ALTERNATIVE HIGH-PERFORMANCE ARCHITECTURES FOR COMMUNICATION PROTOCOLS

by

Parag Kumar Jain

B. Tech., Indian Institute of Technology, Kanpur, India, 1988

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

DEPARTMENT OF COMPUTER SCIENCE

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

February 1995

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date March 26, 95

DE-6 (2/88)

Abstract

The traditional communication protocol architectures have a number of components that present bottlenecks to achieving high performance. These bottlenecks include the way the protocols are designed and the way protocol stacks are structured and implemented. With the advent of high speed networks, the future communication environment is expected to comprise of a variety of networks with widely varying characteristics. The next generation multimedia applications require transfer of a wide variety of data such as voice, video, graphics, and text and have widely varying access patterns such as interactive, bulk transfer, and real-time guarantees. Traditional protocol architectures have difficulty in supporting multimedia applications and high-speed networks because they are neither designed nor implemented for such a diverse communication environment.

This thesis analyzes the drawbacks of traditional protocol architectures and proposes alternative high-performance architectures for multimedia applications and high-speed network environments. Three protocol architectures are proposed: *Direct Application Association, Integrated Layered Logical Multiplexing*, and *Non-Monolithic Protocol* Architectures. To demonstrate the viability of these architectures, a protocol independent framework for each proposed protocol architecture is implemented in the context of a parallelized version of the *x*-kernel. Implementation of the TCP/UDP-IP-Ethernet protocol stack for each of the proposed architectures demonstrates that the performance of these protocol architectures is comparable to that of a traditional protocol architecture. In addition, the proposed architectures are more scalable on multiprocessor systems than the traditional protocol architecture and enable some of the key requirements (*Application specific Quality of Service*, *Application Level Framing*) and optimizations (such as

Integrated Layer Processing) necessary for the future communication environment.

Table of Contents

List of Tables

List of Figures

Acknowledgement

Thank you Norm and Sam for excellent supervision and encouragement throughout my graduate program. Thank you both for encouraging me to experiment with new ideas and to help convert them into concrete results.

Many thanks to Stuart Ritchie for helping me get familiarized with the Raven kernel and the Motorola Hypermodule. Many thanks to Helene Wong who gladly accepted to read preliminary drafts of this thesis and provided valuable comments. Thanks to Don Acton and Daniel Hagimont for proofreading a paper which was the starting point of this thesis.

Many thanks to my office mates, Ying Zhang, Runping Qi, and Sreekantaswamy, for maintaining a friendly environment and their advice on miscellaneous things. Many thanks to Daniel Hagimont, Sanjoy Mukherjee, Mandeep Dhami, Sameer Mulye, Nevin Lianwen, Jinhai Yang, Helene Wong, Xiaomei Han, Catherine Leung, Sree Rajan, Stephane Gagne, Sidi Yu, and Shi Hao for being good friends and making my stay at UBC enjoyable. Thanks to my friends from Simon Fraser University (Dayaram Gaur, Graham Finlayson, Sanjeev Mahajan, Stephane Wehner, Sumeet Bawa, and Sanjay Gupta to name a few) for arranging outdoor activities and numerous gettogethers.

Finally, I thank my parents, brother Pankaj, and sister Mamta for their love and encouragement throughout my graduate studies.

Chapter 1

Introduction

Computer technology has changed rapidly during the last two decades. Processor speed has increased from a fraction of Million Instructions Per Second (MIPS) to 1000 MIPS. The computer network speed has increased from kilobits per second to several gigabits per second and has the potential of going up to terabits per second. Advances in computer technology have made the technology simple and cost effective so that it is within the reach of a common man, making the personal computer a household commodity. The rest of this decade is envisioned to connect every household computer on a global network which will support services such as video on demand, video telephony/conferencing, on-line public libraries, multi-media news, home banking, home shopping. Many of these services have high bandwidth and real-time traffic requirements.

The infrastructure for global connectivity will be provided by a high-speed network called *The Information Superhighway* facilitated by evolving global standard *Asynchronous Transfer Mode (ATM)* technology. The network speed of the information superhighway is expected to range from a few megabits to a few gigabits per second at the user network interfaces and several gigabits to terabits per second on the information superhighway backbone. However, before diverse communication networks, such as the information superhighway, become operational, there are a number of challenges that must be tackled. One of the major challenges is providing high performance computer communication protocols to support emerging time critical services on such diverse communication networks.

1.1 Motivation

Traditional protocol architectures have been able to meet the demand of the last generation of computer networks and applications because networks were quite slow and they were used to carry only one type of data. With the advent of high speed networks, the future communication environment is expected to comprise of a mixture of networks with widely varying characteristics. Future networks are also expected to support *multimedia* applications which transfer a wide variety of data such as voice, video, graphics, and text and have widely varying access patterns such as interactive, bulk transfer, and real-time guarantees. Traditional protocol architectures are not suited for multimedia applications and high speed networks because they are neither designed nor implemented for such a diverse communication environment.

An ideal communication system for the future communication environment would be the one that can provide communication over diverse networks with data transfer rates ranging from kilobits per second to gigabits per second and network diameters ranging from Local Area Networks (LANs) to Wide Area Networks (WANs). It would also efficiently support "all" applications: connection oriented, connectionless, stream traffic, bursty traffic, reliable transport, best effort delivery, different data sizes, and different delay and throughput requirements, etc. To achieve the goal of such a communication system, research work is required at all level of computer systems: starting from physical transmission media, LAN interface design, CPU, memory, disk, operating systems, protocol architectures to application design and implementation.

1.2 Research Contributions of this Thesis

This thesis analyzes the drawbacks of traditional protocol architectures and proposes alternative high-performance protocol architectures for the next generation applications

and high-speed network environments. The primary contributions of this thesis are as follows:

- Three new protocol architectures are proposed:
  - Direct Application Association Protocol Architecture,
  - Integrated Layered Logical Multiplexing Protocol Architecture, and
  - Non-Monolithic Protocol Architecture.
- Parallelization of the *x*-kernel [HP88, HP91] including TCP [Pos81c] / UDP [Pos80] – IP [Pos81b] – Ethernet [MB76] protocols stack for a shared memory multiprocessor machine.

  The hardware platform for our experiments is the Motorola MVME188 Hypermodule, a quad-processor 88100-based shared memory multiprocessor machine [Gro90] with the Raven kernel [Rit93, RN93] running on the bare hardware.
- Demonstration of the viability of these protocol architectures.

  Protocol independent frameworks for each of the proposed protocol architectures are implemented in the context of the parallel *x*-kernel.
- Demonstration of efficiency of the proposed protocol architectures.

  Sample protocol stacks for each of the proposed protocol models are implemented using the most widely used TCP/UDP-IP-Ethernet protocol stack. The performance of these models are measured and compared with that of traditional protocol models under real network traffic conditions.

1.3 Thesis Overview

Chapter 2 surveys the previous work relevant to the understanding of the thesis and presents the thesis proposal. It first summarizes the research trends in protocol architectures to meet the requirements of the future networks and application environments. It then identifies and discusses in detail the requirements of future communication protocol architectures. Finally, this chapter presents the three proposed protocol architectures: Direct Application Association Protocol Architecture, Integrated Layered Logical Multiplexing Protocol Architecture, and Non-monolithic Protocol Architectures.

Chapter 3 first presents an overview of the *x*-kernel, which is an essential prerequisite to understanding the rest of this thesis. It then describes the scheme used for parallelization of the *x*-kernel to make it run on shared memory multiprocessor machines.

Chapter 4 describes the design of the frameworks of each new protocol architecture proposed in the thesis in the context of the *x*-kernel. It first describes the framework of Direct Application Association protocol architecture, followed by that of the Integrated Layered Logical Multiplexing protocol architecture. It finally describes the design of the framework of the Non-Monolithic protocol architecture.

Chapter 5 first describes the run-time environment of our experiments. It gives an overview of the hardware platform and the Raven kernel. It then gives the implementation details of frameworks of each of the three proposed protocol architectures. This chapter also describes the implementation details of the sample TCP/UDP-IP-Ethernet protocol stacks implemented for each the three protocol architectures.

Chapter 6 analyzes the performance of the example protocol stack for each protocol architecture. It first describes the benchmark tools used for performance measurements and then gives performance results of the round-trip latency, incremental cost per protocol per round-trip, latency breakdown, throughput, and connection setup cost. Some

multiprocessor performance experiments and their results are also reported.

Chapter 7 presents a summary of the results. This chapter concludes with a description of the research contributions of this thesis and the identification of some future research areas.

Chapter 2

Related Work and Thesis Proposal

Traditionally, communication protocols have been implemented in a monolithic fashion. In these implementations, the complete protocol stack is implemented either in the operating systems kernel [LMKQ89] or in a trusted user address space or server [GDFR90]. The monolithic protocol architectures have been quite successful in meeting the demands of the last generation of networks because networks were quite slow and they were used to carry only one type of data, that is, time insensitive data. With the advent of high-speed networks such as *Asynchronous Transfer Mode*, future communication environments are expected to comprise of a mix of networks with widely varying characteristics [CT90, Par90, Par93]. Therefore, future communication protocols not only have to support a wide range of networks but also a wide variety of applications that will transfer multimedia data (e.g. voice, video, graphics, text) and will have different access patterns such as interactive, bulk transfer, and real-time guarantees. Widely varying data characteristics mixed with various access patterns will require wide range of *Qualities of Service(QoS)* that are not addressed by current protocol architectures. Researchers have taken different directions to meet these requirements[1]:

- Analyze the shortcomings of current protocols and extend them for future network requirements [Cal92, CSBH91, Dee92, Fra93, JBB92].

[1]The following choices are not mutually exclusive. The references given are by no means exhaustive.

- Re-examine the overheads in the current protocol implementations [BOP94, CJRS89, Jac88, Jac90, KP93a, KP93b, MG94, PP93, PDP93, WVT93], and adapt them for higher performance in the future high-speed network environments [BB92, BOP94, Jac88, Jac90, KP93b, NGB+91] .

- Develop new light-weight protocols which can guarantee high throughput, low jitter and low latency [Che86, Che88, CLZ87, Fel93, PS93, Top90].

- Implement transport protocols in dedicated hardware [AABYB89, Che88, CSSZ90, KC88, NIG+93] or implement some protocol functionality in hardware [BPP92, DWB+93, TS91].

- Propose parallel protocol architectures [BG93, Dov90, GKWW89, GNI92, JSB90, WF89, Zit89].

- Propose new techniques for the implementation of protocol stacks such as *Integrated Layer Processing(ILP)* [CT90], *Application Level Framing* [CT90], *Application-oriented Light-weight Transport Protocols(ALTP)* [PT89], Flexible Protocol Stack [Tsc91] and Non-Monolithic Protocol Architecture [JHC94a, MRA87, MB92, MB93, TNML93].

Traditionally, there has been little coordination between the design of the host operating systems and design of communication subsystem, thereby resulting in poor performance of the communication subsystem. The major sources of overhead that lead to the poor performance of communication subsystem are identified as multiple data copying, poor management of timers, buffers, interprocess communication, interrupts, context switches, and scheduling. A lot of effort is now being expended in the direction of coordinated design of network interfaces, structuring the protocol implementations

and designing suitable operating systems primitives to minimize the host's overhead for communication subsystems.

In this thesis, we concentrate on new protocol architectures for future gigabit networking environments. For this purpose, we first analyze the traditional protocol architectures and identify their drawbacks, and then propose new protocol architectures which can potentially avoid these drawbacks. We believe that it will take several years before new protocols become acceptable for commercial use, and even if they gain acceptance, traditional protocols will remain in use for several years (potentially forever in some parts of the globe). Hence, it is necessary to adapt traditional protocols to run in the future networking environment.

2.1 Requirements for High-Performance Protocol Processing

Clark, Tennenhouse and Feldmeier [CT90, Fel90, Ten89] have analyzed traditional protocol architectures and identified their key idiosyncrasies that present bottlenecks to future high-speed network environments. This section discusses the shortcomings of traditional protocol structures in the context of the following key requirements for future network environments.

2.1.1 Providing Wide Range of Quality of Service to Applications

Next generation multi-media applications will require a variety of services from communication protocols, such as real-time guarantees and bulk data transfer. We expect that in systems supporting these applications, both time sensitive and time insensitive data will compete for shared system resources. A proper sharing strategy is essential to provide performance guarantees while maintaining high system throughput and efficiency. ATM networks guarantee QoS negotiated on an application specific basis. However, this is not

generally true for higher layer protocols that are executed in the host operating systems and account for a large part of the protocol processing overhead.

To provide different QoS to different applications requires that each application be treated according to its individual needs at all resource sharing points. This purpose, however, is defeated by the *logical multiplexing* that occurs at several layers in a traditional protocol stack. By logical multiplexing, we refer to the mapping of multiple streams of layer n into a single stream at layer $n-1$. The problems with layered multiplexing can be summarized as follows [Fel90, Ten89]:

- Loss of individual QoS parameters of multiple higher layers at the lower layer. Streams with different QoS are multiplexed into a single lower layer stream. As a result, all upper layer streams are treated identically at the lower layer.
- Layered logical multiplexing complicates protocols and their implementations. Similar header fields found at multiple layers may result in reduced throughput.
- Multiple context state retrieval as a result of demultiplexing at several layers is slower than a single but larger context state retrieval.
- Flow control functionality is duplicated at multiple layers.
- In the context of multithreaded and parallel processors, layered multiplexing causes control data to be shared and hence, restricts the degree of parallelism.

From the discussion in the previous paragraph, the following conclusions can be made:

- Multiplexing/demultiplexing should be done in a single layer and in the lowest possible layer. The demultiplexing at a single layer needs to determine the application identity and hence, there should be a one to one correspondence between the demultiplexing *key* contained in the header and the application to which the packet is destined.

- QoS information should be processed and acted upon as early as possible after the demultiplexing point. This implies that QoS information should be contained in the same layer header which contains the demultiplexing information.

We propose an architecture called *Direct Application Association* which addresses the issues raised above.

2.1.2 Integrated Layer Processing

Traditionally, network protocols are implemented in a layered fashion. International Standard Organization (ISO) has proposed a seven-layer model. The advantage of layering is that it provides modularity and hides details of one protocol layer from the other layers. The drawback is poor performance because of the sequential processing of each data unit as it passes through the protocol layers[2]. Furthermore, multiple layers may perform *data manipulation* on the data unit. The data manipulation functions are those which read and modify data. Examples of these functions include encryption, compression, error detection/correction, and presentation conversion[3] [CT90]. The data manipulation functions suffer from serious performance problems on modern processors because they cause high traffic over the CPU/memory bus in the presense of cache misses. In a traditional protocol implementation, data manipulation operations are performed independently of one another because of layering. Thus, the cache performance is seriously affected as the cache may be invalidated when going from one layer to another.

Data manipulation functions of different protocol layers are similar and hence, can be performed in an integrated fashion so as to get maximum benefit from the cache.

[2]The layers can be arranged in a pipeline to increase the performance. This purpose, however, is defeated because pipelining requires buffers and has overhead of synchronizing activities in adjacent layers.

[3]Presentation Conversion refers to the reformatting of data into a common or external data representation such as Sun XDR [Inc87] or ASN.1 [fSIPSOSI87].

This approach is termed *Integrated Layer Processing (ILP)* [CT90]. The ILP approach restructures the data manipulation operations so that most of the time, data is found either in the cache or in the processor registers so that there is minimum data transfer over the CPU/memory bus. Thus, n read and n write operations which would have resulted in $2n$ external memory operations, require only 2 external memory operations in the ILP approach. In an ILP approach, a protocol stack is still arranged in a logical layered fashion while its implementation does not follow strict layering. The ILP is an engineering principle which should be applied to the implementations only when performance gains can be achieved.

Abbot and Peterson [AP93] propose an ILP scheme in the context of the x-kernel. They arrange the data manipulation steps of the various protocols into a pipeline. Data is loaded into the pipeline word by word, and the data manipulation functions of the various layers are performed on the data while it remains in the registers, with the data finally being stored back in the memory when processing is complete. So, data is read and written over the memory bus only once instead of being read and written once for each function at each layer. They report substantial performance gains for data manipulation operations by employing the ILP technique [AP93]. The main problem with integrating the data manipulation functions of different protocol layers is that different layers may have different views of what constitutes data in a packet. A higher layer protocol header is typically considered as data by a lower layer protocol. For example, the IP layer views the TCP header as data. This complicates the integration. One solution is to integrate only that part of the message that each layer regards as data. This solution requires that the data and header boundary be known to each layer. For outgoing packets, this boundary is already known and hence, integration can be handled easily. For incoming packets, the data and header boundary can be located only after the demultiplexing operation has been performed at each layer in the protocol stack. Thus, it is difficult to

achieve ILP for incoming packets. The packet filter [MRA87] has been suggested as a tool to peek into the headers and locate the header-data boundary before the packet is processed. However, this solution is inefficient for the following reasons:

- It duplicates protocol demultiplexing: once during packet filtering and once again during the packet processing at each layer.
- Except for locating header-data boundary, it does not really provide any additional support for easing the design of Integrated Layer Processing because demultiplexing is still performed at multiple protocol layers, thereby hindering ILP.

This thesis proposes an efficient protocol architecture called *Integrated Layered Logical Multiplexing* which overcomes these drawbacks.

2.1.3 Application Level Framing

Clark and Tennenhouse [CT90] have argued that on modern processors, presentation conversion may seriously affect the performance of protocol processing. The important aspect of presentation conversion, in general, is that it is performed in the context of the application. A typical example [CT90] is *Remote Procedure Call (RPC)* [BN84]. In an RPC call, transferred data represents the arguments and results of the remote execution of a procedure and are interpreted by the application. So, the presentation conversion should also be dictated by the application for the best performance.

Another problem in supporting real-time applications is that of lost and mis-ordered data [CT90]. In present protocol implementations (e.g. TCP-IP), data loss and misordering blocks the application. Hence, presentation conversion also stops. This also presents a bottleneck to the integration of data manipulation functions starting from protocol layers up to the application. For example, applications delivering video and voice data

can tolerate some loss of data and hence, can process misordered data if they can be informed. In a traditional protocol such as TCP, this cannot be achieved because TCP is based on a byte stream. The byte stream is meaningless for the application because the presentation layer between TCP and the application reformats the data. An alternative option is to have the application, and not the transport protocol layer, retransmit the lost data.

From the above discussion, it can be concluded that for future network environments, data units should be application specific and not protocol layer specific and protocol layers should respect the boundaries of these data units. These data units become the basic units of data manipulation and error recovery. This design principle is called *Application Level Framing (ALF)* [CT90] and the data units are termed as *Application Data Units (ADU)*. Application level framing is protocol stack specific and requires that some functionality, such as recovery from lost or misordered packets which was traditionally part of the protocol layers, be shifted to the application layer. In the traditional protocol architecture, it is very difficult to meet the goal of application level framing [CT90]. It is simpler to implement application level framing in user-level protocol architecture [JHC94a, MRA87, MB92, MB93, TNML93]. In user-level protocol architecture, complete or parts of protocol stacks are linked to the application as user-level library and hence, application has better control on the protocol processing than that available in kernel based protocol implementations.

2.1.4 Getting Bits Faster to Applications

Multimedia applications requiring real-time guarantees must receive data from the network as soon as it arrives. These applications also need to be informed of the lost or misordered data as quickly as possible. Monolithic implementations have difficulty supporting these applications because packets are processed in the kernel thereby delaying

the delivery of data contained in the packets to applications. In user level protocol implementations, the kernel simply determines the application to which a packet belongs and sends the packet to the application. The packet processing is performed in the application. Hence, user level protocol processing provides the fastest way to deliver packets to applications. In user level protocol implementations, an application also has more control on packet processing as required by multimedia applications.

2.1.5 Parallelization of Protocol Stacks

One of the main reasons of slower protocol processing on modern machines is more than one order of magnitude difference between processor speed and memory speed. How does the processor keep packet processing up with the network speed? A common solution is to allocate a number of processors to process network packets and hence, parallelize protocol stacks.

Parallel Protocol Models

Several models for parallel implementation of protocols have been proposed in the past [BG93, Dov90, GKWW89, GNI92, JSB90, WF89, Zit89]. All these models were concerned with the parallelization of monolithic protocol stacks and have had little success. The primary hindrance to parallelization comes from the fact that protocol processing is layered and hence, serialized. In this section, we briefly describe these models[4]. In Section 2.3.3, we show that Non-Monolithic protocol implementations are more parallelizable than the traditional monolithic implementations and therefore, are architecturally superior to traditional monolithic ones.

[4]They have also been summarized in [BG93] and [GNI92].

Processor per Protocol

In the processor per protocol scheme, one or more processors are dedicated to each protocol and the protocol stack is essentially a pipeline of processors. The packets move from processor to processor in the pipeline in both directions. Each processor performs protocol specific processing of packets and send them down or up the pipeline. To reduce the interference between incoming and outgoing packets, separate unidirectional processor pipelines can be used to form a bidirectional pipeline. However, the protocol state still needs to be shared among multiple processors.

This scheme has the advantages and disadvantages of standard pipeline processing. The main advantage is that this scheme fits well with the protocol layering approach. It is natural to implement each layer on a separate processor and therefore not to share the protocol state with processors dedicated to other protocol layers. The number of packets processed in parallel could be as high as the number of stages in the pipeline.

The main disadvantage of processor per protocol scheme is that the pipeline performance is determined by the slowest segment in the pipeline. The slowest segment can be subdivided into multiple segments to increase its performance but it may make the implementation complex and introduce the overhead of synchronization between boundaries if such divisions do not fall on natural boundaries. Another main drawback of the scheme is that communication between the processors is required at each boundary. If the number of processors are less than the number of layers, extra context switches for each packet may seriously degrade the performance. Also, each packet is processed by a different processors at different layer, so the packet will seldom be found in a processor's cache, thereby reducing the performance. On fast processors, overhead of context switch and cache miss is so high that it is difficult to justify using this model for supporting high-speed networks.

Processor per Connection

In the processor per connection scheme, each connection is assigned one processor implying that there is no parallelism within a connection. However, packets related to multiple connections may be processed in parallel on different processors. Synchronization is required for resources that are shared by more than one connection. All packets belonging to the same connections are processed serially and hence, no synchronization is required within the data structures that are local to a connection. The main advantage of this approach is that each packet is processed by a single processor and connection state is not shared among processors. The main disadvantages of this scheme are as follows:

- In traditional protocol stacks which follow the Layered Logical Multiplexing model, several upper protocol layer's connections may be multiplexed over a single lower protocol layer's connection. In this case, all such connections are serviced by a single processor, thereby drastically restricting the available parallelism.
- Applications which open a small number of heavily used connections do not benefit from this model even though a large volume of data may be transmitted back and forth.

Processor per Function

In the processor per function approach, one processor is dedicated for a specific function within a protocol or a specific function common to more than one protocol. This scheme takes parallelism to the extreme of fine grain parallelism. The main advantage of this approach is that it can exploit the maximum available parallelism. The main drawback of this scheme is that the overhead of fine grain synchronization and extra communication among the processors may dominate the processing cost, thereby resulting in poor performance.

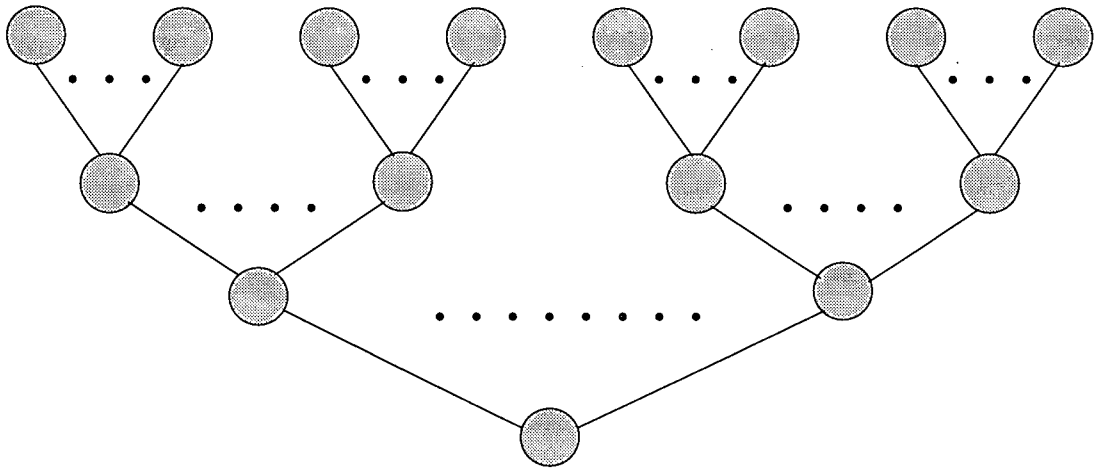

Figure 2.1: Layered Logical Multiplexing Protocol Architecture

Processor per Packet

The most promising approach is to assign one processor per packet. Each processor shepherds the packet through the complete protocol stack. This scheme minimizes the context switches and maximizes the cache hits resulting in better performance. There is no extra communication overhead between adjacent layers. Synchronization is still required for the parallel processing of packets. We adopted this approach for all our protocol implementations.

2.2 Traditional Protocol Architectures

2.2.1 Layered Logical Multiplexing (LLM) Protocol Architecture

In traditional protocol architectures, multiplexing and demultiplexing operations are performed at multiple protocol layers. Such protocol architectures can be termed as *Layered Logical Multiplexing* architectures. Figure 2.1 depicts the behavior of a LLM model. Multiple streams (or sessions) of layer n are mapped to one stream (or session) at layer $n-1$. As discussed in Section 2.1, this architecture has difficulty supporting

application specific QoS and implementing protocol stack in an ILP fashion.

2.3 Alternative Protocol Architectures

In this section, we describe the alternative protocol architectures proposed in this thesis for future gigabit networking environments.

2.3.1 Direct Application Association (DAA) Protocol Architecture

It has been argued by Tennenhouse [Ten89] and Feldmeier [Fel90] that logical multiplexing/demultiplexing at multiple layers in the protocol stack significantly reduces throughput and therefore should be performed only in a single layer in the protocol stack. For user level protocol implementations, the simplest and most efficient approach would be to have an application identifier enclosed in the MAC layer header. The device driver can interpret the application identifier and have the packet routed directly to the application.

As its name suggests, in the DAA protocol architecture, an application specific identifier is encoded in the header of the lowest protocol layer to achieve a single demultiplexing point at the lowest protocol layer in the stack. In ATM networks, a *Virtual Connection Identifier* and *Virtual Path Identifier* pair can be associated with an application to implement protocols in a DAA fashion. Traditional networks do not provide such support and require incompatible extensions.

Figure 2.2 shows a DAA protocol architecture. In the DAA protocol architecture, demultiplexing is performed only once at the lowest protocol layer which identifies the application that is to receive each incoming packet. An analytical study of DAA and LLM models [Zhu92] shows that the LLM model has inherent performance bottlenecks and that the DAA model is superior to the LLM model. These results are in agreement with the arguments given by Tennenhouse [Ten89] and Feldmeier [Fel90]. Other advantages of

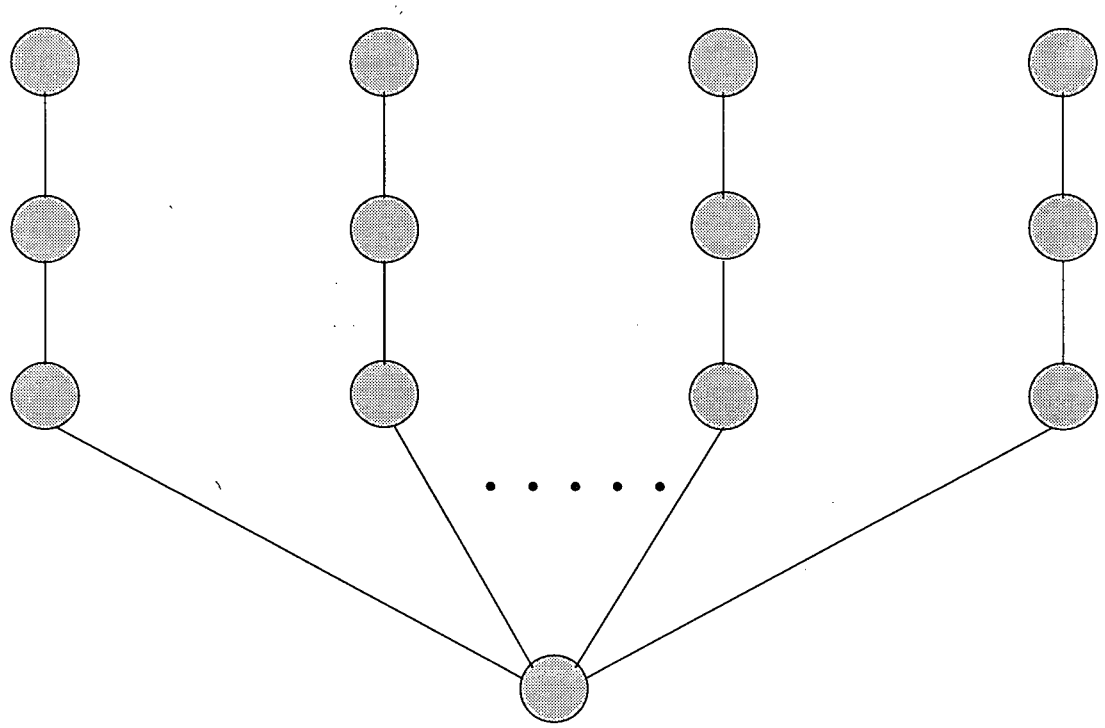

Figure 2.2: Direct Application Association Protocol Architecture

having a one-to-one correspondence between applications and demultiplexing keys are as follows:

- Extra data copies can be eliminated because the network device can directly copy the data in the recipient application's buffer.
- The system can check whether the recipient application has sufficient resources to receive the packet as early as possible. This is very difficult to do in a LLM protocol stack [Pet94].
- For applications requiring real-time guarantees or other kinds of QoS, system resources can be allocated appropriately to the packet so that the packet can meet its QoS.

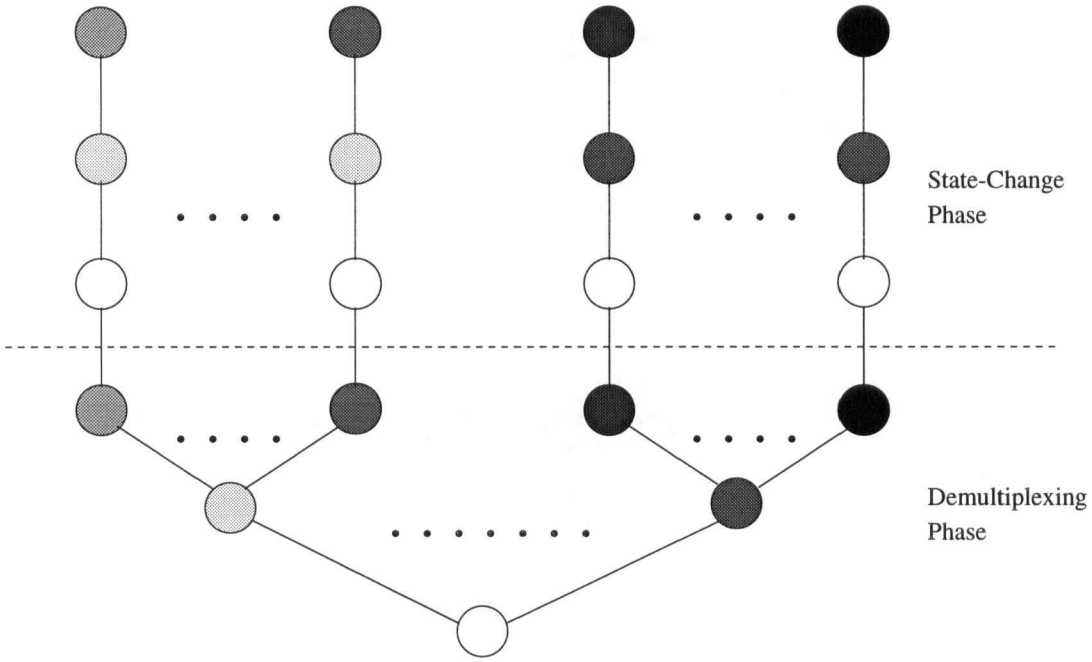

Figure 2.3: Integrated Layered Logical Multiplexing Protocol Architecture

- In high-performance multimedia systems, the destination could be an I/O device such as a frame buffer or a video decompression board. In the DAA scheme, the data can be routed directly to the I/O device without processor involvement.

We design a framework for implementing a DAA architecture stack in the context of the *x*-kernel [JHC94b]. We also implement an example DAA architecture protocol stack by extending the Ethernet header to enclose an application identifier and a QoS information field. These extensions are described in Section 4.2.

2.3.2 Integrated Layered Logical Multiplexing (ILLM) Protocol Architecture

In the ILLM architecture, packet demultiplexing which is normally performed at different protocol layers, is decoupled from these protocols and is performed successively

in an integrated fashion. When the final destination session[5] has been determined, the protocol specific state change processing is then performed in succession on the protocol layers' sessions as shown in Figure 2.3.

Figure 2.3 distinguishes the sessions at different layers with different shades. In the ILLM architecture, the processing of received packets can be divided into two phases: a *demultiplexing phase* and a *state-change phase*. The demultiplexing phase identifies a list of sessions of different protocol layers that the network packet will be visiting. The state-change phase operates on the sessions identified by the demultiplexing phase to update the state of connection represented by these sessions. This architecture achieves the ideally expected behavior (i.e. single demultiplexing point in the complete protocol stack) proposed by Feldmeier [Fel90] and Tennenhouse [Ten89] while still maintaining compatibility with existing protocols.

We observe the following advantages of the ILLM architecture: the demultiplexing operations of multiple protocol layers are decoupled from the rest of the protocol processing. This avoids the drawback of the traditional LLM architecture [Fel90, Ten89] where demultiplexing is performed at multiple layers intermixed with protocol state-change operations. As pointed out in Section 2.1.2, this intermixing hinders ILP because data manipulation functions (which need to be integrated) are part of protocol state-change processing. The ILLM architecture also solves another major problem of ILP optimization for traditional protocol architectures - determining the boundary between headers and data in network packets. As discussed in Section 2.1.2, we need to know the boundary between headers and data as early as possible. For outgoing packets, this boundary is already known. For incoming packets, it is difficult to locate this boundary immediately after the packet is received. Again, this is because in the LLM architecture, state-change operations are sandwiched between demultiplexing operations. A simple but inefficient

[5]In the sense as used in the *x*-kernel defined in Section 4.

solution suggested by Abott and Peterson [AP93] is to use a Packet Filter [MRA87] before the beginning of protocol processing of the received network packet. The Packet Filter technology essentially performs the protocol demultiplexing operation and hence, in this solution, demultiplexing is performed twice: once to locate the boundary between headers and data and then, to locate the sessions of the different protocol layers. In the ILLM approach, demultiplexing is performed only once. Another interesting advantage of the ILLM architecture is that it allows easy integration of data manipulation functions of different protocol layers during the state-change phase and therefore, eases ILP. The ILLM model achieves all the advantages of a DAA architecture described in Section 2.3.1.

2.3.3 Non-Monolithic Protocol Architecture

Traditionally, protocols have been structured in a monolithic fashion with the protocol stack implemented either in the operating systems kernel or in a single trusted user-level server. In a Non-Monolithic protocol architecture, each application is linked with its own private copy of the protocol stack. This implies that protocols are implemented as a user-level library. However, a number of issues must be resolved before these implementations can be made truly functional.

There have been a number of previous attempts to implement protocol stacks at user level [MB92, MB93, TNML93, MRA87] but these implementations were ad hoc. We propose a new approach to implement Non-Monolithic protocol architecture. In this section, we describe the advantages of user-level protocol architectures, previous user-level protocol implementations and their drawbacks. Finally, we propose an efficient, structured, modular, and general framework for the implementation of Non-Monolithic protocol architectures [JHC94a] in the context of the x-kernel.

Two previous efforts to split the implementation of communication protocols between the kernel and user space are described by Maeda and Bershad [MB93] and Thekkath

et al [TNML93]. 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 is easy 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.
- For the new generation of applications that deal with video and audio data, it is important to get the data to the applications as fast as possible after the data is received from the network. Getting bits faster to the desktop is no longer sufficient.
- Implementation of application level framing [CT90] architecture is possible in Non-Monolithic protocol architectures. Clark and Tennenhouse found that the implementation of application level framing is extremely difficult, if not impossible, in the traditional monolithic protocol architectures [CT90].
- Performing only demultiplexing operations in the kernel integrates the demultiplexing operation of different protocol layers into a single "one-shot" operation while maintaining compatibility with existing protocols.

Previous User-Level Protocol Implementations

In previous user-level protocol implementations[6] [MB93, MRA87], a Packet Filter installed in the kernel demultiplexes the packets to the correct recipient address space. Examples include the CMU/Stanford Packet Filter (CSPF) [MRA87], the BSD Packet Filter (BPF) [MJ93], and the Mach Packet Filter (MPF) [YBMM94]. 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 [MB93]. However, we observe that the packet filter technology [MJ93, MRA87, YBMM94] 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 [Pit87].

- Interpretation of some header 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 specify every protocol in the path between the network device and the user. For example, while the transport protocol (UDP [Pos80] or TCP [Pos81c]) installs the packet

[6] In [TNML93] implementation, an application identifier is encoded in the link-level header in the case of AN1 network[SBB+91] for demultiplexing. It is not clear how the demultiplexing is performed in the case of Ethernet network.

filter, that packet filter depends on the network protocol that will eventually deliver the data, thus losing the modularity advantage of IP [Pos81b] 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 [Pos81b] protocol. It is difficult for a packet filter to handle the reassembly of IP fragments, IP forwarding, broadcasting, IP multicasting and IP tunneling[7]. Of all the packet filter implementations reported in literature, only the Mach Packet Filter [YBMM94] 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 and that making the packet filter do more (like handling IP fragments) is as easy as adding a few new instructions [YBMM94]. 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

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

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. Our Non-Monolithic protocol architecture achieves exactly that.

Proposed Framework for Non-Monolithic Protocol Architecture

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. What was once an unstructured kernel level protocol implementation becomes, with the addition of a packet filter, an unstructured user level protocol implementation. Our architecture 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 [JHC94a]:

- 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 in the kernel. However, the design goal would be to place as much functionality as possible in the user's address space.

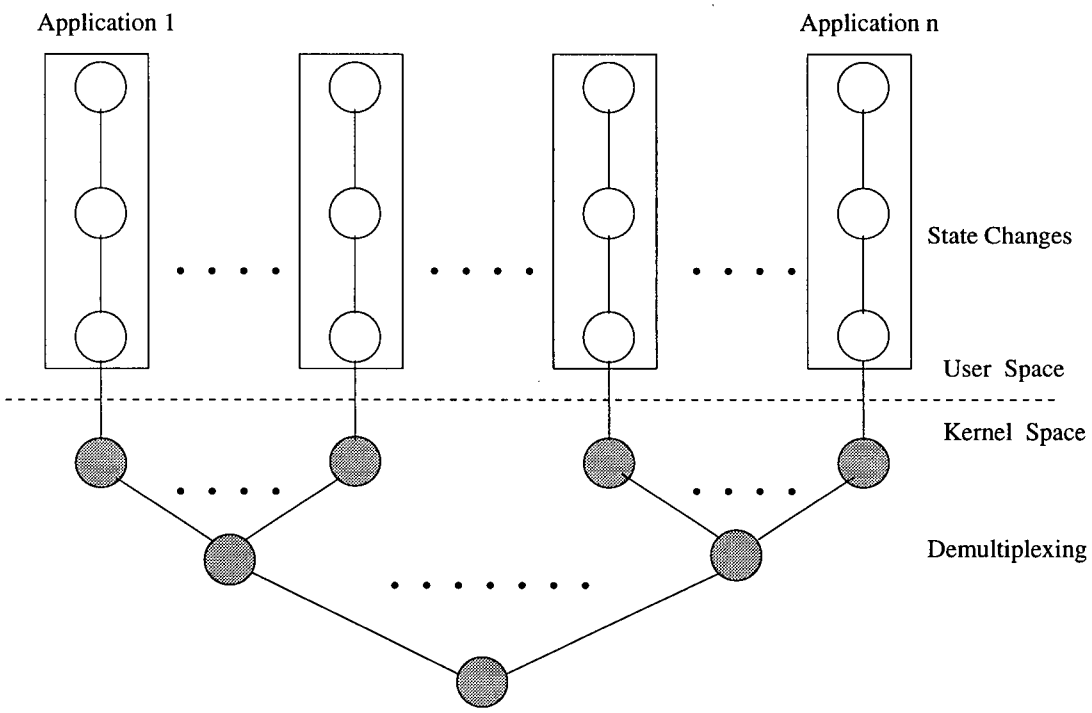

Figure 2.4: Non-Monolithic Protocol Architecture

- In the extreme, it is possible to configure the same protocol in the kernel for one application and in user space for another application.

Our Non-Monolithic protocol framework can be characterized 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.

Figure 2.4 depicts the proposed Non-Monolithic protocol architecture whereby kernel performs at least the demultiplexing of incoming network packets and the rest of the protocol processing is performed in appropriate applications.

Parallelization of Non-Monolithic Protocol Architecture

Bjorkman *et al.* [BG93] report an analysis and performance of the most widely used protocol stack, TCP/IP, on a multiprocessor machine. They report that TCP has so much shared state that it does not scale well beyond three processors. In the Non-Monolithic protocol architecture, each application has its own copy of the protocol stack. For instance, in a TCP implementation for Non-Monolithic protocol architecture, each application has a separate TCP protocol state which is not shared with any other applications' TCP protocol state. Therefore, different user level instances of the TCP protocol can be executed in parallel on different processors without any synchronization among them. Hence, Non-Monolithic protocol architectures provide virtually unlimited parallelism in case of a large number of applications. If an application opens a number of TCP connections, it can create a new TCP protocol instance for each connection so as to get maximum benefit of available parallelism. On the other hand, parallelism may be limited in the kernel level demultiplexing. Although demultiplexing is a read-only operation, kernel sessions corresponding to user connections may be created or destroyed asynchronously. Hence, for consistent demultiplexing, kernel level sessions must be locked during kernel level demultiplexing. In a simple approach, read and write locks can be used for this purpose. Two or more applications can hold read-read locks simultaneously while read-write and write-write locks are mutually exclusive. This locking scheme will result in a fairly good performance if the kernel level sessions are not created and destroyed rapidly. An alternative to read-write locks is to use wait-free synchronization [Her91, Her90, MCS91] and data structures known as lock-free objects [Ber91, MP91]. With these data structures it is possible to totally parallelize the kernel level demultiplexing.

From the above discussion, it can be inferred that the Non-Monolithic protocol architecture is highly scalable with the number of processors and the theoretically best (i.e. linear) speedups can be obtained. There is no inherent limitation to parallelization of protocol stacks implemented in Non-Monolithic fashion. The Non-Monolithic protocol architecture is independent of the protocol complexity and shared state. Although any of the parallelization models for protocols described before could be applied to Non-Monolithic protocol stack implementations, the processor per packet model suits best. We use this model for our implementation.

Chapter 3

The Parallel x-kernel

This chapter first presents an overview of the x-kernel, which is essential in understanding the rest of this thesis. It then describes the scheme used to parallelize the x-kernel for a shared memory multiprocessor machine.

3.1 The x-kernel Overview

The x-kernel [HP88, HP91] is a protocol implementation architecture that defines protocol independent abstractions thereby facilitating efficient implementation of communication protocols. The x-kernel provides three primitive communication objects: *protocols*, *sessions,* and *messages.* Each protocol object corresponds to a communication protocol such as IP [Pos81b], TCP [Pos81c], or UDP [Pos80]. A session object represents the local state of a network connection within a protocol and is dynamically created by the protocol object. Message objects contain the user data and protocol headers and visit a sequence of protocol and session objects as they move through the x-kernel.

The x-kernel maintains a *protocol graph* that represents the static relationship between protocols. Each protocol implements a set of methods which provide services to both upper 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 hierarchies is deferred until link time. 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. The anchor protocols are used to define the application interface and the device driver interface. The anchor protocols allow easy integration of the *x*-kernel into any host environment.

3.1.1 The *x*-kernel Support Tools

The *x*-kernel provides a set of support tools [HMPT89, HP91] to implement protocols efficiently and quickly. These tools consist of a set of routines that are commonly used to implement protocols. The most commonly used tools are as follows:

- **Map Manager:** The Map Manager provides a set of routines to translate an *external identifier*[1] extracted from the protocol headers (such as address and port numbers) to the *internal identifier* (e.g. receiver of the packet i.e. a session) within the protocol. The map manager consists of a set of hash tables (called *maps*) and functions to add, remove, and map bindings between external and internal identifiers. The maps cache the last key that was looked up.

- **Message Manager:** The Message Manager provides a message abstraction to packets and provides a common set of operations on messages so that protocols can add headers and trailers to the messages, strip headers and trailer from messages, and fragment and reassemble the packets.

- **Process Manager:** The Process Manager provides a set of thread management, semaphore synchronization, and spin locks routines in a system independent manner. This tool facilitates easy portability of the *x*-kernel to different environment.

- **Event Manager:** The Event Manager provides the ability to schedule various events asynchronously in the future. The event manager is used for scheduling

[1] Also referred to as *key*.

procedures for execution after a specified amount of time. A good example of this, in the context of communication protocols, is a *timeout* event. By registering a procedure with the event manager, protocols are able to implement timeout events to retransmit unacknowledged messages or to perform periodic maintenance functions such as collection of unused sessions.

3.1.2 Operations on Protocol Objects

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

Protocol objects support 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 the *open* routine of a lower protocol. Alternatively, a high level protocol can pass a capability to a lower level protocol for passive creation of a 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 identifiers such as port numbers, protocol numbers, or type fields used by the protocols to identify the message recipient session or enable objects. The mapping between external identifiers and internal identifiers (which are capabilities for session objects or enable objects) is maintained in the *active* and *passive* maps maintained by each protocol.

In addition to managing sessions, the protocol object also demultiplexes incoming messages to the appropriate sessions. 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 identifier* to search through the two types of maps. The active maps are searched first. If a binding to a session is found in the active map, the message is delivered to that session. If no session is found in the active map and search in the passive map yields an enable object, a new session is created from the enable object and the message is passed to the newly created session. The message is dropped if search in both type of maps fails.

3.1.3 Operations on Session Objects

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

3.2 Parallelization of the *x*-kernel

The standard *x*-kernel's release from the University of Arizona is multiprocessor safe but does not support multiprocessing in the *x*-kernel. This means that every thread of execution that needs to enter the *x*-kernel must synchronize on a coarse grain lock called *MasterLock*. When a thread blocks in the *x*-kernel (e.g. while waiting on a semaphore), the thread must release the *MasterLock* lock before blocking. This scheme does not allow multiple threads execution in the *x*-kernel simultaneously and hence, restricts parallelism. To enable multiple threads of execution in the *x*-kernel, a fine grain locking scheme is required.

A fine grain locking scheme must take special care to avoid deadlocks because in communication systems, multiple messages travel in upward and downward directions

at any given time. Hence, multiple threads executing in different directions through the protocol layers may lead to cyclic waiting for locks and end up in a deadlock [Hut92]. Another requirement of the fine grain locking scheme is to make the job of protocol writers as easy as possible from the locking point of view.

An ideal fine grain locking scheme would make locking transparent to the protocol writer by handling locking in the *x*-kernel UPI library. This scheme would make locking totally transparent to the protocol writer. Designing an ideal locking scheme has been shown to be a difficult goal to achieve [BG93, Hut92]. On the other hand, the simplest locking scheme would be to perform all the locking in the protocol code. This implies that the protocol writer must do all the locking explicitly and must know the internals of the *x*-kernel. This approach is also not viable because it expects too much from the protocol writer and makes the implementation error prone. Besides, this approach may result in poor performance of the complex *x*-kernel operations.

An intermediate approach, whereby the *x*-kernel protects its own data structures and the protocol writer protects protocol specific data structures by locking, seems most appropriate. This thesis adopted such an intermediate approach for the multiprocessor implementation of the *x*-kernel. Bjorkman [BG93] has also taken a similar approach to parallelization of the *x*-kernel.

In an intermediate locking scheme, the parallelization of the *x*-kernel can be divided into two main components: parallelization of the *x*-kernel itself and parallelization of the protocols. This locking scheme avoids deadlocks. An alternative scheme would be to detect deadlocks and take corrective action to break deadlock situations. This alternative scheme has been used in database systems but it is too expensive for performance critical communication systems.

3.2.1 Locking Scheme in the *x*-kernel

The *x*-kernel tools are parallelized to support fine grain parallelism. The following rules are used to avoid deadlock within the *x*-kernel:

- The *x*-kernel does not lock protocol data structures. Similarly, protocol code does not lock the *x*-kernel data structures.
- The *x*-kernel does not hold locks on its own data structures while calling protocol routines.
- If more than one lock need to be obtained in the *x*-kernel, a predefined order of locking is followed to avoid deadlocks.
- Thread scheduling is made non-preemptive if a thread is holding a lock. This support is provided by the Raven kernel. In the Raven kernel, a thread holding a lock is never preempted. All interrupts are queued during the critical period activity of the thread and processed when the thread releases the last lock. This feature is specially useful to avoid deadlocks for asynchronous event handling.

The above rules guarantee that the *x*-kernel is deadlock free.

3.2.2 Locking Scheme in the Protocols

Assuming that the *x*-kernel is deadlock free, the protocol writer must ensure that the following conditions hold for deadlock free operation of protocol processing:

- Protocol code does not lock the *x*-kernel data structures.
- If more than one lock is obtained in the protocol code, a predefined order of locking that avoids deadlocks is followed.

- The protocol writer may have to invoke a *x*-kernel routine while holding one or more protocol locks. In such situations, the protocol writer should know what protocol routine may be called as a result of the call to a *x*-kernel routine and make sure that circular waiting on locks does not occur.

3.3 Implementation Details of the Parallel *x*-kernel

This section gives the implementation details of the scheme that we adopted for parallelization of the *x*-kernel.

3.3.1 Locking in the *x*-kernel

The *x*-kernel's UPI library and other *x*-kernel tools require locking for supporting fine grain parallelism. In the UPI library, the reference counts of `XObj` data structure are protected by locks for consistency. The stack manipulations of the `Part` data structure are protected by locks.

Map Manager

The Map manager's hash table manipulations are protected by locks.

Message Manager

Because of the processor per message model, only reference count manipulations of the `Msg` data structure are protected by locks.

Process Manager

The global lock (*MasterLock*) has been removed to enable fine grain parallelism in the *x*-kernel. The process manager inherits most of the multiprocessor support from the

underlying Raven kernel's thread management, memory management, and semaphore library for supporting the *x*-kernel's routines for thread, memory, and semaphores management functions.

Event Manager

Event queues are protected by proper locking. Interrupts are disabled whenever a thread manipulates the event queue[2]. This feature is required to avoid deadlock due to asynchronous events.

3.3.2 Locking in the Protocols

We implemented the UDP, TCP, IP, ICMP, ARP, VNET, and Ethernet protocols for multiprocessor environment. The fine grain locking in TCP is not completely implemented. We deferred the analysis of TCP to support fine grain parallelism because previous results have shown that TCP does not scale well on multiprocessors [BG93]. Furthermore, by implementing the TCP/IP stack using the non-monolithic protocol architecture, parallelization of a complex protocol like TCP can be avoided because a separate TCP/IP protocol stack can be allocated for each TCP connection.

UDP

UDP is a simple protocol that does not have any shared state. The reference count manipulations of session and enable objects are protected by locks. The UDP port management data structures are protected by locks.

[2]This feature is supported by the Raven kernel. A thread never gets preempted if it is holding a lock. The interrupts are queued and serviced when the thread releases all the locks.

IP

The reference count manipulations of IP session and enable objects are protected by locks. The IP reassembly table which stores IP fragments is also protected by locks.

ARP

The ARP tables are protected by locks for consistency in multiprocessor environment.

Ethernet

The Ethernet layer in the *x*-kernel is divided into two parts. The top layer is the Ethernet protocol and the bottom layer is the Ethernet driver called *Simeth.*

The reference count manipulations of the Ethernet session and enable objects are protected by locks. The Simeth driver is linked to the Raven kernel's Ethernet controller which provides fine grain locking for multiprocessors.

3.3.3 Support for Processor per Message

The multiprocessor version of the *x*-kernel supports the processor per message paradigm. Each packet is assigned a processor and the processor shepherds the message through the protocol stack. For performance reasons, a global pool of threads is maintained at the uppermost protocol and at the lowermost protocol. Each thread in the thread-pool is associated with its own semaphore. When a message is received from either a user or a network device, a semaphore signal awakes a thread. The thread copies the message and shepherds it through the protocol stack. A good example is thread-pool management at Ethernet driver. When a receive interrupt of the Ethernet driver is received, the interrupt handler routines awakens one of the threads waiting on the semaphore. When the thread gets scheduled, it copies the received packet from the Ethernet driver buffer, converts it

to the *x*-kernel message object, and shepherds the message through the protocol stack. When the message is delivered to the application, the thread goes back to sleep on the semaphore again to wait for the arrival of another packet. The thread pool management data structures are protected by locks.

In communication subsystems, a message may change its direction e.g. a TCP/UDP message destined to another application on the same machine. In this case, the IP protocol code detects that the message is destined for an application on the local machine and sends the message up the protocol stack. In a naive implementation, a deadlock may result if a thread is allowed to reverse its direction. In the *x*-kernel implementation, even though a message may turn back, a thread never does. In such cases, a new thread is created and the message is assigned to the new thread for shepherding the message in the reverse direction.

The same convention, as used by the standard *x*-kernel to free message objects, is employed in our implementation. In the standard *x*-kernel, a message object is destroyed by the same entity that originally constructed it. For example, the Ethernet driver is responsible for destroying messages after delivering them to upper protocol. Similarly, the top layer protocols are responsible for destroying messages that have been sent out to their destination.

Chapter 4

Design Overview

This chapter describes the design of the protocol architectures proposed in this thesis in the context of the *x*-kernel. We first describe the design of the DAA protocol architecture, followed by the design of the ILLM protocol architecture. We finally present the design of a non-monolithic protocol architecture which allows protocol stack implementation as user level libraries.

4.1 Design of a Framework for LLM Protocol Architecture

The standard *x*-kernel follows an LLM protocol architecture. We used standard TCP/UDP-IP-Ethernet protocol stack implemented in the *x*-kernel and adapted it for our multiprocessor environment. The locking scheme used is described in Chapter 3. Details on standard TCP/UDP-IP-Ethernet stack implementation in the *x*-kernel are given by Hutchinson and Peterson [HP88, HP91].

4.2 Design of a Framework for DAA Protocol Architecture

A number of changes are required in the *x*-kernel architecture to support a protocol independent framework for DAA stack. Our goal has been that these changes be transparent to the protocol writer and compatible with the original *x*-kernel so that existing protocol code need not be modified to run in the new *x*-kernel.

In a DAA protocol stack, demultiplexing is done only at a single layer and, thereafter,

the received packet is simply passed to the appropriate sessions of different protocol layers before being delivered to the application. This requires that the different layer sessions belonging to the same connection be linked together in an upward direction. These sessions also need to be linked downward for sending packets out efficiently. This leads to the creation of a doubly linked session stack called a *path.* A path in the protocol stack corresponds to a unique connection and contains one session from each protocol layer that sits vertically above each other in the protocol hierarchy. We have extended the *x*-kernel to support paths.

4.2.1 Path/Session Stacks

In the traditional *x*-kernel, different layer sessions corresponding to a connection are linked downward by a pointer. However, there is no way to locate an upper layer session from a lower layer session without going through the upper layer protocol object. We provide a way to link the sessions in both directions with an additional pointer, *up_s*, in each session object.

Another requirement of the DAA architecture is to not share a lower layer session among a number of upper layer sessions. Every session on a path should be physically distinct. This requires that more than one session with the same attributes exist together in a map of a protocol layer. In the traditional *x*-kernel, multiple sessions with the same attributes cannot exist in a map. The sessions are stored in a map as internal identifiers. The traditional map manager allows only *one-to-one* correspondence between external identifier and internal identifiers. To support the new architecture, we need to modify map management functions to support a *one-to-many* scheme. We modified the map manager such that it allows more than one internal entry to be stored in the map having same external identifier. While this scheme makes searching through the map difficult, we do not need to search through the maps at every layer in the DAA protocol architecture

as demultiplexing is performed only at the bottommost layer. However, we still need to store sessions in the map for other purposes such as garbage collection, timeouts, etc. Searching through the map is performed only at one layer in the DAA stack, so we only need to guarantee that there is a *one-to-one* relationship between session and external identifier in the map of that layer. We achieve this by including an *Application Identifier* in the external identifier for search through the map. The application identifier must be unique networkwide.

4.2.2 Networkwide Unique Demux Key

The networkwide unique demux key is constructed by using the machine's physical address, a protocol identifier, and an application identifier. The protocol identifier is used to support different protocol stacks (e.g., DAA stack comprising Ethernet and X.25). The simplest way to provide systemwide unique application identifiers is that the application identifier be assigned by a central authority. The protocol that perform the demultiplexing is the obvious choice for this functionality as it is the only protocol that interprets this field. An application may prefer to have a particular application identifier assigned to it so that it can be identified by remote applications with a known unique identifier. This is particularly important for servers. The application identifier can be viewed as equivalent to an IPC port identifier. In our scheme, an application can demand that a particular application identifier be assigned to it in the *open* call. If the application identifier is already in use, the *open* call returns an error. The user specified application identifier is passed in the *open* call and propagated down the protocol stack.

4.2.3 Specifying QoS

QoS parameters can be of two types: *application specific* and *packet specific.* The packet specific QoS service information must be contained in each packet. Examples of such

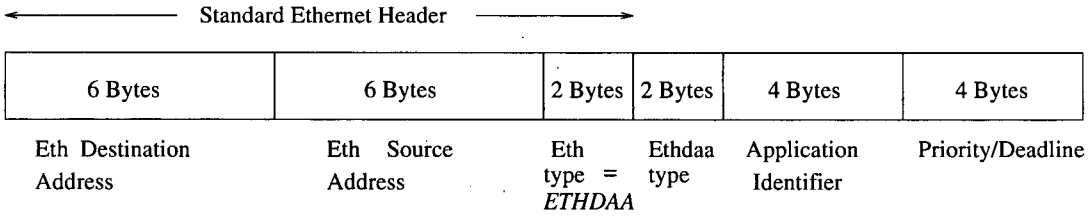

Figure 4.5: Format of Extended Ethernet Header

QoS information include protocol specific deadline and priority. Application specific QoS parameters can be directly linked to the application and are not required to be included in each packet.

For packet specific QoS, the QoS information must be enclosed in the lowest possible protocol layer header. This requires a mechanism to pass QoS information from the upper protocol layers to the bottommost layer. For this purpose, we exploit the notion of *message attributes* provided by the *x*-kernel message library. We have implemented a sample DAA architecture protocol stack consisting of the extended Ethernet, IP, and UDP protocols.

4.2.4 Extended Ethernet Header

An extended Ethernet header is shown in Figure 4.5. Logically, we have added a new protocol layer above the Ethernet protocol. However, the implementation is integrated with the Ethernet layer so that it is more appropriately considered as an extension of the Ethernet protocol. This new protocol is identified as *ETHDAA* in the *eth type* field of the Ethernet header. The new protocol layer contains a 2-byte field *ethdaa type* to identify the upper layer protocol. It may seem that demultiplexing is performed at two layers because two fields specify the upper layer protocols. However, in the implementation, the *eth type* and *ethdaa type* fields are combined together to form a single demux key. Two new 4-byte fields have also been added: an *application identifier* field and a *deadline/priority* field.

As argued in the Section 2.3.1, these fields are essential to obtain the best performance from the DAA architecture protocol stack. The *Application Identifier* field is interpreted as follows: if the Most Significant Bit (MSB) is 0, the *Application Identifier* is assigned by the local machine. If the MSB is 1, the *Application Identifier* has been assigned by the remote machine. When the key for demultiplexing is constructed at the extended Ethernet layer, the MSB is treated as zero. The application may choose how to interpret *deadline/priority* field. For our experiments, the *deadline/priority* field is used as the priority of the thread that shepherds the packet through the protocol stack.

We configure IP and UDP protocols above the extended Ethernet protocol to measure performance. IP and UDP are not well suited for the DAA architecture as some functionality of these protocols that are related to demultiplexing (e.g. UDP port numbers) are not used after the first packet is transferred on a session.

4.2.5 Creation of Session Stack

The DAA stack of sessions can be created either *actively* and *passively* on behalf of the application. In the former case, an application specifies the remote application's coordinates it wants to communicate with. In the passive open case, an application tells the lower layer protocol that it is ready to accept a connection from remote applications.

Active Open

An application initiates the creation of a path by calling the open routine of a protocol. The protocol creates a session and invokes the open routine of the lower layer protocol. This way the open propagates down the protocol stack until the bottommost protocol layer is reached. Each open call returns the session it creates. In the traditional *x*-kernel, the sessions thus created at each protocol layer are linked together in a downward direction by *downv* pointer. In the DAA protocol model, sessions are also linked together

in an upward direction by a *up_s* pointer. Therefore, the active open call leads to the creation of a doubly linked stack of sessions or a path. The open call also propagates the application identifier supplied by the application. The bottommost protocol layer checks whether the application identifier is already in use. If the application identifier is not in use, the session created at the bottommost protocol layer is stored in the map with application identifier as the key. If the application identifier is already in use, the call fails, the sessions created at various protocol layers are destroyed, and an error is returned to the application. If the application does not need any specific application identifier, a unused application identifier is assigned to it.

Passive Open

In case of a passive open, a path is created when the first message is received from the network on a connection. The lowest level protocol layer uses the application identifier field along with the remote machine's physical address and protocol identifier to form the key to demultiplex the packet to a session. If a session is not found, the passive map is searched to find a corresponding *enable* object. If an enable object is found, a session is created and stored in the map using the demultiplexing key. This initiates the creation of a session stack. The message is passed to the upper protocol layer and either a session is created or the message is dropped. The *downv* and *up_s* pointers are initialized at each protocol layer so that the message is shepherded up the protocol stack until it reaches the topmost layer protocol and passed to the application. This completes the creation of a path. If the message is dropped at any layer, the sessions created at the lower layers are destroyed.

4.2.6 Sending Out Packets

Once a path is created, network packets can be sent out and received on that path. Sending out packets is similar to sending out packets in the traditional *x*-kernel. A series of *push* operations starting from the top level session in the path results in sending out a packet.

4.2.7 Processing Incoming Packets

For a received packet, the lowest level protocol demultiplexes the packet to a path (perhaps creates it first). Once the path is found, the packet is passed through the sessions in the path. At each layer, headers are stripped, the connection state is updated, and the packet is passed to the upper layer session pointed to by the *up_s* field of the session object.

The QoS information field is also interpreted by the demultiplexing protocol and acted upon. This allows the implementation to provide guaranteed QoS by acting on the QoS information as soon as the packet is received. For example, let the QoS field of a packet contain the deadline information of the packet. When the packet is received by the lowest level protocol, the protocol can initiate systems scheduler to schedule the processing of the packet so that its deadline is met. If the QoS field contains information related to buffer space or priority, there is no need to waste network bandwidth. These type of QoS information can be associated with the *demux key* at the lowest layer protocol and processed when the *demux key* is formed from the information in the packet.

4.3 Design of a Framework for ILLM Protocol Architecture

In the ILLM protocol framework, only the processing of incoming packets is different from that in the LLM protocol framework (or in the traditional *x*-kernel). Other operations, such as active/passive open, processing outgoing packets, are the same as in the traditional *x*-kernel. Therefore, we describe only the modifications required for processing incoming packets to support the ILLM protocol framework in the *x*-kernel.

4.3.1 Processing Incoming Packets

In the *x*-kernel, a protocol specific *demux* routine performs the demultiplexing operation at each of the protocol layers and returns a session. Normally, a protocol specific *pop* routine updates the state of the session returned by the *demux* routine and is invoked immediately following the *demux* routine. Hence, as a packet visits different protocol layers, the protocol specific *demux* and *pop* routines are invoked alternately at each protocol layer. For example, in the TCP-IP-Ethernet protocol stack, the following sequence of protocol specific routines is invoked: *ethDemux, ethPop, ipDemux, ipPop, tcpDemux, tcpPop.* For an ILLM protocol architecture, the *demux* routines of all protocol layers are invoked first followed by the *pop* routines of all protocol layers. Hence, the following sequence will result for the example given above: *ethDemux, ipDemux, tcpDemux, ethPop, ipPop, tcpPop.* The ILLM framework is implemented by extending the UPI interface of the *x*-kernel. During the demultiplexing phase, a list of the sessions returned by the *demux* routines and the message objects are stored at each protocol layer. The saved information is used during the state-change phase. A protocol independent scheme is designed for this purpose so that it is transparent to protocol writers.

In the ILLM architecture, it is necessary to know when to switch from the demultiplexing phase to the state-change phase. For this purpose, a bit called *upBoundry* is

added to the session objects. This bit indicates the user-protocol boundary. Usually the sessions at the top protocol layer (e.g. tcp, udp sessions) will have this bit set. At every protocol layer, this bit is checked to see if the session is at the user-protocol boundary. If the session is at the user-protocol boundary, the packet processing is switched from demultiplexing phase to state-change phase.

4.4 Design of a Framework for Non-Monolithic Protocol Architecture

This section describes the design of a framework for a Non-Monolithic protocol architecture in the context of the *x*-kernel. The architecture splits the protocols into two separate address spaces: one sits in the kernel or trusted server and the other is linked with each application as a communication protocol library. This decomposition is protocol independent and at the same time it provides a mechanism to take protocol dependent idiosyncrasies into account. This framework also gives the protocol writer the flexibility to decide what to put in kernel and in the user level library. The design of Non-Monolithic protocol framework is motivated by providing a Packet Filter[YBMM94, MJ93, MRA87] mechanism in the x-kernel. Hence, the framework provides a fast path in the kernel to demultiplex the network packets and route them to the destination address space.

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 the kernel and the user is referred to as *IPC.* If the kernel is an operating system kernel, the IPC refers to a system call or upcall. If the kernel is a trusted server, the IPC refers to 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 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 [MRA87], 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 [YBMM94]. 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 Chapter 3. 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 application by having the user register the user session identifiers with the kernel.

4.4.1 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 4.6 shows the protocol graphs of three different users and the corresponding kernel protocol graph. The entire suite of protocols which is available to each user appears in the user's protocol graph even if some of those protocols are actually implemented in the kernel.

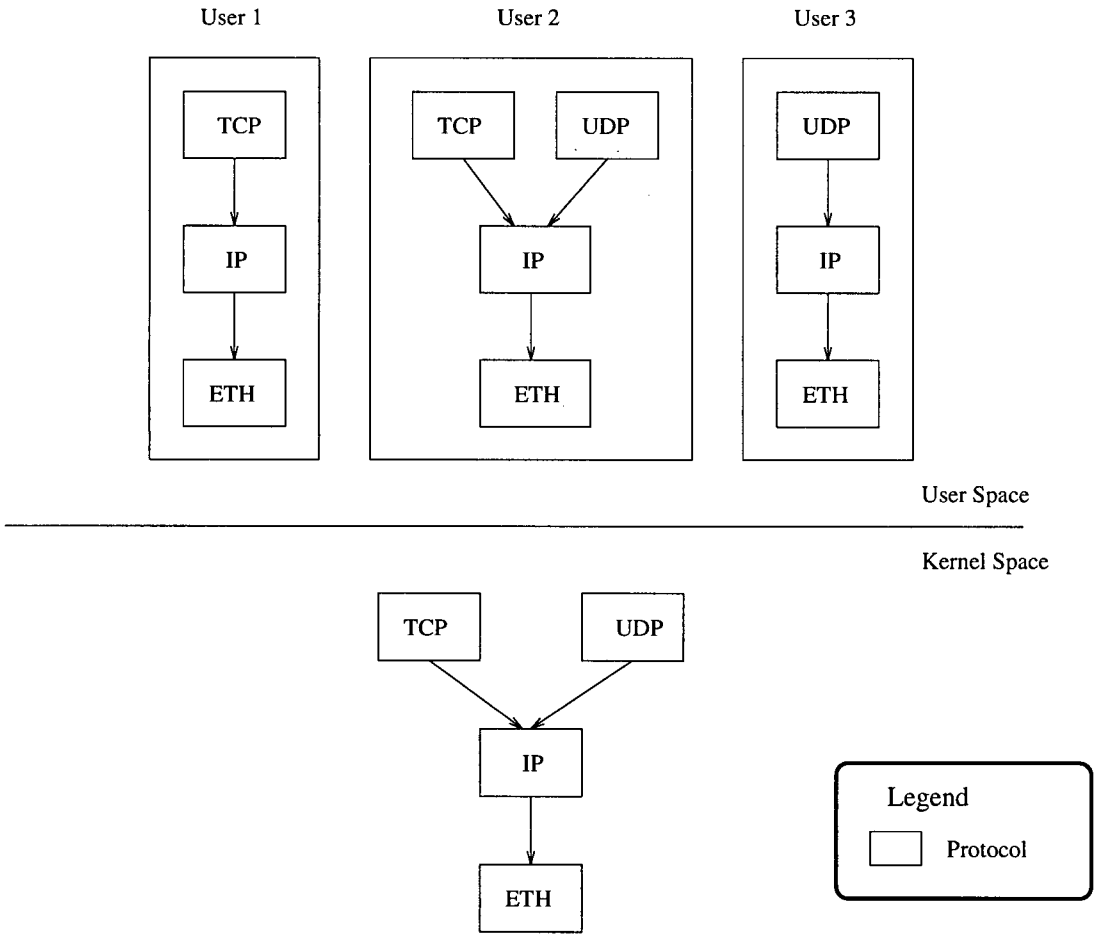

Figure 4.6: User and Kernel Protocol Graphs

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 [YBMM94]).

4.4.2 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. They are called *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.

4.4.3 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, sessions must also be present in the kernel. The kernel level sessions are proxies for their user level counterparts and are called *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 identifier 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. Thus, extra demultiplexing in the user space to locate the appropriate session is avoided.

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.

4.4.4 Demultiplexing in the Kernel

Demultiplexing in the kernel is performed in a similar fashion as it is performed in the traditional *x*-kernel. 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 playback later at the user level.

At each protocol layer in the kernel, 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 the user-kernel boundary protocol contains the user level IPC port number and the user session identifier list. The message is then transferred to the destination user space along with the user session identifier list and the bookkeeping information recorded by the kernel level *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 is the case in a traditional implementation. This is equivalent to combining the demultiplexing operation of multiple protocol layers into a single "one-shot" operation. This integration also makes

it possible to take an Integrated Layer Processing approach to the data manipulation [Fel90, Ten89, CT90].

4.4.5 Packet Processing at User Level

The user level receives the incoming packets along with bookkeeping information and the user session identifier list from the kernel. The connection specific protocol state is updated by invoking a series of *pop* routines and finally the user data is passed 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.

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.

Figures 4.7 and 4.8 depict the receive and send paths for incoming and outgoing messages respectively. In the figures, TCP, UDP, and IP protocols are configured in the user space and have corresponding skeleton protocols in the kernel. The Ethernet [MB76] protocol is configured in the kernel, but a skeleton Ethernet protocol is also configured in the user space to improve the performance of outgoing packets. Figures 4.7 and 4.8 also illustrate that the kernel level sessions that are not at the user-kernel boundary (e.g. IP and Ethernet) may be shared by more than one user level protocol belonging to the same or different applications.

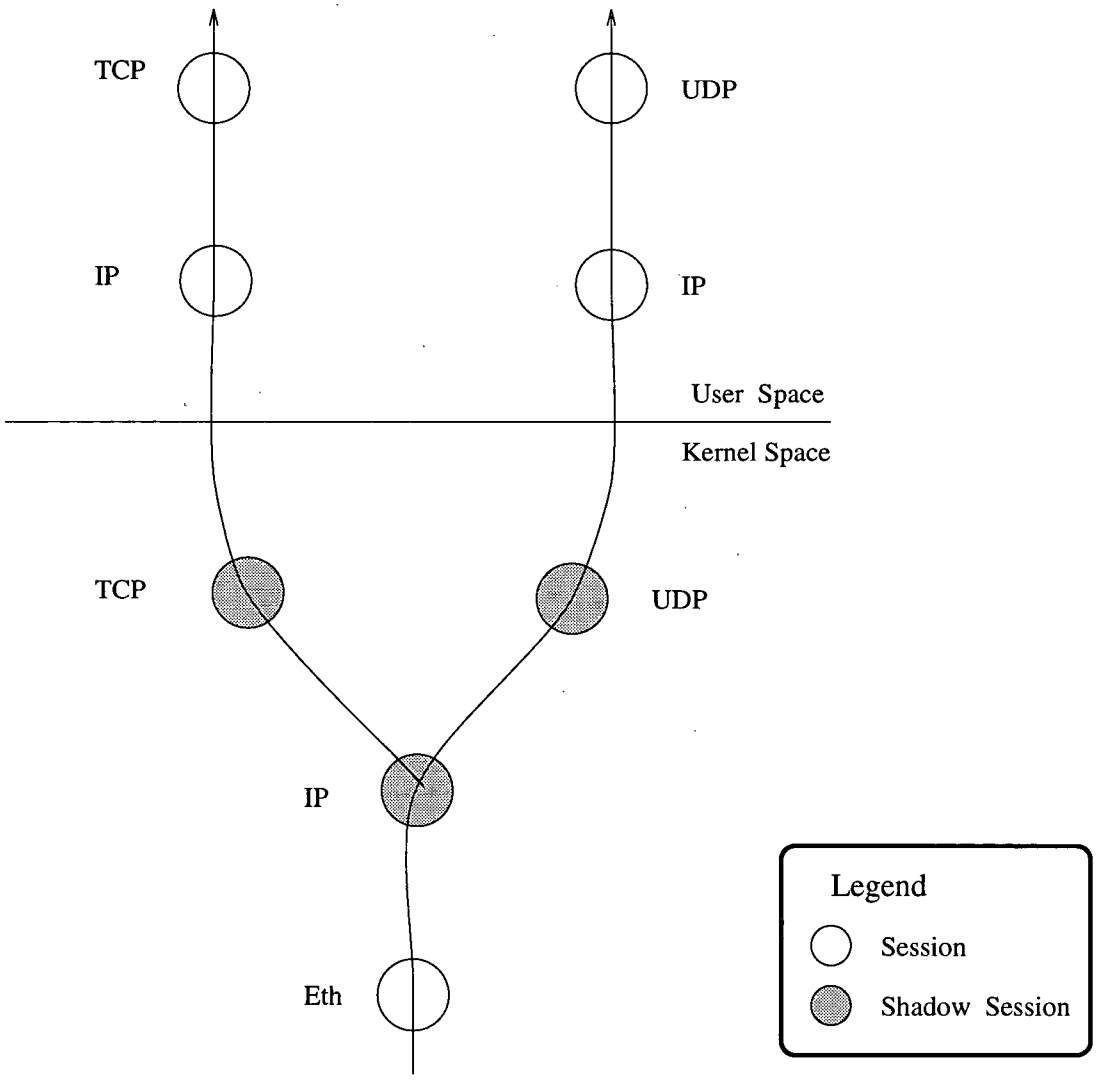

Figure 4.7: Incoming Message Paths

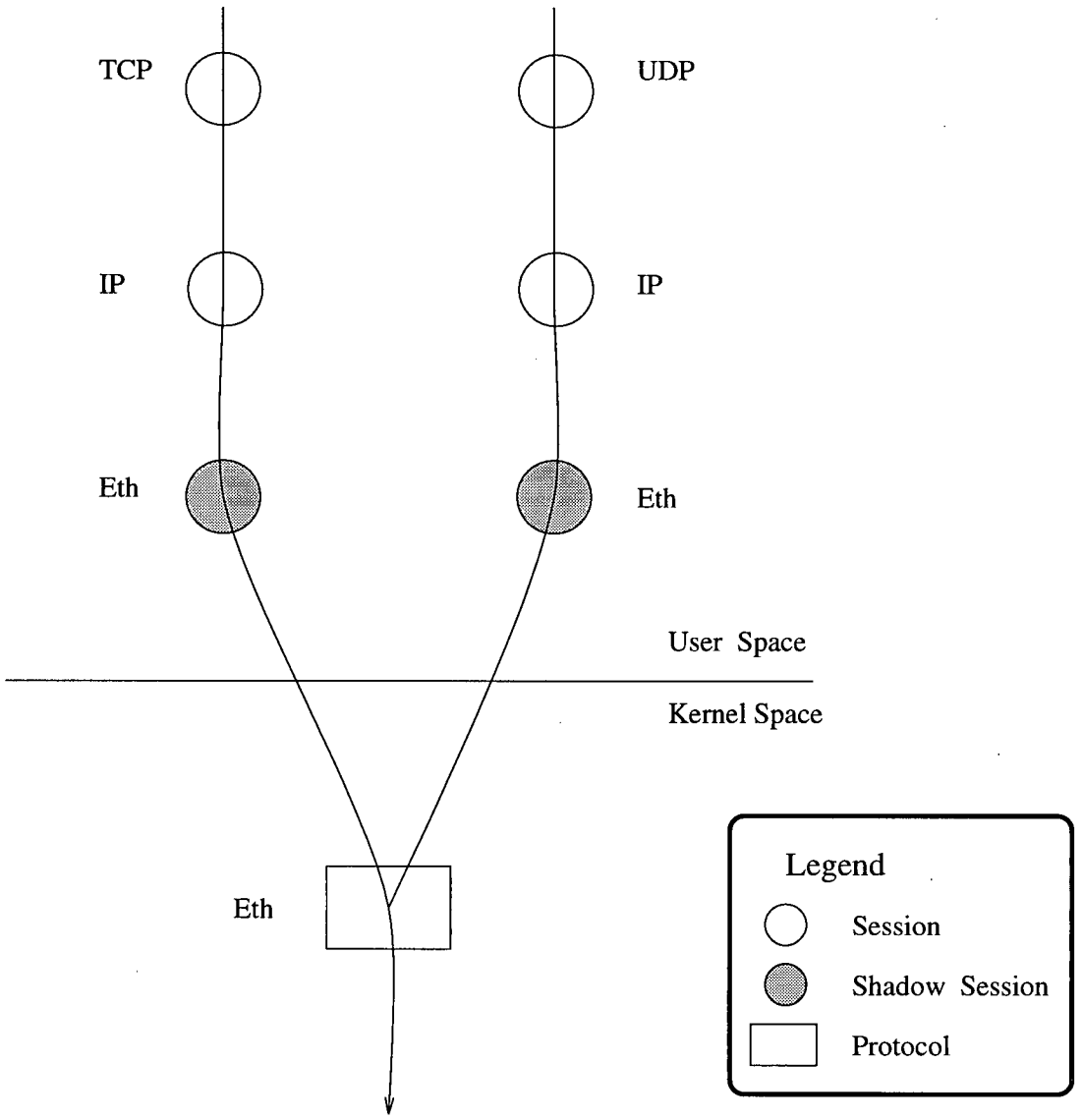

Figure 4.8: Outgoing Message Paths

4.4.6 Shared State

One of the problems that must be addressed in any protocol implementation that allows multiple implementations of protocols is that of shared protocol state. Examples of shared protocol states include IP routing information and ARP tables. Our framework supports a simple but general protocol independent mechanism to manage such information with the assumption that one address space (typically the kernel) maintains the master copy of such information and that other address spaces (typically the user applications) wish to maintain caches of subsets of the information. In this mechanism, an IPC broadcast between the kernel implementation of a protocol and all of the user level implementations of that same protocol is made whenever the shared state changes. This facility is also required to handle IP broadcast and multicast packets.

4.4.7 Details of Skeleton Protocols

Skeleton protocols must manage kernel sessions for demultiplexing incoming network packets. It is possible to provide a set of generic routines such as *open, open_enable, close,* and *open_disable* to perform these connection management functions in the kernel but the *demux* routine is protocol specific and must therefore, be implemented by the protocol writer[1]. 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 that supersede the generic routines whenever necessary.

We describe the generic *open/open_enable* routines to illustrate how the skeleton protocols function, The *close* and *open_disable* routines are similar. The *open/open_enable* routines take a participant list as one of their arguments. The participant list is supplied

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

by the user in the *RegisterOpen/RegisterOpenEnable* IPC calls and is created during the *open/open_enable* call in the user level protocols. As the user level active open call (i.e *open* routine) 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 topmost level protocol has the list of keys each protocol used for identifying its sessions. The topmost layer protocol 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[2]. 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.

4.4.8 Protocol Configuration

Each protocol, session, and enable object contains an additional bit called the *ukBoundaryBit* which specifies whether the object resides at the user-kernel boundary. The user level protocols that are at the bottom of the user level protocol graphs will have this bit set to indicate that there is no lower user level protocol. Similarly, the kernel level shadow sessions at the topmost protocol in the kernel protocol graph have this bit set to indicate that there is no upper kernel level session implying that the packet should be transferred to the user. The shadow session's *state* data structure stores the user level port and user session identifier list.

Similarly, an *apBoundaryBit* is defined for user level objects. This bit indicates whether a user level protocol or session object is at the application-protocol boundary.

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

The application-protocol boundary is defined by the topmost user level protocol registered in the kernel protocol graph for the application (i.e. the protocol hierarchy ends at this boundary for the user protocol stack). The combination of *ukBoundaryBit* and *apBoundaryBit* allows different applications to register different parts of their protocol graphs with the kernel.

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 thereby making it 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).

Chapter 5

Implementation Details

This chapter first describes the run-time environment for our implementation and gives an overview of the Raven kernel that runs on the bare hardware. It then presents the implementation details of the frameworks for each of the three proposed protocol architectures. These frameworks are protocol independent and implemented in the the context of the *x*-kernel. This chapter also describes the implementation details of sample TCP/UDP-IP-Ethernet protocol stacks as implemented in the context of the three proposed architectures.

5.1 Run-time Environment Overview

The hardware platform for our experiments is the Motorola MVME188 Hypermodule, a 25 MHz quad-processor 88100-based shared memory machine [Gro90]. A light weight, multi-threaded kernel called *Raven* [Rit93, RN93] is developed in the Computer Science department at UBC and is the microkernel running on the bare hardware. The three proposed protocol architectures are implemented in the context of a parallel *x*-kernel: the *x*-kernel [HP88, HP91] has been ported as a user level server in this environment and has been parallelized to take full advantage of parallel processing.

The Raven Kernel Overview

The Raven kernel is a lightweight operating system for a shared memory multiprocessor machine. In the Raven kernel environment, several traditional kernel level abstractions

have been implemented at the user level. Such abstractions include thread management, semaphores, device drivers, interprocess communications (*IPC*) [RN93]. Task management, virtual memory management, and low level interrupt dispatching are implemented in supervisor mode. The Raven kernel makes extensive use of shared memory between user/kernel and user/user to avoid kernel mediation. One of the interesting features of the Raven kernel is that interrupts are handled at user level. This facilitates the complete implementation of device drivers in user space, thereby eliminating the costs of moving device data between kernel and user. The user level IPC library provides synchronous send/receive/reply interface as well as an asynchronous send/receive interface. Both kinds of interfaces use shared memory between clients and server and a task signaling facility to avoid kernel mediation as much as possible. However, IPC call semantics are not general. For our purposes, we would have liked to get a pointer to a shared buffer between client and server so that we could fill the buffer ourselves. But the IPC library does not provide such a facility. The IPC library assumes that a contiguous buffer is supplied by the user as an argument to the library function call and the library routine copies the data from the user supplied buffer to the shared buffer area. This results in an extra copy of the data if the user data is not in a contiguous buffer. Such cases are common in communication protocol processing.

5.2 Implementation Details of the LLM Protocol Framework

The standard implementation of the *x*-kernel follows the LLM protocol architecture. The implementation of the standard *x*-kernel release is described in the *x*-kernel manual [IO93] and papers [HP88, HP91]. The standard *x*-kernel is parallelized for our multiprocessor environment. Our sample LLM protocol implementation consists of the parallelized version of the standard TCP/UDP-IP-Ethernet protocol stack which is available in the *x*-kernel

release from the University of Arizona. Details on the standard TCP/UDP-IP-Ethernet stack implementation in the *x*-kernel can be obtained from the *x*-kernel manual [IO93].

5.3 Implementation Details of the DAA Protocol Framework

The sample UDP-IP-Ethernet protocol stack implementation for the DAA architecture is derived from the UDP-IP-Ethernet protocol stack for LLM architecture. The logical relationship between the UDP-IP-Ethernet protocol stack in the LLM model and the UDP-IP-Ethernet protocol stack in the DAA model is depicted in Figure 5.9. However, the two models are implemented in an integrated fashion as depicted in Figure 5.10. The application specifies the protocol model it wishes to use at open time. The `Part` data structure is extended for this purpose. The application includes the desired *application identifier* field in the `Part` data structure at the open time. The default value of this field is zero to imply that the open is on a LLM protocol model[1]. This provides compatibility with the standard protocol code that comes with the standard *x*-kernel release. The standard code need only be recompiled in our environment and it will run in the LLM protocol model without any modification.

If the application identifier field in the `Part` data structure is set to a non-zero value, the open call will result in the creation of a session stack in the DAA protocol model. Every active open on UDP will result in a creation of a new session at every protocol in the UDP-IP-Ethernet stack i.e. a new session is created at UDP, IP and extended Ethernet protocol layer. The application can set the *application identifier* field to a desired value if it wishes to use a specified application identifier. Otherwise, it uses a special symbol *ANY_APPLICATION_IDENTIFIER* in the application identifier field and the system assigns a systemwide unique application identifier. Since only the Ethernet

[1]The standard *x*-kernel follows the LLM protocol model.

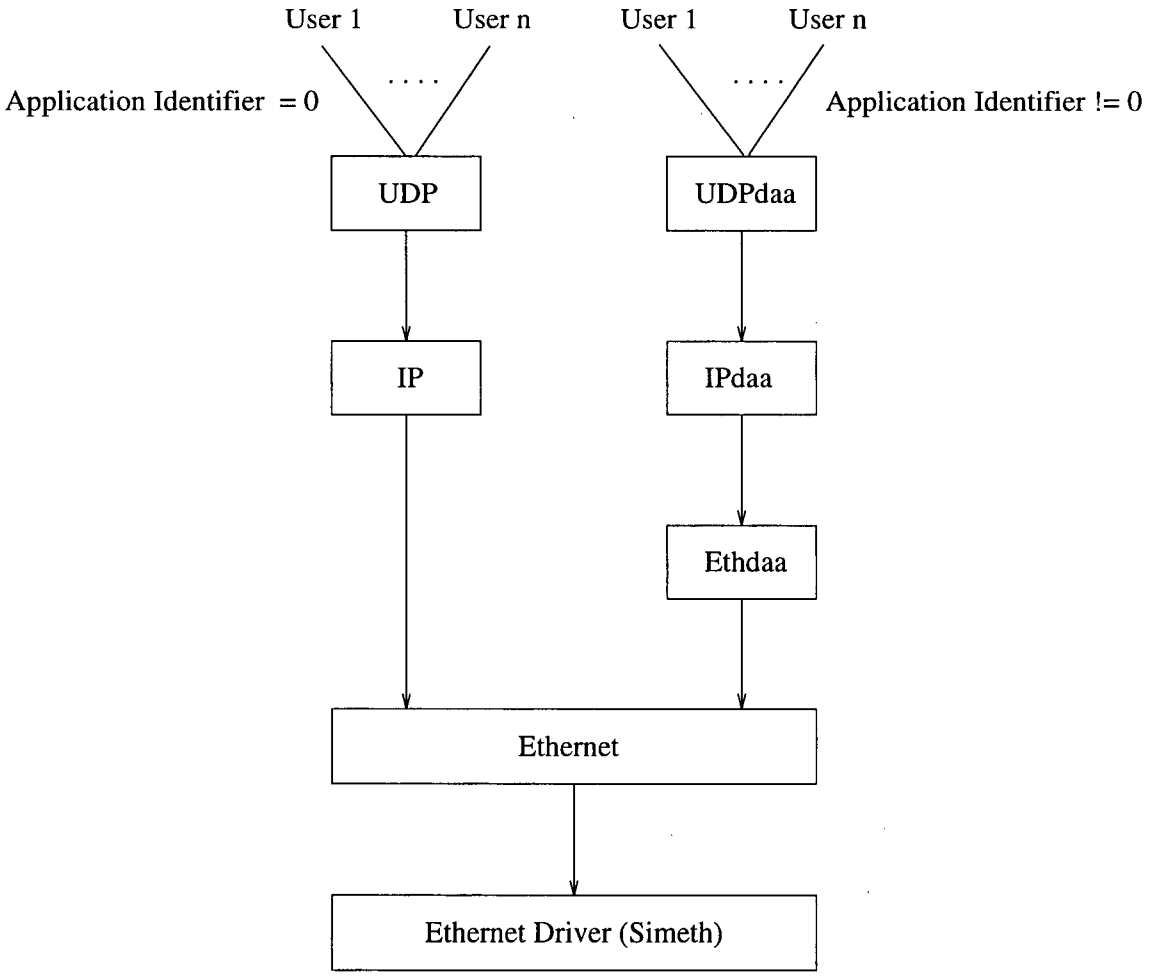

Figure 5.9: Logical Relationship between DAA and LLM Protocol Stacks

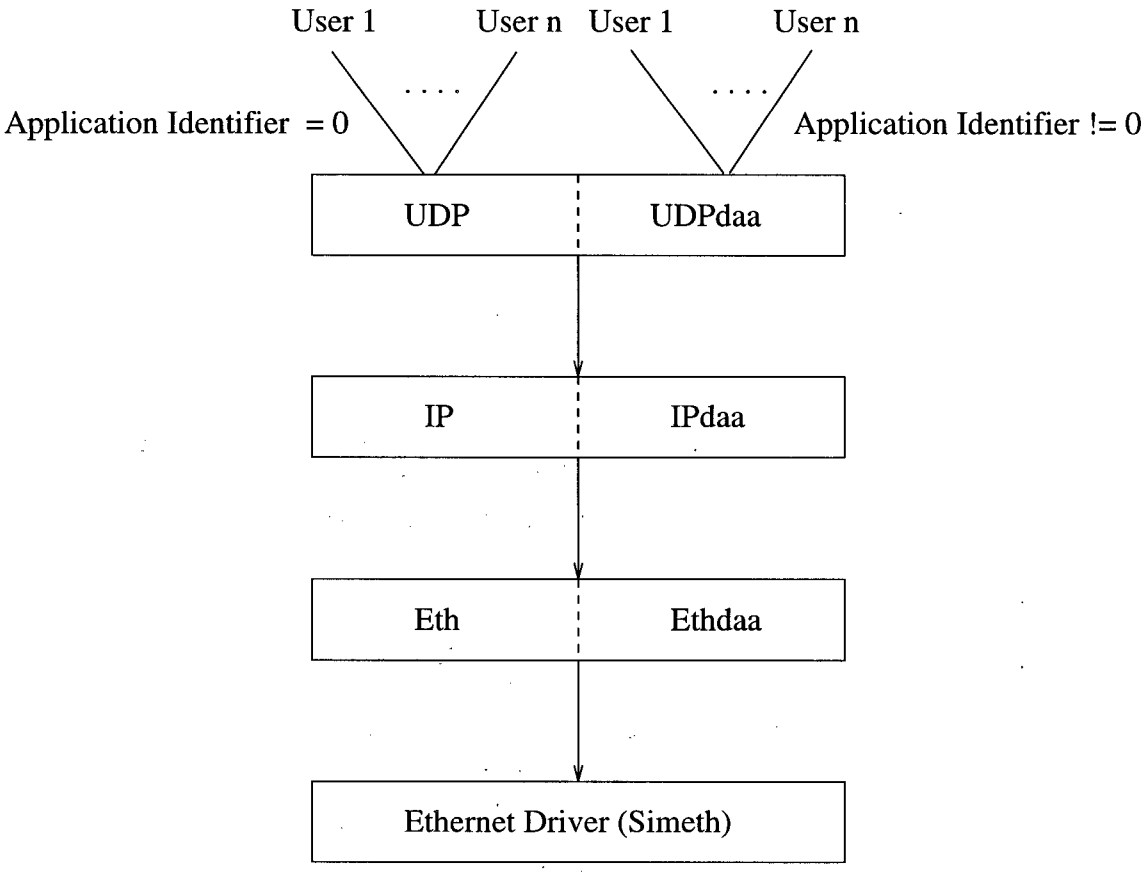

Figure 5.10: Implementation Relationship between DAA and LLM Protocol Stacks

protocol layer needs to interpret the *application identifier* field, the Ethernet protocol layer is responsible for assigning this field.

The `Map` data structure is extended to allow for two kind of maps: traditional *x*-kernel maps that do not allow multiple entries with the same attributes and maps to support DAA architectures which allow multiple entries for the same attributes. A bit field has been added to the `Map` data structure to distinguish between the two kind of maps. The `mapCreate` routine is modified to initialize this bit at creation time. The `mapBind` routine that stores a binding in the map is modified to allow multiple entries

with the same attributes for DAA type maps. Similarly, a bit field is added to the `XObj` data structure to distinguish the protocol/session objects belonging to the LLM protocol model and the DAA protocol model. The `xCreateXobj` routine is modified to initialize this bit field at protocol/session creation time. For all of the new fields added to various data structures, new macros or routines have been defined to set, initialize, and return the values of these fields. Again, to keep the extended *x*-kernel compatible with the old *x*-kernel, all these new fields assume a default value so that no modifications are required for the standard protocol code to run in the LLM model.

UDP, IP, and Ethernet protocol code is modified to support the DAA protocol model. In fact, we have extended the protocol code of these protocols to support both the LLM and DAA models simultaneously. Each protocol now maintains two types of active maps: one for the LLM protocol model and the other for the DAA protocol model. The sessions objects are stored in appropriate maps. The information whether an open is on the LLM or DAA protocol model is readily available from the *application identifier* field of the `Part` data structure which is passed as an argument in the open call. For the passive open, when a session is created from an enable object as a result of a packet arrival for which no session object is found in the active map, the *Eth type* field in the received packet's Ethernet header is used to determine if the packet is a DAA type or LLM type. A special value *ETHDAA* is reserved for the *Eth type* field of the Ethernet header for identifying the DAA type packets. The Ethernet protocol is extended to accomodate the new fields added to support the DAA protocol model.

In our implementation, the *Priority/Deadline* field of the extended Ethernet header is interpreted as the priority of the thread that shepherds the packet through the protocol stack. A scheduling action is taken based on the *Priority/Deadline* field to dynamically change the priority of the thread at the Ethernet protocol layer. The Raven kernel's thread data structure is modified to support dynamic priority changes. Every thread

is assigned a base priority and a current priority. The current priority is the priority at which the thread is running at any instant. For a normal thread, i.e. one which does not require a dynamic change of its priority, the base priority is equal to the current priority. For a thread belonging to the communication thread pool, the current priority is initially set to the base priority. The priority of the thread is changed while shepherding the packets belonging to the DAA protocol model as a result of the scheduling action taken at the Ethernet layer. When the packet has been delivered to the application, the thread's priority is changed back to the base priority. Two routines are defined for changing the priority of the executing thread at run time: `ChangeToCurrentPriority` and `ChangeToBasePrioirty`.

5.4 Implementation Details of the ILLM Protocol Framework

This section describes the implementation details of the framework for the ILLM protocol architecture in the *x*-kernel. As described in Section 4.3, in the ILLM framework, demultiplexing phase is decoupled from the state-change phase and precedes the state-change phase. This requires that the information necessary to perform state-change processing be stored during the demultiplexing phase at each protocol layer. For this purpose, each thread in the *x*-kernel thread pool that is responsible for shepherding a received packet up through the protocol stack is allocated a buffer to store this bookeeping information. This buffer is uniquely associated with each thread and is globally accessible to the code executing in the thread context. The thread control block data structure is extended to accommodate a pointer to this buffer. The buffer is treated to contain the following data structure:

```
typedef struct {
    int      proto_id;                  /* Protocol id */
    XObj     s;                         /* Pointer to this layer session */
    XObj     lls;                       /* Pointer to lower layer session */
```

```
    Msg      msg;                        /* Pointer to this layer message */
    int      hdr_len;                    /* hdr length */
    char     hdr[MAX_HDR_SIZE];          /* hdr itself */
} phdr_t;

typedef struct RecvPacketBuffer {
    int      num_phdr;                   /* header count in phdr array */
    phdr_t   phdr[MAX_PROTOCOL_STACK]; /* Headers information */
    int      cur_phdr_ndx;               /* index into the phdr array */
    int      packet_len;                 /* packet length */
    char     packet[MAX_PACKET_SIZE];  /* packet */
} RecvPacketBuffer_t;
```

The `RecvPacketBuffer` structure stores the received packet in the `packet` data structure. To avoid multiple copies, the thread copies the received packet from the Ethernet board directly into the `packet` field and initializes the `packet_len` field to the length of the received packet. At each protocol layer, the `phdr` data structure is filled with the necessary information so that the state-change processing can be performed later. This information is captured in the `phdr_t` data structure. Pointers to each of the following entities are stored in the `phdr_t` data structure for each protocol layer: protocol layer session, lower layer protocol session, session, and message object. The protocol layer header that is stripped from the received packet during the demultiplexing phase is stored in the `hdr` field of the `phdr_t` data structure by the protocol specific *demux* routine. The `num_phdr` field reflects the count of protocol headers stored in the `phdr` array during the demultiplexing phase. The `protocol_id` field stores the current protocol identifier for any sanity checks that may be performed later during the state-change phase. The `msg` field pointer points to what the protocol layer considers as data in the received packet. As the packet is shepherded through the protocol stack, the `phdr` array is filled at each protocol layer until the packet reaches the user-protocol boundary. This boundary is indicated by the `upBoundary` bit in the `XObj` data structure. At user-protocol boundary protocol, the state-change phase is started.

In the standard *x*-kernel, each protocol's *demux* routine calls the *xPop* routine which, in turn, calls the protocol specific *pop* routine. The protocol specific *pop* routine performs the state-change processing for that protocol. The protocol specific *demux* routine in the ILLM framework is the same as that in the LLM framework except that the former calls *xPopTcb* instead of *xPop*. Figure 5.11 gives a sample implementation of *xPopTcb* routine. The *xPopTcb* routine performs the following operations:

- Stores the sessions object, lower layer session object and message object pointers in the `phdr` buffer.
- Checks to see if *upBoundary* bit of the current session is set. If the bit is not set, it calls the `xDemux` routine with the current session and message objects as arguments. If the bit is set, the routine initiates the protocol state-change processing by extracting the session, lower layer session, and message object pointers of the appropriate protocol from the `RecvPacketBuffer` data structure and invokes the *xPop* routine.

In the standard *x*-kernel, the protocol specific *pop* routine normally calls the *xDemux* routine after it completes the protocol state-change processing. In the ILLM framework, the protocol specific *pop* routine calls the *xDemuxTcb* routine. Figure 5.12 gives a sample implementation of the *xDemuxTcb* routine. The *xDemuxTcb* routine performs the following functions:

- It performs sanity checks to ensure that the various fields of the `RecvPacketBuffer` data structure are valid.
- It extracts the session, lower layer session, and the message pointers from the `RecvPacketBuffer` data structure using `cur_phdr_ndx` as index into the `phdr` array.

```
xkern_return_t xPopTcb( s, lls, msg )
    XObj s;
    XObj lls;
    Msg  *msg;
{
    int return_status;
    RecvPacketBuffer_t *st =
                    (RecvPacketBuffer_t *)(this_thread()->dstate);
    int ndx;

    xAssert(st != NULL);

    /*
     * save msg, s, and lls for pop later;
     */
    ndx = st->num_phdr;
    msgConstructCopy(&(st->phdr[ndx].msg), msg);
    st->phdr[ndx].lls = lls;
    st->phdr[ndx].s = s;
    st->phdr[ndx].proto_id = getProtocolId( s );
    st->num_phdr++;

    if ( s->upBoundary ){

        /*
         * Reached upBoundary. So switch to state-change phase.
         */
        int ndx = st->cur_phdr_ndx;
        Msg *msg = &(st->phdr[ndx].msg);
        XObj s = st->phdr[ndx].s;
        XObj lls = st->phdr[ndx1].lls;

        st->cur_phdr_ndx++;
        return_status = xPop( s, lls, msg );

    } else {
        /*
         * Continue, demultiplexing at upper layer protocol.
         */
        return_status = xDemux( s, msg );
    }
    msgDestroy(&(st->phdr[ndx].msg));

    return return_status;
}
```

Figure 5.11: An Example Implementation of xPopTcb Routine.

- It checks the *upBoundary* bit. If the bit is set, the state-change phase processing is completed for the packet. Therefore, the routine sends the data in the message to the user. If the bit is not set, it calls the *xPop* routine with session, lower layer session, and message as arguments to perform protocol state-change processing.

5.5 Implementation Details of the Non-Monolithic Protocol Framework

5.5.1 IPC Design

In this section, we describe the IPC scheme used for communication between application level communication library and the kernel. The IPC scheme uses shared memory between each application and kernel pair. The use of shared memory also helps reduce the data copying and number of kernel interventions required. To maximize the use of available parallelism, there are two sets of shared buffers - one set for outgoing packets and one set for incoming packets.

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. The following set of IPC calls have been defined:

```
xkern_return_t xDemuxTcb( sesn )
XObj sesn;
{

      RecvPacketBuffer_t *st =
                        (RecvPacketBuffer_t *)(this_thread()->dstate);
      int ndx;
      XObj s, lls;
      Msg *msg;

      xAssert(st != NULL);
      xAssert(st->cur_phdr_ndx <= st->num_phdr);

      ndx = st->cur_phdr_ndx;
      s = st->phdr[ndx].s;
      lls = st->phdr[ndx].lls;
      msg = &(st->phdr[ndx].msg);
      st->cur_phdr_ndx++;

      xAssert(xIsSession(s));
      xAssert(xIsXObj(lls));

      if ( s->upBoundary ){

            /*
             * Send data to the application
             */
            return xDemux(s, msg);
      } else

            /*
             * Do the state-change processing at this protocol layer
             */
            return xPop( s, lls, msg );
}
```

Figure 5.12: An Example Implementation of xDemuxTcb Routine.

RegisterOpen:

An active open on a user level protocol is registered with the kernel using `RegisterOpenReq` IPC call.

```
typedef struct RegOpenReq{
     int      type;                          /* type of IPC Message */
     int      proto_id;                      /* destination protocol id */
     int      uport_id;                      /* user ipc port id */
     int      uhlp_type;                     /* id of hlp_type argument */
     int      uhlp_recv;                     /* id of hlp_recv argument */
     int      part_len;                      /* length of the part array */
     char     part[MAX_PART_LEN];            /* participant list */
     int      num_sessn;                     /* size of valid usessn */
     unsigned long  usessn[MAX_NUM_SESSN]; /* session id list */
} RegOpenReq_t;
```

The `proto_id` field is set to the kernel level protocol identifier to which the IPC call is directed for processing. The `uhlp_type`, `uhlp_recv`, and `part` fields contain the arguments of the `Open` call and they are used by the kernel level protocol to register a user level open. The user level protocol encodes the participants list in the `part` data structure in an externalized form that the kernel can interpret to reconstruct the participant list suitable for the kernel level open routine. This is achieved by using the `partExternalize` routine at the user level and the `partInternalize` routine at the kernel level. The user level session identifier list is stored in the `usessn` array.

The response of the register open call is returned in the `RegOpenRep` IPC call. If the request is successful, the status is set to OK by the kernel. Otherwise, the status is set to the error code indicating the reason of the failure.

```
typedef struct RegOpenRep{
     int      type;                          /* type of IPC Message */
     int      status;                        /* status of the request */
} RegOpenRep_t;
```

RegisterOpenEnable:

The passive open is registered with the kernel using the `RegOpenEnableReq` IPC call. The fields of this call are used in a way similar to the one described for the `RegisterOpen` IPC call.

```
typedef struct RegOpenEnableReq{
     int     type;                          /* type of IPC Message */
     int     proto_id;                      /* destination protocol id */
     int     uport_id;                      /* user ipc port id */
     int     uhlp_type;                     /* id of hlp_type argument */
     int     uhlp_recv;                     /* id of hlp_recv argument */
     int     part_len;                      /* length of the part array */
     char    part[MAX_PART_LEN];            /* participant list */
} RegOpenEnableReq_t;

typedef struct RegOpenEnableRep{
     int     type;                          /* type of IPC Message */
     int     status;                        /* status of the request */
} RegOpenEnableRep_t;
```

DeregisterOpenEnable:

The user level passive open is unregistered with the kernel by using the `DeregOpenEnableReq` IPC call. The contents of the `part`, `uhlp_type`, and `uhlp_Recv` fields must be the same as the ones supplied to `RegisterOpenEnable` at the time a passive open was registered with the kernel.

```
typedef struct DeregOpenEnableReq{
     int     type;                          /* type of IPC Message */
     int     proto_id;                      /* destination protocol id */
     int     uport_id;                      /* user ipc port id */
     int     uhlp_type;                     /* id of hlp_type argument */
     int     uhlp_recv;                     /* id of hlp_recv argument */
     int     part_len;                      /* length of the part array */
     char    part[MAX_PART_LEN];            /* participant list */
} DeregOpenEnableReq_t;

typedef struct DeregOpenEnableRep{
```

```
    int     type;                       /* type of IPC Message */
    int     status;                     /* status of the request */
} DeregOpenEnableRep_t;
```

SendPacket:

The user level sends a packet out on the network through the kernel using the `SendPacket` IPC call. The `buf` field stores the user data along with all the protocol headers that are appended by the protocol processing at the user level. The `proto_id` field is set to the driver identifier that sends the packet over the network. The other fields of this call are self explanatory.

```
typedef struct SendPacketReq{
    int     type;                       /* type of IPC Message */
    int     proto_id;                   /* destination protocol id */
    int     buf_len;                    /* size of the buf */
    char    buf[EMAX_PACKET_SIZE];      /* packet with headers */
} SendPacketReq_t;

typedef struct SendPacketRep{
    int     type;                       /* type of IPC Message */
    int     status;                     /* status of the request */
} SendPacketRep_t;
```

ReceivePacket:

The kernel sends the packets received from the network to the appropriate application using the `RecvPacketReq` IPC call. The `proto_id` field is set to the user level protocol identifier to which the received packet is sent for processing. The `usessn` array stores the list of user level sessions. This list is supplied by the user either at the time of an active open registration or in reply to the first `RecvPacketReq` IPC request i.e. in the passive open case when the first packet is transferred to the user. The `num_sessn` stores the count of the number of the sessions in the `usessn`. Therefore, in the latter case, the `usessn` field is set to zero indicating that the kernel does not yet have the user-level

session identifier list. The `phdr` array stores the bookkeeping information related to the demultiplexing performed at each protocol layer in the kernel. The `buf` data structure stores the complete packet received from the network.

```
typedef struct {
    int      proto_id;                  /* Protocol id */
    int      hdr_len;                   /* hdr length */
    int      hdr_offset;                /* offset from beginning of buf */
} phdr_t;

typedef struct RecvPacketReq {
     int     type;                          /* type of IPC Message */
     int     proto_id;                      /* destination protocol id */
     int      num_sessn;                    /* number of sessions */
     unsigned long usessn[MAX_NUM_SESSN];   /* session ids */
     int      num_phdr;                     /* header count */
     phdr_t   phdr[MAX_NUM_SESSN];          /* Headers information */
     int      buf_len;                      /* packet length */
     char     buf[MAX_PACKET_SIZE];         /* packet */
} RecvPacketReq_t;
```

For performance reasons, each thread belonging to the kernel level demultiplexing thread pool is allocated a buffer to store the `RecvPacketReq` structure. This buffer is globally accessible to the code executing in the thread context. The Ethernet driver copies the received packet in the `buf` array directly from the Ethernet board and initializes the `buf_len` field to the length of the packet.

The user level sends the reply to the `RecvPacketReq` IPC call in the `RecvPacketRep` data structure. The *status* field is set to indicate any error that occurred during the protocol processing at the user level. The user level also registers the user-level session identifier list by filling the list in the `usessn` array and setting the `num_sessn` field to the number of session identifiers in the list.

```
typedef struct RecvPacketRep{
     int      type;                         /* type of IPC Message */
     int      status;                       /* status of the request */
     int      num_sessn;                    /* number of sessions */
```

```
    unsigned long usessn[MAX_NUM_SESSN];  /* session ids */
} RecvPacketRep_t;
```

ProcessControl:

The `ProcessCntlReq` IPC call is used for sending user level control messages to the kernel. Although, we did not find it necessary to send any user level control messages to the kernel in our implementation, it may be required in general. The `cntl_req_id` field is set to indicate the type of control request. The result of the processing of the `ProcessCntlReq` IPC call is sent back to the user level in the `ProcessCntlRep` IPC message. The `status` field indicates the status of the processing of the control message call. This field must be set to `OK` to indicate that the call was successfully processed and that the results are in the `rep_data` field.

```
typedef struct ProcessCntlReq{
    int     type;                       /* type of IPC Message */
    int     proto_id;                   /* destination protocol id */
    int     cntl_req_id;                /* Id of the control request */
} ProcessCntlReq_t;

typedef struct ProcessCntlRep{
    int     status;                      /* status of the request */
    int     rep_len;                     /* length of the reply data */
    char    rep_data[MAX_CNTL_DATA_LEN]; /* reply */
} ProcessCntlRep_t;
```

AssignPort:

The `AssignPortReq` IPC call is currently used by TCP and UDP protocols to obtain the local port number for a connection. Port numbers are assigned by the kernel for correct functioning of user level protocols. The `port_num` field of the `AssignPortReq` call is set to the desired port number by the user and the `proto_id` field is set to the protocol identifier for which the port number is desired. The kernel checks whether the port is

a reserved port or is already in use. If so, the kernel sends a reply indicating an error. Otherwise, the kernel assigns the requested port number and sends the reply. In case, a user does not need a specific port number assigned to it, the user can set `port_num` to *ANY_PORT* and the kernel assigns a unused port.

```
typedef struct AssignPortReq{
     int     type;                          /* type of IPC Message */
     int     proto_id;                      /* destination protocol id */
     int     port_num;                      /* desired port number */
} ProcessCntlReq_t;
```

The kernel sends the results of the `AssignPortReq` IPC call in the `AssignPortRep` structure. The status field is set to indicate status of the results of processing of the IPC request. The `port_num` field contains the port number assigned by the kernel.

```
typedef struct AssignPortRep{
     int     type;                          /* type of IPC Message */
     int     proto_id;                      /* destination protocol id */
     int     status;                        /* return value */
     int     port_num;                      /* assigned port number */
} ProcessCntlRep_t;
```

5.5.2 An Example Non-Monolithic Protocol Implementation

We have implemented the split protocol stack for UDP [Pos80]/TCP [Pos81c], IP [Pos81b], ARP [Plu82], ICMP [Pos81a], and Ethernet [MB76] protocols. UDP, TCP, and most of IP, ARP, ICMP protocols are in the user library while the remaining parts of IP, ARP, and Ethernet protocols are in the kernel. Some IP functions are implemented in the kernel as will be explained later. ARP is implemented at the user level as well as in the kernel. As described in Section 4.4.6, the kernel level ARP is the master and periodically broadcasts the ARP table changes to the user level instances of ARP.

TCP and UDP port assignment must be centralized for correct functioning. In our scheme, kernel is the only central point, 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 the Ethernet header information from the real Ethernet driver located 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 a 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, to 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.

Chapter 6

Performance

In this chapter, we describe the performance of the sample DAA, ILLM, and Non-monolithic protocol stacks implemented in the *x*-kernel and compare it with the performance of a sample LLM 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 stacks for all three architectures. Finally, we present the effect of parallel processing on the round trip latency by using more than one processor.

The performance measurements are made by connecting two Motorola MVME188 Hypermodules through a 10 Mb/sec Ethernet network in light network traffic conditions.

6.1 Benchmark Tools

We analyzed the performance of our implementation in two ways.

- Time measurements 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.

Protocol Stack	*Round-Trip Time(ms)*				
	User Data Size(bytes)				
	1	100	500	1000	1400
Server-Server					
ETH	0.88	0.95	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

Table 6.1: LLM Protocol Stack: User-to-User Latency

Before giving the performance results, we briefly describe the difference in the various implementations. We describe three versions: *Server-Server*, *User-User Server-Stack* and *User-User Partitioned-Stack*. A *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*, 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. Figures 6.13 and 6.14 depict the architectures of User-User Server-Stack and User-User Partitioned-Stack implementations respectively.

6.2 Performance of the LLM Protocol Stack

Tables 6.1 and 6.2 show the round trip times and the incremental cost per protocol respectively for the LLM protocol stack. The latency test used is a simple ping-pong test between two applications called client and server. The client sends data to the server and the server sends the same amount of data back. To eliminate the delay

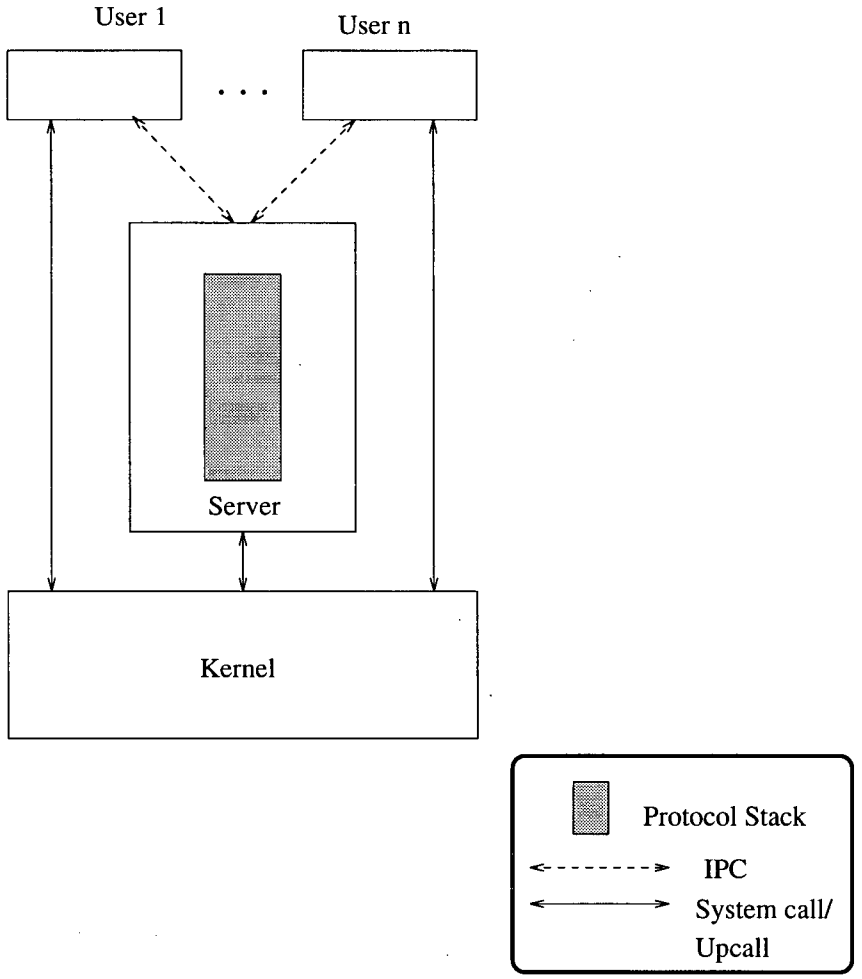

Figure 6.13: User-User Server Stack Architecture

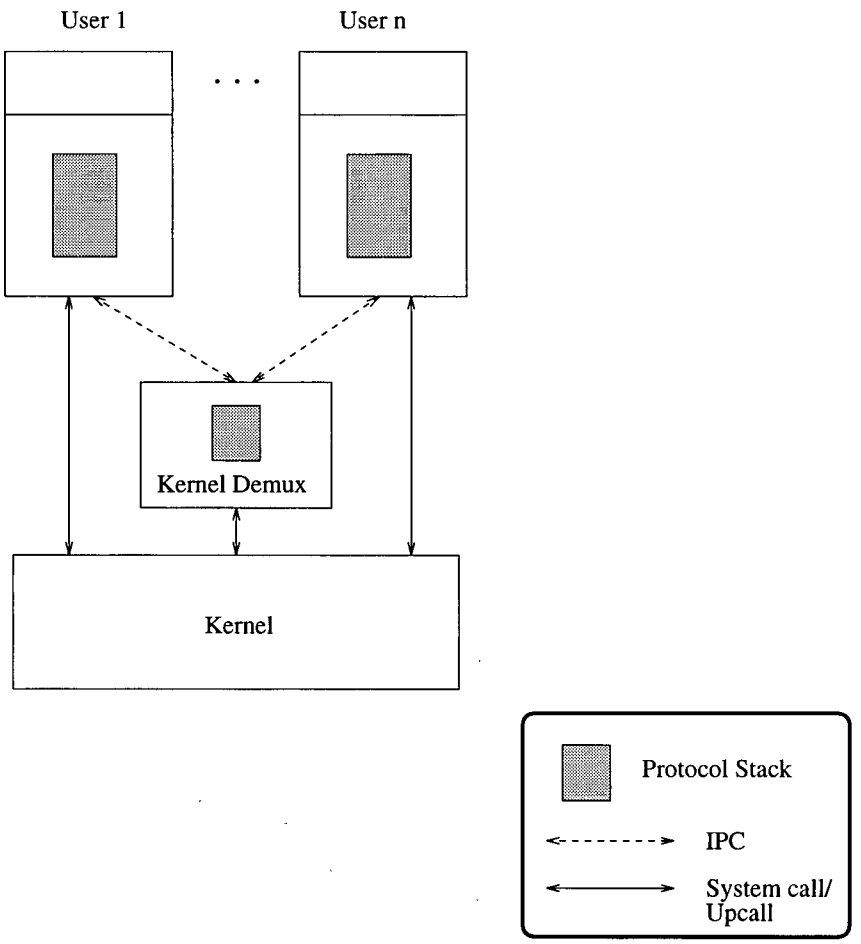

Figure 6.14: User-User Partioned Stack Architecture

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

Table 6.2: LLM Protocol Stack: Incremental Costs per Round Trip (1 byte user data)

variance introduced by the network, the latency figures reported are measured for a single processor averaged over 10,000 transactions. The UDP session is configured to neither compute nor verify the checksum. The incremental cost is calculated by taking the difference between the measured round trip latency for pairs of appropriate protocol stack for 1 byte of user data transfer. For example, UDP latency is computed by subtracting the latency of the IP-ETH stack from the latency of the UDP-IP-ETH stack for 1 byte of user data.

The performance is measured for Server-Server and User-User Server-Stack. The latency difference between the two implementations gives the overhead of IPC between applications and the server.

It can be observed that the incremental cost of each protocol in User-User Server-Stack case is higher than that of the corresponding protocol in Server-Server case. This, we suspect, is due to unfavorable cache performance since two address spaces are involved in User-User Server-Stack case.

6.3 Performance of the DAA and ILLM Protocol Stacks

We measured the round trip time of the sample DAA and ILLM protocol stacks and compare it to that of the corresponding LLM implementation to determine the relative overhead of the protocol architectures. The DAA and ILLM architectures do not

Protocol Stack	*Round-Trip Time(ms)* *User Data Size(bytes)*				
	1	100	500	1000	1400
Server-Server					
ETHDAA	0.92	1.06	2.14	3.47	4.53
IP-ETHDAA	1.07	1.25	2.33	3.70	4.76
UDP-IP-ETHDAA	1.20	1.40	2.49	3.86	4.94
User-User Server-Stack					
ETHDAA	1.90	2.01	3.17	4.58	5.78
IP-ETHDAA	2.09	2.32	3.49	5.01	6.29
UDP-IP-ETHDAA	2.24	2.51	3.68	5.23	6.49

Table 6.3: DAA Protocol Stack: User-to-User Latency

Protocol	*Incremental Costs(ms)*	
	Server-Server	**User-User Server-Stack**
IP	0.15	0.19
UDP	0.13	0.15

Table 6.4: DAA Protocol Stack: Incremental Costs per Round Trip (1 byte user data)

implement ILP optimization. The latency test used is a simple ping-pong test described in Section 6.2. The UDP session is configured to neither compute nor verify the checksum. Tables 6.3 and 6.5 show the round trip times for the DAA and ILLM protocol stacks respectively. Tables 6.4 and 6.6 show the incremental cost per protocol for DAA and ILLM protocol stacks respectively.

6.3.1 Performance Comparison of DAA with LLM

The cost of ETHDAA is higher than that of ETH. This is because ETHDAA has an extended Ethernet header and hence takes more processing time. UDP in the DAA protocol architecture performs marginally better than in the traditional LLM architecture. This performance gain can be attributed to the fact that demultiplexing is performed

Protocol Stack	*Round-Trip Time(ms)* *User Data Size(bytes)*				
	1	100	500	1000	1400
Server-Server					
ETH	0.89	0.95	2.03	3.39	4.47
IP-ETH	1.14	1.29	2.37	3.73	4.80
UDP-IP-ETH	1.27	1.49	2.57	3.89	4.97
User-User Server-Stack					
ETH	1.87	1.94	3.04	4.51	5.73
IP-ETH	2.15	2.36	3.51	5.01	6.34
UDP-IP-ETH	2.31	2.60	3.79	5.27	6.52

Table 6.5: ILLM Protocol Stack: User-to-User Latency

Protocol	*Incremental Costs(ms)*	
	Server-Server	**User-User Server-Stack**
IP	0.25	0.28
UDP	0.13	0.16

Table 6.6: ILLM Protocol Stack: Incremental Costs per Round Trip (1 byte user data)

only at one layer in the DAA protocol architecture. The latency figures of IP and UDP protocols also indicate that the DAA stack performs better when there are more number of layers in the protocol stack.

The incremental costs of IP and UDP in the DAA protocol architecture are smaller than that in the LLM architecture. This is because no demultiplexing is performed for these protocols in the DAA protocol architecture.

6.3.2 Performance Comparison of ILLM with LLM

The ILLM performance is marginally worse than that of the LLM protocol stack. The additional 60 microseconds of UDP latency for 1-byte user data in ILLM can be attributed

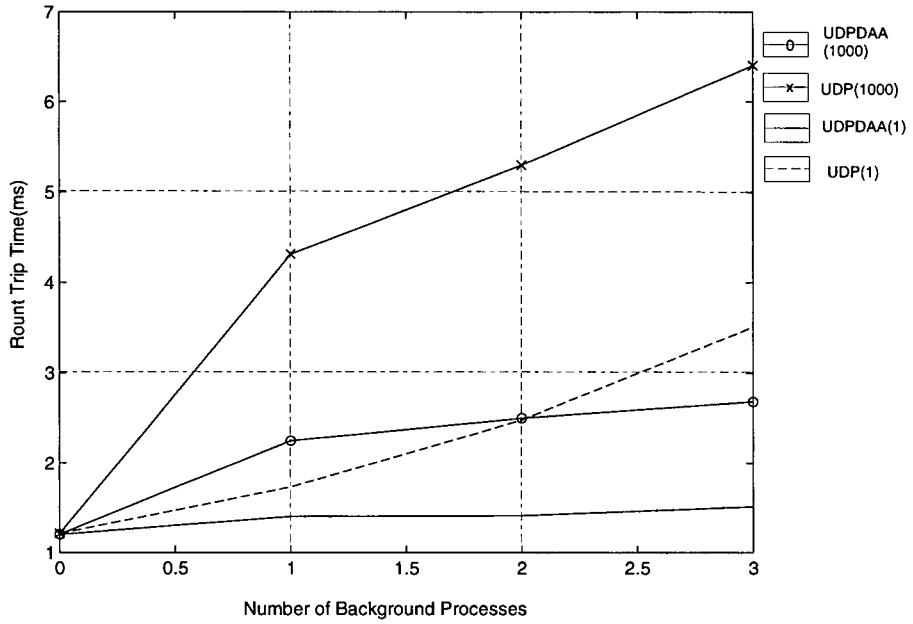

Figure 6.15: Realtime vs. Non-realtime Latency

to saving (or storing) the session and message object pointers during the demultiplexing phase at the Ethernet, IP and UDP layers and reading them again during the state-change phase at each layer. The saving and reading operations require a few instructions and are independent of the protocol complexity. Hence, their overhead is small and is constant for each protocol layer. The performance of the ILLM stack can be improved by appropriate customization. For example, if a protocol does not do any useful processing in the state-change phase (e.g. Ethernet), it is not necessary to save the protocol's session and message objects. For such protocols, the state-change phase can be eliminated altogether. However, the real performance gains in ILLM implementations will come from optimizations such as ILP which ILLM enables.

6.3.3 Real-time Network Traffic

Figure 6.15[1] shows the latency of UDPDAA and UDP stacks for 1-byte of user data in the presence of background UDP network traffic. The background UDP network traffic is increased by running more threads which send data simultaneously. The measurements are made for two data sizes for background network traffic: 1 byte and 1000 bytes. In UDPDAA, the priority field in the ETHDAA header is interpreted as the priority of the thread processing the packet. As can be seen in Figure 6.15, the increase in latency of UDPDAA is much slower than that in UDP as the background network traffic is increased for both data sizes.

6.4 Performance of the Non-Monolithic Protocol Stack

6.4.1 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 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 6.7 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

[1]The number in the brackets indicates the data size of the background network traffic.

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 6.7: 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 6.8: Incremental Costs per Round Trip (1 byte user data)

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 6.8 lists the incremental cost of a round trip of 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 the IP-ETH stack from the

latency for the 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.

Latency Breakdown

Table 6.9 gives the time spent in various protocol layers in the kernel and the user space for both send and receive paths. Tables 6.9 and 6.10 also list the overheads involved in recording and playback of bookkeeping information[2]. The *IP PopTcb* and *TCP/UDP PopTcb* components reflect the cost of recording bookkeeping information while *IP DemuxTcb* and *TCP/UDP DemuxTcb* components 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* component reflects the cost of function calls and *x*-kernel UPI library function calls. The *User IP Preprocessing* component 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 implementation.

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

[2]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.

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 6.9: User-User Partitioned-Stack: Latency Breakdown

Protocol Component	TCP # Instructions	UDP # 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 6.10: User-User Partitioned-Stack: Instruction Counts

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 due to the poor cache performance in the former.

6.4.2 Kernel Level Demultiplexing

The kernel demultiplexing takes between 61 to 98 microseconds. The large variation in the demultiplexing cost is related to cache performance[3]. 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 executed for demultiplexing in the kernel. Table 6.10 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

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

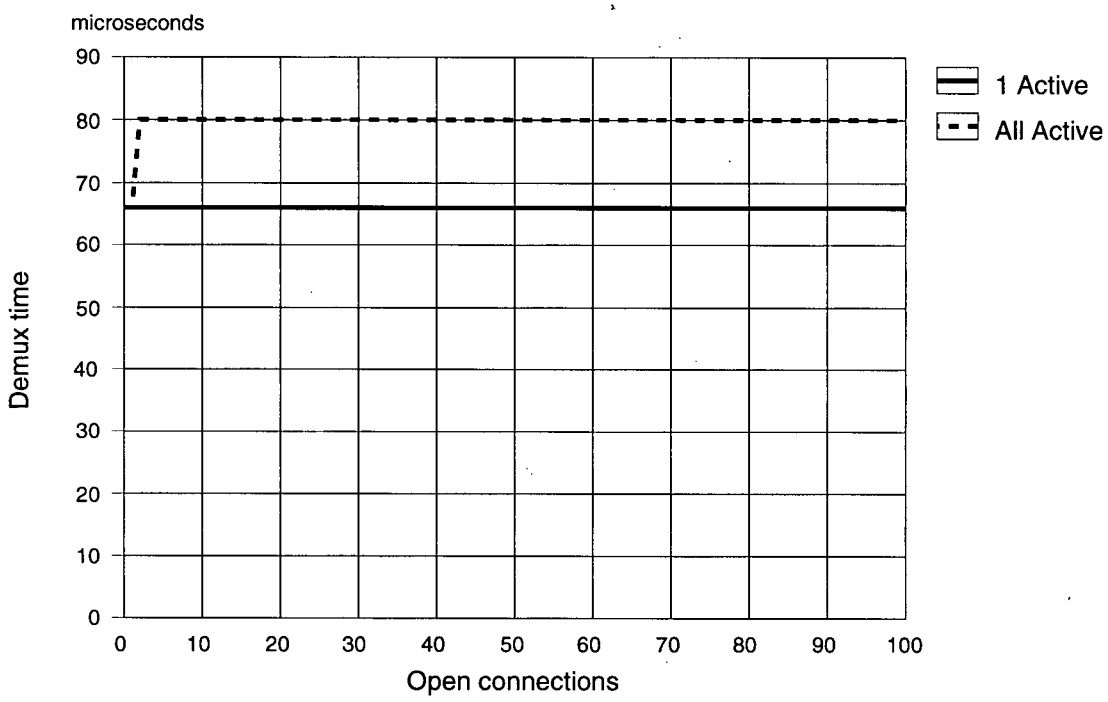

Figure 6.16: Demultiplexing Cost as a Function of Number of Open Connections

buffer shared between the recipient address space and the kernel.

Figure 6.16 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 to only one connection. This is the best case demultiplexing scenario and it 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 the latter, 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

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

Table 6.11: TCP Throughput

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 based implementation [MB93].

6.4.3 User-to-User Throughput

Table 6.11 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, 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.

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

Table 6.12: Connection Setup Cost

6.4.4 TCP Connection Setup Cost

Table 6.12 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 the 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.

6.4.5 Performance on Multiprocessors

Figures 6.17 and 6.18 show the effect of active multiprocessors on the TCP and UDP round trip latencies respectively for all three implementations. An 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 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 we believe that for high-speed networking, shared memory multiprocessor machines will be the most successful.

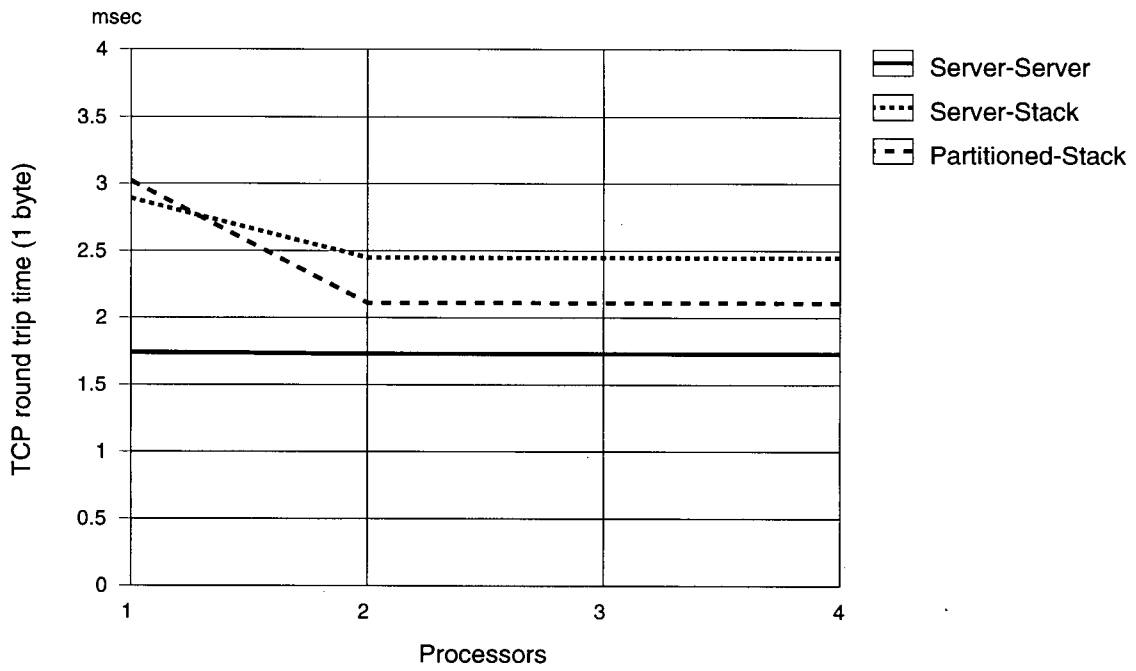

Figure 6.17: TCP Latency as a Function of Number of Processors

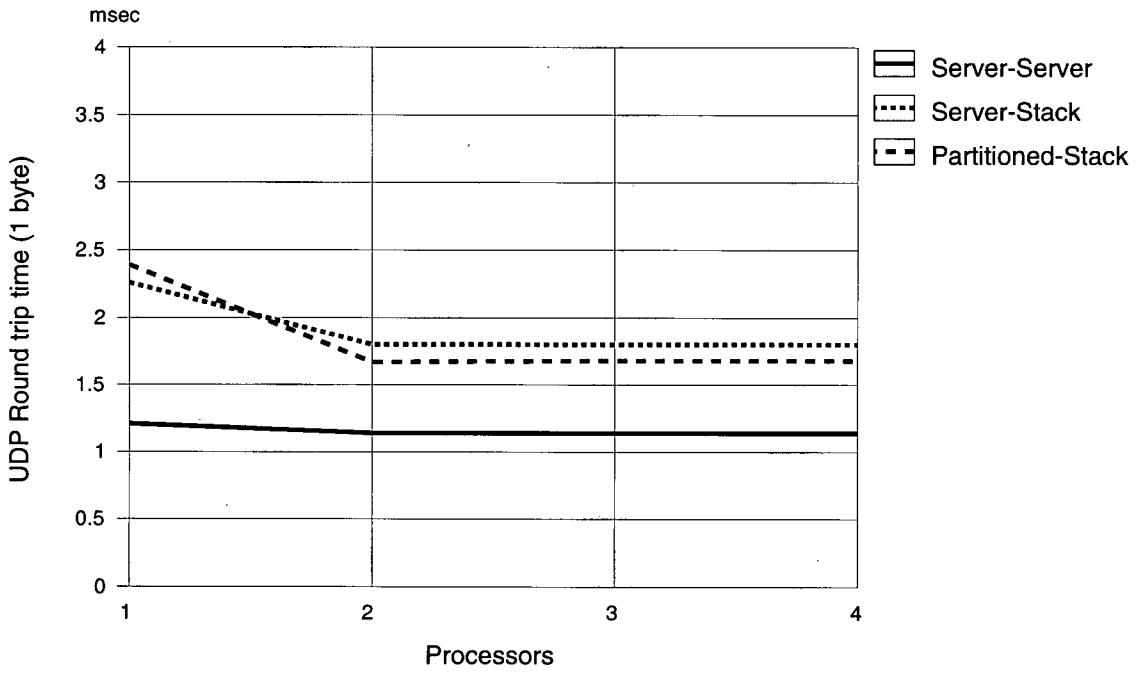

Figure 6.18: UDP Latency as a Function of Number of Processors

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.

Chapter 7

Conclusions and Future Research

In this thesis, three protocol architectures for future high-speed multi-media communication environment have been proposed. This chapter concludes the thesis by presenting a summary of the major results and some future research directions.

This thesis examined the drawbacks of traditional protocol architectures and proposed new protocol architectures which overcome some of the limitations of traditional protocol architectures. These limitations include the following:

- Traditional protocol architectures do not support application specific quality of service due to their use of layered logical multiplexing.
- Traditional protocols do not provide support for real-time guarantees and low latency because there is no provision in the Media Access Control (MAC) layer protocols such as Ethernet and X.25 to accommodate a priority/deadline field.
- New generation multimedia applications require that part of the protocol functionality reside in the applications itself. This goal requires splitting protocol functionality between the kernel and applications on an application basis.
- The performance of traditional protocols has not scaled well with the increase in processor speed because of the increased gap between processor speed and memory speed. ILP has been proposed as a means to minimize data transfer over the memory bus. However, it is difficult to implement traditional protocol stacks using ILP.

- Parallelization of traditional protocols and in particular, that of TCP has been difficult because TCP has a lot of shared state and requires extensive locking.

7.1 Summary of Results

This thesis proposed three new architectures which overcome the above limitations of traditional protocol architectures. These new architectures are DAA, ILLM, and Non-Monolithic.

The DAA protocol model essentially provides low latency and high throughput to applications. In the DAA protocol architecture, an application identifier and the deadline/priority of a message is enclosed in the lowest protocol layer. The application identifier field is the only demultiplexing point in the protocol stack. This field provides a direct channel to the remote application. The priority/deadline field provides proper scheduling of the packet to meet its deadline as soon as the packet is received. The application identifier field can be processed in the network interrupt handler itself, and the data can be copied directly to the application buffer, thereby minimizing the number of data copies. Application resources can also be checked in the interrupt handler and packets can be dropped in the interrupt handler if the application does not have enough resources to receive the packet.

In the ILLM framework, protocol demultiplexing is decoupled from the rest of the protocol functionality and it is performed in an integrated fashion for the complete protocol stack. The rest of the protocol processing for each protocol in the stack is performed after the demultiplexing phase. There are several advantages of the ILLM protocol model over the traditional protocol model. The demultiplexing phase is a read-only operation and constitutes only a small fraction of the total protocol computation. Hence, it can be

performed very fast. Besides, after the demultiplexing phase, the identity of the destination application is known. Hence, the ILLM protocol model essentially achieves all the advantages of the DAA protocol architecture. In addition, the ILLM remains compatible with existing protocols. The ILLM protocol model also facilitates the ILP implementation which is considered essential for high performance protocol processing on modern processors. After the demultiplexing phase, the header-data boundary is known and this eases ILP implementation.

The Non-Monolithic protocol architecture provides a flexible way to decompose protocol services between a trusted kernel address space and user address spaces and gives the protocol implementer the flexibility 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, such as IP reassembly and routing table management, may be added to the kernel implementation. The Non-Monolithic protocol architecture is scalable to thousands of processors. There is no inherent limitation to parallelization because each connection can be allocated its own protocol stack, thereby having no protocol state shared with other connections. In addition, the Non-Monolithic protocol model enables application level framing which is a key requirement of multimedia applications. In multimedia applications, part of protocol processing is required to be performed in the application itself. This model also achieves all the advantages of DAA and ILLM protocol models.

Another contribution of this research is the implementation of protocol independent frameworks for each of the proposed protocol models in the context of the parallel *x*-kernel. The TCP/UDP-IP-Ethernet protocol stack is implemented and parallelized in each of the three frameworks for performance measurements. The same protocol stack is also parallelized in standard LLM protocol architecture for performance comparison

purposes. The performance of these models was measured under real network traffic conditions on two separate machines communicating with each other. Each of the proposed models achieved performance comparable to that of the traditional LLM protocol model without employing any key optimizations such as ILP. In addition, these protocol models achieve all the advantages described above.

Another contribution of this research is the parallelization of the *x*-kernel for shared memory multiprocessor machines. Fine grain parallelism is incorporated in the *x*-kernel to maximize its performance for multithreaded and multiprocessor environments. The TCP/UDP-IP-Ethernet protocol stack is also parallelized.

7.2 Future Research Directions

There are a number of important issues that are left unaddressed in the thesis. They are listed as follows:

- Implementation of a sample protocol stack in each of the three protocol architectures using Integrated Layer Processing optimization and demonstrating the performance gains for each of the three protocol models.
- Experimentation with Application Level Framing using the Non-Monolithic protocol architecture for sample multimedia applications.
- Demonstration of the scalability of the Non-Monolithic protocol architecture on larger multiprocessor machines.
- Implementation of the demultiplexing operation in ILLM and Non-Monolithic protocol architectures using wait-free data structures. This will parallelize the demultiplexing phase totally, thereby giving better performance than that of the LLM protocol model under the same conditions.

Bibliography

[AABYB89] H. Abu-Amara, T. Brazilai, Y. Yemini, and T. Balraj. PSi: a silicon compiler for very fast protocol processing. In *Proceedings of First IFIP Workshop on Protocols for High-Speed Networks*, May 1989.

[AP93] M. Abbot and Larry L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1993.

[BB92] J.C. Brusrtoloni and Brian N. Bershad. Simple protocol processing for high-bandwidth low-latency networking. Cmu-cs-93-132, Carnegie-Mellon University, September 1992.

[Ber91] Brian N. Bershad. Practical considerations for lock-free concurrent objects. Cmu-cs-91-183, Carnegie-Mellon University, September 1991.

[BG93] 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.

[BN84] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BOP94] L.S. Brakmo, S.W. O'Malley, and L.L. Peterson. TCP vegas: New techniques for congestion detection and avoidance. In *Proceedings of ACM SIGCOMM'94 Conference on Communications, Architecture and Protocols*, September 1994.

[BPP92] M.L. Bailey, M.A. Pagels, and Larry L. Peterson. The x-Chip: An experiment in hardware multiplexing. In *Proceedings of IEEE Workshop on the Architecture and Implementation of High Performance Communications Subsystems*, Feb 1992.

[Cal92] R. Callan. *TCP and UDP with Bigger Addresses (TUBA), a simple proposal for internet addressing and routing.* Request For Comments 1347, June 1992.

[Che86] D. Cheriton. VMTP: a transport protocol for the next generation of computer systems. In *Proceedings of ACM SIGCOMM'86 Conference on Communications, Architecture and Protocols*, August 1986.

[Che88] Greg Chesson. XTP/PE overview. In *Proceedings of the 13th conference on local computer networks*, pages 292–296. IEEE Computer Society Press, Silver Spring, MD, October 1988.

[CJRS89] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications*, 27(6):23–29, September 1989.

[CLZ87] David D. Clark, M.L. Lambert, and L. Zhang. NETBLT: A high throughput transport protocol. In *Proceedings of ACM SIGCOMM'87 Conference on Communications, Architecture and Protocols*, pages 353–359, August 1987.

[CSBH91] D. Clark, V. Serf, R. Braden, and R. Hobby. *Towards the Future Internet Architecture.* Request For Comments 1287, December 1991.

[CSSZ90] E.C. Cooper, P.A. Steenkiste, R.D. Sansom, and B.D. Zill. Protocol implementation of the nector communication processor. In *Proceedings of ACM SIGCOMM'90 Conference on Communications, Architecture and Protocols*, September 1990.

[CT90] David D. Clark and David L. Tennenhouse. Architectural considerations for new generation of protocols. In *Proceedings of ACM SIGCOMM'90 Conference on Communications, Architecture and Protocols*, pages 200–208, September 1990.

[Dee92] S.E. Deering. *Simple Internet Protocol(SIP) Specifications.* Internet Draft, November 1992.

[Dov90] Ken Dove. A high capacity TCP/IP in parallel streams. In *Proceedings of UKUUG Summer 1990*, pages 233–235, July 1990.

[DWB+93] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner: A network-independent card provides architectural support for high-performance protocols. *IEEE Network*, 7(4):36–43, july 1993.

[Fel90] 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.

[Fel93] David C. Feldmeier. An overview of the TP++ transport protocol project. In Ahmed Tantawy, editor, *High Performance Communication.* Kluwer Academic Publishers, 1993.

[Fra93] Paul Francis. A near-term architecture for deploying PIP. *IEEE Networks*, 7(3):30–37, May 1993.

[fSIPSOSI87] International Organization for Standardization - Information Processing Systems - Open Systems Interconnection. *Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).* International Standard Number 8825, ISO, Switzerland, May 1987.

[GDFR90] 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.

[GKWW89] Dario Giarrizzo, Matthias Kaiserswerth, Thomas Wicki, and Robin C. Williamson. High-speed parallel protocol implementation. In *Proceedings of First IFIP Workshop on Protocols for High-Speed Networks*, pages 165–180. H. Rudin editor, North Holland Publishers, May 1989.

[GNI92] Murray W. Goldberg, Gerald W. Neufeld, and Mabo R. Ito. A parallel approach to OSI connection-oriented protocols. In *Proceedings of third IFIP Workshop on Protocols for High-Speed Networks*, May 1992.

[Gro90] Motorola Computer Group. *MVME188 VMEmodule RISC Microcomputer User's Manual.* Motorola, 1990.

[Her90] Maurice Herlihy. A methodology for implementing highly concurrent data structures. *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 1990.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages*, January 1991.

[HMPT89] 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.

[HP88] Norman C. Hutchinson and Larry L. Peterson. Design of the *x*-kernel. In *Proceedings of ACM SIGCOMM '88*, pages 65 – 75, 1988.

[HP91] 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.

[Hut92] Norman C. Hutchinson. Protocols vs. parallelism. *Proceedings of the 1992 x-kernel Workshop*, 1992.

[Inc87] Sun Microsystems Inc. *XDR: External Data Representation Standard.* Request For Comments 1014, Network Information Center, SRI International, June 1987.

[IO93] Edwin F. Menze III and Hilary K. Orman. x-*Kernel Programmer's Manual (Version 3.2)*. Department of Computer Science, University of Arizona, Tucson, Feb 1993.

[Jac88] Van Jacobson. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM'88 Conference on Communications, Architecture and Protocols*, pages 314–329, August 1988.

[Jac90] V. Jacobson. 4.3 BSD header prediction. *ACM SIGCOMM Computer Communication Reviews*, 20(2):13–15, 1990.

[JBB92] V. Jacobson, R. Braden, and D. Borman. *TCP Extensions for High-Performance.* Request For Comments 1326, Cray Research, May 1992.

[JHC94a] Parag K. Jain, Norman C. Hutchinson, and Samuel T. Chanson. A framework for the non-monolithic implementation of protocols in the *x*-kernel. In *Proceedings of the 1994 USENIX Symposium on High Speed Networking*, pages 13–30, August 1994.

[JHC94b] Parag K. Jain, Norman C. Hutchinson, and Samuel T. Chanson. Protocols architectures for delivering application specific quality of service. Submitted for publication, 1994.

[JSB90] Niraj Jain, Mischa Schwartz, and Theodore R. Bashkow. Transport protocols processing at GBPS rates. In *Proceedings of ACM SIGCOMM'90 Conference on Communications, Architecture and Protocols*, pages 188–199, September 1990.

[KC88] H. Kanakia and D. Cheriton. The VMP network adapter board (NAB): High-performance network communication for multiprocessors. In *Proceedings of ACM SIGCOMM'88 Conference on Communications, Architecture and Protocols*, August 1988.

[KP93a] Jonathan Kay and Joseph Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *Proceedings of ACM SIGCOMM'93 Conference on Communications, Architecture and Protocols*, pages 259–268, September 1993.

[KP93b] Jonathan Kay and Joseph Pasquale. Measurement, analysis, and improvement of UDP/IP throughput for the DECstation 5000. In *Proceedings of the Winter USENIX Conference*, pages 249–258, January 1993.

[LMKQ89] S. J. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating Systems.* Addison-Wesley, 1989.

[MB76] 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.

[MB92] C. Maeda and Brian N. Bershad. Networking performance for microkernels. In *Proceedings of the fourth WWOS*, 1992.

[MB93] C. Maeda and Brian N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993.

[MCS91] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *PROC of the Fourth ASPLOS*, pages 269–278, Santa Clara, CA, 8-11 April 1991. In *CAN 19*:2, *OSR 25* (special issue), and *ACM SIGPLAN Notices 26*:4.

[MG94] Kjersti Moldeklev and Per Gunningberg. Deadlock situations in TCP over ATM. In *Proceedings of Fourth IFIP Workshop on Protocols for High-Speed Networks*. H. Rudin editor, North Holland Publishers, August 1994.

[MJ93] 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.

[MP91] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Department of Computer Science, Columbia University, February 1991.

[MRA87] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: A new architecture for user-level network code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 39–51, November 1987.

[NGB+91] A. Nicholson, J. Golio, D.A. Borman, J. Young, and W. Roiger. High speed networking at Cray research. *ACM SIGCOMM Computer Communication Reviews*, 21(1):99–110, 1991.

[NIG+93] G. Neufeld, M.R. Ito, M. Goldberg, M.J. McCutcheon, and S. Ritchie. Parallel host interface for an ATM network. *IEEE Network*, 7(4):24–34, July 1993.

[Par90] G.M. Parulkar. The next generation of internetworking. *ACM SIGCOMM Computer Communications Reviews*, 20(1):18–43, 1990.

[Par93] Craig Partridge. Protocols for high-speed networks: some questions and a few answers. *Computer Networks and ISDN Systems*, 25:1819–1028, 1993.

[PDP93] M.A. Pagels, P. Druschel, and Larry L. Peterson. cache and TLB effectiveness in the processing of network data. Technical Report 93-4, Department of Computer Science, University of Arizona, Tucson, 1993.

[Pet94] Larry L. Peterson. Resource allocation in the *x*-kernel. Personal Communication, Department of Computer Science, University of Arizona, Tucson., 1994.

[Pit87] 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.

[Plu82] D. Plummer. *An Ethernet Address Resolution Protocol.* Request For Comments 826, DCP@MIT-MC, November 1982.

[Pos80] J. B. Postel. *User Datagram Protocol.* Request For Comments 768, USC Information Science Institute, Marina Del Ray, CA., August 1980.

[Pos81a] J. B. Postel. *Internet Control Message Protocol.* Request For Comments 792, USC Information Science Institute, Marina Del Ray, CA., September 1981.

[Pos81b] J. B. Postel. *Internet Protocol.* Request For Comments 791, USC Information Science Institute, Marina Del Ray, CA., September 1981.

[Pos81c] J. B. Postel. *Transmission Control Protocol.* Request For Comments 793, USC Information Science Institute, Marina Del Ray, CA., September 1981.

[PP93] C. Papdopoulos and G.M. Parulkar. Experimental evaluation of SUNOS IPC and TCP/IP protocol implementation. *IEEE/ACM Transactions on Networking*, 1(2):199–216, april 1993.

[PS93] T.F. Porta and M. Schwartz. A feature-rich transport protocol: functionality and performance. *IEEE Journal on Selected Areas in Communications*, 11, May 1993.

[PT89] G.M. Parulkar and J.S. Turner. Towards a framework for high speed communication in a hetrogeneous networking environment. *IEEE*, pages 665–667, 1989.

[Rit93] 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.

[RN93] 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.

[SBB+91] M.D. Schroeder, A.D. Birrell, M. Burrows, H. Murray, R.M. Needham, T.L. Rodeheffer, E.H. Satterthwaite, and C.P. Thacker. Autonet: A high-speed, self configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318–1335, October 1991.

[Ten89] David L. Tennenhouse. Layered multiplexing considered harmful. In *Proceedings of First IFIP Workshop on Protocols for High-Speed Networks.* H. Rudin editor, North Holland Publishers, May 1989.

[TNML93] 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.

[Top90] C. Topolcic. *Experimental Internet Stream Protocol, Version 2 (ST-II).* Request For Comments 1190, CIP Working Group, October 1990.

[TS91] B.C. Traw and J. Smith. A high-performance host interface for ATM networks. In *Proceedings of ACM SIGCOMM'91 Conference on Communications, Architecture and Protocols*, pages 317–325, September 1991.

[Tsc91] C. Tschudin. Flexible protocol stacks. In *Proceedings of ACM SIGCOMM'91 Conference on Communications, Architecture and Protocols*, pages 197–204, September 1991.

[WF89] C.M. Woodside and R.G. Franks. A comparison of some software architectures for parallel execution of protocols. Telecommunications Research Institute of Ontario, Department of Systems and Computer Engineering, Carleton University, Ottawa., July 1989.

[WVT93] A. Wolman, G. Voelker, and C.A. Thekkath. Latency analysis of TCP on an ATM network. Technical report, University of Washington, 1993.

[YBMM94] 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.

[Zhu92] Wenjing Zhu. Effect of layered logical multiplexing protocol architecture. Personal Communication, Department of Computer Science, University of British Columbia, 1992.

[Zit89] Martina Zitterbart. High-speed protocol implementation based on a multiprocessor - architecture. In *Proceedings of First IFIP Workshop on Protocols for High-Speed Networks*, pages 151–163. H. Rudin editor, North Holland Publishers, May 1989.