Fall 1991

Approaches to High Speed Networks

Frank Charles Paterra
Old Dominion University

Approaches to High Speed Networks

by

Frank Charles Paterra

B.S. Computer Science, May 1986, Frostburg State College

M.S. Computer Science, August 1989, Old Dominion University

A Dissertation submitted to the Faculty of Old Dominion University in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Computer Science

OLD DOMINION UNIVERSITY

November, 1991

Approved by:

[illegible]
C. Michael Overstreet (Advisor)

[illegible]
Kurt J. Maly (Advisor)

[illegible]
Ravi Mukkamala

[illegible]
Mark Pardue

This work is sponsored in part by grants from Sun Microsystems.

Acknowledgements

Few accomplishments can really be claimed by a single person. I would like to acknowledge some of the people who made this effort possible. First I would like to thank my advisors. Mike Overstreet and Kurt Maly. Mike, you have been more than a friend to me. and your true kindness has given me comfort many times. Kurt your drive for work has been an inspiration to me. I am sure that without it, I could never have completed this research. I would also like to thank the other members of my committee. Ravi Mukkamala and Mark Pardue, for their thoughtful comments and guidance.

I would like to recognize my fellow graduate students, especially Sanjay Khanna and David Game. David, thank you for your advice and your friendship. Sanjay, you have more than generous to me, I wish you success in the pursuit of your degree.

The one teacher has made the largest impact on my life was Dr. Robin Kranz of Frostburg State College. You taught me that it is ok to admit when I don't know something.

To my parents, Frank and Mary Elizabeth, I would like to thank you for teaching me that there is really nothing a person cannot do as long as they are willing to work for it. Thank you for letting me try my wings and for letting me fall on my own. You have taught me the most important lessons of my life.

Finally, I would like to thank my wife and best friend, Norizan. You have been a source of energy and encouragement to me and have sacrificed so much to allow me to complete my degree. I love you very much and I hope that I can be of as much support to you in fulfilling your dreams and goals.

Contents

List of Tables

List of Figures

ABSTRACT

Parallelism in High Speed Networking

Frank Charles Paterra

Department of Computer Science

Old Dominion University

Norfolk, VA 23529-0162

Advisors: C. Michael Overstreet and Kurt J. Maly

This work investigates possible methods by which existing potentially available communication bandwidth can be used by communication intensive applications. Presently fiber optic media are available that can provide multiple gigabits of throughput. Unfortunately, because of the computation overhead required to insure that data are reliably transmitted, this capacity has not be tapped.

A survey of work toward enabling the use of the potential bandwidth is presented. The parallel paradigm is identified as a strong candidate for providing significant increases in system usable bandwidth. Performing communication processing in parallel, however, presents the developer with several implementation options. These options are considered and categorized. This categorization represents a framework that is used in later analysis to compare different approaches and architectures.

Because the number of options available represents a combinatorial explosion in the number of software and hardware architectures that could be implemented, a sensitivity analysis is performed to exclude obvious failures, as well as to identify those components that need further study and close consideration. Some components are identified as limiters to total throughput obtainable: these components warrant special attention when implementing a parallel communication system.

Building on results obtained through the sensitivity analysis, a testbed was then built

and used to obtain performance data for one promising architecture and approach. The results for two and three channels implementations show near linear speedups. These results were then used to verify a model of the system used to calculate throughput values for systems with higher numbers of channels.

In order to more fully examine other promising architectures, a simulation program was developed and exercised. The simulation examined the impact of traditional communication parameters, such as window size and timer length, on performance in a parallel system. Further, the simulation confirmed some of the results of the sensitivity analysis and provided insight to the viability of two algorithms to implement flow control in a parallel environment. Additionally, scheduling algorithms to allocate processors to the communication tasks are examined and performance results are presented.

Chapter 1

Introduction and Problem Statement

The field of computer science called data communication encompasses several research areas. This chapter introduces some problems faced in this field and identifies the specific areas of study addressed in this work.

1.1 What is High Speed Networking

Computer to computer communication at rates of kilobit to low megabit per second speeds are commonly available in network computer systems today. This communication speed, however, is inadequate for many distributed applications, as well as the remote use of some high performance computing resources. Research directed toward increasing the speed of communication has, until recently, focused mainly on the physical media used to link computers together and on the access methods needed to utilize that media. Work in these areas has been successful; one can currently purchase fiber optic media capable of carrying multiple gigabits of data, and media access controllers that can provide 100 megabits of throughput. One popular media access control method is Fiber Distributed Data Interface, or FDDI, a fiber optic based network designed to provide 100 megabits of data transfer per second. As a network for connecting several computers with relatively low network demands, FDDI represents at least a 10 fold increase in the communication bandwidth currently available with popular local area networks such as Token Ring and Ethernet. Allowing a single node on a network to use a significant fraction of this bandwidth, however, requires nontrivial computation power to insure that the data are reliably delivered. Some applications, such as remote visualization, can require multiple hundreds of megabits or even gigabits of bandwidth, making even FDDI too slow.

The media are only one of the problems to providing high speed reliable data communication. The computational power required to perform protocol processing is significant. The amount of time spent waiting for packets to traverse the network and be acknowledged before communication can progress is another problem. Performing high speed communication requires that new approaches be developed to address these and other problems.

1.2 Problems in High Speed Communication

The areas of research called data communication include:

- Physical media used to transmit data among computers,
- Methods for accessing the media to insure that fair and timely communication can take place,
- Methods for determining computer addresses and the routes data take when traversing a network,
- Problems of transferring data among networks,
- Methods for controlling and limiting congestion in a network.

- Insuring that data arrive at the destination in a timely manner, complete and uncorrupted, and,

- Protocols used to specify how data are to be used and how they are to be displayed.

Fiber optic media for data communication can be used to provide bandwidths in the multi-gigabit range. Sending data rates of one gigabit or even hundreds of megabits, however, introduces new problems to the communication process. As communication rates approach gigabit speeds, the time available to place data on, and read or remove them from, the media becomes exceedingly small. In fact at one gigabit per second transmission rate, each bit must be physically placed on the media in no more than one nanosecond ($1 * 10^{-9}$ seconds). Hardware can be built to place and read data at that speed, but the speed at which data can be accessed from the sender's memory in order to place it on the network presents a new bottleneck. For example, fast static memory, normally only used as cache because of its high cost, has access times of five to ten nanoseconds per byte, barely fast enough to keep up at one gigabit. The speed of memory can be effectively increased by using multiple independent banks of memory, each feeding the network in turn, but as network speeds increase the number of banks required becomes large and present a problem.

Placing data on the network is only part of the problem. Once data have reached the receiver, they need to be examined to identify the intended recipient and then, if they are for the local host, they must be copied from the network to a buffer for protocol processing. As with the transmitter, at gigabit speeds the receiver only has one nanosecond to read and act on each bit of the data. The time required to identify the recipient becomes critical, and, because buffering data at gigabit speeds for any length of time requires significant memory, once the data have been read they must be processed quickly. Because more than one action is performed on arriving data, less than one nanosecond of time can be allocated to each bit. This means that the instruction time for the receiving processor can be at most sub-nanosecond. Again, this speed is achievable, but the costs may be prohibitive.

Direction	Transport	Network
sending	191	61
receiving	186	57

Table 1.1: Estimated Instruction Counts

Once data have been read from the network, additional processing is required to insure that the data are received by the intended process, have been received as sent, and are in the original order. Two protocol layers, called network and transport, are responsible for these tasks. Table 1.1 gives the values that Clark et al. in [7] estimated to be the minimum number of instructions required to perform TCP/IP, or transport and network layer processing. If the standard transport layer packet size of 2048 bytes is used, gigabit communication requires more than 15 million instructions per second to send data and 9.5 million instructions per second to receive them. These numbers do not include the additional processing required to processes packet acknowledgments, estimated to be one third of the total processing time. Additionally, Clark's optimized instruction count assume a CISC computer and should be increased for RISC computers. For realistic computations a factor of two for RISC computer instruction counts over those of a CISC can be assumed, indicating that a 40 MIP RISC computer would be needed to perform just the transmitting functions of the transport layer protocol processing.

At the network and transport layer, not all bits received need to be directly accessed. Even so, Jain et al. in [19], estimated that memory access times of less than 10 ns would still be required for TCP/IP receiver operations at gigabit speeds. Finally, to support a 40 MIP processor and the required memory cycle time, a sub-nanosecond system clock and support circuitry would also be required.

If data are corrupted during transmission, the transmitting station needs to resend them. Often, the mechanism within a protocol to notify the sender of lost or corrupted data is for the receiver to simply not acknowledge the receipt of data. Because the sender maintains a timer for each packet it sends, it assumes an error in the data occurred if no acknowledgment is received before the timer expires. This solution, while acceptable for lower speed networks, is very expensive for high speed networks because of the loss of potential to transmit data while waiting for a timer to expire. As an example, in a long distance terrestrial network of 5000 km, the round trip time for a packet is 30 ns. The round trip time is the shortest time in which a packet can be received at the destination, and an acknowledgment can be received at the sender. If this number is used for the timer length, a lost packet will cause the sender to wait at least 30 ns after the packet was lost before resending it. During this time the sender continues to send data, but after the timer expires the sender must resend the lost packet and then either wait for the packet to be acknowledged, or resend all of data after the lost packet. In either case this represents 30 megabits in lost transmission capability. Netravali, et al., [27] suggests that with a more reasonable timer length would be two times round trip time, making the loss 60 megabits. At slower transmission speeds the loss would have been much less, making the waiting time less of an issue. As is discussed in Chapter 2, this is a problem that needs to be faced in protocols designed to operate at gigabit rates.

1.3 Why High Speed, Reliable, Data Communication is Important

The computing needs of applications in the fields of medicine, chemistry, aerospace, and remote sensing are immense. The end product of much of these computations is often visual images that are best viewed as a continuous animated movie [8]. Supercomputers are capable of computing the data required for these images in real time, but these computers are expensive and not available everywhere. The user of a supercomputer must either find a way to purchase a machine of their own, or must travel to a supercomputer facility.

Because of cost, the first option is not viable for many users. For time critical applications, such as medical imaging and remote sensing, travel may not be feasible. Another possible solution is to use a supercomputer remotely by supplying it with raw data, and viewing the results over a network. The amount of data involved, however, make this approach unrealistic without faster networks. To give an example of the amount of data that must be transferred, current scientific images typically used for visualization are 1024 by 1024 pixels in size, with each pixel represented by 8, 16, or 24 bits to enable the viewing of different colors. In order for images to be viewed as continuous animation, a new image needs to be delivered to the viewer approximately every 30 ms. For visualization to occur in real time, as is often required for medical and satellite sensed data [13], these specifications represent bandwidth requirements of between 250 and 750 megabits of application data per second. These numbers are likely to increase as higher resolution displays become available.

Remote visualization is not the only application that would benefit from high speed reliable data communication. In the area of high performance computing data often must be shared among processors in order to compute a single result. Physically distributed simulations, where the components contributing to a simulation are not resident on a single computer, is such an application and would be enhanced by access to high speed networking. An example simulation would be that of a tank battle field where humans are in the loop competing against simulated opponents. Often these simulations are performed with tank drivers operating simulated or actual tanks in one location, while the opponent tanks are being simulated elsewhere. The speed at which data can be collected by the simulating computer(s) about each of the tank groups determines the level of realism that can be achieved, both as training for the tank drivers and in the computed outcome of the battle.

Other high data rate applications are becoming evident as our society moves to a more global information based one. For example, stock market operations typically occur during local business hours. However, as more companies become international, and local governmental and commercial actions have global implications, the questions of whose business hours becomes an issue. The idea of truly international stock markets, where transactions no longer occur within national borders alone, is currently being discussed. This sort of market will require that information be made available nearly instantaneously, rather than just when the newspaper comes out the next morning. This need implies high bandwidths and reliability.

Another new user of high data rate networks is the application called virtual reality. This application involves the presentation of sound and video images to a user, to simulate a particular environment or area of the universe. The displays have to appear as three dimensional and, as the user moves with in the simulated environment, must change to represent current simulated views. As with remote visualization, the bandwidth required for high resolution, real time, color displays will be immense.

1.4 Focus of This Research

The raw bandwidth for each of these applications can be provided with a single fiber optic cable. Unfortunately, fiber optic cables are only the first step to providing higher bandwidths. This work addresses the problems of providing reliable communication at gigabit speed. The focus will be in the transport layer of the ISO communication stack, and will review proposed and existing monolithic protocols and extend parallel methods for solving these problems. Current workstation class RISC computers, such as Sun SparcStations, provide an application with the ability to transmit approximately six megabits of reliable data per second. This work discusses hardware and software architectures that can be used improve that number.

Chapter 2

Survey

As discussed in the previous chapter, several subareas in the domain of data communication need to be rethought in the light of high speed networking. This chapter presents approaches that have been suggested and investigated and gives an outline to the specific work performed in this thesis.

2.1 Solutions to Communication Problems

Traditionally the communication problem has been represented by a layered concept called the OSI communication stack. The layers in this stack are shown in Table 2.1; the lower four are examined in the following sections of this chapter.

Layer	Description
Application	Application to application, such as sendmail
Presentation	Syntax and semantic compatibility
Session	Simple program interfaces, such as FTP
Transport	Provide reliable end to end communication
Network	Provide address resolution and routing
Media Access	Provide access to the physical media
Physical	The media and its physical and electrical specification

Table 2.1: The OSI Communication Stack

2.1.1 Media

An area that has been successful in providing high bandwidths is media. Fiber optic media have the ability to provide bandwidths in the multiple gigabit range. Local area networks (LANs), and metropolitan area networks (MANs), have been developed with fiber optics as the connecting media. Long distance networks, such as inter-continental phone links, have been successfully implemented via the use of undersea fiber optic cables. For data networks, however, the bandwidth capacity of fiber optics has barely been tapped. As discussed later, this is mainly due to other problems in the data communication domain.

For short distances, 25 meters or less, twisted pair copper wire is also a viable medium. In fact, as is discussed below, twisted pair has been used as a base for one high speed media access control protocol called High Performance Parallel Interface or HiPPI, and modifications to one fiber optic based media access control protocol, FDDI, have been proposed to allow it be to implemented with twisted pair as well.

2.1.2 Media Access Control Protocols

In order for any media to be of use, a protocol is needed to access it. Common desirable attributes of a media access control protocol, called a MAC, often include: fairness, low latency, the ability to handle synchronous as well as asynchronous data, and physical fault tolerance. This section discusses three relatively new MAC protocols. Modifications to one to provide higher throughput are also discussed.

FDDI

As discussed previously, FDDI is a fiber optics based protocol capable of delivering 100 megabits of data per second. Formed from work performed to develop the IEEE 802.5 LAN [29], FDDI is a token ring network that employs two counter rotating fiber optic rings. Because fiber connections are really point to point, with a receiving and sending pair in each station enabling a ring, and the FDDI specification allows only 1000 ring connections. up to 500 stations can be attached. The maximum length for an FDDI network is 200km. There are no minimum station count or network length requirements [29].

When a station has data to transmit, it waits for and acquires a token before sending them. Internal to each station are timers that are used to determine the target token rotation time, and control how long a station can hold the token once it had received it. These two timers provide fairness and the ability to handle synchronous traffic. The token holding time determines how long a station can transmit once it has the token, and the target token rotation time, the time that a token takes before completely traversing the ring, makes the token arrival rates deterministic.

Of the two fiber optic rings forming a FDDI network, only one is normally in use for data transmission, with the second only being used in case of a primary ring failure. If a ring breaks, a new ring is formed by each of the two stations on either side of the break forming a loop internally, using the second fiber. This newly formed ring is approximately twice as long as the original. Figure 2.1 shows two FDDI networks. The first is the normal configuration, and the second shows a healed ring after the line between two stations failed. If a station had failed, the ring would be healed by the two stations on either side of the failed station. FDDI represents a significant increase in bandwidth over existing LANs, but falls short of the needs outlined for remote visualization and distributed computing.

Game in [14] proposed modifications to the basic FDDI network to enable gigabit operations. His work centered on modifications to the timer values, removal of data from the ring by the receiver, use of both fibers concurrently, and reuse of slots on the network, created by the removal of data at the receiving station. Game showed that these modifications allow transmission at rates of 400 to 500 megabits, and could make FDDI viable at gigabit speeds.

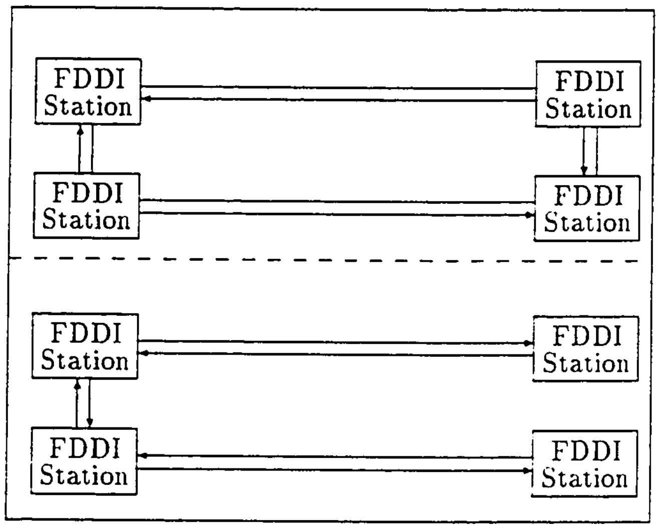

Figure 2.1: Before and After FDDI Ring Failure

CSMA/RN

Carrier Sensed Multiple Access/Ring Network, or CSMA/RN[12], is also a fiber optic based network. However, unlike FDDI, CSMA/RN is a carrier sensed, multiple access network, rather than token based, allowing it to exploit the fact that multiple transmission packets, with different source and destination addresses, can be physically resident on the fiber ring concurrently. CSMA/RN was designed to operate at a nominal speed of one gigabit, but because it employs destination removal of transmitted data, an average throughput of two gigabits is actually possible.

Because data are removed from the ring when they are received, and must not be removed before they arrive at their destination, the destination address of the data must be determined before they passes a station. As discussed previously, buffering data in computer memory when communicating at gigabit speeds is not reasonable. In order to determination of destination of data before they pass a node, CSMA/RN employs delay loops at each station. These delay loops, easily realized as coils of fiber cable, have connections to the controller at each end of the coil. As data enter the loop, they are read by the controller.

If they are addressed to the local host, they will be read and removed from the network upon exiting from the loop. These loops allow CSMA/RN to enjoy zero loss of data due to collision on the media. Normally, in a CSMA network, stations sense to see if the network is free before they begin transmitting, and then they attempt to transmit a complete frame. If two stations sample the network at the same time, both will transmit concurrently and the result will be a collision of data and loss of bandwidth. By presampling the network at the beginning of the delay loop, the transmitting station can be sure that enough free space exists on the fiber for it to transmit at least a packet header and an abort flag. If, after beginning a transmission, data are sensed to be entering the delay loop, the transmitting station can transmit an abort flag and stop transmitting until more free space is detected on the network. Of course, if data entering the delay loop are for the transmitting station, the station can continue to transmit and simply remove incoming data as they enter the receiving portion of the station.

Additionally, CSMA/RN provides synchronous data transmission through the use of a circulating reservation packet (CRP) [11] . When a station requires synchronous transmission service, it accesses a CRP currently on the ring, adds information about its needs to that packets and places it back on the network. When the CRP passes a station, it is examined to determine when unallocated space on the ring for asynchronous traffic will exist.

CSMA/RN enforces no limits as to network lengths or number of stations, and represents a viable MAC for gigabit networks.

HiPPI

HiPPI is a media access protocol that was originally designed to connect a host computer to peripherals such as disks or printers. It was later expanded, however, to allow multiple hosts to be connected to each other or to share peripherals. Working as a point-to-point network. HiPPI can be used to provide single or multiple 800 and 1,600 megabit simplex channels. For most applications, duplex operations are desirable so separate interfaces are provided for input and output. As can be seen in Figure 2.2, the interface consists of several control lines and a set of parallel data lines. The control lines are used to setup a connection, indicate that a host is ready receive data, provide a synchronizing clock, and test the circuit. Two additional control lines are used to indicate the end of a burst of words, and the end of a packet of bursts. These are labeled Burst End and Packet End in the figure respectively.

Because of its original design goals, to connect one host to its peripherals, the length of a HiPPI network is only 25 meters. This, and the fact that it operates as a point to point network, requiring processor attention for each data transfer that passes through, makes it impractical for traditional LANs and MANs.

What the HiPPI interface does provide, and indeed was its design goal, is a standard interface that can be used to link processors to peripherals. The peripheral of interest for this work would be a high speed network interface. The HiPPI interface would make the addition of new types of hardware to an existing network a simpler task.

2.1.3 Transport and Network Protocols

The processing required to perform reliable communication has been identified as a bottleneck for high speed data communication networks [21, 7]. Reduction of this bottleneck has been the focus of much research interest and efforts fall into two broad categories, monolithic and parallel. In both of these categories multiple approaches have been proposed. Independent work by several researchers has resulted in performance gains in both categories. The next two sections of this chapter discuss some of the approaches taken in each category.

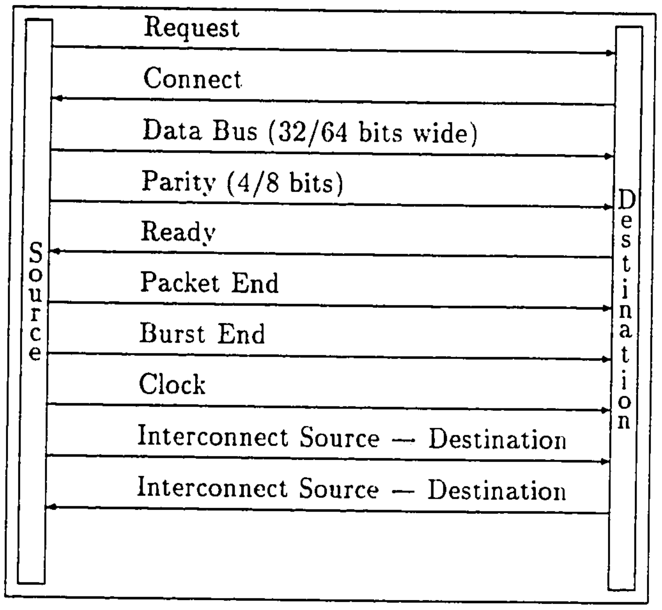
Request
Connect
Data Bus (32/64 bits wide)
Parity (4/8 bits)
Ready
Packet End
Burst End
Clock
Interconnect Source — Destination
Interconnect Source — Destination
Source
Destination

Figure 2.2: The HiPPI Interface

Monolithic Approaches

Monolithic approaches attempt to either optimize existing protocols, or developing new protocols that reduce the amount of processing required to insure reliability.

Working towards optimization, Clark et al. [7] gives a thorough analysis of the processing required at the transport and network layers of the OSI stack, and then goes on in [6] to identify other bottlenecks in communication layers and supporting hardware. One solution suggested by Clark was to move some of the processing, currently being performed by the transport and network layer, to the application. This has the advantage allowing the application program to decide which functions it requires and to use the available processing power accordingly. The obvious disadvantage is that the clean interface between layers of the OSI stack become blurred. As discussed earlier, a highly optimized version of TCP/IP has been estimated by Clark et al. to potentially produce as much as 800 megabytes of throughput using a modern RISC processor. Clark's work, however, did not address hardware bottlenecks other than processor speed. Van Jacobson [18] implemented a somewhat optimized version of TCP/IP and was able to verify that throughput as high as 8 megabits could be achieved with a Motorola 68020 and existing hardware.

Silicon Graphics' optimizing approach, XTP, resulted in a new protocol that places much of the transport layer processing into VLSI hardware [4]. This dedicated hardware's performance is not limited by the speed of a host CPU, and the CPU is not burdened with transport layer processing. The result is an increase in throughput independent of CPU speed and load, and only limited by the speed of the protocol VLSI hardware. PSi is another protocol that is being developed for implementation in VLSI [1]. Krishnakumar et al. [20] describe work toward the automatic translation of a protocol specification into VLSI design.

Delta-t [33] and NETBLT [5] are optimized transport layer protocols which speed the communication process by reducing the amount of processing required. In the case of NETBLT, this reduction has been achieved by providing two unidirectional paths, separating the data and control information flow. This allows the most efficient implementation for each, rather than a compromise for both. Additionally, NETBLT supports a selective acknowledgment strategy which allows individual packets to be resent as soon as their loss is detected, rather than the go-back-N algorithm found in TCP.

La Porta and Schwartz in [28] suggest that bunching packets into larger blocks and setting only one timer for each block would result in faster communication. They justify this approach by saying that new networks, especially fiber optic based networks, are significantly quieter than previously achievable so less loss will be observed.

In most new protocols, the fact that waiting for notification of lost packets represents a significant loss in bandwidth has played a significant part in their development.

One additional monolithic approach that has been examined is the use of a supercomputer, such as a Cray, to perform transport processing. This approach, while very expensive to implement, was shown by Borman in [3] to produce throughputs as high as 750 megabits per second.

Parallel Approaches

In the arena of parallel protocol processing, Jain et al. in [19] and my work in [22] has examined the potential of providing multiple processors at the transport layer, each performing a complete transport processing task.

Jain et al. classify several schemes by type of memory, granularity of parallelism, and scheduling policies used. In their selected proposed implementation they use a pool of specialized protocol processors to transmit and receive data from a single high speed source. Each of the protocol processors operates independently on individual packets but maintain context records in a global memory so that acknowledgments, retransmissions, and resequencing can be handled properly by a centralized processor. In their implementation, special attention must be given to insure that the two common buses, the common memory, and the centralized queue processor do not become a bottleneck for communication because all packets must pass through each of these at least once. Existing memory speeds, as discussed in Chapter 1, will limit the throughput in their proposed architecture because of the need for context records in a global memory.

Maly, Wiencko et al. in [21] initially proposed a general solution that provides parallelism at the transport, media access, and physical layers of the communication stack. They kept the functionalities of the traditional protocol stack and introduced parallelism at some points where performance bottlenecks could develop. Independent of network layers, all protocol functions are viewed as processes to which multiple general purpose processors and channels are allocated to maximize throughput. The key concept is to structure the processes so that full advantage can be taken of parallelism at all levels. This provides additional benefits of increased reliability and graceful degradation in the case of a processor failure. Additionally, this approach allows the use of existing protocol standards and hardware technologies. This method is extended in Chapter 3 to include the network and application layers.

In [16], Haas describes an approach to parallel communication that uses multiple processors to work on each transport layer packet. In this protocol, called Horizontally Oriented Protocol or HOPs, the traditional OSI stack is recast into three layers. The first three layers of the OSI stack, media, media access control, and network, becomes the first layer of HOPS, called the network access layer. The transport, session, and presentation layers of OSI are combined into the new second layer, called communication interface, or *CI*. Finally the application layer of OSI becomes the third new layer. The most important layer for performance improvements is the *CI*. To obtain this improvement, parallelism is used by assigning separate processors to each of the tasks that must be performed for each incoming or outgoing packet. For example, one processor could be dedicated to packetization, while another is dedicated to performing flow control functions. A similar approach was briefly discussed by Jain in [19], and it was noted that, with packet structures such as is found in OSI TP4 (TCP), little parallelism can be achieved. Haas has found success by developing his protocol based on functions rather than a layered concept. Because functions can be performed independently, little communication is required among processors, so parallelism on the packet level can be achieved. This approach has an implicit limitation on the degree of parallelism that can be achieved because there are only so many tasks to be performed for each packet. If each of the tasks functions are replicated for multiple packets, higher parallelism can be achieved but only if parallel data streams of data are present.

Netravali et al. in [27] discuss a hybrid approach, first optimizing the protocol and then implementing it on a parallel computer platform. The optimization presented by Netravali et al. was based on the observation that the sooner an error is detected, the less bandwidth is lost waiting. To take advantage of this they exchange state information from the sender and receiver with every packet sent. This information transfer requires little bandwidth when included with normal data packet transmissions and results in near immediate notification of lost packets. The protocol was also simplified because the need for active timer processing was eliminated.

Each of the above parallel communication methods are software proposals. Little has been done to measure their true performance, to examine the hardware implications of their implementations, nor to classify their approaches so they can be compared and extended. In the remainder of this work I classify the options available to parallel communication systems and identify the components that are of particular concern when performing high speed communication.

Chapter 3

Options in Parallel Communication

The previous chapter introduced some of the approaches taken to perform protocol processing in parallel. In order to evaluate these methods, a categorization scheme is needed to assist in analyzing and discussing them. This chapter identifies and examines the options available in parallel communication to form a basis for that comparison. Because so many options exist, and their potentials and limitations are not always immediately evident, it is important to carefully select candidates for detailed analysis. A model was built which allows a quick first look at possible option combinations to aid in identifying those that seem promising. A sensitivity analysis is performed with this model to quickly identify promising approaches for further study.

3.1 Introduction

Comparing methods used in parallel communication protocol processing is difficult because researchers use different terminology to describe the functions being performed, and use different grouping of subfunctions and approaches. Classification of the methods and the results observed by each approach would be beneficial.

The objective of this chapter is to identify the different options, approaches. algorithms available when implementing parallelism at single and multiple levels of the communication process. Section two of this chapter evaluates classifications of parallelism in other applications and specifically defines the terms used in parallel computation and how they relate to parallel communication. Section three examines options opportunities for parallelism of different stages of the communication process. Section four uses these opportunities and options in a sensitivity analysis to identify promising approaches and to indicate which components need further study.

3.2 Taxonomies in Parallelism

Parallelism to increase performance and fault tolerance in computation has been studied for more than 20 years [10, 30, 9]. Early work by Flynn resulted in four major categories for describing parallel computing architectures. These are:

- SISD – Single Instruction, Single Data
- SIMD – Single Instruction, Multiple Data
- MISD – Multiple Instruction. Single Data
- MIMD – Multiple Instruction, Multiple Data

Working at any single level in the traditional OSI communication stack. Flynn's categorization can be applied as shown in Table 3.1. This classification describes the relationship of CPUs to packets. but is incomplete in that the granularity of computation as it relates to the number of data streams and in the allocation of CPUs to data streams as well as the

Flynn	Communication
SISD	Single CPU performing all protocol processing
SIMD	Multiple CPUs working on multiple packets in lockstep
MISD	Multiple CPUs working on a single packet at a single layer
MIMD	Multiple CPUs working on multiple packets, independently, at one or more layers

Table 3.1: Application of Flynn's Classifications to Communication

interface between multiple communication levels. Additionally, this classification is incomplete for parallel communication as it does not address the interaction among processors within each layer.

In [19], Jain discusses three categories of granularity for processors to data streams. The finest level of granularity provides for multiple CPUs operating on a single packet within a stream. This relates to my use of Flynn's MISD, as well as work by Haas in [16] and Netravali et al. in [27]. The medium level granularity Jain et al. discuss has the individual packets being processed on individual processors as a single stream of data. In this method, processors are given packets as they arrive from the input stream and, once they have been processed, the data and control information is passed to a separate processor for resequencing and maintenance of data stream information prior to deliver of data to application layer. This level of granularity is most closely related to my use of Flynn's MIMD category. The most course level of granularity discussed by Jain assigns individual processors to specific data streams. This method reduces the need for a separate processor to maintain data stream control information, and reduces the amount of interprocessor communication required to properly handle packet timeouts and acknowledgments. This level of granularity also falls into my use of the MIMD category. Jain's work however, was only at the transport layer, so the issues of communication, control, and scheduling between layers still have not been addressed.

Classification	Description
DRDA	Distributed Retransmissions, Distributed Acknowledgments
CRCA	Centralized Retransmissions, Centralized Acknowledgments
DRCA	Distributed Retransmissions, Centralized Acknowledgments
CRDA	Centralized Retransmissions, Distributed Acknowledgments

Table 3.2: Retransmission and Acknowledgments

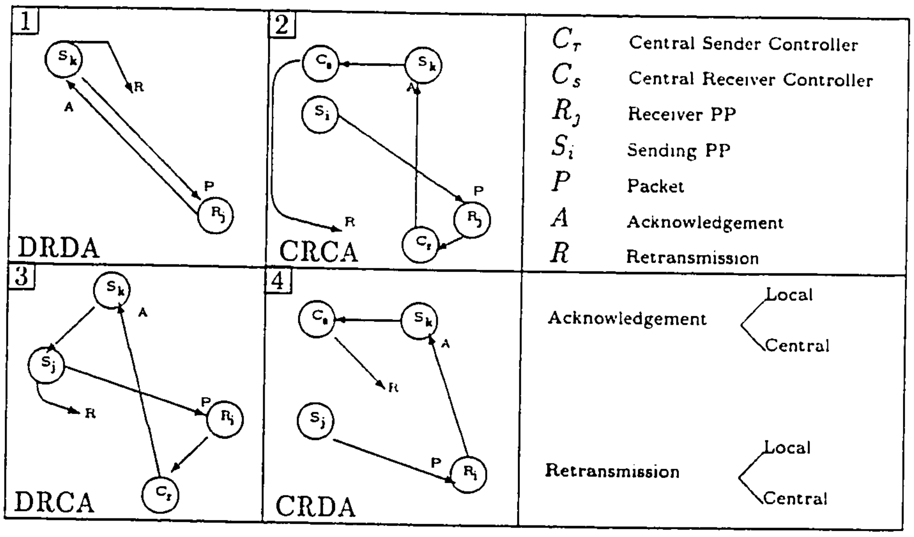

Figure 3.1: Acknowledgement and Retransmissions

Working with Maly et. al in [22], I addressed some of the issues of interlevel communication. Specifically, the placement of scheduling mechanism, the type of scheduling available, and the supporting hardware that may be required for each. In this paper I also described four categories of methods to generate acknowledgments and retransmissions, enabling reliable communication when multiple processors are being used at the transport layer. The four categories are show in Figure 3.1, and are defined in Table 3.2.

DRDA requires that a separate data stream is present between a sender S_k and a receiver R_j, and as such can be considered to fall into Jain's course grain classification. In this solution, sender S_k maintains timers and performs any required retransmissions for the packets that it sends, while receiver R_j is responsible for generating acknowledgments for the packets it receives. My work in [22], falls into this category.

CRCA employs separate processors to perform retransmission and acknowledgment generation. In this solution, timers for packets sent by sender S_k can be maintained either by S_k, or the centralized processor C_s. In the former case, when a timer has expired, sender S_k notifies C_s and C_s forces a retransmission if an acknowledgment was not received by any other S_i. The retransmission can be performed by any S_i. In the later case, once sender S_k has sent a packet, it discards the packet information and begins sending its next packet. If the timer for a packet expires, C_s forces the packet to be retransmitted by some S_i. Because with CRCA the S_k does not receive the acknowledgment, its window size must be very large or it will be stuck waiting for window movement. In the later case all of the timer processing is off loaded to a single processor, because timer processing is a significant part of the transport processing overhead, it would be advantageous if this task can be parallelized. This classification fits the medium granularity model that Jain et. al concentrated on in [19].

DRCA requires that the sending processor perform their own timing task, but the acknowledgments are generated by a single processor on the receiving side. In this scheme, sender S_k sends a packet to any receiver R_i. When a packet is received by some R_i, the centralized acknowledgment processor C_r is notified and it sends an acknowledgment to any S_i. When the acknowledgment is received, it is forwarded to the particular S_i that sent the packet and the packet's timer is stopped. Sender S_k will resend any packets for which its timers have expired.

The final classification, CRDA provides a centralized processor to perform retransmission of lost packets, but the receiving processor R_k generates the acknowledgments for any packets it receives. The problems outlined for the sending side of CRCA need to be addressed for this solution as well. The problem of resequencing received packets has not been addressed in CRDA nor DRDA.

This work only relates to the transport layer where acknowledgments are needed.

As can be seen, there are several issues in parallel communications and existing classification schemes for parallel processing and none are complete. It is tempting to simply use all of the classification schemes available, as a collection, to describe each system. However this may result in ambiguous classifications, and incomplete descriptions. For example, two ways to define a system with processors on the sending and receiving side maintaining multiple transport layer connections were presented. Jain et al. call this a course grain system and my work calls it DRDA. Further a majority of the work in parallel communication has been at the transport and media layers, and little has been done at the network and application layers, so existing classification schemes may not adequately address the other layers. Finally, none of the above systems can be used to completely describe the work done by Haas [16] and Netravali [27], where the protocol has been defined based on functionality rather than on the layers of the OSI stack.

3.3 Opportunities and Problems in Parallelism

A majority of work in the published literature has been in the area of parallelizing the transport layer. Some work was done earlier to enable parallel media access, but little has been done at other layers of the ISO stack. In this section the layers of the ISO stack from the application to the media are evaluated and the opportunities for parallelism at each level are described.

Performance gains should be achievable from the application and network layers. In [15] Gitlin shows that when using a Sun Sparcstation, 20 MIP machine, if a single FDDI interface is provided(100 Mbits/second), the throughput of the network layer (IP) is a little more than 20 Mbits/second, the throughput of the transport layer is only about 15 Mbits/second, and the throughput of the application layer with the application being file transfer (FTP) is only about 1.3 Mbits/second. The higher layers are actually a greater bottleneck than the transport layer, so if an increase can be seen there, total system throughput will benefit.

3.3.1 Application Layer

At the application layer, consider applications running on a single CPU and those running on parallel machines. First, for those running on a single CPU, when block transfers of data, such as file transfer applications are running, other CPUs resident in the machine can be used to perform the application's data communication functions. In previous work [23], I observed near linear speedups of throughput by simply replicating the communication processors. This means that a pool of processors could be used to speed this type of application.

In the case of a parallel processor machine with a single application running on multiple CPUs, spare processors or cycles on application processors can be used to send data in parallel as was described for the single CPU application. With parallel applications, however, additional opportunities for parallelism exist. To exploit these opportunities the following questions need to be answered.

- Do data generated by one application CPU need to be delivered to a specific CPU on the receiving end, or will any CPU suffice?
- Can received data produced by one CPU be consumed before data produced by another CPU have been received?
- How does the parallelism available at the transport layer affect the use of parallelism at the application layer, both for the application and its communication?

While the answers to these questions are primarily driven by the application being considered, some are also partially application independent. For example, regardless of the application, some form of scheduling of application processor data to transport protocol processors will need to be provided. This will have an impact in so far as computing resources may be needed for this control and scheduling.

3.3.2 Alternative Architectures to Parallelism at the Transport Layer

The transport layer has been the focus of significant research recently. As discussed in the first and second sections of this chapter, numerous ways to exploit parallelism at the transport layer exist. The transport layer, identified as a significant bottleneck for high speed communication, also represents one of the most interesting processes to parallelize. Because the transport layer is responsible for providing reliable communication, all data must be delivered to the receiver correctly, and data must be delivered from the receiver's transport layer to its application layer in the same order as sent. These two attributes would initially seem to require that either the sending processors maintain some sort of global information or maintain separate data streams in order to insure that all data are accounted for. On the sending side, because data are being sent on more than one path the transport layer must have a large buffer so that data can be resequenced before it is delivered to the application. Specifically, if each sending processor is maintaining a separate transmission window, the receiving transport layer needs to have n * (window size) * (packet size) bytes of storage, where n is the number of transmitting transport layer processors. As is discussed below, this is not the only way to perform these tasks.

Scheduling Issues

As introduced above when discussing the application layer, the resources of the transport layer must be scheduled to fit the needs of the application layer. This problem is multi-faceted, and several solutions for each dimension can be examined. Table 3.3 gives some of the issues to be considered and some broad option categories for each. The location of the scheduler has an impact on the supporting hardware used, the amount of processing that needs to be performed at each level in addition to communication, and the amount of interprocessor communication required. As shown in Figure 3.2, the scheduler can be in any of three locations. The location chosen has an impact on the utilization of processors, buses, and memory. Additionally, the location determines if the scheduling process is a single processor or a distributed process, and it can indicate the sophistication of the scheduling mechanism.

The scheduling algorithm has an impact on the system's performance. In [22], I showed that when data channels are evenly loaded, FCFS and Round Robin performed nearly equally. However, if the transport processor loads are very different, neither of these solutions could exclude a slow processor from being assigned the final segment. This could result in higher system latency and slower throughput if a nearly complete transmission had to wait a significantly long time for its final segment to be completely sent. A more intelligent algorithm could exclude slower processors from participating in the transmission of the final segments, thereby increasing throughput. The slower processors could be used them early in the transmission. This has the potential to exact higher throughput by using each of the processors when they can make a contribution to the communication task.

Data Sizes

The data blocks generated by the application should be sized to meet the processing speed and memory storage capacity of the transport processors. If block sizes are too small, more communication between the layers is required and if block sizes are too large, the application layer may be forced to wait until data is sent from the transport processors before it can begin producing more data. Adaptive block sizes could change with application types and needs, as well as anticipated network capacity. As with adaptive algorithms discussed earlier, the availability and cost of additional information needs to be considered.

The size of a segment of data from a block that is given to the transport layer processors

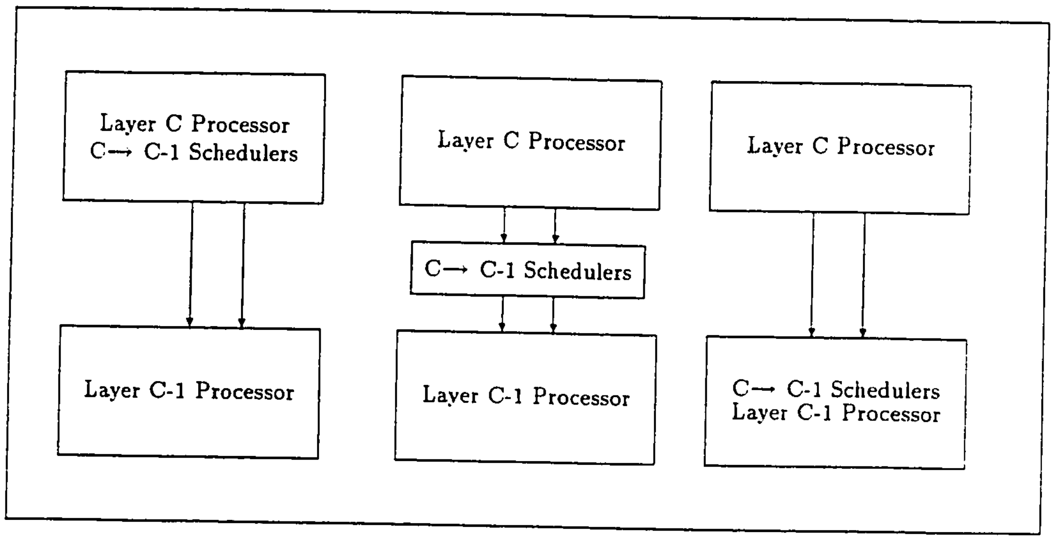

Figure 3.2: Scheduler Placement Options

must also be determined. If segments consist of only one packet of data, the use of each transport processor can be finely adjusted with its load to insure that the complete transmission is not waiting for a slow processor. Unfortunately, this fine tuning comes at the cost of additional interprocessor communication, and possibly wasted wait time. If segments are too large, little fine tuning can be done, and in fact, some processors may be excluded from the transport processing because all data has been allocated to just a few of the available processors. If timely information about processor load and previous performance can be obtained, the size of the segment given to a particular processor can be tailored so that all the processors complete their segments at nearly the same time, resulting in a higher degree of parallelism.

Timer Maintenance and Packet Retransmission

In addition to scheduling processors at the transport layer and determining the correct data sizes to generate and assign, issues of timer maintenance and retransmission need to be addressed. Both of these problems can be solved in several ways. The specific

Attribute	Options	Implementation
Location of Scheduler	Application Transport In Between	Centralized/Distributed Distributed Centralized
Type of Scheduler	Simple Intelligent	FCFS/Round Robin Adaptive
Block Size (data size used by application)	Fixed Variable	 Remainder/Adaptive
Segment Size (size of data used at transport layer)	Fixed Variable	 Adaptive

Table 3.3: Sending Scheduler for Transport to Application Mapping

problems and categories of solutions are given in Table 3.4. Because the transport layer must insure reliable transmission, timers or some other way of insuring that lost packets are eventually resent must be incorporated. Maintenance of timers normally require some processing to update and react when a packet has been lost. The task of maintaining this information can be performed in a central location, or distributed among the processors performing the transport processing. This task in particular is a large part of the overhead of transport processing and distributing is potentially beneficial. To determine the desired solution for this problem one must also decide where the incoming acknowledgments will be received and what processors will be responsible for acting on them. If the timers are located centrally, any transport processor can receive the acknowledgments and forward them to a dedicated acknowledgment processor. Alternatively a dedicated acknowledgment processor can receive all the acknowledgments directly and then stop the timers. If the timers are located in the sending processors, the acknowledgment for a packet either has to be received directly by the sending processor, or received by any processor and then forwarded to the sending processor. Alternatively if the sending processor keeps timers in a global memory and acknowledgments are received by any processor, the processor receiving the acknowledgment can stop the timer directly and the sending processor will simply check the global memory when a window movement is needed.

Task	Options
Timer Execution	Centralized/Distributed
Receipt of Acknowledgment	Sender S_k, Any S_i, Dedicated
Location of Window	Individual Windows for each S_k. Single Window
Generation of Retransmission	Original Sender, Any S_i

Table 3.4: Timers and Retransmission Options

Placement and size of the sending processor window is also an issue that needs to be addressed. In the last example, it was implicitly assumed that each sending processor maintained a separate window. An alternative implementation. however, could have a single, large global window that is used by all processors. The solution selected for this problem also has an impact on the other issues in Table 3.4

Acknowledgment Generation

The receiving end of the transport processing is responsible for generating acknowledgments for received packets, and for reordering the data received before passing them to the application layer. The categories of solutions to these problems in a parallel implementation are given in Table 3.5. As packets are received by a processor, the single processor may not be able to determine if a packet should be acknowledged. For example. if a single stream of data is received by several processors, before any one packet can be acknowledged all the packets sent prior to the packet must have been received. Without the use of some global memory, the individual processor may not know if all the packets prior to its packet have been received, so an acknowledgment cannot be sent. If on the other hand each processor receives a separate stream of data. it knows what packets it has and has not received for that stream and can act accordingly without communicating with the other receiving processors. A third alternative would be to provide a separate processor responsible for the generation of acknowledgments. and have each of the receiving transport processors report to it when they receive a packet. This solution simplifies the acknowledgment problem. but

Attribute	Options
Acknowledgment Generation	Receiving R_i/ Central Processor C_r
Number of Streams	Single shared, Multiple to be Merged
Reordering Storage	Global, Local to one Processor

Table 3.5: Reordering and Acknowledgment Generation

could require that a significant global memory is available to store all of the packets as they are received. Because all protocol processors must in some way access the global memory, memory access may be a performance bottleneck.

Reordering of Data

Once data have been received, they need to be ordered correctly before they can be passed to the application layer. This can be accomplished by using a single queue processor as suggested by Jain et al. in [19], or by having a single processor poll the receiving processors for complete segments and selecting the lowest available one to pass to the application layer. These solutions, while not the only ones, demonstrate the dependence of the issues at the transport layer. The first solution selected by Jain assumes that all the receiving processors are working on a single stream of data and that they do not build complete segments themselves. The second solution assume that each processor is working on a separate data stream and can store a complete segment locally before passing it to the application layer. Additionally, both of these solutions require completely different supporting hardware. In the former case a large global memory is prescribed, in the later, a significant local memory must be available at each of the protocol processors and a separate processor must be available to poll for the next segment to be passed. The later solution could also implement this polling function as a distributed process among the transport processors.

3.3.3 Network Layer

In addition to breaking data blocks into data segments and scheduling these segments to transport processors, packets generated by the transport layer need to be scheduled to network layer processors. As with the transport and application layers, the number of processors assigned to network layer processing may be different than the number assigned to other layers. As discussed earlier, Gitlin [15] showed that the network layer represents a factor of five reduction in available bandwidth from that offered at the media level, so extracting speedups from it would be beneficial.

Parallel network layer processing introduces new problems and issues over those in serial communication systems. Some of these problems, shown in Table 3.6, are similar to those of the transport layer. As with the scheduling between the application and transport layer, the scheduler between the transport and network layer can reside either in the transport layer, the network layer, or in a processor between the two layers. Additionally, media access processors must be scheduled to network processors. As with the scheduling mechanism discussed for the transport layer, the algorithms can be simple FCFS or Round Robin, or they can incorporate information about system and component performance to attempt to increase throughput. Unlike the transport layer, reliability is not guaranteed for the network layer, so the issues of acknowledgments and retransmission do not need to be addressed. Also, data can be delivered to the transport layer in the order it was received by the network layer, so there is no need to buffer incoming data for resequencing. However, if transport layer packets are fragmented at the network layer and multiple paths are used to send a single transport packet, it is important that the complete transport packet is delivered to one transport processor on the receiving end. If this is not done, many packets could be successfully sent, but thought to be lost because several transport processors are holding a small part of each packet. This implies that the fragments must be reassembled at the receiving network layer. This can be done by requiring that all network packets from a

Task	Options
Scheduling	Simple, Adaptive
Fragmentation	Yes or No
Recombination	Single Network Layer Processors, Distributed
Addressing	Single Address, Multiple Addresses

Table 3.6: Parallel Network Layer Problems

single transport packet be sent to the same network processor on the receiving side, or that the fragments forming a single transport packet be address to a particular receiving transport processor. In the later case, some mechanism to resequence the single transport packet must also be provided.

If the transport layer cannot accept data from a network layer processor, the data can simply be discarded. This means that the problems of timer and acknowledgment maintenance, and buffer memory are not imposed at this layer. Interprocessor communication may still be needed however, because the incoming data packets need to be scheduled to the available network layer processors, and the network layer processor may want to communicate to determine the best route to transmit a particular packet.

3.3.4 Media Access Layer

Multiple media access processors or controllers can be used to provide high throughput by aggregating the throughput of each. Parallelism at this layer is significantly easier than at the network and above layers because not only is there no concern for proper delivery and reordering of data, any fragmentation that occurs at this layer is handled at the receiving end by the receiving media access controller. In many computer systems multiple media access controllers are already in used, but no real parallelism can occur because normally only one is processor performing the communication protocol processing, and it can only use one media access controller at a time. As the throughput of the higher layers increase, the data are more quickly available to the network layer and, hence, to the media access controllers. Multiple media access controllers can then be driven by the parallel network layer. This is not meant to imply that the efficient use of multiple media access controllers is a trivial problem. Mukkamala et al. in [26], however, showed improvement in throughput when multiple FDDI channels were provided.

3.3.5 Physical Media

Parallelism at the physical media, that is the use of multiple paths concurrently to deliver more data, is not a new idea. The telephone system has used this to perform congestion control and as well as to provide the needed bandwidth through major cities [17].

Parallelism at the media level also has been used to add fault tolerance and bandwidth. In addition to the telephone network efforts mentioned above, the FDDI network/protocol contains two counter rotating rings that are used to improve fault tolerance. The use of the second ring for parallel traffic is a possibility, but has not yet been added to the standard. In 1987 an Ethernet compatible network using counter-rotating fiber optic rings was developed. This technology is currently in use for fault tolerance, but, as with FDDI, could be used to provide parallel communication paths [34, 35].

An additional view of parallelism at the media level is to use multiple slower media channels driven by several faster media access controllers and a multiplexor to match the output of the media access controllers to the speed of the aggregation of slower media.

3.3.6 Parallelism in the Complete Communication System

The opportunities for parallelism in both protocol layers and schedulers are shown in Figure 3.3. A parallel communication system may implement fewer parallel layers and schedulers depending on available components, cost considerations, and performance needs. The blocks outlined with solid lines are actual layers in communication systems. The dashed line boxes represent the scheduling functions that must interface the various layers. As discussed above, these are not actual layers, but rather logical layers whose functions can be placed in any of three places.

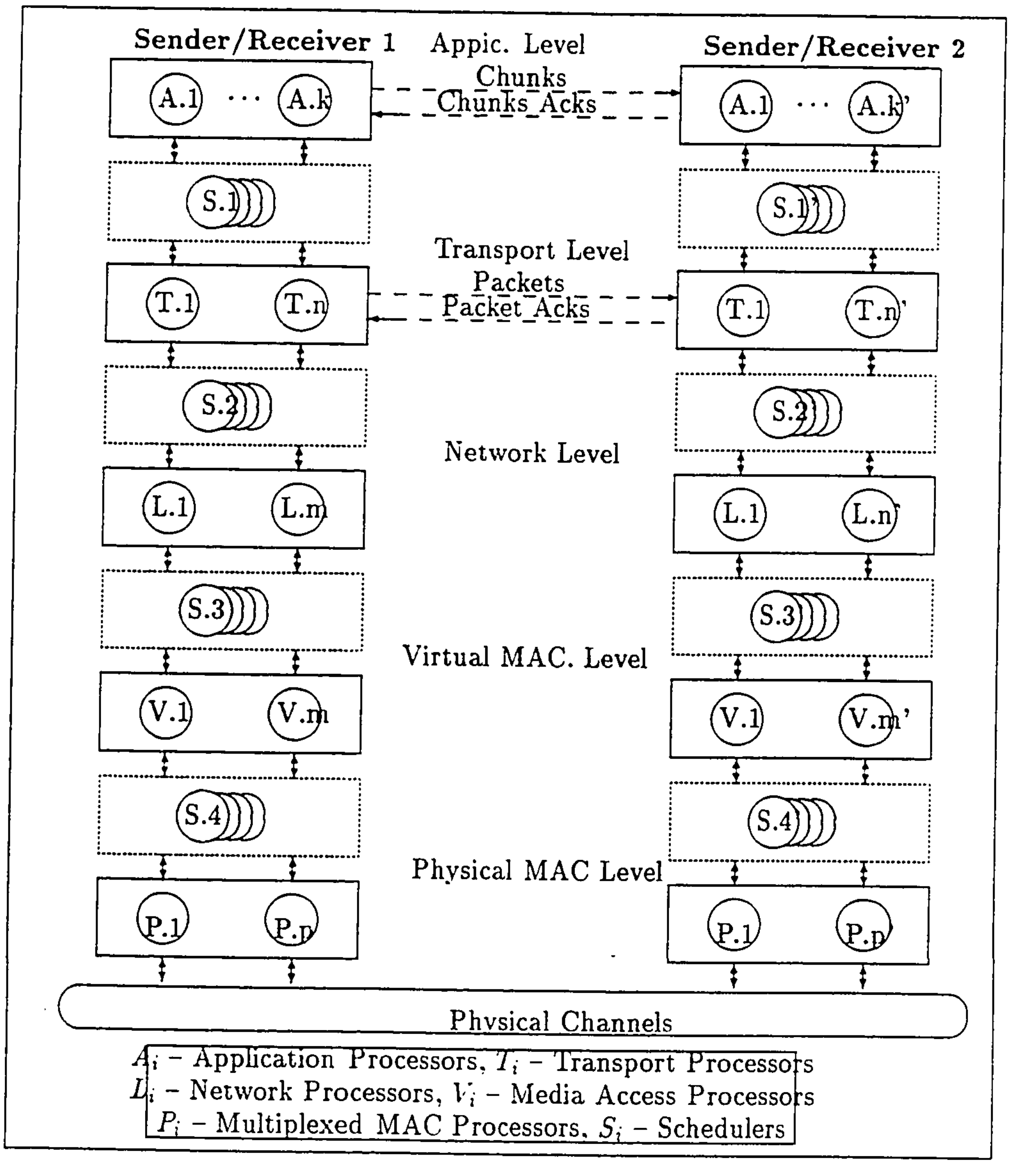

Figure 3.3: Levels and Scheduling

Many of the opportunities and problems with parallelism at each of the lower 5 levels of a traditional ISO communication system have been discussed. Note that, while parallelism is conceptually possible at all levels, it may not be needed or useful at some levels. Additionally, the degree of parallelism provided at each level need not be the same. That is if transport processing takes twice the time of network processing, it might be sufficient to provide X processors at the transport layer, but only X/2 processors at the network layer. The various schedulers discussed above can be used to match the different degrees of parallelism at each level.

3.4 Sensitivity Analysis

The options presented in the previous section could be used to build a large number of different implementations of parallel communication systems. Some approaches would provide increases in performance, depending on the underlying hardware configuration, and some would not. As a quick first look to determine which seem more promising, a sensitivity analysis is being performed.

3.5 Description of System Components

The system being considered, shown in Figure 3.4, is based on the Sun MP architecture, where multiple Sparc processors, each equipped with their own cache, are connected to a common high speed bus. The only memory on the system is global and is also attached to the common bus. In this system, one Mbus module is dedicated to performing application processing. The remaining processors are all performing transport and network layer processing. As is discussed below, all scheduling of data to processors is assumed to occur within the protocol processors.

Component	Performance Rating	Confidence
Processor Speed (PS)	20 MIPs	80%
Cache access time (CAT)	10 ns	40%
IU Bus speed (IBS)	100 Mbytes/sec	80%
Mbus to Sbus speed (MSI)	80 Mbytes/sec	70%
IU to Mbus speed (IMI)	100 Mbytes/sec	80%
Sbus speed (SBS)	80 Mbytes/sec	90%
Mbus speed (MBS)	320 Mbytes/sec	90%
Mbus Arbiter (MA)	50 ns/request	75%
System memory access time (SMAT)	70 ns	80%
Mbus to Sbus speed (MSS)	80 Mbytes/sec	90%
FDDI speed (FS)	100 Mbytes/sec	100%

Table 3.7: Expected Component Performance

as operating system actions, making the availability and load of each component difficult to assess. In this table, I assume 5 processors, 4 performing communication functions and one performing application processing, and three FDDI controllers.

3.7 Analysis Definitions

The values for component utilization are computed as follows -

- Application Processor
  - IU – Must execute IPBO instructions for each byte in the image. The time for this equals total instructions executed divided by the speed of the processor. This number is then divided by the time available to compute each image to obtain percentage of utilization.

$$\frac{(IPBO * IS)/(PS * 10^6)}{(FOI/1000) * 100}$$

  - IU bus – Must carry each of the input for each byte output as well as the output bytes of each image. The instructions that are required to compute each byte

Component	Performance Rating	Confidence
Processor Speed (PS)	20 MIPs	80%
Cache access time (CAT)	10 ns	40%
IU Bus speed (IBS)	100 Mbytes/sec	80%
Mbus to Sbus speed (MSI)	80 Mbytes/sec	70%
IU to Mbus speed (IMI)	100 Mbytes/sec	80%
Sbus speed (SBS)	80 Mbytes/sec	90%
Mbus speed (MBS)	320 Mbytes/sec	90%
Mbus Arbiter (MA)	50 ns/request	75%
System memory access time (SMAT)	70 ns	80%
Mbus to Sbus speed (MSS)	80 Mbytes/sec	90%
FDDI speed (FS)	100 Mbytes/sec	100%

Table 3.7: Expected Component Performance

as operating system actions, making the availability and load of each component difficult to assess. In this table, I assume 5 processors, 4 performing communication functions and one performing application processing, and three FDDI controllers.

3.7 Analysis Definitions

The values for component utilization are computed as follows -

- Application Processor
  - IU - Must execute IPBO instructions for each byte in the image. The time for this equals total instructions executed divided by the speed of the processor. This number is then divided by the time available to compute each image to obtain percentage of utilization.

$$\frac{(IPBO * IS)/(PS * 10^6)}{(FOI/1000) * 100}$$

  - IU bus - Must carry each of the input for each byte output as well as the output bytes of each image. The instructions that are required to compute each byte

Processing Type	Component	Processing Requirement	Confidence
Application			
	Bytes In per byte out (BIPO)	5	60%
	Instr. per byte out (IPBO)	10	60%
	Image Size (IS)	1 Mbyte	90%
	Frequency of images (FOI)	100 ms	60%
	Segment Size (SS)	10 K bytes	60%
Protocol operations			
	Num. of instructions (NOI) (for both scheduling + TCP/IP)	2048	60%
	Cache hit rate (CHR)	80%	40%
	Bytes for FDDI check (FC)	4	60%
	Packet Size (PKS)	2 K bytes	80%
	Data accesses/packet (DAPP)	3 Bytes	60%
	Packet header size (PHS)	24 Bytes	90%

Table 3.8: Expected Resource Demands

Component	Item	Utilization
Appli. Processor		
	IU	525%
	IU bus	210%
	IU to Mbus interface	30%
	Cache Memory	160%
Trans. Processor		
	IU	1 %
	IU bus	10%
	IU to Mbus interface	5%
	Cache Memory	5%
System Level		
	Mbus	20%
	Mbus Arbiter	15%
	System Memory	300%
	Mbus to Sbus inter.	15%
	FDDI	30%

Table 3.9: Component Utilization - Baseline Assumptions

are either in cache, or system memory. If they reside in cache. they just traverse the IU bus once. If they reside in the system memory. they are first placed in the cache, and then sent to the IU. The total of these bytes for each image is then divided by the throughput of the IU bus and normalized to the allowable time for each image.

$$\frac{(BIPO + 1 + IPBO + 2 * IPBO * (1 - CHR) * IS}{FOI * 10^6/(FOI/1000) * 100}$$

– IU to Mbus Interface – For the application processor. the complete image must traverse this connection, as well as the instructions that do not reside in cache. The total number of bytes that must traverse the IU to Mbus interface is then divided by the speed of the interface and the number is normalized as before.

$$\frac{(IS + (1 - CHR) * IPBO * IS)/IMI * 10^6}{FOI * 10^6/(FOI/1000) * 100}$$

– Cache Memory – All instructions executed for each byte of the image must come from the cache memory. The instructions that are not in cache. i.e. a cache miss. must also first be placed in cache and then accessed from cache and sent to the application IU. As with the instructions, each of the input bytes are first looked for in cache and then, if not found there, gotten from the system memory and placed in the cache. The sum of the accesses to cache for each image is then multiplied by the speed of the cache memory and the number is normalized to the image time.

$$\frac{(IPBO * IS + BIPO * IS + (1 - CHR) * 2 * IS) * CAT * 10^{-9}}{FOI * 10^6/(FOI/1000) * 100}$$

- Protocol Processor

The work of the protocol processors is divided equally among all processors, so to make the formula below easier to read, the following new variables are defined.

- NOP – Number of processors
- NOS – Number of segments in an image for which each processor will be individually responsible.

  $$NOS = (IS/NOP)/SS$$

- POS – Number of packets per segment

  $$POS = SS/PKS$$

- SIP – The data each image for which a processor is responsible

  $$SIP = IS/NOP$$

- POI – Packets per images for which each processor is individually responsible

  $$POI = NOS * POS$$

- TPD – The total packet data sent by each processor, of each image sent

  $$TPD = (PKS + PHS) * POI$$

- NOF – The number of FDDI controllers available

With these definitions, we can define the processing responsibilities of a protocol processor's components.

- IU – For each of the packets that a protocol IU is responsible for, it must execute NOI instructions. This instruction count is divided by the speed of the processor and then normalized to the time available for each image to find a usage percentage.

  $$\frac{(POI * NOI)/PS * 1^{-6}}{FOI * 10^{6}/(FOI/1000) * 100}$$

- IU bus – The IU bus in each communication processor carries that processor's share of the image being transmitted to the processor for checksum calculation, the completed packets sent by the module, the acknowledgments for the packets sent, the instructions needed to process each packet, and bytes needed to select an FDDI channel for transmission. The acknowledgment is assumed to only be the size of a packet header. As before, the instructions that are not found in cache must first be placed there from the system memory, so they must traverse the bus twice. The total bus traffic is divided by the speed of the bus and then normalized to the time allowed for each image.

$$\frac{SIP + TPD + POI * (PHS + FC) + (1 + 2 * (1 - CHR)) * NOI * POI + DAPP * POI}{FOI * 10^6/(FOI/1000) * 100}$$

- IU to Mbus interface – Each processor's share of the image must travel through this interface to be assembled into packets. After the packets have been built they must travel through on their way to the FDDI controllers. The acknowledgments must also traverse this interface as must the instructions that are needed to built the packets. Finally, the FDDI check bytes (FC) must also traverse the connection. All of these data are summed and divided by the speed of the bus. As before this number is then normalized to the time allowed for each image.

$$\frac{(SIP + TPD + POI * (PHS + FC) + NOI * (1 - CHR) * POI)/IMI * 10^6}{FOI * 10^6/(FOI/1000) * 100}$$

- Cache memory – The cache memory on the communication processor is accessed for each instruction executed for each packet. The instructions that are not currently located in the cache memory require two accesses, once to place them in the cache and once to retrieve them for the IU. The total number of accesses is multiplied by the access time for the cache to determine the total time required for cache processing. This number is then normalized as before.

$$\frac{(CHR * NOI + (1 - CHR) * 2 * NOI) * POI * CAT * 10^{-9}}{FOI * 10^6/(FOI/1000) * 100}$$

- System Components

These components are used by both the application and the protocol processors. As with the definitions of the protocol processor component's utilization, the new variables defined above will be used.

  - Mbus – This bus must carry all of the traffic that passes through the UI to Mbus interfaces for both the application processor and the protocol processors. This includes the passing of the image from the application processor to system memory, the passing of image segments to the protocol processors, and the passing of packets to the FDDI controllers. Additionally all of the instructions not found in the caches of processors comes from system memory so they must traverse this bus as well. Lastly the FDDI check bytes (FC) and responses, and the packet acknowledgments must also traverse the bus.

    This gives the total utilization of the Mbus for data to be

    $$DATA = \frac{(2 * IS + (PKS + 2 * PHS + 2 * FC) * POI * NOP)/MBS * 10^6}{FOI * 10^6/(FOI/1000) * 100}$$

    and the total utilization for instructions as

    $$INSTR = \frac{((NOI * POI * NOP + IPBO * IS) * (1 - CHR))/MBS * 10^6}{FOI * 10^6/(FOI/1000) * 100}$$

    Making the total Mbus utilization equal to

    $$DATA + INSTR$$

  - Mbus Arbiter – Every time a request is made to send data across the Mbus, the Mbus arbiter must grant access privilege to the requester. This occurs once for every image, once for every instruction miss of either the application processor or protocol processor, once for every segment, every packet, every packet acknowledgment, and every FDDI check. The total number of requests for each image are totaled and multiplied by the time to service each request. This number is the normalized to the time available. It is noted that this formula assume that once access to the bus has been granted, the requester owns the bus until it releases the bus. If this is not the case, the formula would have to be changed to reflect the maximum amount of data that could be transferred for each request.

$$\frac{(1 + IS/SS + 4 * NOP * POI + (POI * NOP + IS) * (1 - CHR)) * MA * 10^{-9}}{FOI * 10^6/(FOI/1000) * 100}$$

– System Memory – The system memory is used to store the images being produced as well as the instructions required to produce them and to transmit them. The total number of access to the system memory is multiplied by the access time of the memory to determine the total time needed for the memory accesses of each image. Again, normalization is performed.

$$\frac{(2 * IS + POI * NOP * 3 + (NOI * POI + IPBO * IS) * (1 - CHR)) * SMAT * 10^{-9}}{FOI * 10^6/(FOI/1000) * 100}$$

– Mbus to Sbus Interface – The packetized data as well the acknowledgments and the FDDI checks must pass through this interface. The total amount of data that passes through is divided by the available bandwidth and normalized.

$$\frac{(PKS + 2 * FC + 2 * PHS) * POI * NOP/MSI * 10^6}{FOI * 10^6/(FOI/1000) * 100}$$

– FDDI – As with the protocol processors, the FDDI interfaces share the load of transmitting the complete images. The only data that must pass through an interface is it's share of the image packets and the corresponding acknowledgments. Because in this study all controllers are given even loads, simply dividing the total load generated by the system by the number of controllers, and using the bandwidth of the controller and the time allow for each image, gives the percentage utilization for each of the controllers. Unlike the other formula, in this case we need to convert from the byte representation used for other system bandwidth values to the bit representation used to describe FDDI's capacity.

$$\frac{POI * NOP * 8 * (PKS * 1024 + 2 * PHS)/NOF/FS * 10^6}{FOI * 10^6/(FOI/1000) * 100}$$

3.8 The Analysis

Because the loads and performance characteristics are educated guess rather than hard numbers, the exact performance of the system is not accurately know from the above calculations. For this reason, in all of the following tables the values have been rounded up or down to the nearest 5%.

As shown in Table 3.9, the demands on some the system components are beyond their capabilities even with 100% of their capacity dedicated to the application and communication processes of this study. An obvious problem occurs in the application processor which has a utilization factor of over 500%. This problem, however, is not of concern for the communication processes, and can be ignored. What is of concern is the utilization of the system memory. Even with base performance assumptions and the removal of the application processor's use, the load would still be approximately 151%. As discussed in an earlier chapter, multiple independent banks of memory might be used to address this problem, but that is a limited solution. Providing faster memory is a possible solution, but this must be weighed against its cost.

The values shown in Table 3.9 assume that 100% of the capacity of each of the components are available for the communication task. The values given for the bus speeds, however, are peak speeds rather than sustained. If the speeds of the Mbus, Sbus, and IU bus are lowered to their sustained speeds, one fourth of peak, the results shown in Table 3.10. These results show that the application processor module will not keep up, but the communication and system components, other than memory speed, are still capable of delivering the data.

Because transport layer processing is the known bottleneck in reliable communication

Component	Item	Utilization
Appli. Processor		
	IU	525%
	IU bus	840%
	IU to Mbus interface	125%
	Cache Memory	160%
Trans. Processor		
	IU	1%
	IU bus	35%
	IU to Mbus interface	25%
	Cache Memory	5%
System Level		
	Mbus	70%
	Mbus Arbiter	15%
	System Memory	295%
	Mbus to Sbus inter.	55%
	FDDI	30%

Table 3.10: Component Utilization – Sustained Bus Speeds

processing, it might be useful to increase the number of instructions required for each packet in order to emulate a high processing demand. On a 12 MIP SparcStation, the peak throughput achieved was 6 megabits. If the 2K byte packet size it assumed, we can compute an estimate of the number of instructions executed per packet.

12 MIPs/6*10^6 bits = 2 instructions/bit

2*2048*8 = 32768 instructions/packet

Of course these numbers do not take into account that a significant amount of time is spent waiting for acknowledgments and window movement. The number of instructions used in the calculation, however, simply represent the time taken to process the data, whether the IU is processing, or being forced to wait. It is tempting to round the number of instructions up because the processors also spend time scheduling segments and FDDI interfaces, however, as Table 3.11 shows even without this rounding the IU bus in the protocol processing module is already over subscribed. Additional problems are also now apparent with the speed of the Mbus as it's utilization is now over 100%. The problems with

Component	Item	Utilization
Appli. Processor		
	IU	525%
	IU bus	840%
	IU to Mbus interface	125%
	Cache Memory	160%
Trans. Processor		
	IU	1%
	IU bus	255%
	IU to Mbus interface	55%
	Cache Memory	50%
System Level		
	Mbus	110%
	Mbus Arbiter	15%
	System Memory	350%
	Mbus to Sbus inter.	55%
	FDDI	30%

Table 3.11: Component Utilization - Realistic Instruction Counts

the system memory are even worse. It is apparent that for any meaning full detailed results to be obtained, a more indepth analysis of TCP/IP processing will need to be done. For this sensitivity analysis, however, the above assumptions will suffice. This table maintains the assumption that only one fourth of the specified bus speed will be usable.

Finally, the cache hit rate should be examined. The initial hit rate was 80%. This number is very optimistic, and probably should be reduced. If reduced to 25%, a more reasonable assumption, the problem of the IU bus on the protocol processor gets worse, and a new problem is seen for the UI to Mbus interface. The problems of system system Mbus and memory are worse as well. The results are shown in Table 3.12. As before, the previous assumptions about bus utilization and instruction counts are maintained in this table.

These numbers make the prospect of a feasible parallel communication system with this architecture look dim, but by adding local memory or cascading caches to each of the processor modules, the utilization numbers are made much better. The values in Table 3.13 assume a 100% cache hit rate. This make the amount of traffic across the IU and M buses

Component	Item	Utilization
Appli. Processor		
	IU	525%
	IU bus	1300%
	IU to Mbus interface	355%
	Cache Memory	175%
Trans. Processor		
	IU	1%
	IU bus	440%
	IU to Mbus interface	150%
	Cache Memory	75%
System Level		
	Mbus	295%
	Mbus Arbiter	45%
	System Memory	920%
	Mbus to Sbus inter.	55%
	FDDI	30%

Table 3.12: Component Utilization – Realistic Cache Hit Rate

reduce significantly. Unfortunately, the utilization of the IU bus on the protocol processor modules is still too high. The previous assumptions where still in effect when computing this table.

Consider this utilization in the light of the number of protocol processors available. If the number of processors increases, the amount of data processed by each will decrease, meaning the amount of data and number of instructions that traverse the IU bus will also be reduced. Table 3.14 shows the values obtained when the number of processors is increased to eight. The IU bus utilization is reduced to less than 100%, meaning the protocol processors can keep up. The number of processors was doubled and the load on the IU bus was halved, indicating a linear speedup. This is explainable for the IU bus because no interaction between the local IU bus of one processing module and any other system component occurs. The same linear speedup probably would not be seen for the entire system, rather just for the protocol processing modules. As before, the previous assumptions as to bus use, instruction count, etc., are still used when computing this table.

Component	Item	Utilization
Appli. Processor		
	IU	525%
	IU bus	670%
	IU to Mbus interface	40%
	Cache Memory	155%
Trans. Processor		
	IU	1%
	IU bus	190%
	IU to Mbus interface	20%
	Cache Memory	40%
System Level		
	Mbus	40%
	Mbus Arbiter	5%
	System Memory	145%
	Mbus to Sbus inter.	55%
	FDDI	30%

Table 3.13: Component Utilization – With Local Memory

Component	Item	Utilization
Appli. Processor		
	IU	525%
	IU bus	670%
	IU to Mbus interface	40%
	Cache Memory	160%
Trans. Processor		
	IU	1%
	IU bus	95%
	IU to Mbus interface	10%
	Cache Memory	20%
System Level		
	Mbus	40%
	Mbus Arbiter	5%
	System Memory	145%
	Mbus to Sbus inter.	55%
	FDDI	30%

Table 3.14: Component Utilization – With Eight Processors

Component	Item	Utilization
Appli. Processor		
	IU	525%
	IU bus	670%
	IU to Mbus interface	40%
	Cache Memory	160%
Trans. Processor		
	IU	1%
	IU bus	95%
	IU to Mbus interface	10%
	Cache Memory	20%
System Level		
	Mbus	40%
	Mbus Arbiter	5%
	System Memory	85%
	Mbus to Sbus inter.	55%
	FDDI	30%

Table 3.15: Component Utilization – With Fast Memory

While increasing the number of communication processors to eight lowered the load on the IU buses and made it possible for the processors to keep up. there is still a problem with the global memory. In the previous table, each communication processor was equipped with local memory that could be used for instruction storage. All the application data. however. must still pass through the global memory, so its performance is critical. In table 3.15, the access time of the global memory was reduced to 40ns. This results in a working system from a communication point of view, but the use of 40ns memory is expensive and may not be available on most computing platforms. Again. none of the previous assumptions where excluded when computing the values for this table.

3.9 Discussion

The purpose of this analysis was to provide pointers to architectures and issues that warrant further analysis. The results indicate that parallel communication is feasible. but special attention needs to be given to actual component performance. In the baseline performance table, is appeared that, other than the speed of memory, the system was workable with no local memory. As the analysis was refined with more realistic assumptions about component loads and utilization, it became evident that a large global memory would not allow communication to occur anywhere near the desired speed of 80 Mbits/second. This problem was solved by including a local memory at each of the communication processors that could be used for instruction storage.

Even though, as shown by Clark, four communication processors was more than enough computing power for gigabit communication, the connecting buses and cache memory hardware proved to limit the throughput to less than the target 80 Mbits/second this application needed.

Memory speed was shown to have a dominate impact on the speed of communication possible. The assumptions used in this analysis are valid for commonly available memory. Because the memory speeds are unlikely to be reduced by orders of magnitude, as will be needed for gigabit communication, memory access and placement will be important issues in parallel communication.

Chapter 4

Experimental Results

The previous chapter gave initial insight into some of the problem areas and more promising approaches to parallel communication. The analysis presented, however, made many assumptions about workloads, and could only give top level insight into the problems that may be encountered when implementing a parallel communication system This chapter describes a testbed environment developed to gain that insight and used to collect experimental performance data from one approach to parallel protocol processing. Specifically this testbed is used to collect performance results from a system that employs multiple processors to perform the transport and network protocol layers. One goal of this study is to determine the communication speedups attainable with existing hardware and software standards. To achieve this goal, existing hardware was reconfigured and software was replicated to realize a parallel environment.

4.1 Scope and Objective

The objective of this study is to collect data to show possible performance gains and overhead costs incurred when performing protocol processing in parallel. This information allows one to predict if a given hardware architecture will be able to provide the required service, to more easily predict the amount of hardware needed to perform a particular communications task, and to know when adding more hardware will no longer be beneficial. Different application classes have different communication needs, including the need for resequencing, the frequency and size of transmissions, and acceptable level of latency and reliability. Each of these needs to be considered when developing a parallel communication system.

Remote visualization was selected as the target application for the testbed environment. As discussed previously this application requires large blocks of data to be sent in regular intervals and data latency is an issue. The testbed takes advantage of the suggestion by Clark in [6] discussed earlier and removes the requirement for data reordering from the transport layer. This task could be placed in the application itself, but for remote visualization the function is not required. Windows are still employed in the transport layer, so lost data will cause the system to wait and time out. Because the "wait and time out" process has been identified as a major bandwidth cost its inclusion in this study is appropriate.

In order to maintain the goal of identifying performance gains possible with existing hardware, and to develop a testbed to measure performance and overhead results quickly, the testbed was implemented with commonly available processors, networks, and software wherever possible. Custom software was needed to provide parallel communication, and to measure the performance gains achieved. An important consideration in developing this testbed was to insure that the components selected did not introduce any new bottlenecks by reducing any capability previously provided.

The selected processors for the testbed were Sun SparcStations. These machines were attractive because of the ease in setting up, reconfiguring, and tearing down the testbed. Additionally, the transport layer source code was easily accessible to help in analysis the results, and several machines were available for me to use. The communication processors selected were Ethernet interface cards. These cards are readily available and more than one can be added to a SparcStation. Additionally, the TCP/IP software provided with Sun workstations in SunOS 4.0 has been optimized for Ethernet networks, making it ideal. The transport processing performance for TCP/IP in the selected processors is able to produce approximately 6 Mbits/second under ideal circumstances on a single channel. Because there were only two machines on each network, the Ethernets could theoretically be driven at 10 Mbits, so they did not pose a new performance limitation. A software interface, called Parallel TCP/IP, was developed and placed between the application and the individual TCP connections. This software enabled the emulation of a parallel communication system in which individual physical processors communicate through a common bus, each having a significant amount of local cache, and sharing a common memory. It is noted, however, that the purpose of this study is not to show that Ethernet was better than token ring in a parallel environment, nor that parallel Ethernet is superior FDDI, but rather to determine what the costs are for using parallel TCP/IP. Ethernet is simply the least expensive easily obtainable interface available. FDDI would have been a more interesting choice, but at the time the testbed was being developed, it was not available.

4.2 Design of the Case Study

4.2.1 Testbed Description

Figure 4.1 shows the components required to perform parallel communication using the testbed from a process point of view. While bottlenecks can occur at several points, the one addressed here is transport layer processing. In order to provide parallelism, multiple processors working on the communication process concurrently are needed. Since these processors are used to provide service to a single data stream, some scheduling of data to the processors is required. The use of a single processor to perform the scheduling operations

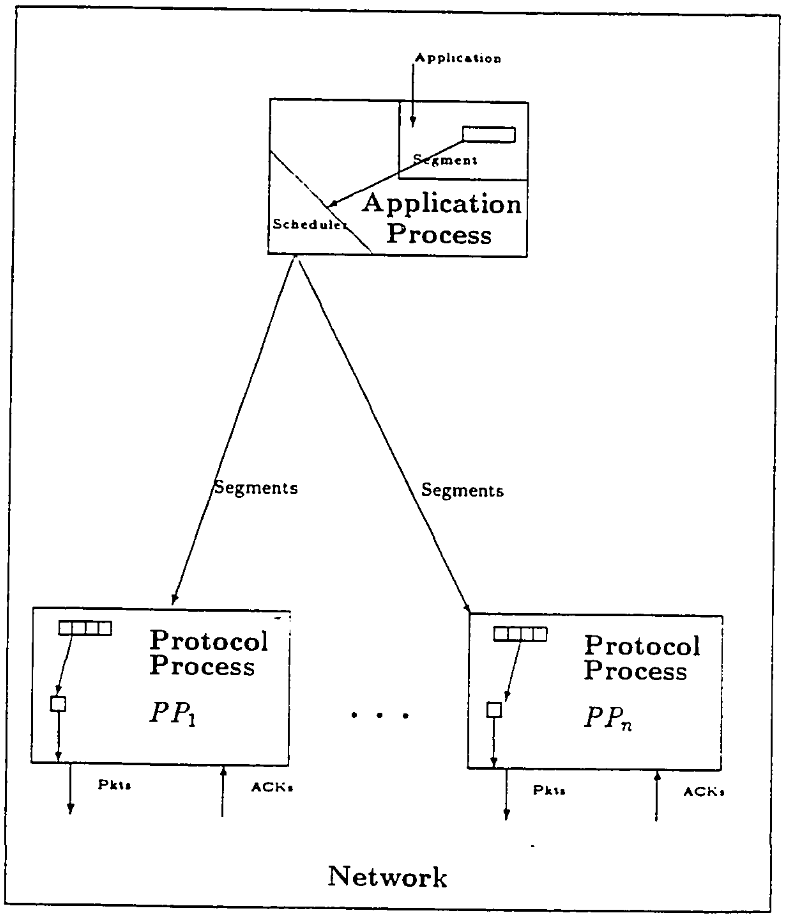

Figure 4.1: Scheduling and Control - Application — Protocol Process

in a centralized manner, or a distributed scheduling mechanism, allowing the transport processors to schedule themselves are both viable. Finally, the protocol processors need a way to place data on the final medium for transmission. This can take the form of one very high speed media interface placing data on a single high speed medium, or several slower interfaces placing data on separate lower speed media.

Figure 4.2 shows the testbed with processes allocated to physical processors. The scheduling process is handled by the node marked *Master*. The processors marked *Slave* perform transport layer processing. Each slave processor is connected to its master by a separate network interface and does not need to communicate with other processors. However, because The master and slave processors were connected via Ethernet, and communication could not begin before all data were transmitted from the master to slaves, it is impossible for the master to send the actual data to each of the slaves without introducing an artificial bottleneck into the testbed. This means that for parallel transmission of data to occur, all data to be transmitted had to be pregenerated and prepositioned on the local disks at the sending slaves. During each run of the testbed, the master processor sent the scheduling control messages to the slave processors, indicating which data to transmit. With this approach, the delay required of a control message to travel from master to slave processors more closely approximates the delay that would be seen on an operational implementation which uses a high speed, common bus connecting the processors and a global memory. A symmetric bottleneck occurs at the receiving end of the testbed, so data were also prepositioned at the receiving master. During all experiments, data were actually transmitted from the sending to the receiving slaves; however, they were discarded when received. Figure 4.2 shows the logical control and data lines that would be needed for an operational implementation. The lines marked "Control Path" have been implemented in the testbed, but, as described above, the lines marked "Data Path" have not. Section 4.4 describes an architecture using multiprocessor workstations with true parallel end-to-end delivery of

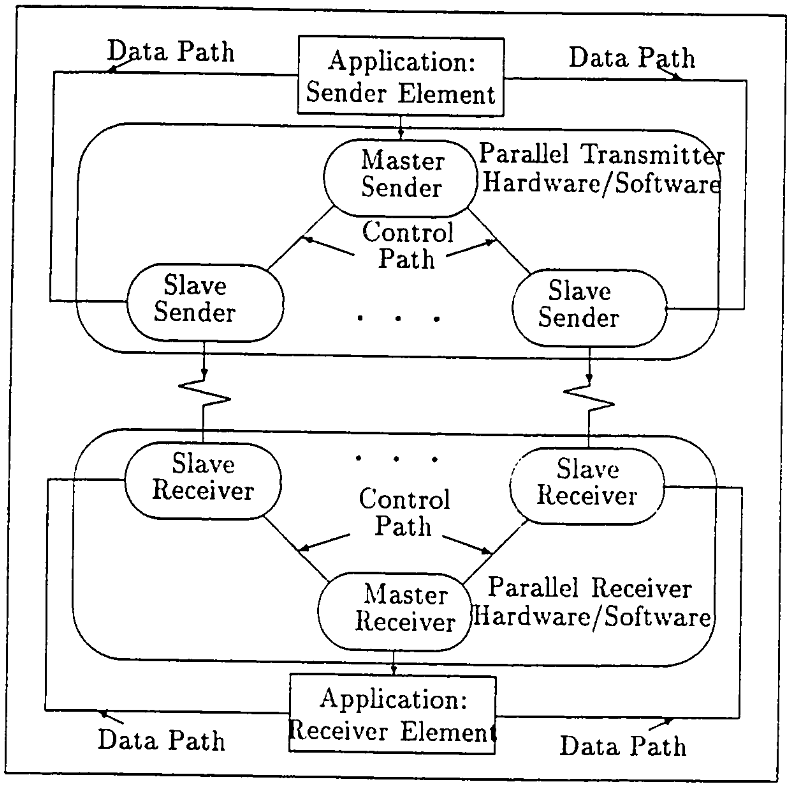

Figure 4.2: End to End System Design

data.

The testbed was configured optionally with two or three channels, requiring six and eight machines respectively. All of the slave machines required two Ethernet cards, and the master machines both required one Ethernet card for each channel in the testbed. By using multiple Ethernet cards each machine to machine connection was a separate physical network, so the operation of one slave processor could not affect the amount of bandwidth available to another. Additionally, no other computers were attached to any of the physical networks (except as noted in the 'background load' section below). This was done to insure that no uncontrolled factors influenced measured results.

Packet Processing in the Testbed

When the master process receives a transmission request from the application, it divides the identified data buffer into multiple segments, based on slave segment size, and issues a send command to each slave, giving a buffer segment number and a buffer segment offset. The segment given to slaves for transmission are normally much larger than the underlying transport layer protocol packet size so that a slave process can generate several packets before going back to the master for more data. After the send command is issued to the slaves, control is returned to the application. The slave processes, running in parallel on separate physical processors, transmit all of their data and then report back to the master for more when finished.

When data are received from the transmitting slaves, the data are discarded and a received segment number is sent to the receive master. When the master determines that an image has been complete received, either by an end-of-image signal or a transmission-abort signal, it makes its local copy of the data available to the receiving application.

The transmitting master is responsible for managing the connection of new slaves, assigning segments to slaves when they are ready, and providing graceful degradation if channels fail. The services provided can be thought of as operating system communications functions that can be used by more than one user process concurrently. The software to perform the sending and receiving, master and slave tasks required approximately 1100 lines of code. A detailed explanation of the software operations is given in appendix A.

4.2.2 The Experiment – Parameters and Environment

To measure the performance and costs associated with parallel TCP/IP, throughput was measured and plotted as a function of the number of physical channels (which equals the number of transport processors). The theoretical limit of a linear speedup is plotted as well, for comparison. Clearly, throughput and costs observed depend on several parameters and network conditions.

- TCP/IP Parameters

  In the testbed, each of the transmitting and receiving slave processors maintain transport connections in pairs. The standard TCP/IP as delivered by Sun is tuned for the local area network used in the testbed so none of its default parameters, relating packet size, timer length, or window size were changed.

- Size of Data Segment Given to Slave Processors

  Because the overhead of communication between a slave processor and master processor introduces a possible bottleneck, or at the very least a delay in transport processing, a critical parameter is the size of the data given to the slave processors each time. When all processors and channels are evenly loaded, the obvious solution is to equally divide the data into as many segments as there are protocol processors, giving each processor only one segment. In unevenly loaded conditions, however, this may result in high latency in data transmission on those channels with heavy background traffic from other sources.

- Background Load

  In a serial network, when background traffic increases on the channel, the total bandwidth is reduced by at least that amount. In a parallel network a similar result occurs, but the routing of data may reduce the amount of degradation seen by the application. Because the testbed employed Ethernet, a completely fair network, adding significant traffic to one channel in a controlled fashion was difficult to achieve because of the limited number of processors available. However, because each transport processor has a dedicated channel, the same result can be achieved by slowing a transport processor. In this study, after sending a packet of data, the transport processor was put to sleep for a period of time to simulate delay in accessing a heavily loaded channel.

- Data Latency

  Because equal segment sizes are being used, it is possible for overall data latency to increase significantly if one channel is slower than others. Another reason for differential latency can be if the physical channels are routes of different lengths. In some applications, such as remote visualization, this is less of an issue because the data do not need to be reordered before being passed to an application. In other applications, if one slow channel holds some packets of data and reordering must be performed by the receiver, total throughput could be significantly reduced. To obtain data on channels with very different latencies, the same approach as used in background traffic is employed.

4.3 Performance

The parameters discussed above have an impact on the network performance. In this study, results concerning the total throughput of the parallel network, the overhead incurred under different loading conditions, channel latencies, the effect of segment size, and an indicator of when adding more channels results in minimal performance gains were of interest.

Ideally if no additional overhead is seen and the system is evenly loaded with uniform latency on all channels, a linear increase in speed versus the number of channels should be observed. However, because new work is added to the total processing in order to achieve parallelism this is not the case and, after a point, adding more channels or transport processors will add little to the total throughput.

Data points for Figure 4.3 were obtained with the two and three channel testbed, and are calculated for higher numbers of channels. For details of the calculations, see appendix B. As a baseline, the graph in Figure 4.3 shows the total throughput that could be achieved with the testbed when the optimal segment size is used, no other traffic is on the network, and channels have equal latency. The diagonal line shows the theoretical limit if no overhead

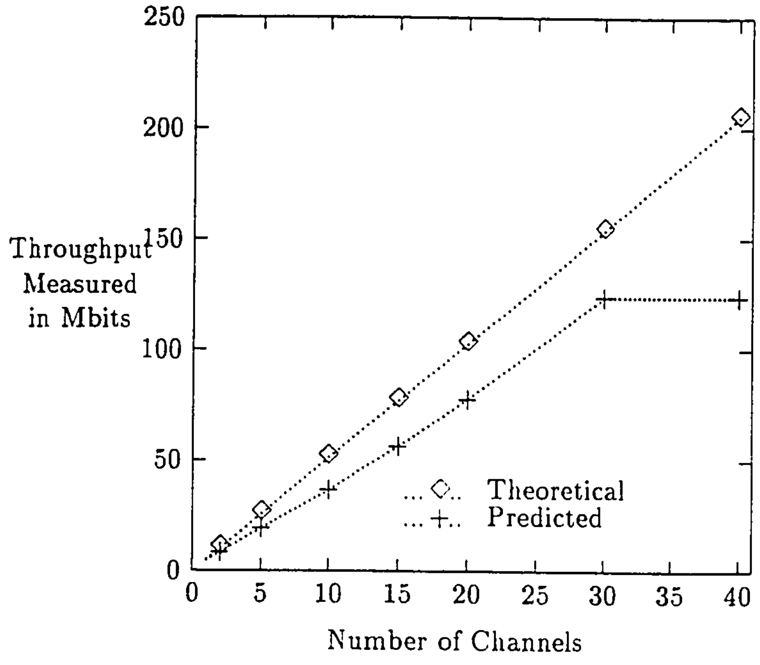

Figure 4.3: Baseline Performance for Testbed

exists. With the testbed, the amount of increase in performance drops off significantly at 30 processors. This should not be interpreted to mean that 30 channels is the limit for parallelism in communication, rather, for the configuration selected to form this testbed, 30 is the limit. Other configurations will have different limits.

Another baseline is shown in Figure 4.4. These data assume a hardware architecture similar to that shown in Figure 4.7, a single bus with sustained throughput of 80 Mbytes/second, and multiple 12 MIP processors each equipped with a local memory and dedicated network interfaces. As before, after a point in the performance curve, adding additional processors has little benefit. For this architecture it occurs at 10 processors. This can be attributed to a new bottleneck occurring at the common bus serving the protocol processors. If more than 15 processor are added, the total throughput actually decreases. This can be attributed to the fact that all of the processor are vying for a common memory, so adding processors beyond what can be served by that memory's speed will result in high contention and lower overall throughput. This architecture is discussed in more detail in section 4.4. The pro-

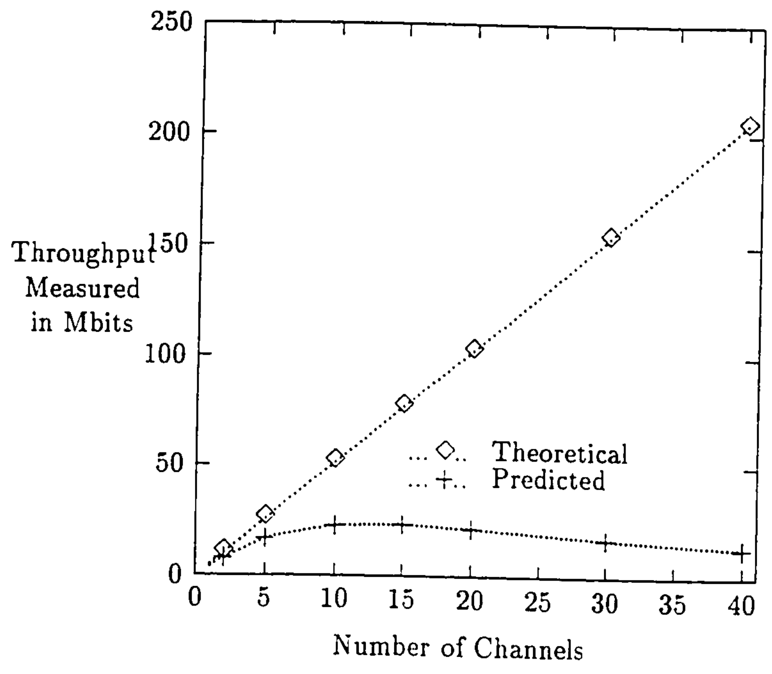

Figure 4.4: Baseline Performance for Multiprocessor Architecture

cessor limit for the testbed is significantly higher than the architecture in Figure 4.7. This is because the testbed more closely approximates an architecture with direct, independent connections from each protocol processor to common memory, where as the architecture in Figure 4.7 uses a single bus to connect all processors and common memory.

The segment size selected will greatly effect overhead and total throughput. This study used two and three channel testbed configurations with no background traffic and similar channel latencies with different segment sizes. As expected the highest throughput occurs when the segment size given to each protocol processor is equal to the message size divided by the number of available protocol processors. If the processing speeds of the protocol processors or their channel loads were different, but static, the best segment size to assign to each processor would be equal to the fraction of the total work that that processor could achieve times the message size. For example, in a 2 processor system, if 1 processor was half as fast as the second, it would be able to perform 1/3 of the total work so it would be assigned 1/3 of the message size. If the channel or processor characteristics can change

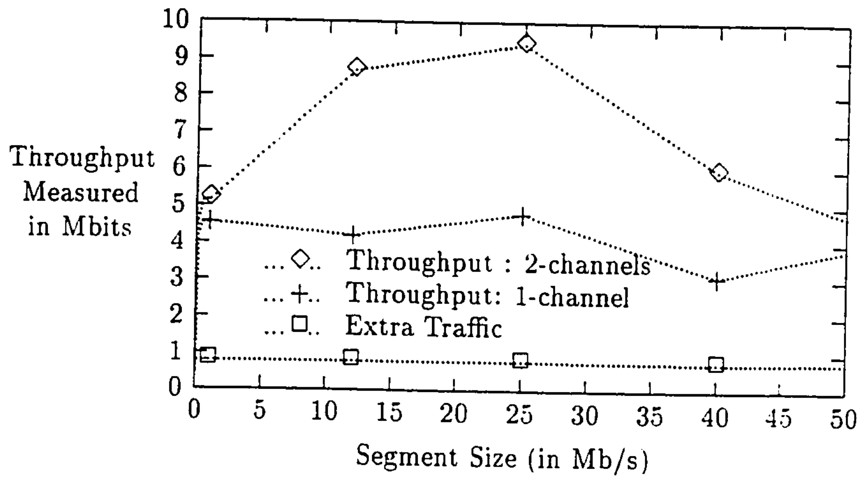

Figure 4.5: Single vs multi-channel under bursty network traffic

over time, providing an optimal segment size is the same load balancing problem as in other multiprocessor systems.

The next parameter considered is the effect of background traffic. To determine this, a single channel of the prototype was loaded with bursty background traffic at a rate of 2K of data every 2ms for a 10% background load. Figure 4.5 shows the results obtained for a single channel and multiple channel testbed configurations. No allowances where made in the data segment scheduling to account for the loaded channel. However, since the segments are assigned on a first-come-first-serve basis, if a channel falls very far behind, one of the remaining channels would eventually be assigned the loaded channel's data. This figure shows that, because the second channel was unaffected, there was almost a 100% increase in throughput. The minimal ability of the default scheduler to handle the difference in channel loads was able to make up for the additional overhead incurred by parallelism. If a more intelligent scheduling mechanism had be in place, that could vary segment size, greater total throughput likely would have been observed.

The next environmental characteristic considered was the effect of differing channel latencies. The throughput observed was significantly effected by the segment scheduler's ability to use information about channel latency when assigning segments. The scheduler

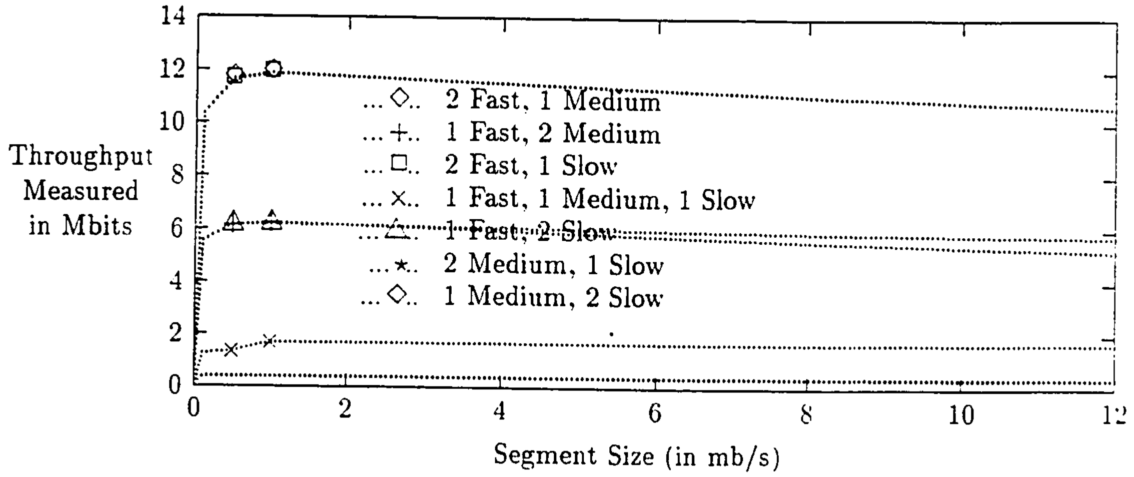

Figure 4.6: Three channel testbed with Different Channel Latencies

was not changed during any of the testbed experiments. but segment sizes were varied. When smaller segment sizes were used, as can be seen in Figure 4.6, the first-come-first-serve scheduling mechanism was able to better control the channels in use. This is because when the slower channels were stuck, they were not holding much data and the faster channels could progress. In generating this figure, some channels were operated at full speed, some were reduced to only 20% of their capacity, and some were reduced to only 2 % of their capacity. The different channel capacities used are labeled in the graph as Fast, Medium, and Slow respectively. All the segment sizes used were static and equal across all channels. If a scheduler could have changed the sizes dynamically, or assign different sizes to different processors, it is likely that the total throughput would have increased. It is interesting to note that fastest channel drives the throughput. The graph lines for two fast and one medium, and for two fast and one slow channels lie almost on top of each other. The same is true for the lines representing one fast and two medium, and one fast and two slow channels. This can attributed to the first-come-first-serve scheduling mechanism. Specifically, while the slower channel(s) are busy with their first small segments, the faster channel can carry the bulk of the traffic.

To show that graceful degradation could be handled by parallel TCP one channel was literally disconnected during transmission. All the data were still transmitted, albeit at a reduced rate. Throughput was comparable to a system with one less channel with the only an additional cost due to retransmission of the segment in progress on the disconnected channel.

4.4 System Implementation

The system described in this section, and emulated via the testbed, can be implemented with existing off-the-shelf hardware. The architecture described here assumes a Sun M-bus based system with several Sparc-based processors, each equipped with a local cache large enough to hold a segment of data as well as the transport code and some control information. Additionally a common memory is attached to the single bus to store the image data to be sent. Both a feasible hardware configuration and a possible software architecture for such a system are described. The system uses distributed first-come-first-serve scheduling of fixed segments to protocol processors.

Figure 4.7 shows the architecture proposed. The table attached to the shared memory is a global data structure used by protocol processors to determine which data have not been sent and which data need to be retransmitted.

Before transmission of data can occur, each protocol processor must open a transport connection to a corresponding receiving transport processor. This can be performed when the application processor requests a transport connection, and can be driven by configuration tables stored at the transmitting and receiving ends.

When data are available to send, they are placed in the common memory, and the data structure, with a separate entry for each segment to be sent, is initialized. Each record in the data structure contains a segment identification number, a status field, and an offset of the segment's data in common memory. Initially the status field contains a value of zero. Each of the protocol processors caches the global data structure so changes to that memory

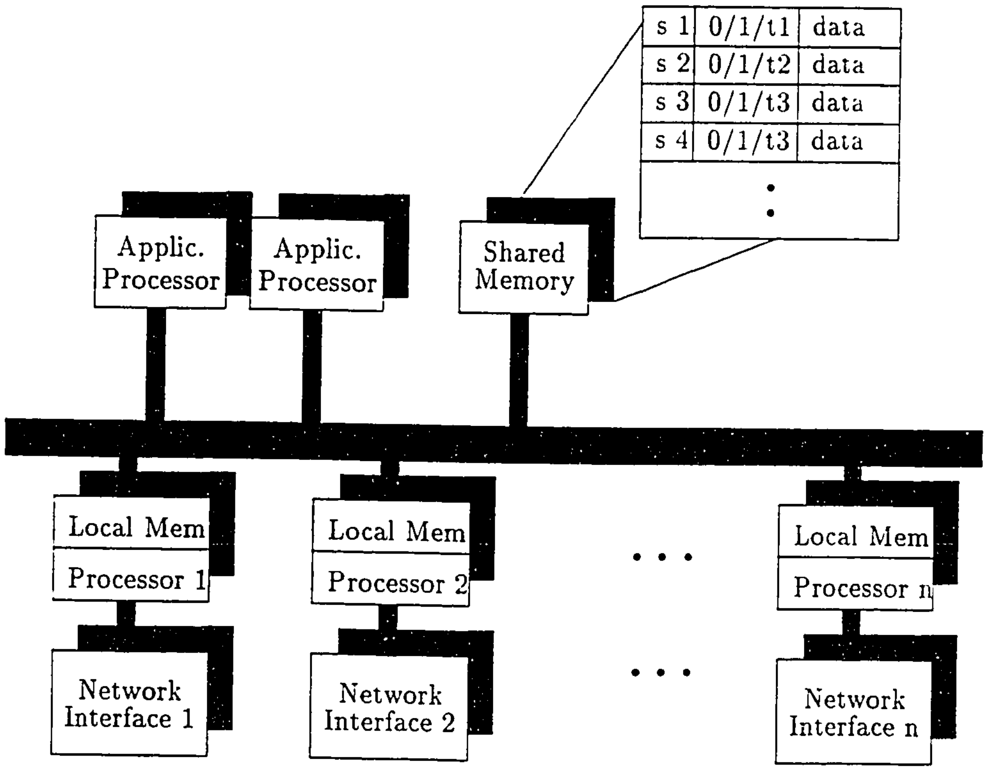

Figure 4.7: MultiProcessor Protocol Processing Architecture

are propagated to each processor as described below. Once the data have been placed and the data structure initialized, the application processor notifies each protocol processor to begin transmitting the data.

Each processor in turn gains ownership of the data structure and scans it for the lowest outstanding segment, indicated by a zero in the status field. After a data segment has been selected, the protocol processor changes the value in the status file to the current time, releases ownership of the data structure, transfers the data to its local cache, and begins transmitting this data. Once the data have been transmitted the responsible protocol processor gains control of the data structure again, changes the value in the status for that segment to indicate that it was successfully sent, and then scans the data structure for the next pending segment. If, while scanning, the protocol processor finds a record with a status field containing a time value that is too old, indicating a timeout, it selects that segment for transmission, and updates the status field to the current time. This data structure/scheduling mechanism insures that all segments will be sent.

On the receiving end, as data are received, each protocol processor collects packets, using standard TCP for packet sequencing, until a complete segment has been received. Because each segment contains a offset or local address, a complete segment can be forwarded directly to the receiving application for consumption. No corresponding data structure or global memory requirement is use at the receiver.

Each protocol processor has a dedicated network interface card, and based on the underlying network, either separate physical networks or a shared high speed network could be used.

4.5 Discussion

The testbed showed a relationship between performance and four parameters: granularity of data sent to the transport layer, nature of background traffic (burstiness), number of channels, and variation in latency on different channels. Since different parameter values significantly effect performance it is useful to dynamically allocate resources (processors and bandwidth) to incoming traffic at all levels in the communication stack and not just between application and transport layers. This extension was discussed in a previous chapter. Providing a mechanism to direct packetized data to network adapters that show the lowest latency will reduce frequency of need to change segment sizes to protocol processors. If latency is very different on each of the channels performance of a parallel network system can be significantly degraded though reducing segment size provides a partial solution. This solution, however, comes with the added expense of increasing overhead between transport and application levels. The ability to dynamically alter segment size and the time and frequency of scheduling segments on channels depending on traffic characteristics would also increase throughput. These enhancements are not possible with the version of the testbed described here, but a more general approach such as that described in the previous chapter allows use of such algorithms.

The specific results of the testbed study can be summarized as follows. At a minor cost, the feasibility of using parallelism in communication has been demonstrated. For a parallel system that provides individual, non-interacting, transport processors, performance gains close to the theoretical limit for two and three channels have been observed. These results have been extrapolated to develop predictions of system performance that can be expected by adding system components and to determine when adding components will no longer increase performance. The testbed network is able to delivered over 14Mbits/s to a single application where no other traffic is present. If latency on all channels is comparable, the relationship between segment size, that is, the size of the data blocks delivered to the transport processors, and total throughput is a parabola with a maximum roughly at $1/k$ of the message size where k is the number of channels available. Also, if segment size is small, simple schedulers can do a good job of balancing the load of the communication processors.

but more intelligent schedulers are needed if larger segment sizes are used.

The testbed system, sending only control messages to the protocol processors actually delivers up to 1.96 and 2.7 times as a single channel on the two and three channel testbed configurations respectively. These figures are likely indicative of speeds achievable if data were transferred from the master to slave over a common bus. It has been shown that the key parameters of segment size, background traffic, and latency have a significant impact on throughput; monitoring and acting on their values needs to be weighed against the increase in overhead that may be introduced. Further, graceful degradation is possible, and demonstrated by literally disconnecting one channel and continuing to deliver on the other channel(s); all data were delivered after the channel was disconnected, albeit at a slower rate.

Because the use of larger segment sizes requires less interprocessor communication, the processing costs and viability of intelligent scheduling algorithms should be evaluated. The testbed presented here was implemented by approximately 1,100 lines of custom code, and allow the evaluation of a single set of options. It also gave insight into the actual work that is required to perform communication in parallel, was well as the implementation issues that need further study.

Chapter 5

Simulation Results

5.1 Introduction

Building on the experimental results obtained via the testbed, a simulation was developed to analyze other implementations of a parallel communication system This chapter discusses that simulation and the results obtained from it.

5.2 Simulation Environment and Goals

The following terms and environmental context are used throughout this chapter, as with the testbed the target application is remove visualization.

The system being considered consists of high performance image generators and displays, called high performance producer/consumer pairs (HPPC). The consumer can be logically thought of as the frame buffer, while the producer can be thought of as a high performance computer producing 10 Mbits of data every 200ms, 100ms (or 30ms for 5, 10,and 33 images/sec.). The scope of this study encompasses HPPCs which reside in a metropolitan or local area network, equipped with multiple FDDI and/or Ethernet media. The goal of the simulated architectures is to provide the bandwidth required by the simultaneous use of

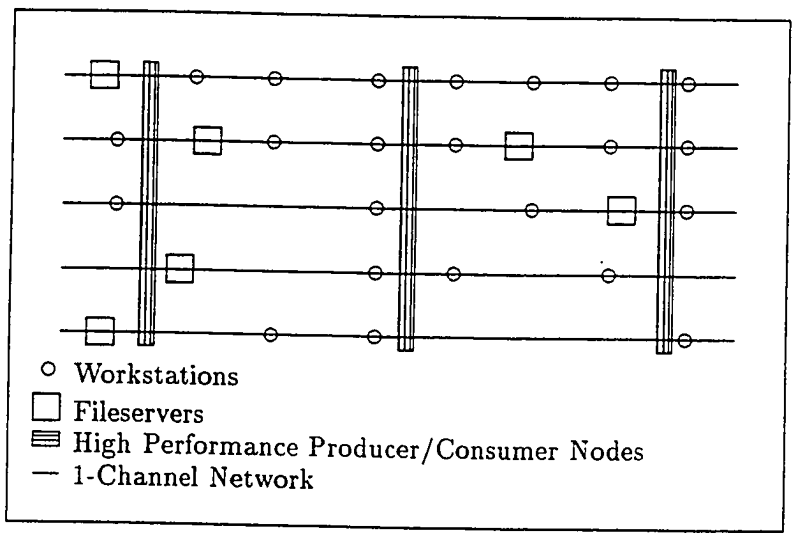

Figure 5.1: Target Network Architecture

multiple paths within the network. Figure 5.1 shows a sample network with multiple paths between the high performance sender and receiver, with all of the channels being shared by other computers. As will be discussed later, this is the networking environment assumed for this study.

5.3 Approach

In order to provide the bandwidth required for interactive remote visualization two general approaches are considered in the simulation. The first connects HPPCs in a point-to-point network with dedicated lines of sufficient bandwidth and switches to handle connections, as suggested in [32]. The second approach uses the traditional multiple access networks with arbitrarily complex topologies.

The former approach guarantees that sufficient raw bandwidth can be provided at the media level. This is a practical solution for short distances, however, it would be prohibitively expensive for longer distances because a dedicated network must be provided.

From a network point of view, as more HPPC pairs need to be connected, the number of interconnections grows exponentially, making larger networks infeasible. Finally, in this approach the possibility of a single point of failure exists.

With the later approach, the total bandwidth is achieved through the aggregation of several channels. If this could be done with no additional overhead costs, the solution would be obviously preferred. Unfortunately, effectively using each of the smaller bandwidth channels requires that the cost of scheduling transmitting requirements to network capacity be incurred [25, 24]. Additionally, the costs of media access time is incurred for each channel used in parallel, rather than just once for a single high speed link. Finally, the system's performance can be more variable due to the different loading and latencies of the channels being used in parallel.

In both of these approaches several bottlenecks are possible. The producer or the consumer could be too slow to generate or display the images fast enough for visually effective animation. This problem is not network related. A network related bottleneck is the processing required to provide reliable communication between the producer and consumer. Finally, the physical medium could be too slow. For older networks, this could be a problem, and multiple channels would have to be used. Newer network technologies such as FDDI, however, provide increased speed and should be able to provide these kinds of services.

5.4 Software Design Issues - The Transport Layer

Conceptually, the simulation parallel processing at the transport layer can be visualized as a set of processors executing the transport protocol in parallel. This is illustrated in Figure 5.2. In this figure, there are three basic components, an application processor, n protocol processors, and m network interface units (NIUs). The application processor is running the image generation application, building images for transmission to a remote site. The protocol processors are performing the transport and network layer processing. Finally the network interface units are performing the media access task. The NIUs can be though of as Ethernet or FDDI controllers.

As discussed previously, to allocate data from the application to the transport processors a scheduler is required. In Figure 5.2 this scheduling task has been placed in the application processor and labeled it as the ATS (or Application/transport scheduler). The ATS can schedule using several different algorithms, but its basic job is to provide the transport processors with data segments, one or more packets, from the application processor. As shown in the figure, the input to ATS consists of the application's estimate of required bandwidth, the frequency and size of data chunks from the application, the idle and active times of the application process, the current load on the protocol processors, and other channel information such as the error rates, idle periods, and current load on channels. Based on this information, ATS dynamically determines the segment size to be assigned to a processor and the processor to which it is to be assigned.

The transport processors, in addition to performing the transport and network layer processing, must schedule packets from the data segments to the available NIUs. As with the ATS, the scheduling algorithm used can vary, but the basic task is to supply the NIUs with packets for transmission. This scheduler is labeled TNS in figure 5.2. The TNS could also be placed on the NIUs in a distributed manner if intelligent NIUs were used. Like ATS, TNS dynamically determines schedules packets to NIUs. As shown in the figure, the input to TNS consists of routing tables, channel error rates, channel loads and latencies, and the size of data remaining with the application. Based on this information, TNS determines the size of the packet and the NIU to which the packet is to be assigned.

5.4.1 Application-Transport Layer Interface

The application, generating data frames, requires the communication system to deliver the frames to a remote site. Here, it is assumed that the application to be residing in the Application Processor, AP, and is interfaced with n protocol processors $(PP_1, PP_2, \ldots, PP_n)$ in the transport layer. The AP generates data in the form of data frames consisting of data segments, where each data segment is associated with a specific location in the frame. With respect to the communication system, the data segments within a data frame are assumed to be independent of each other (i.e., no resequencing of segments is necessary at the receiver end).

The simulation implements the transport layer as a set of general purpose processors, referred to as Protocol Processors, PP and assumes the availability of n such processors. The data from the application are split among the parallel PPs in the transport layer. The task of splitting the data into smaller distinguishable units, or segments is assigned to the ATS. The physical location of ATS is a design decision. For example, if the ATS resides in the application processor, then the application alone controls the allocation of data segments to the transport processes in the protocol processors. As discussed previously, this is referred to as a *central* scheduling policy. A second possibility is to implement ATS in a distributed manner where agents of ATS are placed on the protocol processors. The responsibility of coordinating the allocation of data segments to individual protocol processors then would lie with ATS agents located at the same processors.

A centralized implementation is examined first. Here, at least two methods of segment allocation to the transport processes (TP) can be considered. Under the *First-Come-First-Served* or FCFS policy, ATS allocates a data segment to a TP as soon as transmission of its previous segment has been successfully completed. This assumes a close interaction between the TPs and ATS. Under the *Round-Robin* or RR method, ATS allocates data segments to TPs in a cyclical fashion. Reduced interaction between the scheduler and TPs is required in this case. The reduced interaction may result in reduced control (hence, reduced communication between AP and PP) as well as reduced performance (due to a lack of feedback from TPs). Given a scheduling policy, a scheduler can control the performance of protocol processing by adapting itself to the environment. For example, when a scheduler detects that a particular TP is slow to respond, it could allocate smaller data segments to that TP. The simulation specifically addressed the results obtained from FCFS and RR. Under both of these scheduling schemes, the size of the segment passed from the AP to the TPs needs to be addressed. If all TPs and channels are loaded exactly the same, the optimal solution is to assign segments of size I/M, where I is the size of the image to be sent and M is the number of TPs. If processor and channel loads vary, other segment sizes should be considered. For example, if the load on each of the TPs is highly erratic, if a TP receives its segment and then becomes significantly slower than the other TPs, total system latency will increase and total throughput will decrease. Alternatively, assigning small segments to TPs means more work for ATS, and more interaction between the TPs and the AP. This could also result in lower throughput. In addition to fixed segment sizes, variable segment sizes can be considered. With variable segment sizes, the segment size given to each TP can change as that TPs ability to process data changes.

In the case of a distributed implementation of the ATS, the ATS agents on different PPs need to coordinate the allocation of data segments. The actual details of such an implementation depend on the interconnection architecture connecting the PPs. Once again FCFS, RR and adaptive policies can be considered as candidates.

5.4.2 Segment Processing at the Transport Layer

Once a segment is allocated to a sender TP, it is the TPs' responsibility to deliver it to the transport layer at the receiver end. It achieves this objective by breaking the segment into data packets and sending them to the transport layer at the receiver. The responsibility of rebuilding segments is left to the receiver.

At the sender end of the transport layer, the management of packet acknowledgments and retransmissions are the two major design issues. Even though their implementation is well understood in a single TP case, their extension to parallel TPs is not obvious.

When an acknowledgment for a packet arrives at the transport layer, it can be processed either by the sender TP or by a central server. In case it is processed by the sender TP, the network/transport layer interface should be responsible for directing the acknowledgment to the TP. In the case of a central server, all acknowledgments are directed to it. The central server may be located on a separate processor or its functionality may be shared among a set of PPs.

In the absence of an acknowledgment within a specified time period, the transport layer at the sender should retransmit the packets. The retransmission function is closely related to the acknowledgment processing function at the sender. In other words, if we assign the responsibility of processing acknowledgments to the sender TP, then the same TP should also be responsible for retransmissions. Similar arguments hold in the case of central server policy.

The receiving transport layer receives packets and assembles them into segments to be delivered to the application. In addition, it sends packet acknowledgments. The simulation assumes that all data packets from a single segment are handled by a single TP at the receiver. In addition, we let this TP be responsible for sending the acknowledgments. Alternate policies possible include a central server to receive and distribute packets to TPs, and a distributed policy where decisions are made by each individual TPs.

5.4.3 Transport-Network Layer Interface

A sender TP divides a segment and builds transport packets these packets for transmission, and delivers them to the network layer. Whether the network layer should be implemented on a single processor or a set of processors is a design decision. The simulation considers a system where the network layer is implemented as a set of network processes, or IPs. A single IP is associated with a single TP and resides on the same processor as the TP, thus the two processes can share memory, information, etc.

5.5 Software Design Issues - The Network Layer

While TP hands-out data packets to the network layer (IP), it forms network data units, and sends them to the media access control layer or MAC. Since the simulation assumes a one-to-one relationship between a TP and an IP, both residing on the same processor, the design issues in IP are the same as a traditional network layer, especially with respect to its interaction with a TP. However, its interface with the media access layer may be different in the context of parallel processors at MAC level. The processor that handles the MAC and data layer functions is referred to as a network interface unit, or NIU. The simulation assumes the existence of m parallel network interface units interfacing the IPs with parallel physical channels. It is also assumed that a one-to-one correspondence between a TP packet and an IP data unit exists, and the IP data unit contains a complete transport packet.

Each network layer has to select an NIU to transmit the data packets. The responsibility of selecting an NIU for the transmission of a data packets from a TP/IP pair to an NIU is assigned to a Transport Network Scheduler, or TNS. The design decisions for TNS are similar to the the ones in ATS design: location and scheduling. If TNS is located in a PP, it can distribute the data packets using either FCFS or RR policy to the network interface units. In the case of FCFS policy, we assume that the NIUs request for data packets from the IPs. In the case of the round-robin policy, the TNS distributes data packets to NIUs in a round-robin fashion. In addition, other intelligent scheduling policies are possible, such as an intelligent RR policy that allocates a data packet to an NIU only when the NIU's current queue is less than a threshold. The simulation, however, only considers FCFS and RR policies.

If the TNS is distributed and located at each of the NIUs, then it can select a PP using either the FCFS or RR policy to obtain data packets. In the case of FCFS, the network layer in a PP seeks to transmit a packet, and the TNS gives permissions in a First-Come-First-Serve fashion, with possible queuing of data units. In the case of RR, the TNS obtains the data units in a round-robin fashion from the network layers. Several other policies are also possible.

At the receiver end, the NIUs receive packets and acknowledgments from the parallel physical channels. These need to be delivered to the network layers in PPs. The allocation of data packets to PPs depends on the packet processing policies. In this study it is assumed that a single PP completely processes a single segment.

Figure 5.2 shows the logical connection of components for the architecture of this study. In this architecture all protocol processors are operating as independently as possible while still contributing to the task of transmitting a block of data in parallel. In this architecture it is assumed that timers and acknowledgments are performed at the protocol processors. This is not the only way to handle timers and acknowledgments and four basic ways to perform this processing, with varying protocol processor independence, were identified in chapter 3 and illustrated in Figure 3.1.

5.6 Architectural Design Considerations

The earlier sections of this chapter have primarily discussed the software components of communication system design for visualization applications. This section considers the hardware aspects of the proposed system.

Bandwidth and costs are two important criteria to be considered in choosing the hardware architectures. The hardware configuration will also impact the software and control options that can be used. For example, if the mechanism for scheduling segments to protocol processor is to be distributed, the data to be transmitted must be placed in a memory that is accessible to all of the protocol processors. This dictates that a global memory is be included in the hardware architecture. Conversely, if each processor is directly connected

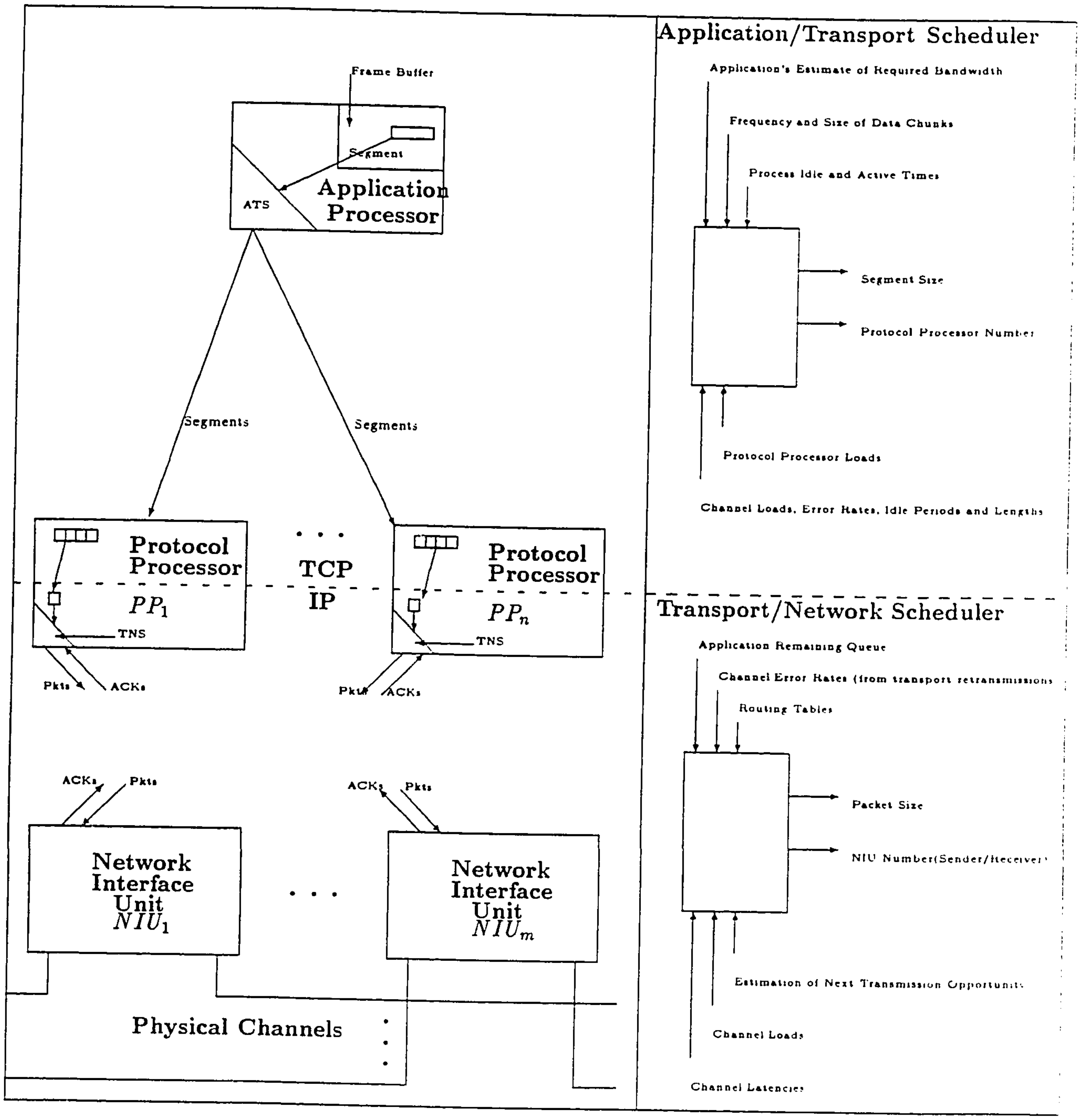
Application/Transport Scheduler
Frame Buffer
Segment
ATS
Application Processor
Application's Estimate of Required Bandwidth
Frequency and Size of Data Chunks
Process Idle and Active Times
Segment Size
Protocol Processor Number
Protocol Processor Loads
Channel Loads, Error Rates, Idle Periods and Lengths
Segments
Segments
Protocol Processor
TCP
IP
PP_1
TNS
Protocol Processor
PP_n
TNS
Pkts
ACKs
Pkts
ACKs
Transport/Network Scheduler
Application Remaining Queue
Channel Error Rates (from transport retransmissions)
Routing Tables
Packet Size
NIU Number(Sender/Receiver)
Estimation of Next Transmission Opportunity
Channel Loads
Channel Latencies
ACKs
Pkts
ACKs
Pkts
Network Interface Unit NIU_1
Network Interface Unit NIU_m
Physical Channels

Figure 5.2: Scheduling and Control - Application — Protocol Processors

to a single network interface unit, there is no need for transport to network interface unit scheduling. Three bus based hardware architectures which represent three classes of cost are considered in this section. Many other configurations, utilizing other processor connection configurations, multiple buses, mesh, hypercube, differing number and types of processors, and providing separate processors for ATS, TCP, TNS, and IP processing can be conceived. The configurations selected are by no means comprehensive, but are representative of the current architectures proposed for parallel communication systems. For example, we are explicitly excluding separate processors for the TCP and IP functions in the design, whereas it was discussed in chapter 3 that this exclusion is not necessary. Additionally, it should be noted that some of the options will require very high performance components. For example, if ATS is distributed among the TPs, a significant amount of local memory has to be available on each of the TPs, or the global memory and connecting bus will have to be very fast to insure that it does not introduce a bottleneck.

A low cost option is presented in Figure 5.3. Because of the popularity of Ethernet and the proliferation of relatively low speed LANs, a configuration that may be found in places could be several Ethernet LANs acting as subnetworks for a department or division. The aggregate capacity these LANs could be easily tapped to provide a single higher speed connection between two hosts. By simply equipping the high performance producer and consumer machines with extra network interface cards for each subnetwork being utilized, parallel hardware is realized. The figure shows one high performance workstation that could be acting as either a producer or a consumer. As an example, if this architecture is configured with five Ethernet channels, 10Mbits/sec each, the total throughput of the system could theoretically be as high as 50Mbits. Because of the operating characteristics of Ethernet, it will be significantly lower, though a nearly a five fold increase in nominal Ethernet performance could be observed. As with all architectures discussed here, it is assumed that the networks utilized are being used concurrently by other users, so available

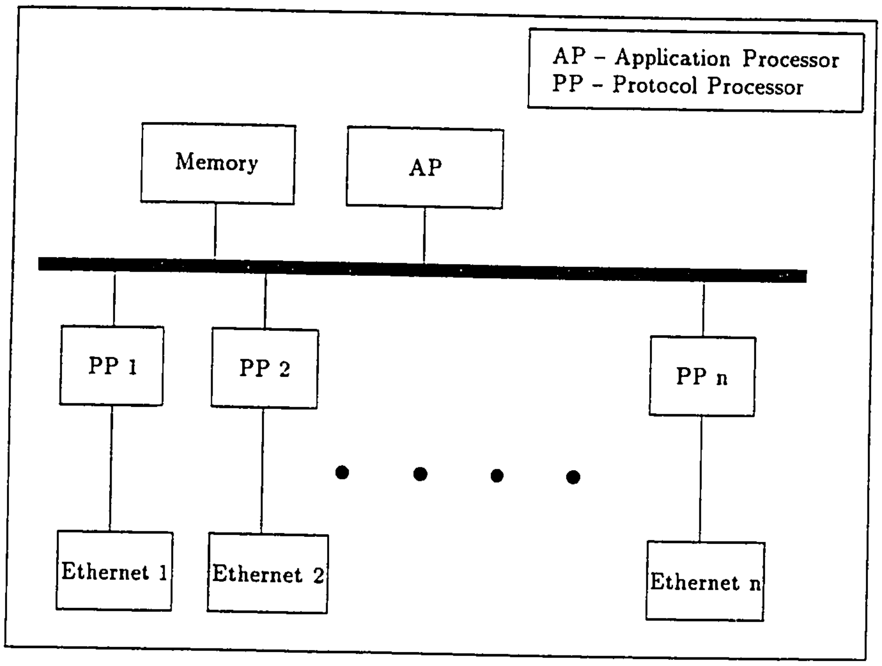

Figure 5.3: Architecture A: Low Cost

network capacity among parallel networks may not be identical. From a remote visualization point of view, the example of five Ethernets could conceivably deliver 10Mbits of data, one frame, every 200ms. The assumption in this example is that the connecting bus could be driven at a sustained rate of greater than 50 Mbits/second. If not, the bottleneck has simply been moved from the protocol processing task to the connecting bus. This represents limited animation that could be used for applications such as viewing weather data obtained from satellites. It should also be noted that in any environment other than a two node network, Ethernet can not be expected to deliver 10 megabits of traffic. This has been taken into consideration in the design of the simulation.

The architecture in Figure 5.3 has obvious expansion and performance limitations. As needs increase, a high performance, and higher cost, media should be considered. Figure 5.4 show the next architecture considered. FDDI is becoming increasingly available, but, because of the processing requirements of transport layer processing, no single client can

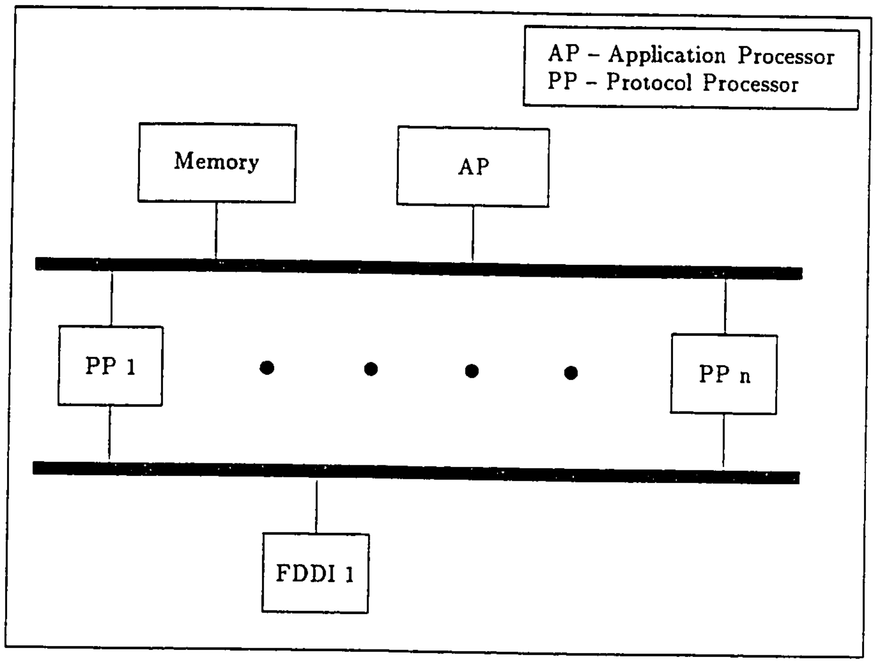

Figure 5.4: Architecture B: Medium Cost

use the 100 Mbits of capacity it offers. The architecture shown in Figure 5.4 is designed to address that problem. A single FDDI LAN is used, but multiple protocol processors are employed to process the data at 100 Mbits speeds. If the full 100 Mbits of capacity could be utilized, a new screen image could be delivered to the viewer every 100ms. Closer to true animation, but not the 1 image every 30ms for seamless animation, this capacity could be used for medical imaging applications, as well as orbital surveillance systems such as that used to track ships in an ocean. To utilize the full capacity of the FDDI channel sufficient processing power would be needed in the form of TCP and IP processors. As with the first architecture, the simulation assumes that the connecting buses are able to keep up with the protocol processors. Additionally, there is a need to insure that the processor performing the ATS task is able to feed the TPs fast enough to not be a new bottleneck.

In order to obtain true animation, a new image needs to be delivered to the viewer every 30ms. To obtain this, multiple FDDI channels could be used. The architecture shown

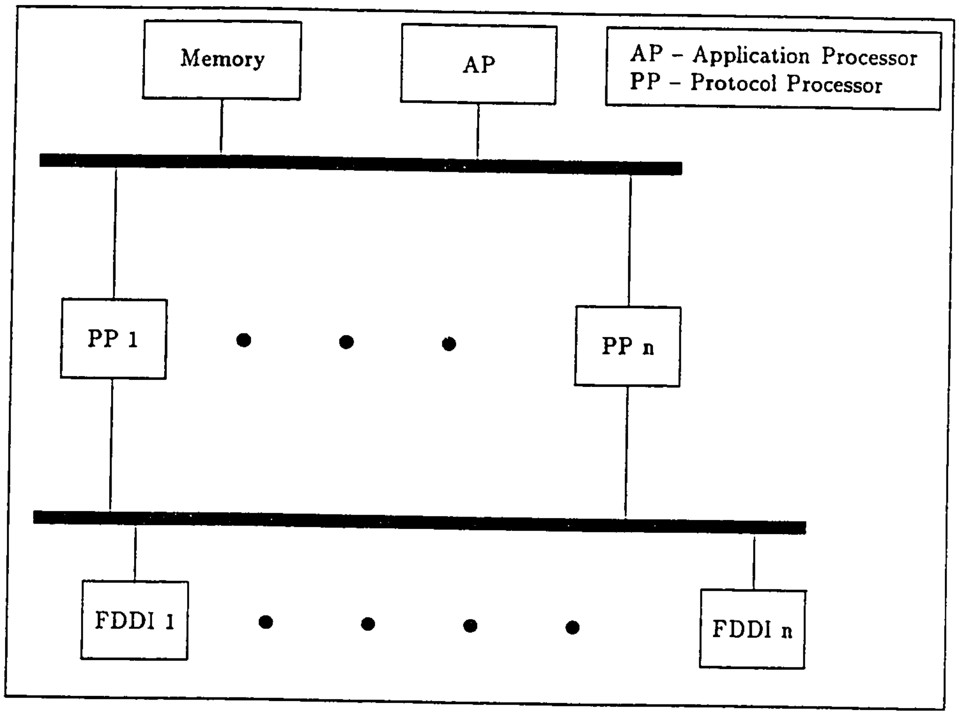

Figure 5.5: Architecture C: High Cost

in Figure 5.5 could provide the needed raw bandwidth. The processing power required to use supply data at that rate would be significant. Multiple RISC machines, such as Sun Sparc processors, could be used for the transport processing. In this architecture we are using the protocol processors for both TCP and IP processing. An alternative configuration could, however, separate these two tasks and place then on separate processors, allowing the transport processor to be completely devoted to TCP processing. It should be noted that, as before, the connecting bus and ATS processor are a source of concern. Additionally, the access time for the memory unit will have to be very fast. This issue was discussed in chapter 1.

5.6.1 Performance Evaluation

Performance data for the three system architectures discussed above was obtained. As outlined before, several software and hardware configuration issues will effect the performance of a system. Many are discussed in chapter 3, but others include network length, error rates, and window size. In addition to these software configuration options, many of the hardware components selected can also induce bottlenecks and limit communication throughput. Specifically, the speed of the processor performing transport processing, of any global memory, of the connecting buses, and of the transmission medium will all effect the throughput that can be achieved. The time to fully examine all of the options available for each of the design categories would be prohibitive. For this study the main interest is in the performance of differing hardware architectures, segment sizes, and window sizes. With this in mind the other options have been fixed as to reduce their impact on the total system performance.

Number of Protocol Processors

For the results presented in Figures 5.6 through 5.10, the simulation was used to examine the three hardware architectures in Figures 5.3, 5.4, and 5.5. In each of these the application processor to protocol processor bus was fixed at 1 Gbit/second. The global memory used has an access time of 20ns. The protocol processors used were assumed to be capable of 12.5 MIPs, and the transport and network layer processing for each packet was fixed at 500 instructions/sec. In architectures B and C, the protocol processor to media adapter bus was fixed at 400 megabits/second. Each of these values were selected to insure that sufficient throughput was available to drive the media adapters at their full capacity. Initially, the application to protocol processor bus was fixed at 400 megabit/seconds. As shown in Figure 5.7, this had no effect on architecture B. but architecture C was not able to provide significantly more throughput than architecture B. even though it was equipped with three

FDDI channels. After examining the results and the options performed, it was realized that the application to protocol processor bus must transfer all of the data at least twice, in addition to being used for interprocessor communication. This meant that although the bus was rated at 400 megabits/second, the best we could achieve would be less than 200 megabits/second throughput, regardless of the amount of processing power at the protocol processing level or bandwidth at the media level.

The graphs shown in Figures 5.6 and 5.8 show the effect of increasing the number of protocol processors for each of the architectures. In architecture A, because each protocol processor had a dedicated network interface, the increase in throughput observed is almost linear with respect to the number of processors available, until the aggregate throughput available at the media level exceeds that of the bus connecting processors and global memory. For Ethernet and the hardware speeds that selected, this occured at approximately 120 processors.

Because architecture B used a single FDDI channel, the maximum throughput that could have been observed would be 100 megabits/second. In Figure 5.8 it was shown that this limit is almost achieved when 15 protocol processors are provided. Until that point, an almost linear increase in performance is seen as the number of protocol processors are added. Beyond that point, the total throughput increases, but at a significantly reduced rate.

Architecture C's results show the same trends as thaose of architecture B, but the total throughput achieved is greater. Beyond 30 protocol processors, little increase in throughput is observed. The maximum throughput is 160 megabits. As with the application to protocol processor bus, the global memory must be accessed at least twice for each bit of data. This means that the highest throughput that could have observed would be less than 200 megabits, even though there was capacity for 300 megabits at the media access level.

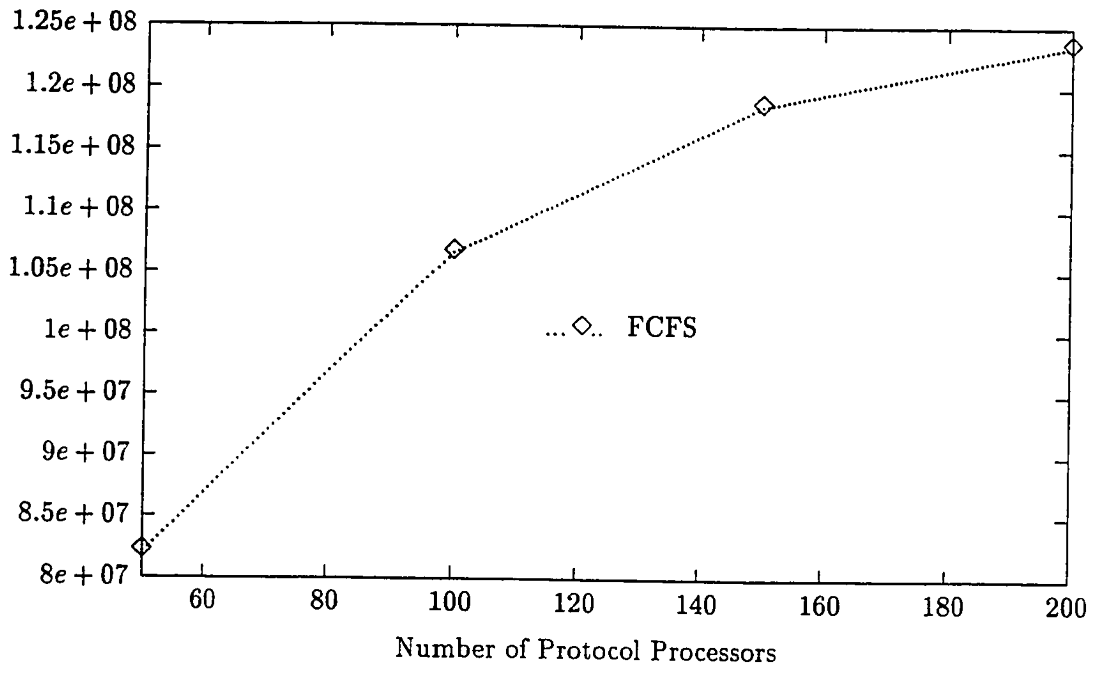

Figure 5.6: Architecture A – Throughput with Large Number of Processors

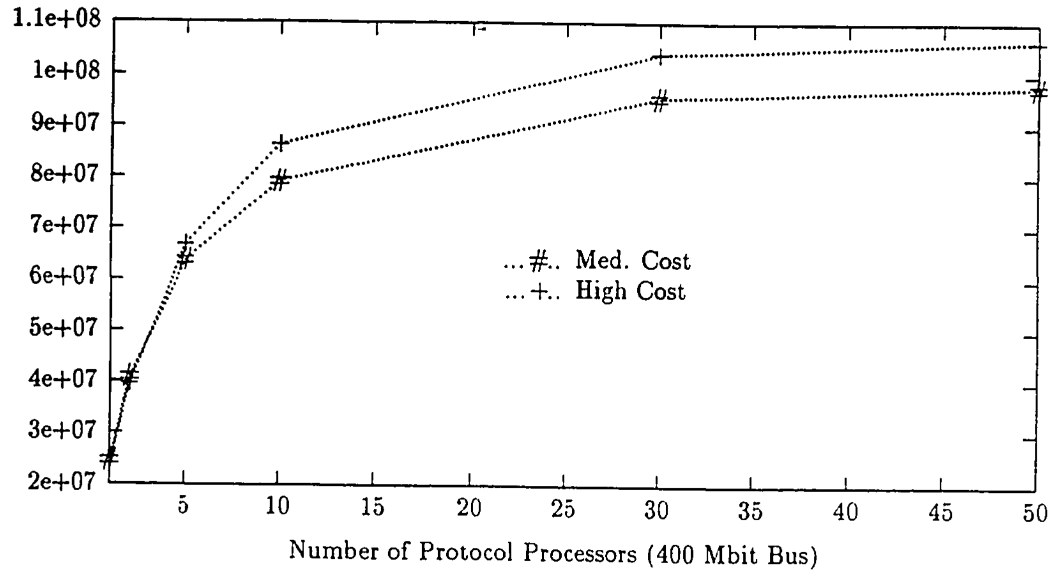

Figure 5.7: Architecture B and C – Throughput vs Number of Processors (400 Mbits/sec Bus)

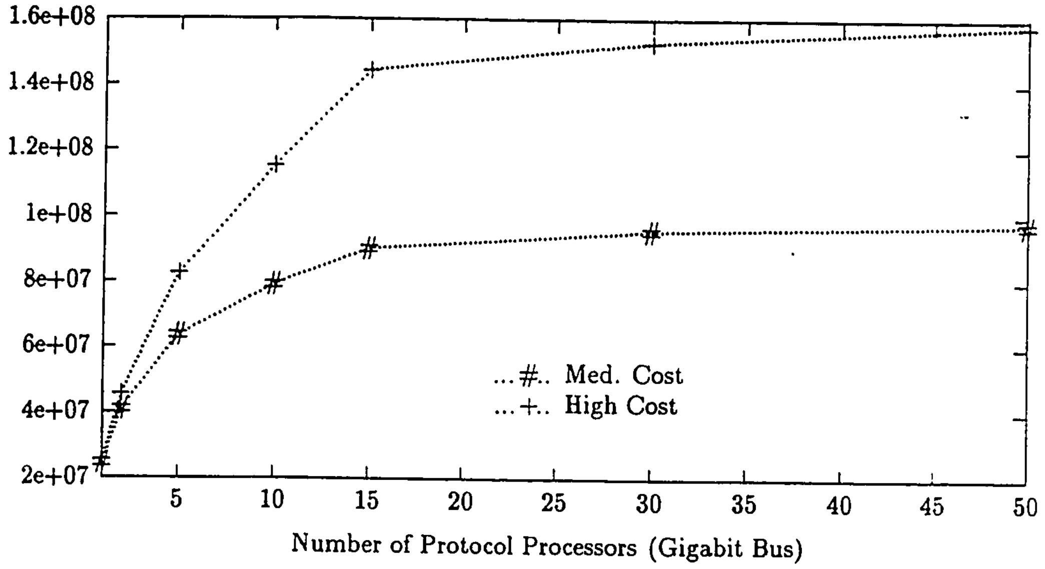

Figure 5.8: Architectures B and C – Throughput vs Number of Processors

Segment Size

With the data obtained above, 30 protocol processors was used as a baseline in the simulation runs involving differing segment sizes. The results obtained are shown in Figure 5.9. The x axis of these graphs is logarithmic. For each of these, the traffic load on the network was low and varied little, and the load on each of the processors was also low and relatively stable. Because of this, the optimal segment size was always equal to (image size)/(number of processors).

The results for all three architecture are shown in Figure 5.9. In architecture A, relatively small segment sizes had little impact on total system throughput. This can be attributed to the fact that the Ethernet adapters used made the total system throughput always low compared to the speed of the application to protocol processor bus and of the global memory. This meant that the contention for these devices was low and each of the protocol processors could be serviced frequently and rapidly. When the segment size exceeded the optimal segment size, some protocol processors were not given any data to transmit because

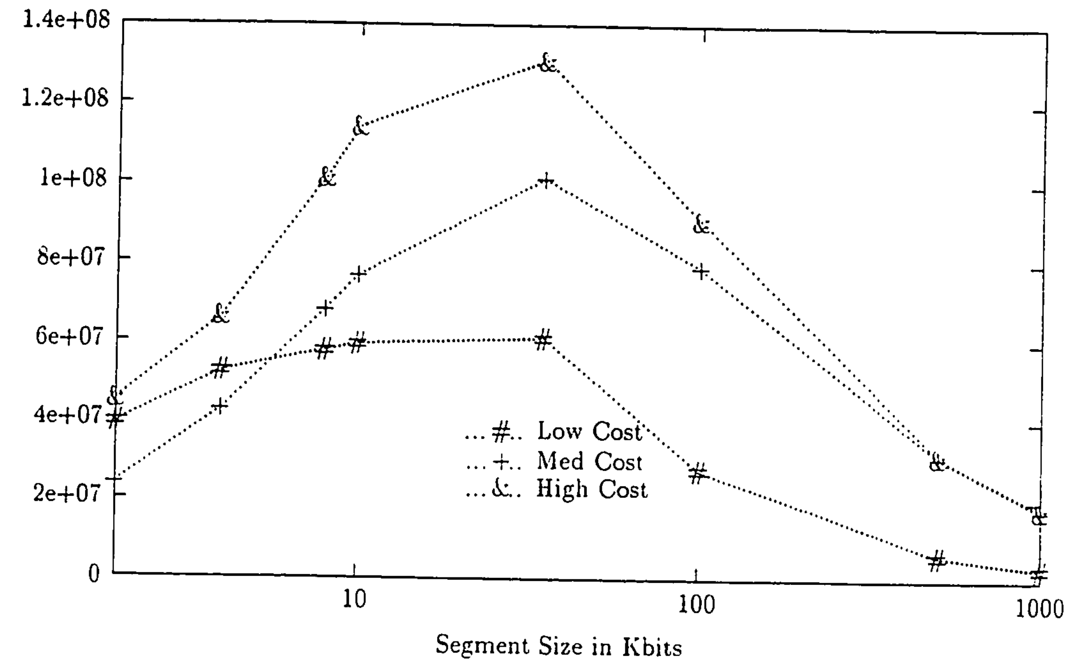

Figure 5.9: Throughput vs Segment Size

the other processors were already working on it. This meant that not all of the protocol processors could contribute to the throughput and hence total system throughput was reduced.

For architectures B and C, smaller segment sizes had a significant impact on the throughput that could be achieved. These architectures were equipped with one and three buffered FDDI channels respectively. This meant that little time was spent by the protocol processor waiting for transmission to occur, so the speed of the application to protocol processor bus and global memory became more important. When the optimal segment size was used, each protocol processor needed to compete for the application processor and global memory only once for each image sent, so the demand was reduced. As with architecture A, when the segment size was increased beyond the optimal size, some processor were excluded from participation and total system throughput was reduced.

Window Sizes

After determining the number of protocol processors and optimal segment sizes to be used. the impact of window size on total system throughput for each of the architectures is considered. The graph shown in Figure 5.10 contains 1 line for each of the architectures.

The results for architecture A show small improvement from a window size of 1 to a window size of seven. Beyond seven. no real benefit is observed. This relates again to the fact that architecture A is using Ethernets and the transmission time is relatively long. In this case the acknowledgment is returned long before the remaining six packets of the window are sent, so normally no time is spent waiting for acknowledgments of packets. If the network environment is extremely noisy, or the network length is extremely long, then the window size would have been more of an issue. Simulation runs for this architecture, with error rates as high as 10^{-6} were made. and the window size still made little difference.

For architectures B and C a window size of one is an extreme impediment. Window size of seven, however. showed great improvement. Beyond seven though. no real increase in speed is observed. As with architecture A, the network error rate for these architectures is very low, 10^{-9}. Given this error rate. the 100 Mbits/second transmission speed of FDDI channels, and the network length of 20 km with 20 stations, the queuing delay of a packet to be approximately $3.65 * 10^{-4}$. (For details of the queuing formula see [2].)

The processing time for a single packet, using the values given above, is $4.0 * 10^{-4}$. This means that on the average a new packet would be ready to transmit about the same time that the last one is received. Since the time for an acknowledgment to be built and to be sent is about the same as for sending a packet, a few packets may be outstanding at any time. Hence, a window size of seven is sufficient.

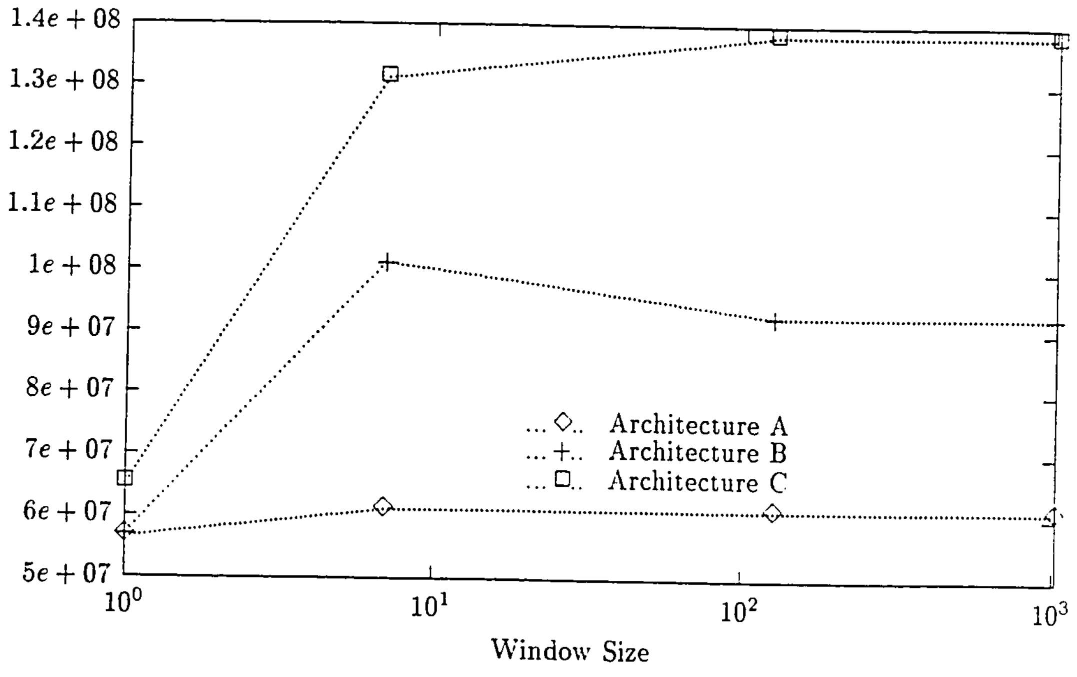

Figure 5.10: Throughput vs Window Size

Chapter 6

Conclusions

While the world is getting politically and industrially smaller, people will remain physically separated. This means that the need for even faster communication will continue.

Parallel communication processing has the potential of providing bandwidths in the gigabit and multi-gigabit speed to a single user. Because this bandwidth can be delivered via single high speed fiber optic cable, or by the use of multiple slower speed media, interoperability with existing network is achievable. Because protocol processing is being performed in parallel, with processors working semi-autonomously, the degree of fault tolerance is increased over that of a single channel system. Additionally, because parallelism is applied separately to individual layers in the communication stack, the degree of parallelism used can be matched to the performance needs of an application. If the processors are general purpose, when they are not performing the communication task, they can be devoted to other tasks such as user application or operating system processing. Finally because separate physical channels can be used, the control and avoidance of congested network channels is easier, and when congestion cannot be avoided, its impact on total system throughput is reduced.

In this work I have presented options and issues important to parallel communication and related them to other areas areas of computer science and specific approaches to parallel communication. In order to deliver the required bandwidth, regardless of the method used, how different hardware and software components that comprise the communication system interact must be understood. This task of obtaining this understanding was addressed in three ways in this work. First, because the number of possible approaches is large, a sensitivity analysis was performed. This allowed a quick look at system options and provided insight in to which approaches are viable and what components need to be further studied. Because of the speed of memory currently available, the use of local memory was shown to be required. Additionally the local bus, connecting a processor to its local memory, needs to provide high bandwidth to handle the instruction and data transfers required for protocol processing.

The second approach used was to prototype a system approach through the use of a testbed. This is a more costly approach, but allows the collection of more accurate performance data for a particular approach. The testbed in this work provided two and three parallel channels and showed speedups of 1.9 and 2.7 over that of a single channel. The testbed also showed that by changing the size of the data blocks given to the protocol processors, the degradation in throughput due to a single busy line could be reduced. This type of fine tuning does not come for free, and the cost to collect the additional information required to realize dynamic segment sizes must be considered. Simple schedulers were also able to balance the load on channels with differing throughputs, but only by using small segment sizes, thus incurring the higher cost of more frequent interprocessor interactions. In addition to collecting actual performance data, the testbed gave more insight into how components of the system interact and the amount of work required to perform communication in parallel. The testbed is limited, however, because changes to the approach taken can require that the testbed be radically changed, so the approaches to be studied must be chosen carefully.

Because the testbed is limited in the ways it can be expanded without building a complete operational implementation, the next tool used to examine approaches and options is simulation. I built a simulation that allowed the comparison of systems with different scheduling mechanism and acknowledgment processing. The simulation showed that, for the approaches being compared, there was little difference in selecting FCFS or RR scheduling between the application processor and the protocol processors. It also showed that the speed of memory and system bus are driving constraints. The speed of memory was the biggest issue for the software and hardware architectures studied. The selected segment sizes became more important as the raw throughput at the MAC layer increased. For Ethernet, the media is slow enough to allow new segments to be retrieved by the communication processor before previous segments have been completely transmitted. For FDDI based architectures this was not the case, and large segment sizes where needed to reduce the time spent by the communication processor retrieving a large number of small segments. Window sizes were also considered during the study, and I showed that a window size of seven was sufficient for all architectures.

This work is a starting point for research in the area of parallel communication. The ability to compare differing methods, and to classify methods based on implementation details such as the type and place of layer scheduling and the method used to process timers and acknowledgments, allows new research to build upon existing work.

In my work, I actually began with the testbed study, and then built the simulation, followed by the model used in the sensitivity analysis. This is because before the simulation or sensitivity analysis models could be built, I needed a better understanding of the problems that needed to be addressed. The sensitivity analysis represents the most general model of the parallel communication process, and can be used as a guide to the areas the need further examination.

My work concentrated mainly on DRDA systems with simple schedulers. There may be a potential for higher throughput and better control of parallel resources, with more intelligent schedulers. This, however, will require a study to identify the possible sources of feedback, or information, that the scheduler can use and the cost of obtaining the information required for intelligent scheduling. Additionally, the scheduling information must be provided to the scheduler quickly so that the scheduler can react to current situations.

In all architectures considered in the framework in Chapter 3 implicitly assumed that data to be transmitted were placed in a central location, before the communication processors received them. This represents a bottleneck through which all data must pass, and eliminating this bottleneck is something that needs to be studied.

Bibliography

[1] H. Abu-Amara, T. Balray, T. Barzilai, and Y. Yemini. *PSi: A Silicon Compiler for Very Fast Protocol Processing*. North–Holland, 1989.

[2] D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, 1987.

[3] D.A. Borman. Implementing TCP/IP on a Cray Computer. *Computer Communication Review*, 19(2), 1989.

[4] G. Chesson, B. Eich, V. Schryver, A. Cherenson, and A. Whaley. XTP Protocol Definition. Technical Report Revision 3.0, Silicon Graphics, Inc., Jan. 1988.

[5] D. Clark, M. Lambert, and L. Zhang. NETBLT: A Bulk Data Transfer Protocol. Network Working Group Request for Comments (RFC) 998, 1987.

[6] D.D. Clark. Architecture for a New Generation of Network Protcols. *Proceedings of the Second Gigabit Testbed Workshop*, February 1991.

[7] D.D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications*, June 1989.

[8] T.A. DeFanti, M.D. Brown, and B.H. McCormick. Visualization: Expanding Scientific and Engineering Research Opportunities. *IEEE Computer*, August 1989.

[9] J.J. Dongarra and I.S. Duff. A spectrum of options for parallel simulation. Technical Report ANL/MCS-TM-57, Argone National Laboratory, September 1989.

[10] M. J. Flynn. Very High Speed Computers. *Proceedings of the IEEE*, 54:1901 - 1909, December 1989.

[11] E.C. Foudriat, K. Maly, C.M. Overstreet, S. Khanna, and F. Paterra. A Carrier Sensed Multiple Access Protocol for High Data Rate Ring Networks. *Computer Communication Review*, 21(2), April 1991.

[12] E.C. Foudriat, K.J. Maly, C. M Overstreet, F. Paterra, and S. Khanna. A carrier sensed multiple access ring protocol for gigabit networks. *ODU CS Tech. Rep. 90-8*, January 1990.

[13] H. Fuchs, M. Levoy, and S.M. Pizer. Interactive Visualization of 3D Medical Data. *IEEE Computer*, August 1989.

[14] David E. Game. *Performance Improvements for FDDI and CSMA/CD Protocols*. PhD thesis, Old Dominion University, 1991.

[15] R.D. Gitlin. LuckyNet: A Gigabit Research Network. In *The 2nd Annual Workshop on Very High Speed Networks*. ACM SIGCOMM, 1991.

[16] Z. Haas. A Protocol Structure for High-Speed Communication over Broadband ISDN. *IEEE Network Magazine*, 1991.

[17] I. Jacobs and H. Leung. Survivable Optical Networks. *Proceedings of the 6th MCI Technology Symposium*, October 30, 1989.

[18] V. Jacobson. Congestion Avoidance and Control. In *Sigcomm 1988*, Aug 1988.

[19] N. Jain, M. Schwartz, and T.R. Bashkow. Transport Protocol Processing at GBPS Rates. *Proceedings of Sigcomm '90*, pages 188 -199, 1990.

[20] A.S. Krishnakumar, B. Krishnamurthy, and K. Sabnani. Translation for Formal Protocol Specifications to VLSI Design.

[21] K. Maly, I. Jacobs, C.M. Overstreet, R. Mukkamala, M. Zubair, E. Foudriat, J. Wiencko, and S. Midkiff. A Reliable Gigabit Network Based on Parallelism. Technical Report 89-46, Old Dominion University, 1991.

[22] K. Maly, F. Paterra, C.M. Overstreet, R. Mukkamala, M. Zubair, S. Khanna, and J. Wiencko. Remote Visualization Using Parallelism in the Context of Existing Networks. *Pending*, 1991.

[23] K. Maly, F. Paterra, C.M. Overstreet, R. Mukkamala, M. Zubair, S. Khanna, J. Wiencko, and H. Srivatsan. Overhead in Parallel Communications: A Case Study. Technical report, Old Dominion University, 1991.

[24] K.J. Maly, E.C. Foudriat, D. Game, R. Mukkamala, and C.M. Overstreet. Traffic Placement Policies for a Multi-Band Network. In *Sigcomm 1989*, Aug 1989.

[25] Kurt Maly, C. Michael Overstreet, Xia-Ping Qiu, and Deqing Tang. Dynamic resource allocation in a metropolitan area network. *Proceedings, SIGCOMM '88 Symposium*, pages 13–24, August 1988.

[26] R. Mukkamala, E.C. Foudrat, K.J. Maly, and V. Kale. Modeling and Analysis of High Speed Parallel Token Ring Networks. Technical Report TR-91-27, Old Dominion University, 1991.

[27] A.N. Netravali, W. D. Roome, and K. Sabnani. Design and implementation of a high-speed transport protocol. *IEEE Transactions on Communications*, 38, no. 11:2010–2024, November 1990.

[28] T.F. La Porta and M. Schwartz. Architectures, features, and implemenation of high-speed transport protocols. *IEEE Network Magazine*, pages 14–22, May 1991.

[29] Floyd E. Ross. FDDI – A Tutorial. *IEEE Communications*, pages 10–17, May 1986.

[30] D. Siewiorek. *Fault Tolerant Computing - Theory and Techniques.* Prentice Hall, 1986.

[31] A. Tannenbaum. *Computer Networks.* Prentice Hall, 1981.

[32] Ultra Network Technologies. UltraNet Product Description, 1989.

[33] R. W. Watson. *The Delta-t Transport Protocol.* North-Holland, 1989.

[34] J. A. Wiencko. SAFENET Committee Presentations. *NASA Tech Briefs*, 1990.

[35] J.A. Wiencko. Fiber optic csma/cd network approaches, November, 1987. presented to the IEEE 802.8 and IEEE 802.3 standard working groups.

Appendix A

Testbed Software Operations

This appendix gives a detailed description of the software operations used to realize the testbed discussed in chapter4. The discussion below assumes that two channels have been implemented. An additional channel was added, but the software operations are the same for any number of channels. Each additional channel requires two more computers.

A.1 Transmission Protocol

A minimum of six machines are needed for this implementation of parallel TCP. Three are required for the transmitter and three for the receiver. The transmitter consists of a master and at least 2 slave computers to achieve parallelism. In order to allow true parallel communication, all machines reside on physically different Ethernets. This section gives function descriptions for both the master and slave programs. followed by a description of their functions and interactions.

A.1.1 Master and Slave Program Functionality

The master program. run on one machine only. is used to interface with the user's program. The program must perform the following functions:

- Recognize the existence and identities of slave processors
- Manage slave processors transmission loads and terminations
- Provide for the retransmission of slave commands for unacknowledged data packets
- Provide an interface to the user program for data and destination address identification
- Provide a mechanism for aborting the transmission of a message

The slave program is run any number of processors. This program performs the actual sending of data to the receiving slave processors. This program's functions are:

- Identify themselves to the running master program
- Interpret the command packets sent from the master and perform the functions.
- Interface to the receiving slave processors

As will be discussed in the next section, the slave processors are not concerned with the number of channels that are being used or the packet order and distribution. Additionally, the underlying transport layer protocol performs all of the retransmission required within the slave processors.

A.1.2 Master and Slave Interactions

This section discusses the functions and interactions among each of the programs' components for both the master and slave programs.

The master program is started first. Ports for sending and receiving are opened. only two ports are used. The communications handler is then installed as an interrupt driven process. Finally the program waits for the user program to call the send routines.

After the master program has started and its address are printed. the slave programs are started. Command line arguments for the slave program are used to specify the master and receiving hosts and ports. The slave program first opens sending and receiving ports, and sends an ADD command packet to the master program to identify itself as a participating slave. The program then is blocked until the master program has acknowledged the command packet.

The master program's interrupt driven communications handler is started by the message from the slave. The communication handler processes new slave commands, data sent commands, and slave removal commands. When a new slave command is received, the command packet is parsed for the slave's host and port number. This information is kept in a circularly linked list. After the new slave has been added to the list it is assigned a unique identification number and the new slave command is acknowledged. While the communications handler is executing communications interrupts are ignored.

After all the slaves have been registered the data transmission can begin. The user program calls the master program program to request transmission of a data file. This is currently done by a 'C' language call to `para_send(`*File name*`)`. After the specified file has been opened and its length determined, the first n data transmission command packets are constructed. After all n have been constructed they are sent and timers are started on each. The packets are preconstructed so that interrupt processing of the SENT command packets sent by the slaves does not interfere with the calculation of packet numbers and file offsets. After the first packets are sent, control is passed back to the application program.

When a slave processor receives a SEND command packet, it parses the packet for the file name, packet number, and offset. Using this information it constructs a data packet containing the file name, offset, packet number and data from the file beginning at the offset. This data packet is sent to the receiving host. After the data packet has been successfully sent, a SENT packet is sent back to the master program to acknowledge that the data has been sent. The slave program then is blocked until the next command packet is received.

When the master program receives a SENT command packet, it first stops the timer for the target data packet, then determines the next data to send and builds a SEND packet for this data. This command packet is then sent to the slave that originated the

SENT command packet and a new timer is started. If the target file has been completely sent, a FILE_COMPLETE command packet is constructed and sent to the slave. The FILE_COMPLETE packet is sent to notify the receiver that the file has been completely transmitted.

If a timer expires before a SENT packet is received, the responsible slave is removed from the list of active slaves and the lost packet is sent to the next free slave.

When the master program terminates, by the user issuing a quit command, a TERM command packet is sent to all slaves and the test is complete. Figure A.1 contains the pseudocode for the master and slave transmitting programs.

A.2 Receiving Protocol

In some ways the receiving component of the parallel TCP can be seen as the reverse process of the transmitter. As with the transmitter, at least 3 machines are needed for the master and slave programs. The programs' functions, outline below, are similar to that of the transmitter, but the program interactions are not.

A.2.1 Master and Slave Program Functionality

As with the transmitting code, the master program is used to provide an interface to the user's program. The master performs the following functions:

- Recognize the existence and identities of slave processors
- Manage slave processors terminations

However, it must also

- Provide an interface to the user program for data delivery
- Recognize aborted messages and deliver them as such
- Block the user program from execution while a message has not been delivered.

As before, the slave program can be run any number of processors. This program receives data from the transmitting slaves and passes it's description to the receive master for ordering and delivery. This program's functions are:

- Identify themselves to the running master program
- Interpret the command packets sent from the master and perform the functions.
- Save the received data in the correct file at the correct location
- Notify the master when packets are received, and when transmission of a message is complete or aborted.
- Interface to the transmitting slave processors

As will be discussed in the next section, the slave processors are not concerned with the number of channels that are being used or the packet order and distribution, so there is no need for them to be concerned with data retransmission. Additionally, because the data link between the transmitting and receiving slaves are all TCP links, data transmission is handled automatically between the corresponding slave programs.

A.2.2 Master and Slave Interactions

As with the transmitting code, the master is started first. Its interrupt driven communication handler code is installed and initialized, then its address is printed so that the slave programs can identify themselves.

The slaves programs are started next, given the address of the master and transmitting slave programs, the communication sockets are opened. As with the transmitting slaves, the receiving slaves send an ADD command packet to the master program to identify themselves as participating slaves. After signup is complete, the slaves wait for an incoming message from the transmitting slaves, or a status request from the receiving master. The status request is only used for debugging.

When a packet is received by a receiving slave, it is parsed for data, file name, buffer length, offset, and packet identification information. The buffer is then written to the specified file at the specified offset from the beginning. Once the data has been saved, the file name and packet number are sent to the receive master for ordering. If a negative packet number is received, this indicates that the transmission is complete or has been aborted. Aborted messages are identified because the data offset of the received termination message is also negative. This information is also passed to the receive master. Because the slaves are not concerned with packet ordering or message completeness, they simply go through the process of receiving messages, processing them, and returning to a blocked, reading state. If it senses a termination of communication with the transmitting slave, the receiving slave also terminates. Because the receiving master is managing the slaves, a final termination message is sent to the receive master prior to the slave's termination.

When the receiving application requests a message, it is blocked until a complete or abort message has been received. As packets are received and their information sent to the receive master, they are read from disk and returned to the requesting receive application. Pseudocode for the receive master and slave program can be found in Figure A.2

```
                              Master Program

1. Call to initialize the communication routines
2. While termination signal has not been received do
   (a) Wait for command packet or transmission call
   (b) If transmission call then
       i. Divide the message into sections
       ii. Send the first set of sections to the slaves
       iii. Return control to the calling program
   (c) If ADD command
       i. Add the new slave to the list
       ii. Acknowledge the ADD command to the slave
   (d) If SENT command
       i. If all other packets have been sent and acknowledged, send end of file message
       ii. If more data needs to be sent, send it via this slave
3. end do
4. call to close communications

                              Slave Programs

1. Initialize communication
2. Signup with the Master
3. While a TERM message has not been received do
   (a) Wait for a command message
   (b) If SEND command
       i. If the current transmission is not complete, send an abort message to the
          receiver
       ii. Read the data and send it along with the file name and new offset
   (c) If TERM command, close communication and stop
4. end do
```

Figure A.1: Transmitter Pseudocode

```
                         Master Program

1. Call to initialize the communication routines
2. While termination signal has not been received do
   (a) Wait for command packet or read request
   (b) If read request then
       i. If a message has been received and processed, return the data
       ii. else block the calling program
   (c) If ADD command
       i. Add the new slave to the list
       ii. Acknowledge the ADD command to the slave
   (d) If RECV command
       i. If this is a end of file message then save the data in a raster format for the
          receive application
       ii. If abort message then
           A. Fill in the blank spots on the image
           B. Save the data in raster format for the receiver
3. end do
4. terminate all slaves
5. call to close communications

                         Slave Programs

1. Initialize communication
2. Signup with the Master
3. While a TERM message has not been received do
   (a) Wait for a command message
   (b) If data packet
       i. Save the data in the correct file at the specified offset
       ii. Send the packet information to the master receiver
   (c) If TERM command. close communication and stop
4. end do
```

Figure A.2: Receiver Pseudocode

Appendix B

Testbed Analytical Model

B.1 Model Definition

For purposes of analysis, this analysis treats the application and its interface with the transport layer as a single process, called the *master* process. Similarly, the transport layer, the media access control, and the data layer are treated as a single process, called the *slave process*. The master process splits the application data into *segments* and sends them, or makes them available, to the slave processes. Since there is a one-to-one relationship (in terms of a physical connection) between the sender slave and the receiving slave, they are refer to as a *slave pair*. Thus the system has N slave pairs. In addition, the following assumptions are made.

- The number of segments transmitted through slave pair i is denoted by S_i.
- Each of the S_i segments could be of different lengths, where $d_{i,j}$ is the size of segment j.
- Since the data in segment j may be transmitted by the media access layer in several packets (depending on the packet size and the segment size), the number of packets in a segment is denoted by $P_{i,j}$.

This model is concerned with determining the end-to-end performance characteristics of the testbed system. To this end the rate of throughput at the application level is the target calculation. Accordingly, given the size of data that needs to be transmitted from sender application to the receiver application D_a, and given the total physical bandwidth available to the application I_a, the total delay from delivery of data to the master process to the time the data are completely received by the receiver application is also of interest.

The delays encountered in delivering the message from application to application are as follows:

Delay at the Sender Slave. This consists of wait time for a slave to receive a segment from the master, preparing the packets for transmission, and the delay in accessing the channel. In addition, depending the acknowledge and sliding-window protocols, a slave may need to wait to receive acknowledgments from its counterpart on the receiving side before continuing the packet transmissions. These terms are further explained below.

- $ts_{i,j}$ - The time slave i must wait before it receives segment j from the master. This is the time between when slave i completes a transmission of segment $j-1$ and when it is assigned segment j. This time is influenced by the communication mechanism between the master and the slave. If they communicate through a bus, the bus contention and bus transmission delay needs to be considered. Similarly, the delay also depends on whether the master sends the entire segment or just a control message indicating the position of the segment in a shared memory. In such a case, the time is influenced by the memory access speed, memory-slave transfer speeds, and any other memory contention.

- $to_{i,j}$ - Total overhead of packetizing segment j at slave i. This overhead is a protocol overhead which packetizes the received segment and adds overhead bits (actually added by several layers of the communication protocol between the slaves) prior to transmitting on the physical channel. Since, the overhead is generally constant for a packet, the total overhead is a product of per packet overhead and the number of packets in segment j, $P_{i,j}$.

- $tak_{i,j}$ - Total delay incurred due to the absence of acknowledgments at slave i in transmitting fragment j. This delay depends on the communication protocol between the slave pair i. For example, in a TCP like protocol, where the transmissions are controlled by a moving window protocol, it is sometimes possible to transmit a message even before receiving the corresponding acknowledgment for the previous packet. In a stop-wait like protocol, however, this delay is incurred before the transmission of each packet. Obviously, this delay depends on the window size, loss of packets, loss of acknowledgments, TCP timer value, etc.

- $tsc_{i,j}$ - Total waiting time at sending slave i to access the channel. This term represents channel contention for all the packets in segment j and depend on the load on the network and the channel allocation protocol. Once again, knowing the average waiting time per packet, the delay may-be obtained by the product of total number of packets and the per packet delay.

Transmission and Propagation Delay. The transmission and propagation between slave pairs is denoted by $tt_{i,j}$ represents the total time required for transmission and propagation of segment j from slave i. For each packet transmitted between slave pair i, a transmission delay (which depends on the channel bandwidth and the packet size) and a propagation delay (which depends on propagation speed and the distance between slaves) are involved.

Effect of retransmissions. Due to loss of messages, transmission errors, etc., it may be necessary to retransmit messages between the slave pairs. This is represent by $nr_{i,j}$ denoting the average number of packet retransmissions at slave i during the transmission of segment j.

Delay at the Receiver Slave. The delay at the receiver is attributed to the overhead in processing the packets, delivering them to the master, and acknowledging the received packets. This delay is denoted by the following terms.

- $ro_{i,j}$ - Total overhead at receiving slave i needed for processing packets corresponding to segment j. This term includes the time to read a message from the channel, perform the transport protocol processing, and to reassemble the packets in to segment j.
- $rsc_{i,j}$ - Total waiting time at (receiving) slave i to access the channel. This is the waiting time that receiving slave i waits for the channel when trying to acknowledge each packet. Indirectly, this term s accounted in the $tak_{i,j}$ and is influenced by the traffic on the channel and the protocol.
- $rt_{i,j}$ - Total transmission and propagation delay to acknowledge segment j. This is the total time that the receiving slave i spends sending acknowledgments for the packets in segment j. Since the acknowledgment messages are generally small in size, the transmission delay may be insignificant.

The total elapsed time between a sending application initiating the transfer of a message to the time it is completely received by the receiving application, denoted by TD, is now expressed as

$$T_i = \sum_{j=0}^{S_i} ts_{i,j} + to_{i,j} + tsc_{i,j} + tt_{i,j} + ro_{i,j} + rsc_{i,j} + rt_{i,j} \quad \text{(B.1)}$$

$$TD = \max(T_1, T_2, \ldots, T_N) \quad \text{(B.2)}$$

$$\text{(B.3)}$$

From here, the offered throughput to the application, NTh, may be expressed as

$$NTh = \frac{\sum_{i=1}^{N} D_i}{TD} \tag{B.4}$$

$$D_i = \sum_{j=0}^{S_i} d_{i,j} \tag{B.5}$$

B.2 Values Used in the Model

To the terms defined above, the following values were assigned.

- $ts_{i,j}$ - In the testbed, after each segment is sent, the slave must send a message to the master to indicate that i has sent segment $j-1$. The master then determines if more data are to be sent, and then sends a message to the slave giving the segment number and offset of its segment j. Code was written to measure this this parameter, and the average observed, when two slaves were contending for the master, was 3400 *µseconds*.

- $to_{i,j}$ - The underlying protocol for the testbed is TCP. According to Clark et al. [7], the number of instructions needed to send a create a TCP packet is 235, and the number of instructions required by IP is 61. These values do not include the instructions required for device drivers and other operating system overhead, but they should give us a magnitude representation of the processing required. The testbed is running on Sun SparcStation 1 computers that have a processing speed of 12.5 MIPs. With this the calculated the overhead of one packet is 24 *µseconds* and the overhead for each segment is 24 * the number of packets in a segment ($P_{i,j}$). For the experiment segment sizes this give values of 24, 600, 3,000, 6,144, 73,728, 153,600, 245,560, and 307,200 *µseconds*.

- $tsc_{i,j}$ - Because Ethernet is being used for machine connections, it is convienent to use the formulas found in [31] and find an average waiting time for accessing a channel as

0.014 *μseconds* for the clear channel and 0.015 *μseconds* loaded channels.

- $tt_{i,j}$ - The sustained throughput measured on the Ethernet channels is 5 Mbits. Based on the segment sizes begin used the segment transmission times is computed to be $819*10^{-6}$,, 0.0204, 0.1024, 2.516, 5.242, 8.388, and 10.485 seconds respectively.

- $ro_{i,j}$ - As discussed earlier, Clark et al. [7] provides the number of instructions for TCP and IP processing of incoming data packets to be 186 and 57 instructions respectively in a highly optimized implementation. As with the value for $to_{i,j}$, these instruction counts are used to give an order of magnitude estimate. No additional headers are used by the testbed when creating the packets, so no additional processing is required at the receiving end. With this information, the time computed on a Sun SparcStation is 19 *μseconds* per packet and 19, 475, 2.375, 4,864, 58,368, 121,600, 194,560, and 243.200 *μseconds* per segment respectively.

- $rsc_{i,j}$ - This is the time spent waiting for the channel when sending the acknowledgment. This time should be the same as was needed to wait when transmitting 1 TCP packet or 0.014 *μseconds* for the clear channel and 0.015 *μseconds* for the channel with background traffic per acknowledgment.

- $rt_{i,j}$ - The transmission and propagation time for the acknowledgment is easily computed based on the sustained Ethernet throughput, the number of packets sent $P_{i,j}$, and the size of the TCP acknowledgment packet (4096 bits). This gives us the segment values of $125*10^{-6}$, $3124*10^{-6}$, $15625*10^{-6}$, $32000*10^{-6}$, 0.384, 0.8, 1.28, and 1.6 seconds.