Protocol Architectures for Delivering Application Specific Quality of Service

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

Abstract

With the advent of high speed networks, the future communication environment is expected to comprise 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 which 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.

In this paper, we analyze the drawbacks of traditional protocol architectures and propose two alternative architectures for the next generation high-speed network environment: Direct Application Association and Integrated Layered Logical Multiplexing. We implement sample protocol stacks for each of the above models using the most widely used protocol stack (TCP/UDP-IP-Ethernet) in the context of the x-kernel. The performance of these protocol architectures is shown to be comparable to that of a traditional protocol architecture. They win by enabling the key requirements (Application specific Quality of Service) and optimizations (Integrated Layer Processing) necessary for future communication environments.

1 Introduction

Computer technology has changed rapidly during the last two decades. Processor speed has increased from a fraction of a 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, multimedia news, home banking, and 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 the 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 a diverse communication network.

This paper proposes two alternative protocol architectures for future high-speed network environments. We first analyze the traditional protocol architectures and identify their drawbacks, and then propose new architectures which can 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.

Organization of this Paper

The rest of the paper is organized as follows: Section 2 describes our motivation for exploring alternative protocol architectures and surveys previous work in this area. Section 3 presents the proposed alternative protocol architectures and Section 4 describes the design of these architectures in the context of the *x*-kernel. Section 5 analyzes the overhead of the im-

*Parag Jain is currently with the Bell Northern Research, Ottawa, Canada.

†Samuel Chanson is on leave at the Hong Kong University of Science and Technology.

0-8186-7216-1/95 $04.00

plementations of each of the proposed models using the UDP-IP-Ethernet protocol stack. Finally, Section 6 offers some conclusions and future directions.

2 Motivation and Previous work

Traditional protocol architectures have been quite successful in meeting the demands of the previous generation networks because networks were quite slow and were used to carry time insensitive data. Future communication protocols must support multimedia applications that transfer a variety of data (e.g. voice, video, graphics, text) and have different access patterns such as interactive, bulk transfer, and real time performance guarantees. Widely varying data characteristics mixed with various access patterns require a wide range of *Qualities of Service (QoS)* that are not addressed by present protocol architectures.

Clark, Tennenhouse and Feldmeier [3, 4, 21] have analyzed traditional protocol architectures, and identified the key idiosyncrasies that present bottlenecks to future high-speed network environments. We briefly discuss the shortcomings of traditional protocol structures in the context of two key requirements for future network environments: Quality of Service and Integrated Layer Processing.

2.1 Providing Wide Range of Qualities of Services to Applications

Multimedia applications 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 [4, 21]:

- 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 above discussion, 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 that contains the demultiplexing information.

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

2.2 Integrated Layer Processing

Traditionally, network protocols are implemented in a layered fashion. Open Systems Interconnection (OSI) 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[1]. 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[2] [3]. The data manipulation functions suffer from serious performance problems on modern processors because

[1] 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.

[2] Presentation Conversion refers to the reformatting of data into a common or external data representation such as Sun XDR [11] or ASN.1 [5].

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 processing migrates 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. This approach is termed *Integrated Layer Processing (ILP)* [3]. 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. Abbot and Peterson [1] propose an ILP scheme in the context of the x-kernel and report substantial performance gains for data manipulation operations by employing the ILP technique.

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 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 [12] 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 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 paper proposes an efficient protocol architecture called *Integrated Layered Logical Multiplexing* (in Section 3.3) which overcomes these drawbacks.

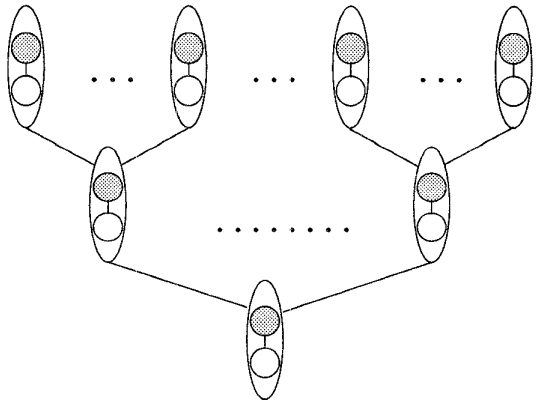

Figure 1: Layered Logical Multiplexing Protocol Architecture

3 Proposed Alternative Protocol Architectures

In this section, we first describe the traditional protocol model and then describe the two alternative protocol architectures proposed in this paper for future gigabit networking environments[3].

3.1 Traditional Protocol Architectures

In traditional protocol architectures, multiplexing and demultiplexing operations are performed at multiple protocol layers. At each protocol layer, for a received packet, first the demultiplexing is performed to identify a correct session and then the state change processing is performed on that session. Such protocol architectures can be termed as *Layered Logical Multiplexing (LLM)* architectures. Figure 1 depicts the behavior of a LLM model[4]. As discussed before, this architecture has difficulty supporting application specific QoS and implementing protocol stack in an ILP fashion.

3.2 Direct Application Association (DAA) Protocol Architecture

As the 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 proto-

[3] We have also proposed a Non-Monolithic protocol architecture which splits the protocol functionality between the kernel and the applications. Details on the Non-Monolithic protocol architecture can be found in [10].

[4] In each of the Figures 1, 2, and 3, the unshaded sessions represent the demultiplexing while the shaded sessions represent the state change processing.

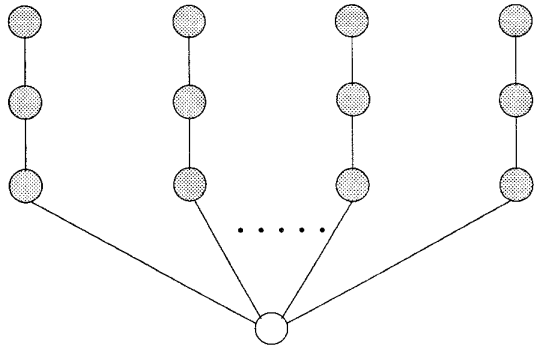

Figure 2: Direct Application Association Protocol Architecture

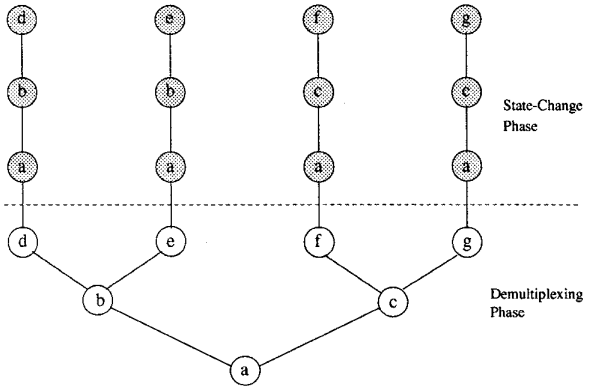

Figure 3: Integrated Layered Logical Multiplexing Protocol Architecture

cols in a DAA fashion. However, traditional networks do not provide such support and require incompatible extensions.

Figure 2 shows a DAA protocol architecture. In this architecture, the demultiplexing is done only once at the lowest protocol layer which determines the application to receive the incoming packet. An analytical study of DAA and LLM models [22] 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 Feldmeier [4] and Tennenhouse [21]. Other advantages of 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 [15].
- For the applications requiring real-time guarantees or other kinds of QOS, the system resources can be appropriately allocated to the packet so that the packet can meet its QoS.
- In high-performance multimedia systems, the destination could be an I/O device such as the 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 have designed a framework for implementing a DAA architecture stack in the context of the *x*-kernel. We have also implemented 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.

3.3 Integrated Layered Logical Multiplexing (ILLM) Protocol Architecture

In the ILLM architecture, 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 3.

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 updates 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 [4] and Tennenhouse [21] 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 [4, 21] where demultiplexing is performed at multiple layers intermixed with protocol state-change operations. As pointed out in Section 2.2, 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 ma-

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

jor problem of ILP optimization for traditional protocol architectures - determining the boundary between headers and data in network packets. As discussed in Section 2.2, 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 solution suggested by Abott and Peterson [1] is to use a Packet Filter [12] before the protocol processing of the received network packet is started. 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 3.2.

4 Design Overview

In this section, we describe the design of the DAA and ILLM protocol architectures in the context of the x-kernel. But first, a brief overview of the x-kernel is given.

4.1 The x-kernel Overview

The x-kernel [8, 9] is a protocol implementation architecture that defines protocol independent abstractions thereby facilitating efficient implementations 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 [17], TCP [18], or UDP [16]. 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.

4.2 Design of a 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 protocol 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 sessions, one from each protocol layer, that sit vertically above one another according to the protocol hierarchy. We have extended the x-kernel to support paths.

4.2.1 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 to have the application identifier assigned by a central authority. The protocol that performs 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.

We have implemented a sample DAA architecture protocol stack consisting of the extended Ethernet, IP, and UDP protocols. We have added two new 4-byte fields: an *application identifier* field and a *deadline/priority* field to the Ethernet layer. The application identifier field uniquely identifies an application. For our experiments, the deadline/priority field is used as the priority of the thread that shepherds the packet through the protocol stack.

4.3 Design of an ILLM Protocol Architecture

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. In this framework, 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 it is at the user-protocol boundary, the packet processing is switched from demultiplexing phase to state-change phase.

5 Performance

In this section, we compare the performance of the sample protocol stacks (TCP/UDP-IP-Ethernet) implemented in each of the LLM, DAA and ILLM styles. We first describe the hardware and software platform for our experiments and the benchmark tools used. We then report on the round trip latency of UDP for each of these models and performance of UDPDAA as compared to the performance of UDP in the presence of varying background network traffic.

5.1 Hardware/Software Platform

The hardware platform for our experiments is the Motorola MVME188 Hypermodule, a 25 MHz quad-processor 88100-based shared memory machine [6]. A light weight, multithreaded kernel called *Raven* [19, 20] has been developed in the Computer Science department at UBC and is used as the micro-kernel running on the bare hardware. The protocols have been implemented in the context of a parallel *x*-kernel: the *x*-kernel [8, 9] has been ported as a user level server in this environment and has been parallelized to take full advantage of parallel processing. Our approach to the parallelization of the *x*-kernel is similar to the one described in [2]. We use the one processor/thread per packet model for the processing of incoming and outgoing packets. The performance measurements are made by connecting two Motorola MVME188 Hypermodules through a 10 Mb/sec Ethernet network in light network traffic conditions. We used the on-board Z8536 Counter/Timer configured as a 32-bit timer with microsecond resolution for the measurements.

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

Table 1: LLM Protocol Stack: Latency

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

Table 2: DAA Protocol Stack: Latency

5.2 Round Trip Latency

We measured the round trip times of the sample DAA and ILLM protocol stacks and compared them to that of the corresponding LLM implementation to determine the relative overhead of the protocol architectures. The DAA and ILLM architectures do not implement any ILP optimizations. 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. The 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. Tables 1, 2, and 3 show the round trip times for the LLM, DAA, and ILLM protocol stacks respectively.

5.3 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

Protocol Stack	Round-Trip Time(ms) User Data Size(bytes)				
	1	100	500	1000	1400
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

Table 3: ILLM Protocol Stack: Latency

better than in the traditional LLM architecture. This performance gain can be attributed to the fact that demultiplexing is performed only at one layer in the DAA protocol architecture. The latency figures of the IP and UDP protocols also indicate that the DAA stack performs better when there are more layers in the protocol stack.

5.4 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 to saving (or storing) the session and message object pointers during the demultiplexing phase at the Ethernet, IP and UDP protocol layers and reading them again during the state-change phase at each protocol 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.

5.5 Real-time Network Traffic

Figure 4 shows the latency of UDPDAA and UDP stacks for 1-byte of user data in the presence of background UDP network traffic[6]. The processes generating the background UDP network traffic are also using simple ping-pong tests in an infinite loop. The background UDP network traffic is increased by running more number of such processes. 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. However, the priority is not respected in the Ethernet driver's send queue. As can be seen in Figure 4, the increase in latency of UDPDAA is much slower than that in UDP

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

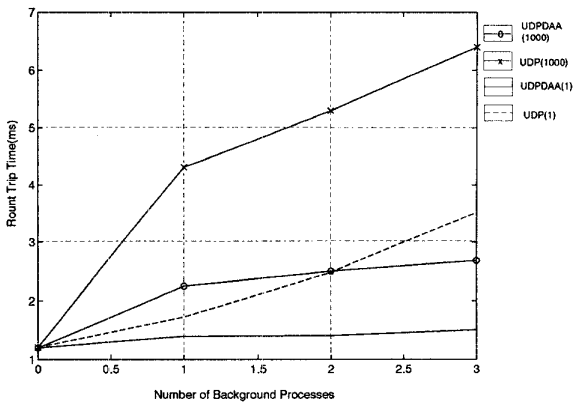

Figure 4: Real Time vs Non-Real Time Latency

as the background network traffic is increased for both data sizes. The results of this experiment show the superiority of DAA protocol architecture over LLM protocol architecture for real-time network traffic. This experiment also shows that the prioritized scheduling to process network packets is important for real-time network traffic.

6 Conclusions

We have proposed two protocol architectures (DAA and ILLM) suitable for the next generation of network environments. These protocol architectures can provide application specific QoS and resource allocation which are hard to achieve in traditional protocol architectures. We have designed protocol independent frameworks and implemented sample protocol stacks for each of these architectures in the x-kernel. The overhead introduced by these protocol models without incorporating any optimizations such as ILP is shown to be comparable to that of the traditional protocol model. We are currently working on incorporating ILP in both the DAA and ILLM protocol stacks which should substantially improve their performance.

Acknowledgements

We would like to thank anonymous referees for their valuable feedback on an early draft of this paper.

References

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

[2] Mats Bjorkman and Per Gunningberg. Locking effects in multiprocessor implementations of protocols. In *Proceedings of ACM SIG-*

COMM'93 Conference on Communications, Architecture and Protocols, September 1993.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

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

[12] 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.

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

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

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

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

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

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

[19] 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.

[20] 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.

[21] 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.

[22] Wenjing Zhu. Effect of layered logical multiplexing protocol architecture. Technical Report, Department of Computer Science, University of British Columbia, July 1992.