters into a single filter [22]. In our scheme, we exploit the notion of protocol graphs to achieve this. These protocol graphs define the protocol hierarchy as explained in the x-kernel overview. In a naive user level protocol implementation the demultiplexing operation will be performed twice – one time in the kernel to locate the recipient application and then again in the application to locate the recipient session (i.e. protocol control block) because, in general, an application may open more than one connection. In an efficient implementation, the demultiplexing in the application can be completely avoided. Our scheme avoids demultiplexing in the user by having the user register the user session identifiers with the kernel as described later in this section.

4.1 Protocol Graphs

In our design, each user has its own protocol graph, as does the kernel. The kernel level protocol graph is the *union* of all user level protocol graphs and is called the *Universal Protocol Graph(UPG)*. The user level protocol graph is registered with the kernel when the user application starts. The kernel incorporates this graph into the UPG by the *union* operation. Figure 1 shows the protocol graphs of 3 different users and the corresponding kernel protocol graph. The entire suite of protocols which is available to each user appears in its protocol graph even if some of those protocols are actually implemented in the kernel.

Each protocol's demultiplexing routine is also registered with the kernel. This routine is protocol specific and not user (or protocol instance) specific. This implies that although there are a number of copies of the same protocol at the user level, the kernel will use a single demultiplexing routine for this protocol. This is equivalent to combining different packet filters (e.g. Mach Packet Filter [22]).

4.2 Protocol Objects

The protocol objects reside in the kernel as well as in the user space. However, the kernel level protocol objects in most cases are skeletons of user level protocol objects and implement only that portion of the protocol functionality that the user decides to move to the kernel. We call them *skeleton* protocols. The minimum functionality of a protocol that resides in the kernel is demultiplexing. However, a user may decide to move more functionality to the kernel, and in the extreme, our design allows a user to move the complete protocol stack to the kernel. This flexibility is achieved by adding new data structures to the protocol and session objects.

4.3 Session Objects

In a user level implementation of protocols, sessions need to reside in the application because they store all the protocol state information of the application's connections. Since demultiplexing of incoming packets is done in the kernel, we need to have sessions in the kernel as well. The kernel level sessions are proxies for their user level counterparts. We call these sessions *shadow sessions*.

In the *x*-kernel, an *open* call leads to the creation of a session at each protocol layer. These sessions are linked by pointers and form a session stack. Every user level open call is registered with the kernel in a *RegisterOpen* IPC message which leads to the creation of a shadow session stack in the kernel. The user level also sends the user IPC port and user level *session id list* in the IPC message. This information is stored in the shadow session at the user-kernel boundary and is passed to the user along with each received network packet. This avoids extra demultiplexing in the user space to locate the appropriate session.

User level passive opens (e.g. a server waiting on a port for connections) are also registered with the kernel by using a *RegisterOpenEnable* IPC message. This enables the kernel to create a shadow session stack when a packet is received for such a server.

4.4 Demultiplexing in the kernel

Demultiplexing in the kernel proceeds in the traditional *x*-kernel way. However, there are no *pop* routines in the kernel. The UPI library's *xPop* routine is modified so that instead of calling the protocol specific *pop* routine, it registers the protocol specific header information for later playback at the user level.

In the kernel, at each protocol layer, the protocol specific *demux* routine is invoked, the header information is stored in an IPC buffer, and the message is passed to the upper layer protocol. At the user-kernel boundary layer protocol (e.g. TCP, UDP, etc), the demultiplexing determines the identity of the destination user level application for the current message. The session returned by the demultiplexing routine at this protocol layer contains the user level IPC port number and the user session id list. The message is then transferred to the destination user space along with the user session id list and the bookkeeping information recorded by the kernel *xPop* routine in a *ReceivePacket* IPC call.

At each protocol layer, the protocol specific demultiplexing routine is invoked. There is no header processing in between the demultiplexing routines as in a traditional implementation. This is equivalent to combining the demultiplexing operation of multiple protocol layers into a single "one-shot" operation. This also makes it possible to take an Integrated Layer Processing approach to the data manipulation [2, 20].

4.5 Packet Processing at User Level:

The user level receives the incoming packets from the kernel along with bookkeeping information and the user session id list. The connection specific protocol state is updated by invoking a series of *pop* routines, resulting in passing the user data to the application. A user may passively wait for connections. In this case, when the first packet is received on a connection, the user level sessions are created. The user level session stack thus created is then registered with the kernel.

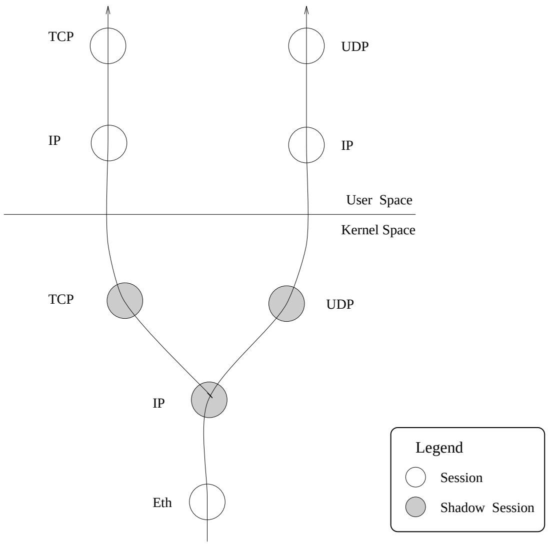

Figure 2: Incoming message paths

Sending packets out is straightforward. For outgoing messages, protocol headers are appended to the message at each protocol layer by invoking a series of *push* routines. At the user-kernel boundary protocol, the message is transferred to the kernel level protocol in a *SendPacket* IPC message. The kernel level protocol to which the message is sent is the one that logically sits just below the bottom user level protocol in the protocol graph hierarchy. This information is readily available from the user level protocol graph. The kernel is free to make any checks required for security reasons in the header portion of the outgoing message.

Figure 2 and 3 depict the receive and send paths for incoming and outgoing messages respectively. In the figures, the TCP, UDP, and IP protocols are configured in the user space and have corresponding skeleton protocols in the kernel. The Ethernet [10] protocol is configured in the kernel, but to improve the performance of outgoing packets, we configure a skeleton Ethernet protocol in the user space. The figures also illustrate that kernel level sessions that are not at the user-kernel boundary (e.g. IP, Ethernet) may be shared by more than one user level protocol belonging to the same or different applications.

4.6 Shared state

One of the problems that must be addressed in any protocol implementation that admits multiple implementations of protocols is the problem of shared protocol state. Examples of such state include IP routing information and ARP tables which map between IP and network addresses. Our framework supports a simple but general protocol independent mechanism to manage such information assuming that one address space (typically the kernel) maintains the master copy of such information and other address spaces (typically the users) wish to maintain caches of subsets of the information. This facility amounts to an IPC broadcast between the kernel implementation of a protocol and all of the user implementation of that same protocol whenever the shared state is changed. This facility is also required to handle IP broadcast and multicast packets.

4.7 More Details of Skeleton Protocols

The skeleton protocols need to manage kernel sessions for demultiplexing incoming network packets. It is possible to provide a set of generic routines such as *open/open_enable/close/open_disable* to perform these connection management functions in the kernel but the *demux* routine is protocol

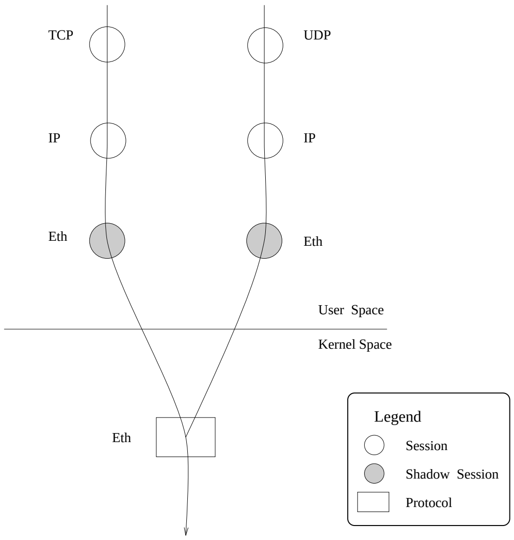

Figure 3: Outgoing message paths

specific and therefore, must be implemented by the protocol writer.[3] In the most common case, this is all that is necessary to construct the skeleton protocol in the kernel. If the protocol writer decides to move additional functionality of the protocol to the kernel, he/she must provide protocol specific routines as necessary that supersede the generic routines.

To give insight into how the skeleton protocols function, we now describe how the generic *open/open_enable* functions proceeds as an example. The *close* and *open_disable* routines are similar. These routines take a participant list as one of their arguments. The participant list is supplied by the user in the *RegisterOpen/RegisterOpenEnable* calls and is created during the *open/open_enable* call in the user level protocols. As the user level active open (i.e *open*) propagates down the user protocol stack, each protocol records the key it used to identify the newly created session object. When the user level *open* call completes, the top level protocol has the list of the keys each protocol used for identifying its sessions. It passes this list to the kernel to initiate the creation of a shadow session stack in the kernel. The kernel level protocols treat this list as a stack – each protocol pops a key from the stack, creates a shadow session, and passes the rest of the stack to the next lower layer protocol to initiate creation of shadow sessions at lower layers.[4] A passive open (i.e *open_enable*) is similar to an active open except that a passive open does not propagate down the protocol stack.

4.8 Protocol Configuration

Each protocol, session, and enable object contains a new bit called the *ukBoundaryBit* which specifies whether the object resides at the user-kernel boundary. The user level protocols which are at the bottom of the user level protocol graphs will have this bit set indicating there is no lower level user space protocol, as will the kernel level shadow sessions at the top of the kernel protocol graph. This permits the kernel to know if the received packet needs to be transferred to the destination application. The shadow session's *state* data structure stores the user level port and user *session id list*.

[3] It is difficult to provide a generic *demux* routine because protocol header length is not always constant (e.g. TCP and IP).

[4] Some protocols such TCP and UDP use the lower layer protocol session (e.g. the IP session) to form the keys for demultiplexing a received packet into a session. We define a special symbol *USE_LOWER_SESSION* as the key to tell the kernel level protocol to use the lower layer session as the key.

Currently, protocol configuration is done statically. The kernel resident active and passive maps for each protocol are created at the system startup time. The key size information required to create these maps is stored in the protocol graph and hence is readily available. Our architecture would easily accommodate dynamic protocol configuration, but our environment does not provide a mechanism for the dynamic addition of code to the kernel (such as loadable device drivers).

5 An Example Implementation

We have implemented the split protocol stack for UDP [12]/TCP[15], IP [13], ARP [17], ICMP [14], and Ethernet[10] protocols. UDP, TCP, and most of IP, ARP, ICMP are in the user library while the remaining parts of IP, ARP, and Ethernet sit in the kernel. Some IP functions are implemented in the kernel as will be explained later. ARP is implemented at user level as well as in the kernel. The kernel level ARP is the master and periodically broadcasts the ARP table changes to the user level instances of ARP as described earlier.

TCP and UDP port assignment must be centralized for correct functioning. In our scheme, the only central point is the kernel, so port assignment is done by the kernel. To improve the performance of outgoing packets, we configure a skeleton Ethernet protocol in the user space library. This caches Ethernet header information from the real Ethernet driver sitting in the kernel.

When a user level application wants to set up a TCP connection to a remote machine, it calls *tcp_open* with the appropriate set of participants. The *tcp_open* routine creates a TCP session and calls open on the IP protocol. IP creates a new session or returns an already existing session. IP also links to the appropriate network interface (in our case Ethernet). This completes the creation of a user level session stack. The user session id list is then registered with the kernel by sending a *RegisterOpen* IPC request to the kernel which creates the shadow session stack in the kernel. The kernel stores the user level session id list in the newly created shadow TCP session's *state* data structure. Next, *tcp_open* tries to establish a connection with the remote machine. If the connection is successfully established, *tcp_open* returns success. Otherwise it destroys the user level session stack that has just been created and sends a *DeregisterOpen* IPC request to the kernel to destroy the kernel level shadow session stack.

On the remote machine, the server waits passively for new connections after having registered a *RegisterOpenEnable* with the kernel. When the connection message is received on the remote machine, the kernel demultiplexes the packet and ultimately creates the shadow session stack in the kernel. It then passes the packet to the server via the *ReceivePacket* IPC message. The protocol library in the server creates the user level session stack as the message flows up the protocol stack. At the TCP level, after the TCP session is created, the user level session id list is registered with the kernel in the IPC reply message to the *ReceivePacket* IPC call. The kernel stores the user session id list in the corresponding TCP shadow session. This completes the creation of the user level session stack as well as the kernel level shadow session stack on the remote machine. Hereafter, packets can be efficiently exchanged between the two machines with demultiplexing done only in the kernel, and protocol specific connection state processing only in user space.

IP Services Decomposition

The IP protocol implementation has been split between the kernel and user space such that the services that are difficult to handle in user space are implemented in the kernel. These services are IP reassembly of fragments, IP routing table management, IP packet forwarding, IP broadcasts, and IP multicast packet processing. The kernel thus maintains enough of the IP protocol context so that it can handle these services. IP Routing tables are maintained in the kernel and periodically broadcast to the user level instances of IP. The kernel maintains maps for collecting IP fragments and when a message is reassembled, it is passed to the upper layer protocol (in our case UDP). The upper level protocol transfers the complete message to the appropriate user space. This implies that the user level IP never has to deal with reassembly issues. Similarly, the kernel maintains the IP forward map which stores the session corresponding to packets that need to be forwarded.

6 Performance

In this section, we report on the performance of the sample application level protocol stack (TCP/UDP-IP-Ethernet) implemented in our framework and compare it with the performance of the monolithic implementation. We first describe the hardware and software platform for our experiments and the benchmark tools used. We then report on the round trip latency, incremental cost per round trip, latency breakdown for receive and send paths, the kernel level demultiplexing cost, TCP throughput, and connection setup cost in the context of the sample protocol stack. Finally, we present the effect of parallel processing on the round trip latency by using more than one processor.

6.1 Hardware/Software Platform

The hardware platform for our experiments is the Motorola MVME188 Hypermodule, a 25 MHz quad-processor 88100-based shared memory machine [4]. A light weight, multithreaded kernel called *Raven* [18, 19] has been developed in the Computer Science department at UBC and is the microkernel running on the bare hardware. The protocols have been implemented in the context of a parallel x-kernel: the x-kernel [6, 7] has been ported as a user level server in this environment and has been parallelized to take full advantage of parallel processing. Our approach to the parallelization of the x-kernel is similar to the one described in [1]. We use the one processor/thread per packet model for the processing of incoming and outgoing packets. The performance measurements are made by connecting two Motorola MVME188 Hypermodules through a 10 Mb/sec Ethernet network in light network traffic conditions.

IPC Design

We use Raven's light-weight synchronous IPC library which associates a shared buffer with each communication port. In order to achieve pairwise shared buffers, each application initializes by going through a registration process with the kernel in which each side creates a communication port and an appropriate number of communication threads and sends the port identifier to the other side. The application stores the kernel port identifier in the protocol objects at the bottom of its protocol graph and directs all but its first request to the newly created kernel port. Likewise, the kernel stores the port identifier of the application in the shadow session, and enable objects that are created on behalf of the application at the user-kernel boundary.

Benchmark Tools

We have analyzed the performance of our implementation in two ways.

- Round trip times and timing of the various components of the implementation have been taken using an on-board Z8536 Counter/Timer configured as a 32-bit timer with microsecond resolution.
- Instructions have been counted by using the instruction tracing facility of a hardware simulator.

Before giving the performance numbers, we briefly describe the difference in the various implementations. We describe three versions: *Server-Server*, *User-User Server-Stack* and *User-User Partitioned-Stack*. *Server-Server* implementation is the traditional x-kernel running as a user level server. There is no boundary crossing between the user and the x-kernel server. In *User-User Server-Stack* case there is a boundary crossing between the x-kernel server and applications, however, all the protocol processing is done in the x-kernel server. The *User-User Partitioned-Stack* version is our new implementation of protocols as a user level library.

6.2 User-to-User Latency

We measured the round trip time of various user level protocols and compared it with the corresponding server-stack implementation. The latency test is a simple ping-pong test between two user

Protocol Stack	*Round-Trip Time(ms)*				
	User Data Size(bytes)				
	1	100	500	1000	1400
Server-Server					
ETH	0.88	1.01	2.09	3.44	4.51
IP-ETH	1.04	1.20	2.27	3.64	4.71
UDP-IP-ETH	1.21	1.42	2.49	3.80	4.87
TCP-IP-ETH	1.74	2.17	3.63	5.47	6.92
User-User Server-Stack					
ETH	1.86	1.94	3.12	4.56	5.77
IP-ETH	2.06	2.27	3.44	4.96	6.25
UDP-IP-ETH	2.26	2.54	3.70	5.18	6.43
TCP-IP-ETH	2.89	3.46	5.01	7.00	8.64
User-User Partitioned-Stack					
ETH	1.88	1.98	3.13	4.51	5.65
IP-ETH	2.17	2.35	3.47	4.88	6.08
UDP-IP-ETH	2.39	2.63	3.78	5.17	6.31
TCP-IP-ETH	3.02	3.51	5.05	6.93	8.47

Table 1: User-to-User Latency

Protocol	*Incremental Costs(ms)*		
	Server-Server	**User-User Server-Stack**	**User-User Partitioned-Stack**
IP	0.16	0.20	0.29
UDP	0.17	0.20	0.22
TCP	0.70	0.83	0.85

Table 2: Incremental Costs of Various Protocols per Round Trip (1 byte user data)

level applications called client and server. The client sends data to the server and the server sends the same amount of data back. The average round-trip time is measured over 10000 such transactions. The figures reported are measured for single processor case. The UDP protocol processing time does not include UDP checksum overhead. Table 1 gives the latency of the various schemes. The latency difference between *Server-Server* and *User-User Server-Stack* is due to the overhead of moving the data between the two address spaces involved. The difference between *User-User Server-Stack* and *User-User Partitioned-Stack* gives the overhead of our scheme. The latency of the Ethernet stack is approximately the same in the server-stack and the Partitioned-Stack cases because their implementations are very similar. We observe that the latency of the Partitioned-Stack is smaller than that of the Server-Stack for user data size larger than 1000 bytes. This, we suspect, is due to the difference in the cache performance for the two implementations.

The following list describes in detail the overhead of our implementation:

- Overhead of recording bookkeeping information at each skeleton protocol layer. The information stored consists of a protocol identifier and the length of the protocol header.
- Longer IPC messages are exchanged because the bookkeeping information must be included in the IPC messages.
- Constructing and interpreting the different types of IPC messages to pass them to the appropriate protocol. This applies to both the kernel as well as the user.

- Preparing the x-kernel message object from the IPC message received from the kernel to invoke the series of *pops* routines in the user space.

All of the above overheads are small, requiring only a few instructions, and are independent of packet length.

Table 2 lists the incremental cost of a round trip for each protocol for each implementation. The incremental cost is calculated by subtracting the measured round trip latency for pairs of appropriate protocol stacks that transfer 1 byte user data. For example, TCP latency is computed by subtracting latency for IP-ETH stack from the latency for TCP-IP-ETH stack. In the Partitioned-Stack case, the IP protocol has a much higher incremental cost. This is mainly due to the overhead of preparing an x-kernel message object from the IPC message. This cost is incurred only by protocols that are at the user-kernel boundary. The incremental cost of TCP and UDP in the Partitioned-Stack case is 20 microseconds higher than that in server-stack case. This can be attributed to the small overhead of bookkeeping information and the larger IPC messages that are exchanged. We also observe that the incremental cost of each protocol in the server-stack case is higher than that of the corresponding protocol in the Server-Server case. This, we suspect, is due to unfavorable cache performance since two address spaces are involved in the server-stack case.

6.2.1 Latency Breakdown

Table 3 gives the time spent in various protocol layers in the kernel and the user space for both send and receive paths. Tables 3 and 4 also list the overheads involved in recording and playback of bookkeeping information.[5] *IP PopTcb* and *TCP/UDP PopTcb* reflect the cost of recording bookkeeping information while *IP DemuxTcb* and *TCP/UDP DemuxTcb* reflect the cost of the playback of bookkeeping information. As can be seen, the number of instructions executed for these functions is extremely small. The user-kernel boundary protocols spend more time in recording the bookkeeping information because they also store the user level session list in the IPC message. The *Other Overheads* reflect the cost of function calls and x-kernel UPI library function calls. The *User IP Preprocessing* reflects the cost of converting an IPC message back into an x-kernel message object. This component is the largest single overhead of our Partitioned-Stack implementation over the server-stack one.

In the send path, the cost of the Ethernet layer increases dramatically with the size of the user data because the message is copied to a contiguous buffer before making each IPC call. This extra copy could be avoided by modifying the IPC library. The IPC cost from user to kernel is more than the IPC cost from the kernel to the user. Again, we suspect this is the due to the poor cache performance in the first case.

6.2.2 Kernel Level Demultiplexing

The kernel demultiplexing takes 61 to 98 microseconds. The large variation in the demultiplexing cost is related to cache performance.[6] Therefore, we give a breakup of the number of Motorola 88100 instructions executed in each kernel level protocol layer's demultiplexing routine and the total number of instructions taken for demultiplexing in the kernel. Table 4 lists the number of instructions taken by various components of the kernel demultiplexing and user level processing. The demultiplexing cost for the TCP-IP-ETH stack is the same as that of the UDP-IP-ETH stack and they take 687 and 686 instructions respectively. Given this low cost of kernel level demultiplexing, we could perform the complete demultiplexing function in the interrupt handler. This would avoid one extra copy of the data as we would be able to copy each packet directly to the IPC buffer shared between the recipient address space and the kernel.

Figure 4 shows the cost of kernel level demultiplexing as the number of open TCP connections is varied from 1 to 100. The solid line indicates the cost of demultiplexing when the data is sent

[5] It is possible to get rid of the bookkeeping overhead in a customized implementation. In that case, the number of instructions taken for kernel demultiplexing is further reduced.

[6] The cache size on our machine is 16KB instruction and 16 KB data cache per processor.

Protocol Component	*TCP(microseconds)*		*UDP(microseconds)*	
	Length(Bytes)		*Length(Bytes)*	
	1	1400	1	1400
Send Path				
TCPtest/UDPtest	24	26	15	15
TCP/UDP layer	179	561	45	47
IP layer	30	30	30	30
Other Overheads	6	6	6	6
Ethernet Layer	70	235	65	225
IPC to Kernel	294	391	285	355
Ethernet Driver	66	275	62	270
Total Send Path	669	1524	508	948
Receive Path				
Interrupt Dispatch and Handling	96	89	96	91
Ethernet read	67	361	63	351
Ethernet demultiplexing	17	17	17	17
IP demultiplexing	21	21	21	21
IP PopTcb	4	4	4	4
TCP/UDP demultiplexing	18	17	18	18
TCP/UDP PopTcb	8	8	8	8
Other Overheads	13	13	13	13
IPC to User	246	309	238	306
User IP Preprocessing	46	46	46	47
User IP layer	47	45	47	47
User TCP/UDP layer	186	565	46	47
TCPtest/UDPtest	6	6	6	6
Other Overheads	16	16	16	16
Total Receive Path	791	1517	639	992
Network Transit Time	50	1215	50	1215
Total	1510	4256	1197	3155

Table 3: Latency Breakdown — User-User Partitioned-Stack case

Protocol Component	*TCP*	*UDP*
	# Instructions	*# Instructions*
Ethernet demultiplexing	154	154
IP demultiplexing	230	230
IP PopTcb	13	13
TCP/UDP demultiplexing	172	171
TCP/UDP PopTcb	33	33
Other Overheads	85	85
Total Kernel Demultiplexing	687	686
User IP DemuxTcb	7	7
User TCP/UDP DemuxTcb	7	7

Table 4: Instruction Counts — User-User Partitioned-Stack case

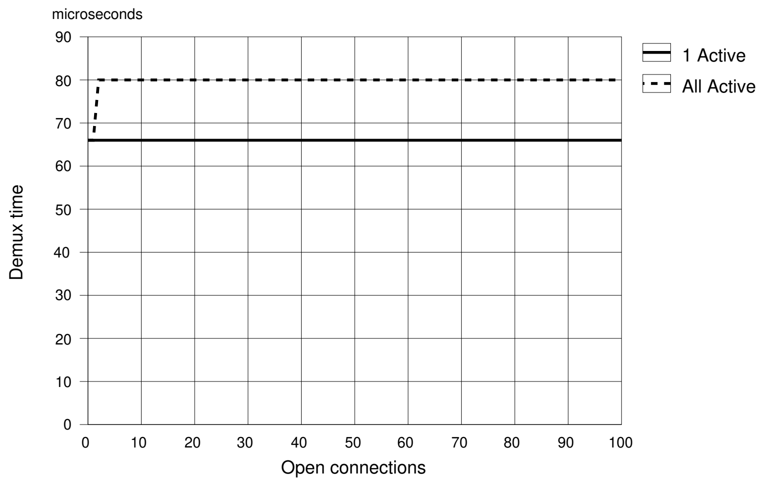

Figure 4: Demultiplexing Cost as a Function of Number of Open Connections

Protocol Stack	*Throughput(Mb/sec)*
Server-Server	6.15
User-User Server-Stack	4.74
User-User Partitioned-Stack	4.30

Table 5: TCP Throughput

to only one connection. This is the best case demultiplexing scenario and costs 66 microseconds. The session entry in this case, is always found in the cache of the TCP active map. The dotted line depicts the demultiplexing cost in the worst case scenario in that data message is sent to each open connection in a round robin fashion. In this case, the session entry is guaranteed not to be in the cache of the TCP map and a search in the map has to be carried out. This results in an extra 14 microseconds of overhead. Another important point to note is that the demultiplexing cost is totally insensitive to the number of open connections. This is similar to the result achieved in traditional kernel based protocol implementations and Mach Packet Filter [22] based implementation [8].

6.3 User-to-User Throughput

Table 5 lists the throughput in Mb/sec for TCP for various implementations. TCP throughput is measured over the TCP-IP-ETH protocol stack by measuring the time taken to send 1 megabytes from the client to a server. TCP uses a 4096 bytes window and fragments the data into 1460 byte data messages. The throughput differences among the three implementations are as expected. In the Server-Server implementation, there is no boundary crossing from the server to the application and hence, its performance is the best. In the Server-Stack implementation there are two boundary crossings on each side, per packet, and a corresponding performance reduction is observed. In the Partitioned-Stack implementation, extra overhead over the Server-Stack implementation is incurred because of longer IPC messages that are exchanged, converting each IPC message to the x-kernel message object, and recording and playback of bookkeeping information. This overhead results in a 9% slowdown over the Server-Stack implementation. In the Partitioned-Stack implementation,

Protocol Stack	*TCP Connection Setup time(ms)*
Server-Server	5.81
User-User Server-Stack	7.42
User-User Partitioned-Stack	11.01

Table 6: Connection Setup Cost

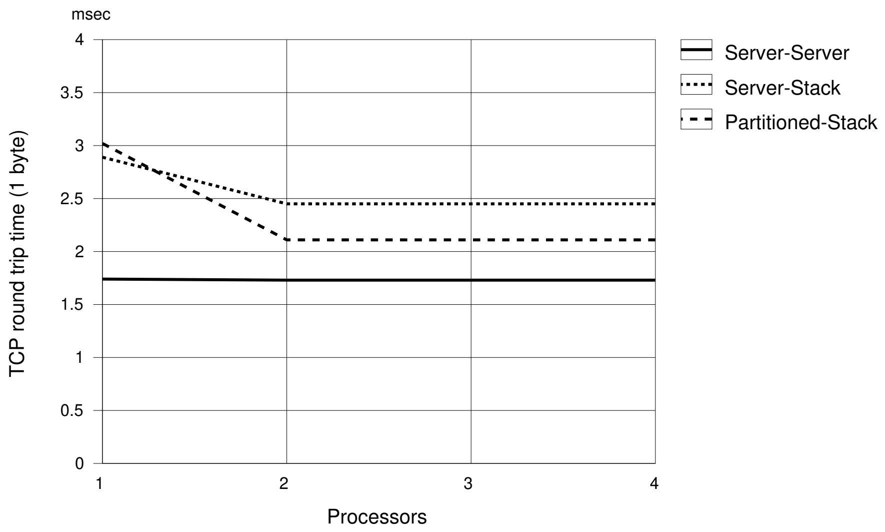

Figure 5: TCP Latency as a Function of Number of Processors

there is one extra data copy on the send path over the Server-Stack implementation which could be eliminated by changing the IPC library interface. This elimination would result in performance closer to that of the Server-Stack implementation.

6.4 TCP Connection Setup Cost

Table 6 indicates the relative connection setup time of the three implementations. The connection setup cost is an important measure of performance for applications that periodically connect to a peer entity, send a small amount of data, and close the connection. The connection setup cost for the Server-Stack implementation incurs IPC overhead over the Server-Server implementation. The 3.59 milliseconds extra for the connection setup cost for Partitioned-Stack implementation is incurred over the Server-Stack implementation because of the following overheads:

- Extra IPC messages exchanged to register the user level session id list with the kernel.
- Creation of a shadow session stack in the kernel.

6.5 Performance on Multiprocessors

Figures 5 and 6 show the effect of active multiprocessors on the TCP and UDP round trip latencies respectively for all three implementations. The interesting point to note is that the latency of the Partitioned-Stack implementation is smaller than the Server-Stack latency for more than one active

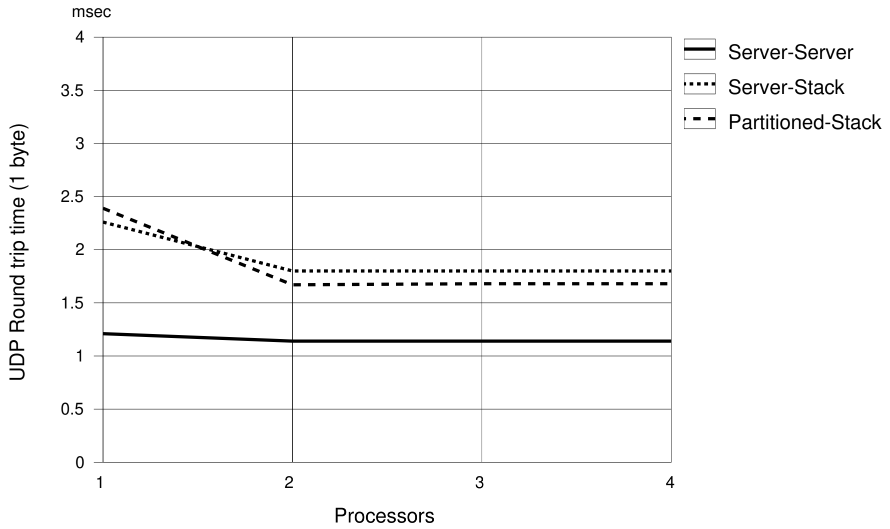

Figure 6: UDP Latency as a Function of Number of Processors

processor. This shows that the partitioned stack implementation is potentially more parallelizable than the Server-Stack or Server-Server implementations. This is an important result because for high-speed networking we believe that shared memory multiprocessor machines will be the most successful. Another interesting point to note is that the latencies of TCP and UDP drop when we go from a single active processor to two active processors and remain constant as the number of active processors is further increased. This is because there is not enough concurrency in the latency tests which are of a stop and wait nature.

7 Conclusions

We have described a protocol independent framework for implementing network protocols as user level libraries. This framework provides a flexible way to decompose protocol services between a trusted kernel address space and user address spaces and allows the protocol implementor to decide how the services should be decomposed. The primary functionality that must be included in the kernel portion of the protocol implementation is the efficient demultiplexing of incoming data to the appropriate recipient address space. Additional functionality, like IP reassembly, routing table management, etc., may be added to the kernel implementation when it is considered worth while. We characterize our approach as the logical conclusion of current trends in packet filter technology as more and more support for the idiosyncrasies of particular protocols make their way into the instruction set of packet filters. The performance of our framework was demonstrated through a TCP/UDP-IP-Ethernet protocol stack implementation which is shown to be competitive with a monolithic implementation. The kernel level demultiplexing is shown to be unaffected by the number of open connections. We have also shown that non-monolithic protocol implementations are potentially more parallelizable than the traditional monolithic implementations. This result is important for future high-speed network environments.

Acknowledgements

We would like to thank Donald Acton, Daniel Hagimont, Jeffrey Mogul, and the anonymous referees for their valuable feedback on early drafts of this paper which greatly improved its quality.

References

[1] Mats Bjorkman and Per Gunningberg. Locking effects in multiprocessor implementations of protocols. In *Proceedings of ACM SIGCOMM'93 Conference on Communications, Architecture and Protocols*, September 1993.

[2] David C. Feldmeier. Multiplexing issues in communication systems design. In *Proceedings of ACM SIGCOMM'90 Conference on Communications, Architecture and Protocols*, pages 209–219, September 1990.

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

[4] Motorola Computer Group. *MVME188 VMEmodule RISC Microcomputer User's Manual.* Motorola, 1990.

[5] Norman C. Hutchinson, Shivakant Mishra, Larry L. Peterson, and Vicraj T. Thomas. Tools for implementing network protocols. *Software - Practice and Experience*, 19(9):895–916, 1989.

[6] Norman C. Hutchinson and Larry L. Peterson. Design of the x-kernel. In *Proceedings of ACM SIGCOMM '88*, pages 65 – 75, 1988.

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

[8] C. Maeda and Brian N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of 14th ACM Symposium on Operating Systems Principles*, December 1993.

[9] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX Conference*, pages 259–269, January 1993.

[10] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[11] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of 11th ACM Symposium on Operating Systems Principles*, pages 39–51, November 1987.

[12] J. B. Postel. User datagram protocol. *Request For Comments 768*, USC Information Science Institute, Marina Del Ray, CA, August 1980.

[13] J. B. Postel. Internet protocol. *Request For Comments 791*, USC Information Science Institute, Marina Del Ray, CA, September 1981.

[14] J. B. Postel. Internet control message protocol. *Request For Comments 792*, USC Information Science Institute, Marina Del Ray, CA, September 1981.

[15] J. B. Postel. Transmission control protocol. *Request For Comments 793*, USC Information Science Institute, Marina Del Ray, CA, September 1981.

[16] Thomas Pittman. Two-level hybrid interpreter/native code execution for combined space-time program efficiency. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 150–152, St. Paul, MN, June 1987. ACM.

[17] David. C. Plummer. An ethernet address resolution protocol. *Request For Comments 826*, DCP@MIT-MC, November 1982.

[18] D. Stuart Ritchie. The Raven kernel: A microkernel for shared memory multiprocessors. Technical Report 93-36, Department of Computer Science, University of British Columbia, April 1993.

[19] D. Stuart Ritchie and Gerald W. Neufeld. User level IPC and device management in the raven kernel. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, September 1993.

[20] David L. Tennenhouse. Layered multiplexing considered harmful. In *Proceedings of IFIP Workshop on Protocols for High-Speed Networks*. H. Rudin editor, North Holland Publishers, May 1989.

[21] C. A. Thekkath, T.D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, 1993.

[22] M. Yuhara, Brian N. Bershad, C. Maeda, and J. Eliot B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proceedings of the 1994 Winter USENIX Conference*, pages 153–165, January 1994.