Packet Train Model: Optimizing Network Data Transfer Performance

by
Cheng Song

Computer Sciences Technical Report #867
August 1989

PACKET TRAIN MODEL: OPTIMIZING NETWORK DATA TRANSFER PERFORMANCE

by

Cheng Song

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1989

ABSTRACT

Network transmission bandwidth of 100 megabits per second is now available for LANs and bandwidth over gigabits per second will soon be available for WANs. A growing number of communication applications demand the support of high speed bulk data transfers. Unfortunately, effective user level data throughput has been far below the network bandwidth due to the bottleneck effect of packet software processing.

This thesis presents a new approach to reducing packet processing overhead, in particular, reducing the number of invocations of host system support and protocol processing routines. The packet train model, based on observed network traffic burstiness, provides a framework in which a large block of data can be transferred as a large packet without incurring various problems, and a proper selection of transport layer functions can be offloaded to the network front-end without overloading the front-end processor.

A packet train is a special form of large data delivery over a network. On a transmission medium, a train is a sequence of consecutive packets, spaced within a maximum allowed inter-packet gap and loaded with data for the same transport layer entity. The size of a train is dynamically determined by the amount of data packetized for transmission and by interruptions of urgent data transmitted onto the medium. At data sending and receiving hosts above the medium level, a packet train is handled as one transfer unit for its protocol processing.

The thesis presents measurement results which provide motivations for the packet train approach. It describes in detail designs for the major packet train mechanism components, both hardware and software. Results of a simulation evaluation of the packet train scheme vs. the traditional non-train scheme indicate that the packet train is a viable approach to improving bulk data transfer throughput under a wide range of system parameters and traffic load conditions. Finally, the thesis discusses extensions needed for the packet train approach to benefit data transfer performance in a WAN environment.

ACKNOWLEDGEMENT

Professor Lawrence H. Landweber, my advisor, has been a perennial source of inspiration and encouragement ever since I started as a graduate student. I am greatly indebted to him for his excellent guidance, continued encouragement, and generous support, as well as his extraordinary patience in improving my writing and communication skills. I will always admire him for his astute thinking and impeccable clarity in communicating his ideas. I will also miss his invaluable advice in matters well beyond this thesis research.

I would also like to thank the readers of this thesis: Professor Miron Livny, who spent much time to help me with his DeNet simulation tool and methodology, and Professor Stuart Friedburg, who gave me valuable comments on my thesis. I would also like to thank Professors James Goodman and Arne Thesen for having the kindness to serve as members of my exam committee, and for providing helpful comments. I feel grateful to all the faculty for their help and advice.

I am grateful to all my friends and fellow students in the CS department and in the deparment's Systems Laboratory. In particular, Paul Anderson was always willing to discuss specific aspects of my work with me. Mike Litzkow's work on Condor greatly speeded up the progress of my simulation work. Amarnath Mukherjee was a considerate and helpful officemate whom I will miss in the future. Phil Pfeiffer's help and encouragement always cheered me up. Sheryl Pomraning's kindness and help always made me happy.

I am deeply grateful to my parents, my sister and my in-laws for their support and encouragement. My wife, Helen, and my sons, David and Stephen, endured years of sacrifice and hardships both material and psychological. I owe my family so much that I could find no words to express my deepest gratitude.

TABLE OF CONTENTS

ABSTRACT ii
ACKNOWLEDGEMENT iv
Chapter 1 Introduction 1
1.1. Background 1
1.2. Scope of the Study 3
1.3. Organization 6
Chapter 2 Related Work 8
2.1. Hyperchannel 8
2.2. LANCE 9
2.3. VMP Network Adaptor Board 11
2.4. XTP and Protocol Engine 12
2.5. Protocol Implementation Strategies 13
2.6. Optimization Techniques Based on Traffic Characteristics 15
Chapter 3 Network Measurement 19
3.1. Measurements of an 80 Mbps Proteon Ring 20
3.1.1. Measurement Environment and Methodology 20
3.1.2. One-to-One Data Transfer Results 21
3.1.3. Many-to-One Data Transfer Results 24
3.1.4. Discussion of the Measurement Results 27
3.2. Other Measurement Results 29
3.2.1. Measurement on Protocol Performance 29
3.2.2. Measurement on Network Traffic Characteristics 31
Chapter 4 Packet Train Model Motivation and Overview 33
4.1. Motivation 33
4.1.1. Reducing Host System Operation Invocations 34
4.1.2. Reducing Packet Interrupt Overhead 36
4.1.3. Problems with Large Packets 37
4.1.4. Proper Offloading of Protocol Functionality 38
4.2. The Packet Train Model 42
4.3. Packet Train Mechanism Design Assumptions and Objectives 47
4.3.1. Traffic Characteristics 47
4.3.2. Network Characteristics 48
4.3.3. Design Objectives for the Packet Train Mechanism 49
4.3.4. Major Components of the Packet Train Mechanism 51
Chapter 5 Network Controller Design Features 53

5.1. Network Controller Overall Architecture 53
5.2. Medium Access Controller 57
5.3. Buffer Memory and Its Organization 59
5.4. Coordinating Accesses to Packet Descriptors and their List Registers 63
5.5. Packet Train Interrupts 64
5.6. On-board processor 66
5.7. Implementation Consideration 67
Chapter 6 Transmission and Reception of Packet Trains 72
6.1. Packet Train Transmission 72
6.1.1. Packet Train Transmission and IEEE 802.5 MAC Protocol 73
6.1.2. Packet Train Transmission and FDDI MAC Protocol 75
6.2. Interruption of a Packet Train for Urgent Packet Transmission 78
6.3. Error Handling for Packet Train Transmission 79
6.4. MAC Controller Operation for Packet Train Transmission 81
6.5. MAC Controller Operation for Packet Train Reception 83
6.6. Packet Train Reception Interrupt Handling 86
Chapter 7 Data Structures and Transport Layer Functions 91
7.1. Major Data Structures for the Packet Train Mechanism 91
7.1.1. Service Request/Status Records 91
7.1.2. Protocol Lookup Tables 97
7.2. Train Packet Encapsulation 101
7.3. Packet Train and Protocol Offloading 102
Chapter 8 Simulation Study 108
8.1. Simulation Model 109
8.2. Simulation Parameters 114
8.3. Simulation Results 116
8.3.1. Bulk Data Transfer Time Comparison 116
8.3.2. Train Sizes 119
8.3.3. Urgent Packet Transfer Time 121
8.3.4. Packet Size vs. Bulk Data Transfer Time 122
8.3.5. Effects of Large Packets 124
8.3.6. Summary 125
Chapter 9 Packet Train Approach in WAN Environments 138
9.1. WAN Traffic Characteristics 138
9.2. Advantages of the Packet Trains 140

9.2.1. Avoidance of Packet Fragmentation and Reassembly 140
9.2.2. Cut-through switching 143
9.3. Issues and Difficulties with the Packet Train Approach 145
Chapter 10 Conclusion 149
10.1. Summary 149
10.2. Contributions 151
10.3. Future Work 153
Appendix A: Train Packet Formats 154
Appendix B: MAC Protocol Finite-State Machines with Extensions 158
Bibliography 163

Chapter 1

Introduction

1.1. Background

In recent years, raw transmission bandwidth for computer networks has been increasing dramatically. At present, 80 Mbps transmission bandwidths has become common for local area networks[31]. The FDDI standard with 100 Mbps bandwidth is being prototyped[36]. For wide area networks, plans to utilize 45 Mbps T3 transmission links for the coast-to-coast NSFNet backbone are being implemented[28]. Current fiber optic technology supports a transmission rate of 1.7 Gbps on a single fiber with an order of magnitude increase likely in the next ten years. In addition, both local area networks and fiber optic technology provide very low error rates, on the order of $10^{(-9)}$ to $10^{(-12)}$.

At the same time, computer communication applications are also rapidly expanding in variety and performance requirements. Certain bulk data transfer applications demand not only the transfer of a large volume of data, but also that the transfer be completed within an extremely short period of time. For example, a remote display of computer graphics requires the transfer of a screenful of graphics data with a response time acceptable to a human eye. Multi megabyte per second data transfer rates are required for real time data collection in fields such as

astronomy, medical imaging, high energy physics and seismic exploration; for CAD/CAM computer graphics applications involving workstation/mainframe interactions; for transfer to a remote display of data generated from supercomputer calculations; and for transfer of full motion color graphics images.

Unfortunately, effective user to user data transfer throughput has been far below the raw bandwidths of existing local area networks. Given the observed low aggregate network traffic volumes[1,26,38], network bandwidth is not now and is not likely in the future to be a throughput bottleneck. Performance degradation occurs because of the characteristics of the network interface device, the connection between the interface device and the host computer, the host operating system, and the communication protocol software.

The goal of our research, as reported in this thesis, is to find ways to narrow the gap between network transmission technology and communication application performance, so as to to bring the benefits of the advances of networking technology to computer users. In more specific terms, we would like to achieve for bulk data transfers a user level throughput close to what the network bandwidth is capable of supporting. To achieve this goal, we must overcome performance bottlenecks for bulk data transfer throughput, and at the same time not compromise response time performance for small urgent high priority data transfers.

The basic approach of our work is to take advantage of traffic characteristics to improve data transfer performance. One can find analogous approaches to improving

performance in other areas of computer sciences. For example, in compiler object code optimization, data flow analysis is done to better allocate registers based on which variables are likely to be referenced soon. By keeping variables in registers that will be accessed soon or frequently, unnecessary register load and store instructions can be avoided and program execution speed increased. In operating systems research, work has been done to analyze program memory reference patterns to determine which pages are more likely to be referenced soon. Knowledge of such reference patterns can be used to design more efficient paging algorithms. These performance optimization techniques are based on knowledge of their own "traffic" characteristics. Our work is new in that a unique model is designed to improve data transfer performance, and this model not only takes advantage of traffic characteristics but also intensifies certain existing traffic burstiness properties to further benefit bulk data transfer performance without having adverse effects on other classes of traffic.

1.2. Scope of the Study

In this thesis, we propose, describe and evaluate a new approach to achieving our research goal: a data packet transmission and processing strategy and a protocol offloading methodology that we call the "packet train model". Our approach is based on measured network traffic characteristics and on the fact that software processing for packets forms the most significant performance bottleneck in recent local area networks. As will be discussed in later chapters, network traffic is highly

bursty. Within each packet burst, there is a high correlation between successive packets with respect to their source and destination addresses. The Poisson distribution model for packet arrival, which is often assumed in theoretical network performance studies, can not be justified by measurement results.

This observation, based on various measurement results, provides the motivation for the packet train model. The bulk of this dissertation deals with the design and evaluation of the packet train model, and a description of how, by minimizing packet processing overhead, this packet train model can improve bulk data transfer throughput.

Packet processing overhead consists of two major components, protocol processing overhead and host operating system support overhead. When data packets move through different system components, such as the network medium, the network interface device, the host bus, protocol modules and various host operating system support facilities, the protocol and host operating system processing, i.e., the time spent by the host CPU or a protocol front-end on processing data packets, is often the most severe performance bottleneck limiting data transfer throughput. This is especially true when the speed of both the network medium and the host bus is at least several megabytes per second. The packet train model is designed to overcome the software processing bottleneck by minimizing repetitive invocations of expensive protocol and system support operations. It should be pointed out that this study does not deal with improving existing protocol functions,

but rather with how they should be organized, offloaded, and executed in the context of the packet train model.

This thesis focuses on

(1) Extensions to the token ring medium access control protocols to accommodate packet train transmission and reception.

(2) Hardware extensions to existing network interface device designs.

(3) Offloading of network and transport layer protocol functions to a network interface front-end for packet train processing.

The thesis presents a study and design of these key components for the packet train model. However, the thesis is not intended to be a study of network interface device or protocol design in the general sense.

The research method includes measurement of actual performance, specification of proposed mechanisms, and evaluation by simulation of the proposed solution. Measurement was done to study raw data transfer performance on an 80 Mbps token ring. We have specified the design for the needed mechanisms, both hardware and software, to support the proposed packet train model. Simulation was employed to study the performance benefits of the proposed packet train mechanism. We avoided the "trap" of simulating every detail of our proposed packet train mechanism. Instead, we concentrated on the major components of the packet train mechanism and on the performance gains that could be achieved under different

traffic and system configuration conditions. This has helped highlight the performance benefits and limitations of the packet train model.

1.3. Organization

The rest of the dissertation is organized as follows. Chapter 2 describes related work. There is a large body of literature concerning bulk data transfer performance. We discuss the most relevant of this work in Chapter 2. Chapter 3 is a summary of our and other network measurement results. This chapter gives empirical evidence as to why the network interface device is a key component in achieving high data transfer throughput at the user level, and why the packet train model is feasible in terms of current network traffic characteristics. Chapter 4 describe the motivation and gives an overview for the packet train model, and our strategy for offloading protocol functions from the host to the network interface front-end. Chapters 5 to 7 describe the various components of the packet train mechanism. Chapter 5 discusses the design of a network controller, which can support transmission and reception of packet trains and efficient protocol offloading. Chapter 6 analyzes both IEEE802.5 and FDDI MAC protocols in view of packet train transmission. This chapter also gives details of the extensions needed for the network controller's medium access controller component for transmission and reception of packet trains in the above two token ring MAC environments. Chapter 7 describes data structures used by the network controller to process packet trains and execute offloaded protocol functions. It also gives further detail on offloading transport layer protocol functions to support

the packet train mechanism. Chapter 8 presents the simulation study. Chapter 9 provides a discussion of packet train model ideas to a wide area network environment. It points out potential benefits from extending the packet train model to such an environment. Chapter 10 summarizes the research results and outlines future work.

Chapter 2

Related Work

In this chapter, we review related work in the following areas.

(1) Local area networks that support high speed bulk data transfer.

(2) Network interface devices.

(3) Protocol implementation strategies.

(4) Optimization techniques for data transfer performance based on network traffic characteristics.

2.1. Hyperchannel

Hyperchannel[10] is one of the earliest packet switched computer interconnection that was intended to support large volume high speed data transfers. Hyperchannel was designed to support communications between large computers and between computers and mass storage devices in a computer center or a large data processing center environment. A Hyperchannel differs from commonly used LAN's in the following two aspects. First, each node machine on a Hyperchannel is connected via a special purpose adaptor to 4 coaxial cables each having a 50 Mbps data rate. Second, the medium access control protocol allows a pair of nodes on a cable to exchange a long sequence of data. Once the sequence of data exchange begins, no other nodes can use the same cable to transmit their packets in the middle of such a

sequence. The combination of long uninterrupted sequence of data exchange and special hardware adaptor board (complex and expensive to build, $40,000 each about three years ago) provides the basis for Hyperchannel's high throughput bulk data transfer capability.

Although the basic ideas (long sequences of bulk data and special processing for such data sequences) used by Hyperchannel shed some light on how one would go about achieving high data throughput, Hyperchannel is unattractive for a broad range of communications applications because it gives little consideration to short bursty traffic generated by applications like remote login and interprocess communication that are very common on LAN's. Due to its cable interfacing and medium access control protocol, Hyperchannel is restricted to a maximum length of 1 Kilometer and allows a maximum of a few tens of connected nodes. Therefore, it is not suitable for a general computing environment.

2.2. LANCE

LANCE, standing for Local Area Network Controller for Ethernet[24], is one of the most commonly used network controller boards for micro- and mini-computers. The key features of a LANCE chip are the following.

(1) It can be easily interfaced to 8086, 68000, z8000, and LSI-II microprocessors. Hence, a co-processor for packet processing can be built on a network interface device using a LANCE chip.

(2) It allows multiple buffer descriptors to be queued in two circularly linked lists for transmission and reception respectively. Further, these descriptors can be set to point to host memory buffers, allowing packet data to be directly fetched/stored from/to host memory.

(3) It provides on-board DMA and buffer management.

A network interface device with a co-processor using the LANCE chip would typically look like Figure 2.1.

We choose to describe the LANCE device here, not because it has unique features that other modern interface cards do not have, but because it is a widely used, typical network interface card. It does have several shortcomings. First, its buffer descriptors are only for packet buffers, and as such do not register complete packet transmission or reception status information needed for interrupt handling. With one interrupt logic, this means that interrupt handling turn-around time can still

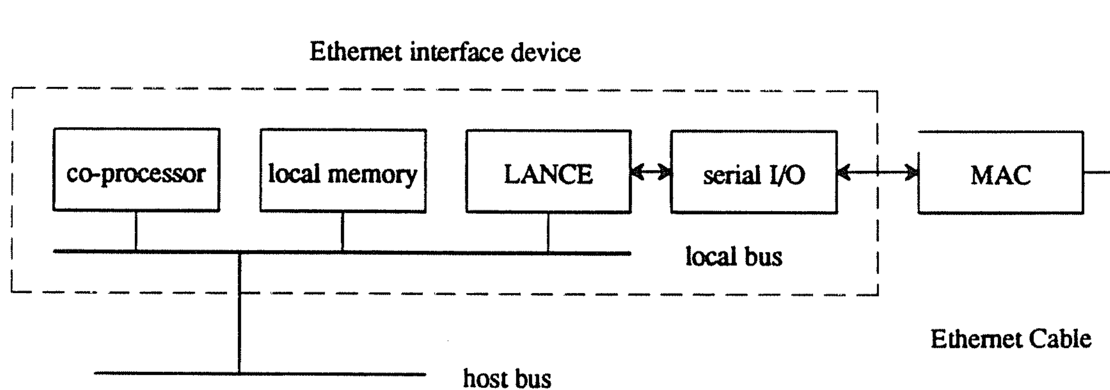


Figure 2.1

cause packet loss, if it is longer than packet arrival time. Second, the packet data transfer path, a co-processor and the co-processor's memory all share the same local bus. Bus contention for packet data and the co-processor's memory access often creates a performance bottleneck. In addition, the LANCE lacks other features that we will discuss later in the thesis.

2.3. VMP Network Adaptor Board

The VMP Network Adaptor Board (NAB) developed at Stanford University[22] has many improvements over the LANCE. Some of its prominent features are as follows:

(1) It has an on-board co-processor. This co-processor manages the buffer memory and a host bus block copier interface, and performs certain packet processing functions.

(2) A packet pipeline provides packet header encryption/decryption and checksumming in a pipelined fashion. These functions can be performed in real time while receiving a packet.

(3) Separate channels are provided for packet data and for the co-processor's access to its local memory.

One problem with the NAB design is that it is tightly bound to the VMTP[6] protocol header structure and is very protocol specific. In particular, its packet pipeline can only handle VMTP packets. Because its design is strongly influenced

by VMTP, a transport protocol based on the request/response transaction model, another problem is that this interface device can not improve bulk data transfer performance by taking advantage of actual packet arrival characteristics.

2.4. XTP and Protocol Engine

XTP (Xpress Transfer Protocol) is a lightweight transport protocol with an internetwork service conforming to OSI layers 3 and 4[9]. The protocol engine is a set of VLSI chips that interfaces with 100 Mbps FDDI networks and provides transport level functions in hardware[8], in particular, XTP functions. XTP and the protocol engine are related parts of a research effort at Silicon Graphics Inc., the premise of which is that network software would not be able to keep pace with the 100 Mbps FDDI or future 1 Gbps networks. The proposed solution is a VLSI implementation of network and transport level protocols. In order to have a transport protocol suitable for such VLSI implementation, XTP was designed to be simple. For example, the distinction between datagram and reliable connection service is "blurred" in XTP by treating each datagram as a short-lived connection. There are only two types of packets, control and data types. In addition, packet header structures are designed to facilitate direct hardware processing.

The "Protocol engine" approach is rather ambitious in that it attempts to implement in a small number of integrated circuits such transport level protocol functions as buffer control, address and routing algorithms. The XTP protocol attempts to be general-purpose by providing traditional stream services, bulk

transport, real-time reliable datagram service, flow/error/rate control, selective retransmission, message boundary preservation, multiple addressing plans, out-of-band signaling, reliable multicast mechanism, and a multipath capability.

The success of this project has yet to be proven. In our view, with the complicated functionalities listed above, we feel that the XTP is no longer a lightweight transport protocol that can be easily implemented in hardware. We also see two other major problems with the "protocol engine" approach. The first is that it is also protocol specific, i.e., a protocol engine can only support the XTP protocol. The second is that the important performance problem of how to interface protocols with host operating systems is not addressed by this approach. As measurements of network software performance indicated[42], host operating system support is a major component of packet processing overhead. Fore example, in a particular TCP implementation, to send a packet, only about 200 80386 machine instructions are executed, but far more code needs to be executed by the host operating system to support the TCP execution (just a timer package could cost that much)[11].

2.5. Protocol Implementation Strategies

One of the most important performance considerations in implementing protocols is how the protocols should be integrated into the host operating system. Two approaches are usually taken.

One advocated by Richard Watson[41] uses a network server that behaves like a network communication agent to the host operating system kernel. The kernel implements process management and inter-process communication functions. When it detects a message addressed to a remote process, it passes the message to this agent and the agent invokes necessary protocols to reliably deliver it to its peer agent on the remote node on which the destination process resides. The advantages of this approach are greater modularity of system structure, minimal requirements for modification to operating system code and greater autonomy for each node. RIG (Rochester Intelligent Gateway)[25], and Accent (kernel for the Spice distributed system at Carnegie-Mellon University)[32] are among the systems that have adopted this approach. The disadvantage is the overhead of one extra process switch from the kernel to the process implementing the network agent. When an application process has a retransmission timeout mechanism, each ACK and retransmitted message will also incur this overhead. There is also the difficulty of choosing the right priority level for the network server process. If its priority is too low, message passing may experience unacceptable delay. If it is too high, it could "kill" a node system and thus destroy its autonomy [4] when it takes all CPU cycles in response to very fast and continuously incoming packets sent by some malfunctioning node.

The second approach is to integrate the protocols into the operating system kernel. This eliminates the extra process switch. V [5] , Locus [30] and Domain [26] use this approach. For example, the Locus kernel has a fixed number of lightweight

processes instantiated at the time of system initialization, which share system global variables and call system subroutines directly. Network requests are put on a system queue and each of these processes looks at the queue for work to do. Performance of systems using this approach is usually better than that of systems using the first approach.

Using the second approach of integrating protocols into the host operating system kernel, it becomes feasible to implement certain protocol functions at packet I/O event interrupt level to guarantee priority and speed for protocol execution[39]. It is also possible to consider offloading protocol execution to a dedicated co-processor on the network interface front-end. Protocol offloading guarantees the availability of processor cycles for protocol execution, and saves valuable host CPU cycles for the tasks for which host CPU is really designed.

2.6. Optimization Techniques Based on Traffic Characteristics

A number of research projects have attempted to take advantage of network traffic characteristics to benefit data transfer performance. The following two are closely related to our work.

The first called "header prediction", was used by Van Jacobson[19] to avoid repetitive TCP protocol execution on BSD Unix systems. The idea of header prediction works as follows. When a packet has just been processed in the middle of a bulk data transfer, the TCP module is likely to know what the next packet will look

like, because of usual network packet arrival patterns, i.e., the next packet is highly likely to be the next packet in the same data transfer operation. The header of the next packet will be just like the one processed with either the sequence number or the ACK number updated (depending on whether the TCP module is sending or receiving). The TCP module can use such "hints" to lay out a tcp packet header, and to try to match this header with that of the next packet (a 14 byte comparison). If the prediction is correct, then only a checksum need be done and the packet data is appended to the socket buffer. It then wakes any process that is sleeping on the buffer. If the prediction fails, full protocol processing must be performed for this packet.

The saving in TCP processing is about 220us on a Sun-3/60 machine. The comparison takes 6us extra time. Some measurements[19] indicated that on bidirectional data flows, the prediction would win slightly more than half the time. On unidirectional flows, the prediction would almost always win.

220 us is only a small fraction of the total packet processing time. This is because the saving is only for TCP processing. Other expensive operations, such as host operating system operations, could not be saved by this header prediction technique. Because a new packet header is predicted by software at the TCP level, the overhead of buffer management, packet queue manipulation, context switching, etc, has already been incurred when packet processing has reached this point.

The second optimization technique, "Optimistic Blast"[3], also takes advantage of the high probability that the next packet received by the destination host is the next packet in the bulk data transfer. A bulk data transfer protocol using this technique makes an optimistic guess that the next packet is the one expected, and sets up its network interface in such a way that the data of the next packet is directly deposited in the anticipated location in the host memory. This way, no data copying is required. However, if the guess turns out to be wrong, some necessary corrective action has to be taken.

This technique was first deviced and experimented with at Rice University[3]. Some preliminary experiments were performed using the LANCE Ethernet interface (see 2.2) on a collection of SUN 3/50 workstations connected by a 10 megabit Ethernet. Results indicate that when the rate of wrong guesses is sufficiently low (i.e., invocations of corrective action are not frequent), the average performance of optimistic blast protocols is superior to that of their pessimistic counterparts (always check packet header and copy packet data), especially on single user workstations.

This technique reduces data copying to a certain extent, but it does not benefit software processing performance. Although it is important to reduce data copying, especially for large packets, its effect on total transfer throughput would not be dramatic when data transfer size is large and data copy for packets can be overlapped with packet transmission and software processing. If software processing overhead is the performance bottleneck, the benefits of such a technique

are limited. To take full advantage of network traffic characteristics, hardware assistance at the level of the network interface device is required. This entails support to "group" packets in order to reduce software processing overhead and data copy expenses.

Chapter 3

Network Measurement

In this chapter, we present results of several network measurements, performed by the author and others, that provide motivation for the packet train model.

Our measurements were performed during late 1986 and early 1987 on the 80 Mpbs Proteon token ring system, using the software developed for the Crystal project[14]. At that time, the 80 Mbps Proteon token ring was one of the fastest commercially available local area network products. Our measurements provide insight into problems at the network interface device level and their effects on maximum "raw" data throughput. Since "raw" throughput puts an upper bound on user-to-user data transfer throughput, an understanding of "raw" throughput bottlenecks is important for understanding how one might achieve user level throughput close to what the network medium can accommodate.

The measurement results by others provide information that we were not able to obtain from measurements within the Crystal experimental environment. Of particular interest are protocol performance and network traffic characteristics in non-experiment (i.e., production) environments.

3.1. Measurements of an 80 Mbps Proteon Ring

3.1.1. Measurement Environment and Methodology

The measured Proteon ring [31] connected 17 VAX 750's and a VAX 780. The VAX 750 software environment consisted of a minimal communications kernel and the measurement test process. Since the goal was to study ring interface device performance problems, artificial traffic was generated to create different ring traffic patterns and intensities. Measurements were taken when there was almost no other traffic on the ring. A simple C program repetitively called routines whose only functions were to send/receive datagrams to/from the interface device. The maximum packet frame size allowed by the Proteon ring was 2046 bytes. With 32 bytes for the packet header (for the transport protocol), the maximum and minimum packet body sizes were 2014 (2K) bytes and 2 bytes respectively. Additional details regarding these routines and their implementation can be found in[14].

The Proteon Ring interface device has: 1) a CTL board that provides primitive packet send, receive and error checking functions and 2) a host specific board that provides a DMA interface to the VAX 750 UNIBUS. The CTL board has two receive buffers and one send buffer, each of 2 Kbytes. DMA transfer over the UNIBUS has a maximum speed close to 5 Mbps.

We also obtained measurements using a locally built "coprocessor" attached to a Proteon ring CTL board (see Figure 3.1). This coprocessor uses a National

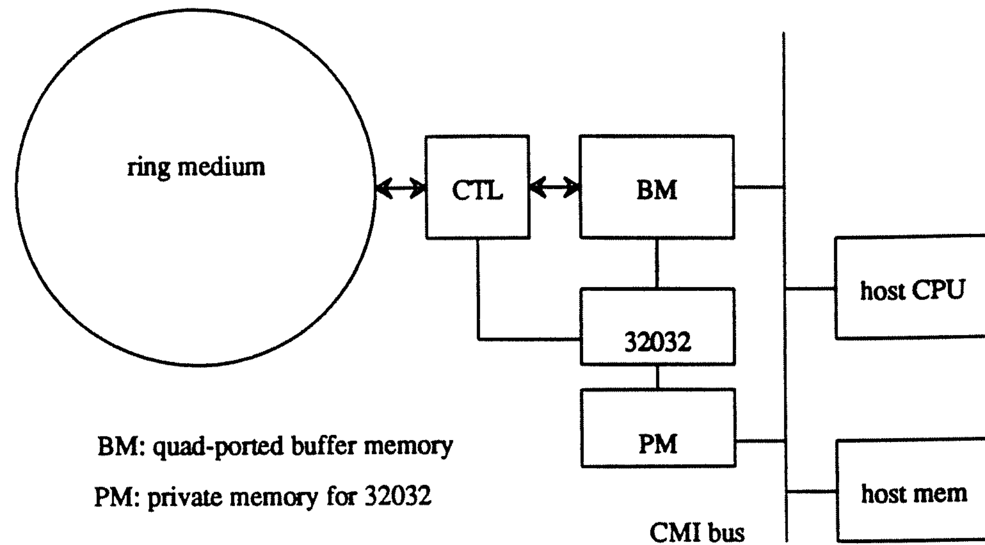


Figure 3.1: Coprocessor Architecture

Semiconductor 32032, and has a 32 Kbyte quad-ported memory for storing packets, connected to the CTL board with an effective bandwidth of 26 Mbps.

3.1.2. One-to-One Data Transfer Results

Tables 3.1 shows one-to-one data transfer rates and packet send delays between two VAX 750 memories for different packet sizes. The time to complete a transfer was from when the sender began to transmit the first packet till the sender received an interrupt indicating that the last packet had circulated back to it on the ring. Packet preparation time (1.42ms), interrupt processing time (0.38ms), and DMA transfer time were not overlapped. The measured rates are close to the maximum possible and, as expected, as packet size increases, the UNIBUS DMA speed limits the number of packets that can be sent/received.

PACKET SIZE (No Hdr)	Kbytes/Sec (No Hdr)	PACKETS/Sec	DELAY (ms)
2	1	502	2.0
8	4	502	2.0
32	16	502	2.0
128	61	477	2.1
512	183	358	2.8
2014	371	184	5.4

Table 3.1: Data Transfer Rate between VAX 750 Memories

Now consider overlapping the DMA transfer with software processing. Software processing plus interrupt processing requires about 1.8 ms. Hence, even with an arbitrarily high DMA rate, the VAX 750's processing power would limit transfers to about 550 packets per second or at the maximum size packet, about 9 Mbps. Note that for this configuration the maximum is actually lower because the DMA transfer cannot be overlapped with interrupt processing (the CTL board has only one send buffer). With a 5 Mbps DMA rate and using the 2 Kbyte maximum packet size, if software processing (1.42 ms) were overlapped with DMA transfer, this would leave about 4 ms to send each packet or about 250 packets per second. Hence, with the Proteon interface device and the VAX 750, one could, by overlapping DMA transfer and software processing, at best achieve a 4 Mbps data transfer rate, about 1 Mbps higher than the measured rate. Increasing processing speed or maximum packet size would not improve throughput because the DMA transfer time for a 2 Kbyte size packet is longer than the processing time. These can only improve throughput when the DMA speed is relatively high.

With only one coprocessor available, we measured the data sending rate using the coprocessor as the sending host. The coprocessor could send packets from its quad-ported memory much faster than a VAX 750 from its memory for the following reasons. First, the coprocessor was a bare machine with only the measurement program running and thus had little software processing overhead. With simplification of packet preparation functions, processing time was less than 0.1 ms for the coprocessor vs. about 1.4 ms for VAX 750. Second, the DMA transfer bandwidth between the quad-ported memory and the CTL board was about 26 Mbps per second. Using the maximum packet size of 2 Kbytes, the data sending rate was about 1000 packets per second or 16 Mbps.

With the coprocessor sending, the VAX 750's receive rate (about 225 of the 1000 2 Kbyte packets sent per second) was consistent with the earlier measurements, the UNIBUS transfer rate and the time required by the VAX 750 to process an incoming packet. When we did not invoke a DMA transfer into VAX 750 memory for packets arriving in the CTL input buffer, but merely handled the packet arrival interrupt and then immediately re-enabled the CTL board to receive, 860 of the 1000 2K byte packets could be received into the CTL input buffers. This means that even if we had an infinitely fast host bus, we would not be able to get higher than a 14 Mbps data transfer rate with the current design of the Proteon interface receive logic.

The VAX 750 took about 1.11 ms to handle each receive interrupt. Hence, when packets were arriving at the interface at an interval of 1 ms and the two CTL

board packet buffers were working alternately, about 140 packets per 1000 were rejected because the receive logic was not enabled in time. Of course, when packets keep arriving for an extended period of time at an interval less than the time to handle an arrival interrupt, any finite number of packet buffers would eventually not be able to store all incoming packets. On an 80 Mbps ring, transmission of a 2 Kbyte packet takes about 0.2 ms, an obvious limit on the packet arrival interval. However, 0.2 ms may not offer sufficient time for a reasonable interrupt handler routine implemented on a workstation/minicomputer with moderate processing speed. With higher transmission bandwidth available in the future, interrupt handling time will certainly pose a serious performance problem.

3.1.3. Many-to-One Data Transfer Results

For the many-to-one traffic pattern, we were interested in studying packet loss problems caused by many senders overrunning the speed of a single receiver. As expected from the earlier measurements, with two senders, a VAX 750 receiver was already driven at about its maximum speed. For example, for 2 Kbyte packets, the receiver received 221 packets per second from two senders and 225 packets from ten senders; for 2 byte packets, 691 packets per second were received from two senders and 722 packets per second were received from ten senders. Effects of different packet sizes on the percentage of received packets were not significant. From 10 senders, 14 percent of sent packets were received for the 2 byte packet size and 12 percent were received for the 2 Kbyte packet size.

Table 3.2 below shows the same data for 10 senders with the National 32032 coprocessor as the receiver.

PACKET SIZE	PACKET PERCENTAGE	PACKETS/ Sec	KBYTES/ Sec
4	99.7	4985	179
2K	73.8	1342	2743

Table 3.2: Packet Receive Percentage v.s. Packet Size (10 senders)

Because the coprocessor has a faster processing speed and a high data rate from the CTL input buffers to the coprocessor's memory, it could receive a much higher percentage of packets from the 10 VAX 750's sending as fast as they could. For the 2 Kbyte packet size, the coprocessor received 6 times as many packets as a VAX 750 receiver. To receive 1342 2 Kbyte packets per second, the 26 Mbps data transfer channel was utilized about 85% of the time. The remaining time when the 26 Mbps data channel was idle (about 0.1 ms per packet) was presumably for packet processing and DMA set up by the coprocessor.

Results for the 4 byte packet size need some careful interpretation. Almost 100 percent of sent packets were received and still the 26 Mbps data transfer channel was utilized for only 5.5% of its capacity. However, there was still a 0.3 percent packet loss even when there was still extra capacity at the receiver. We believe that this packet loss was caused by the receiver logic not being set to receive in time. Theoretically a packet took about 4 microsecond transmission time on the 80 Mbps

ring, but 100 microseconds was needed per packet for processing and DMA set up at the receiver. Due to the difference in the buffer turnaround time and the packet arrival interval, the two CTL input buffers could experience momentary congestion which would cause packets to be lost.

We also recorded the source addresses of received packets in our measurements. We did not find any fixed patterns of packet loss with respect to sender addresses.

We also measured the increase of per packet send delay when there were 10 senders vs. only 1 sender. As Table 3.3 shows, when 10 VAX/750's were sending 2046 byte packets (including 32 header bytes) as fast as they could, they consumed about 37.5% of the 80Mbps medium bandwidth. Per packet send delay for packets of this size was increased by 76 microseconds, only 1.3% of the per packet send delay when only one station was sending. As expected, such increased load on the network medium did not increase packet delay significantly. This also indicates that with ample bandwidth, software processing time is certainly the more significant portion of per packet delay.

PACKET SIZE(No Header)	Microsecond/Packet
2	66
8	66
32	68
128	66
512	75
2014	76

Table 3.3: Increase of Per Packet Send Delay with 10 Senders

3.1.4. Discussion of the Measurement Results

In our measurement, the slow UNIBUS speed is the factor limiting throughput; and it overshadows packet processing overhead. Packet processing and interrupt handling times are only significant to the extent that they cannot be overlapped with DMA transfers. For this reason, a slow host bus also makes the use of large packet sizes less attractive.

With computers having a bus speed as high as or significantly higher than the network medium speed, packet processing and interrupt handling times, as well as packet size and buffer management strategies, become critical in determining system throughput. One could take advantage of a high DMA rate by increasing packet size and overlapping the DMA operation with packet processing. For example, with an 80 Mpbs DMA speed and overlapping, a 16 Kbyte packet would have only about a 2 ms (DMA plus interrupt processing) per packet send delay in our measured system if no other hardware or software components were changed. This would allow the data

sending rate to reach 64 Mbps, 80% of the 80 Mbps network bandwidth.

Further improvement in the data sending/receiving rate would require reduction of interrupt processing overhead by using interface designs that would allow multiple packet send/receive events to be handled in an overlapped fashion or to be handled as one interrupt event. Recall that 0.38 ms VAX/750 send interrupt handling time was included in the measured system. If two sets of send control logic could be used to overlap the interrupt handling of a previous send finish with the transmission of the next packet, the 2.0 ms per 16 Kbyte packet send delay time could be reduced to about 1.6 ms or 1.7 ms. This, together with an 80 Mbps DMA rate, would allow a data sending rate close to the full network transmission bandwidth.

With increased use of bulk data transfer applications requiring high throughput, more frequent and intense packet bursts are likely. In such an environment, use of multiple sets of packet send/receive logic as described above does not have the potential to effectively cope with large packet bursts. Furthermore, current interface device designs do little to reduce packet processing at higher protocol levels. Increasing packet size in "parallel" with increases in bus or network bandwidth will incur serious problems, such as interference with timely transmission of urgent small packets on the medium. These observations lead to our motivations for the packet train model, described in later chapters, to overcome performance problems for bulk data transfer applications.

3.2. Other Measurement Results

3.2.1. Measurement on Protocol Performance

Workers at the Lawrence Livermore Lab measured the performance of a general purpose transport protocol, Delta-t, on a VAX 750 running VMS 2.5[42]. This study was concerned mainly with software processing overhead at various protocol functions and layers. Table 3.4 shows a summary of some specific measurement results.

System Component	Send a Data Packet	Receive a Data Packet
user/system interface	1167	1167
system/transport interface	1133	1102
Delta-t (transport protocol)	565	346
network and lower levels	1326	1578
total	4191	4193

Table 3.4: Execution Time at Different Protocol Layers (μs)

Their measurements led to some important observations.

(1) most transport level services are relatively inexpensive; the transport protocol execution time itself comprises a small proportion of the total time needed to send and receive packets;

(2) context switches and system call software checks are relatively expensive;

(3) since each protocol service was organized in one or more procedures, the procedure call/return overhead is significant.

Note that in the above, the user/system interface and system/transport interface mainly include operations to manipulate packet queues, to allocate and deallocate packet buffers, to wake destination process and to do context switches. The conclusion that operating system related operations account for over half of the packet processing overhead is surprising, and contrary to the usual assumptions about packet processing overhead. This is an important part of our motivation for the packet train model to improve bulk data transfer throughput.

Other people have made similar observations. For example, in an analysis on the number of machine instructions executed for various components of TCP/IP[11], Dave Clark at MIT noted that "the first overhead is the operating system". In a typical operating system, the functions to process an interrupt, allocate a packet buffer, restart the I/O device, wake up a process and reset a timer, may turn out to be very expensive.

In an announcement concerning his recent test on TCP/IP throughput performance between SUN workstations[19], Van Jacobson expressed the following view: "to make the network go faster, it seems we just need to fix the operating system parts we've always needed to fix: I/O service, interrupts, task switching and scheduling."

3.2.2. Measurement on Network Traffic Characteristics

LAN traffic load on the average is often light. A number of measurements[1,15,26] found that traffic load consumes less than 5 percent of network raw bandwidth on the average. Momentary peak traffic with a load beyond 30 percent of network bandwidth does not often last over a few minutes. It has been generally held that computer network traffic is characterized by packet bursts.

Raj Jain and Shawn Routhier performed an interesting analysis on traffic burstiness based on measurements of a 10 Mbps token ring connecting 33 computers, 5 gateways and 3 disk servers at MIT[20]. Major conclusions from this analysis are:

First, the packet arrival pattern does not follow the "Poisson Arrival" that is commonly used in analytical modeling. In a Poisson model, packet inter-arrival time is independent, and the inter packet arrival time, t, is exponentially distributed. That is, the probability density function is $p(t)=\lambda * e^{-\lambda * t}$ Jain and Routhier found that, given an element of the arrival time series, t_i, the covariance between t_i and t_{i-k}, i.e., $E(t_i * t_{i-k}) - E(t_i) * E(t_{i-k})$, is not equal to zero, violating independence. The coefficient of variation of t, i.e., $\frac{\sqrt{E[(t-E[t])^2]}}{E[t]}$, is very high compared with unity, violating exponential distribution.

Furthermore, the analysis found that the average packet burst inter-arrival time is on the order of hundreds of milliseconds whereas packet inter-arrival time within a

burst is a few milliseconds. The latter is subject to the speed at which a node can send packets. There is a high likehood that a packet will have the same source and destination address as the previous one within a burst. In other words, there is a strong address locality and predictability within packet bursts. Based on such findings, they called each packet burst a "packet train".

Others have found similar packet bursts. For example, from a measurement of a medium-size 10 Mbps Ethernet connecting file servers to diskless workstations[17], it was found that the Poisson model is inappropriate for packet arrival model. The arrival process is highly bursty and the independence assumption is almost certainly not justified. There is a high probability that one arrival will be followed by a second one within a deterministic time, which depends on the protocol, the packet sizes, and the traffic intensity.

As will become clear in the next chapter, this notion of packet burst is a key element in our work.

Chapter 4

Packet Train Model Motivation and Overview

A packet train is a closely spaced sequence of packets, like a succession of cars in a train on a railroad, that carry all or part of one large block of user data from one node to another over a network. The packet train mechanism provides a means by which bulk data can be transferred from one user process to another over the network at a granularity level larger than individual packets, but without the problems incurred by the inflexibility of large packets. This mechanism offloads certain transport layer functions to the network interface front end without overloading it, and enables reduction of expensive processing overhead both at the interface device and at the host operating system. We first discuss the motivation for this model, and then provide an overview of the model, followed by an outline of the design objectives, and components of, the packet train mechanisms.

4.1. Motivation

Three largely independent issues motivate the packet train model.

(1) Reducing software overhead above the datalink layer by minimizing the number of host operations.

(2) Approaching raw network bandwidth at the datalink layer by minimizing the number of packet event interrupts at the network interface device level.

(3) Speeding up protocol processing by shifting an appropriate set of protocol functions from the host system to a a dedicated processor on the interface device.

4.1.1. Reducing Host System Operation Invocations

Along the transfer path a packet traverses, software processing overhead forms the dominant portion of the total transfer delay. Consider the following simple formula for single packet delay:

packet_delay = sender_processing +
sender_data_copy +
network_transmission +
receiver_data_copy +
receiver_processing

In the case of the 80 Mbps Proteon Ring and the VAX/750, network transmission time of a maximum size packet (two kilobytes) is 200 microseconds, whereas the total software processing time is in the range of several milliseconds (see Chapter 3), at least an order of magnitude higher than the transmission time. On faster machines, per packet software processing time is still close to one millisecond[19]. The bottleneck effect of software packet processing will become more prominent when network bandwidth is further increased.

The simple formula above did not take into account different degrees of concurrency (overlapping) among bus DMA, network transmission, and software processing operations, which occurs when a sequence of packets are transferred

between two nodes. With software processing time as a dominant factor, overlapping these operations would reduce the total delay only slightly. Software overhead will remain a bottleneck until packet size is increased to the point that per packet transmission and data move times are about as long as the per packet software processing time.

Further consider the total software processing overhead, as illustrated by the formula below.

$$\text{processing_time} = \text{number_of_packets} * \text{interrupt_handling} + N * (\text{protocol_processing} + \text{OS_related_operations})$$

where N is usually equal to number_of_packets.

Measurement results described in Chapter 3 pointed out that among the various components of software processing, the host OS related overhead is the dominant factor. As a packet is processed at different protocol layers, various system related operations are invoked. These include manipulation of packet queues, allocation and deallocation of packet buffers, and context switching to wake up the process that implements a certain protocol function layer. These host system related operations are expensive as compared to other protocol processing operations such as connection state record lookup and update (see Chapter 3). Therefore, reducing the value of N, i.e., reducing the number of invocations of these host system related operations, will achieve substantial data transfer performance improvements.

4.1.2. Reducing Packet Interrupt Overhead

One solution to the performance problem of expensive host system operations is to delay passing packets to a higher layer from the data link layer until a certain number of packets are received. This solution achieves the effect of large packets above the data link layer, but has a number of problems.

Such a solution can only be used at the receiver host. A sender host must still prepare headers one packet at a time. Also, a data link layer receiver does not know the number of packets that a transport layer process expects from its sender peer. Using a timer to prevent waiting too long for a fixed number of packets introduces an unnecessary delay for applications other than bulk data transfers.

More importantly, packet arrival interrupts still need to be generated on a per packet basis. On a typical workstation, interrupt handling time is greater than packet reception time, and this disparity is likely to increase with rapidly increasing network bandwidth. This implies that, for highly bursty traffic in which back to back packets can arrive over an extended period of time, an interface device with a finite amount of packet buffer space will start to lose packets, severely degrading performance. Because throughput at the interface device level puts an upper bound on user-level throughput, the bottleneck effect caused by the traditional one interrupt per packet scheme must be overcome by designing a more appropriate interrupt mechanism.

4.1.3. Problems with Large Packets

A simple and naive way to reduce the number of invocations of host system related operations and the number of interrupts to the network front-end is to use large packets. However, increasing packet size will introduce the following problems.

(1) Larger packets increase the delay in delivering initial data, which is undesirable for a number of applications that can start data processing as soon as the first byte or first record is available. A distributed database is one such example.

(2) Buffer management become less efficient when a wider range of different buffer sizes must be satisfied.

(3) Large packets reduce the benefits of concurrency; that is, the overlapping effect among DMA transfer, network transmission and protocol processing is reduced when each of these operations is performed for a larger data size.

(4) Last, but not least, larger packets occupy the network medium for longer periods of time, hindering the timely transmission of higher priority small packets.

For these reasons, a number of systems, such as AT&T's Universal Transport protocol on Datakit[16], actually use small packets for all data transfer applications.

Our goal is to achieve, for bulk data transfer applications, user-to-user level throughput close to what network raw bandwidth can accommodate, and at the same time, guarantee small delay for urgent data transfers. We must minimize software processing overhead in order to achieve this performance goal. In particular, we must minimize the number of times that expensive host system related operations are invoked, within the current limits of processor speed and host operating system structure. Our motivation is clear: we wish to find a solution that achieves the effects of using large packets for software processing in both the host and the FEP, but avoids the problems incurred with actually using large packets.

4.1.4. Proper Offloading of Protocol Functionality

Designs for network interface devices have been evolving from a model of a primitive I/O port to a model of a sophisticated channel device with its own processing capabilities. Functional requirements have been getting more complex together with interface designs. With hardware costs falling rapidly, it has become desirable to provide a network interface device with an on-board general purpose processor, called a "front end processor" (FEP) and to dedicate this FEP processor to packet processing at a certain protocol function level. The advantages of protocol offloading include guaranteed processing power for protocols and the saving of higher cost host CPU cycles. It is generally understood that moving protocol functions from a host CPU to such FEP processors has potential to to improve data communication performance as well as overall host system performance.

A number of network research projects have used protocol offloading. There are two common strategies: (1) casting some or all protocol functions in VLSI chips that are part of the network interface hardware, and (2) offloading a whole protocol function layer, such as the transport layer, to the front end. Protocol offloading is one essential element in our packet train model. However, our protocol offloading strategy is different from the above two.

Implementing a complete protocol functionality in special hardware chips, the approach exemplified by the design of the Protocol Engine[9] is extremely inflexible. The Protocol Engine, for example, can support only XTP. In addition, this approach does not address the key performance problem of interfacing with the host operating system. Because host system operations can account for more than half of the total processing overhead, simply running simplified protocol functions such as XTP in some special hardware chips is not an adequate solution. We concur with earlier criticism of this approach[11].

Moving a complete protocol layer, like the transport layer, to a FEP appears to offer a big performance win due to the large amount of work being offloaded. A careful examination, however, leaves us with doubts if this would always be true. The transport layer involves elaborate flow control and error control strategies, maintains lists of timers and buffers, and frequently looks up and updates the transport state for a potentially large number of active data transfer processes. These require much more complex operations than data link layer functions, and often need

a fair amount of operating system support. If a "mini-operating system" is built at the front end, there is a potential of overloading even a fairly powerful FEP processor. In this case, the problem of speeding up protocol processing may be merely shifted from the host CPU to the FEP processor.

Furthermore, one may have to provide interaction mechanisms between such a mini operating system at the front end and the host operating system, which will undoubtedly be much more complicated than a simple packet send and receive interface. Simultaneous accesses from the front end and the host CPU to certain shared host system objects, for example, the process contexts, may pose contention problems, and synchronization of such accesses may limit performance improvements achieved through protocol offloading.

Our method differs from the above two in that offloading is not simply in terms of protocol function layers. Rather, it is along some vertical division of protocol functionality. Figure 4.1 depicts our protocol offloading strategy.
Instead of offloading "horizontally" a complete transport layer to a FEP, we select certain functions of the transport layer to be offloaded, leaving the rest still the responsibility of the host CPU. We select those functions that can be executed by the FEP processor without frequent access to host system resources. Such functions include those that check on packet validity, order packet data by their sequence numbers, prepare the transport level packet header, move packet data to user address space, and update protocol state record. These functions need little support from the

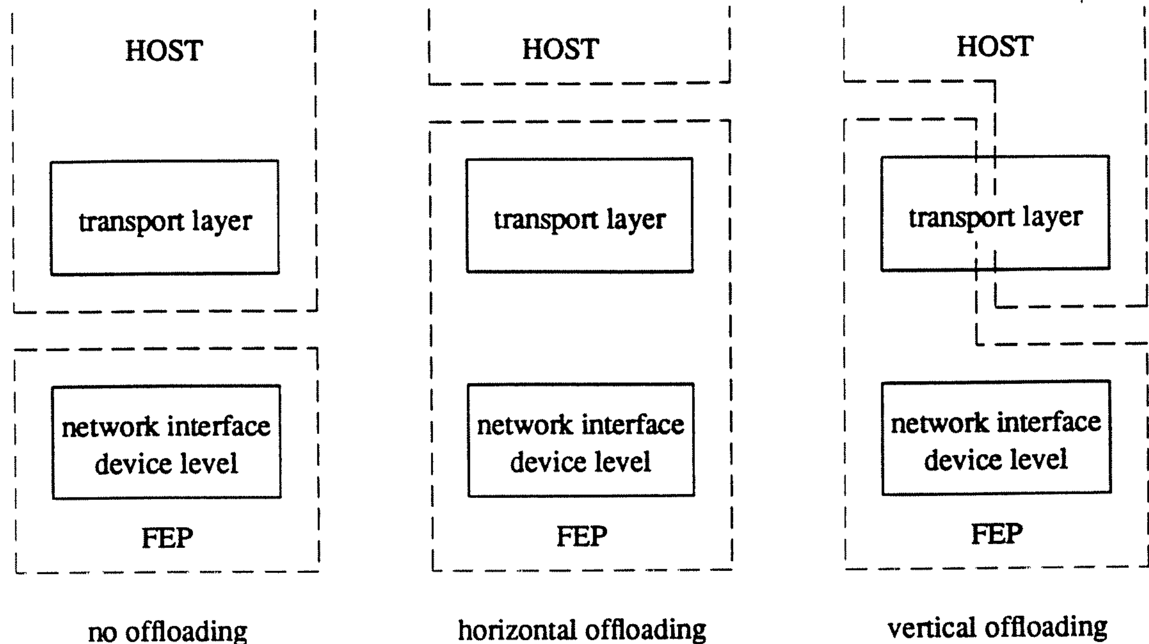


Figure 4.1

host system, and are repetitively executed for a sequence of packets associated with the transfer of a large block of data. Functions that depend on host operating system support or frequently interact with the host operating system are less attractive for offloading. Examples of these are control parameters calculations such as flow control and window size adjustment, KEEPALIVE packet transmission, etc. Furthermore, offloading infrequently used functions, such as abnormal condition handling, offers fewer performance benefits.

However, there is not always a clear-cut division between those functions that should be offloaded and those that should not. The packet train model, described later in this section, provides a natural way of selecting a set of functions for

offloading. By offloading only appropriate functions, the packet train model can avoid the pitfall of overloading the FEP processor, while still achieving the performance benefits of protocol offloading.

4.2. The Packet Train Model

A packet train is a special form of large data delivery over a network. On the network transmission medium, a packet train is a sequence of consecutive packets, spaced within a maximum allowed inter-packet time gap and loaded with data for the same transport layer entity. As such, packets in a train have the same control header and data size (except the last one). The size of a train is dynamically determined at the time of medium transmission. It is bounded by the amount of packetized data ready for transmission. Once transmission starts, the size is further subject to the time a sender node can monopolize the medium access for packet transmission and to interruptions by transmission of more urgent packets. At the sending and receiving hosts above the medium transmission layer, a packet train is handled as one transfer unit for its protocol processing.

The notion of packet bursts is the key element in the packet train model. We achieve the effect of using large packets by deliberately sending intensive bursts, or "trains", of normal size packets on the medium, while the FEP and host are interrupted, and software processing is performed, just once per burst.

As part of the packet train model, a simple data send and receive service is provided to the host transport level, where the data size can be arbitrarily large (the actual data size per send/receive request is subject to some practical limitations such as interface device buffer size, but not to the underlying network maximum packet size limit). Figure 4.2 illustrates the boundary between the host and the network interface. Because the front end can use DMA to load and unload data to/from a whole train of packets with one invocation of transport layer functions, host processing is reduced. The fixed per packet processing overhead is amortized over a train of packets instead of being charged for each individual packet.

At the network medium level, a packet train appears as a burst of adjacent packets carrying all or part of one logic block of user data. Packets in a train can be thought of as cars in a train having chains linking them together. In reality, however, there are no such "chains". The cars are implicitly linked from packet frame format, header control information, and the timing of their arrival. Such linking also requires that there is a maximum separation between packets in a train. Packets in a train are of the same size except that the last one may be shorter. They must be sent from only one sender and destined for only one receiver. For example, there are two trains in Figure 4.3a, both from the node A to node B. There is no chain between the third and the fourth packet because the time gap between the two is "too long". In Figure 4.3b, some packets that arrive closely in time on the network are not part of one train because they are from different senders or to different

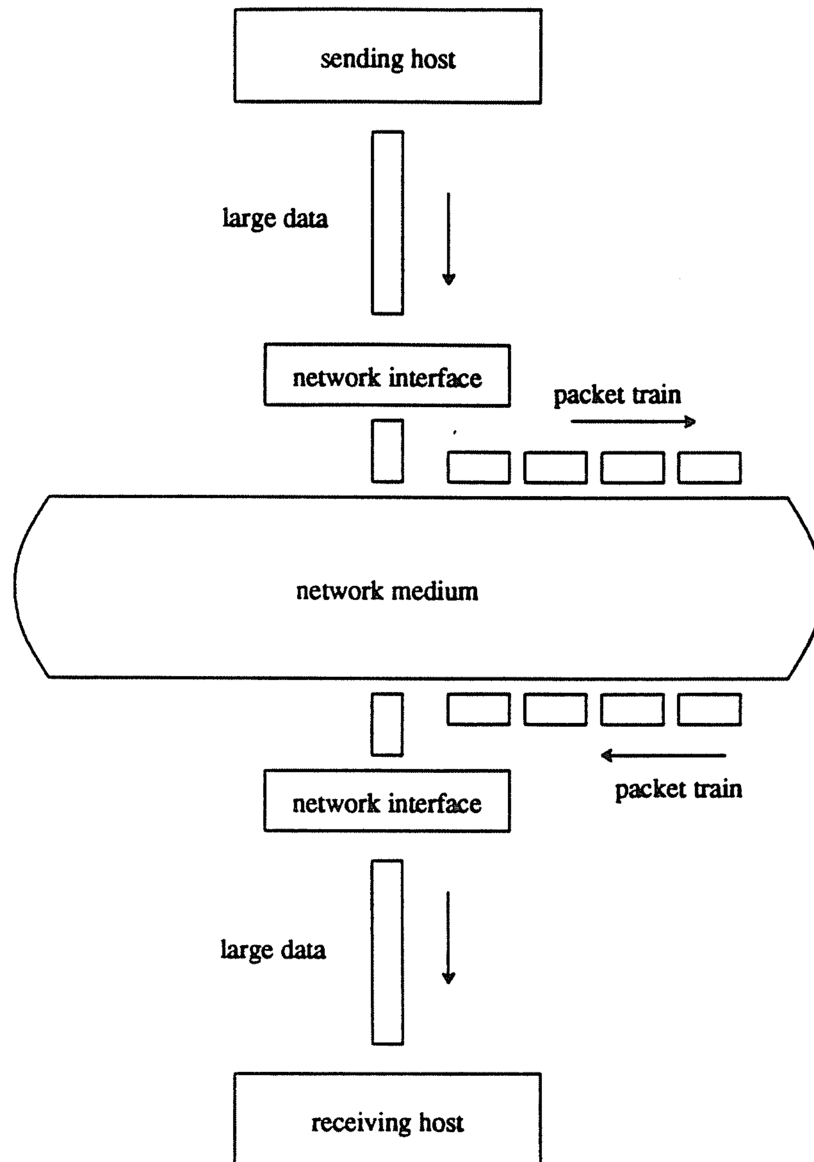


Figure 4.2

receivers. Trains are dynamic objects on the network and their sizes are subject not only to the amount of data to be sent, but also to the intervention of other traffic.

As illustrated in Figure 4.3c, both the sender and receiver hosts attempt to treat a large block of data as one logical unit at the transport and network interface levels. Only when transmitted on the network medium, will the data be split into a sequence of packets. Hosts charge a fixed "handling fee" (processing overhead) for the whole train at a time, rather than for each individual packet.

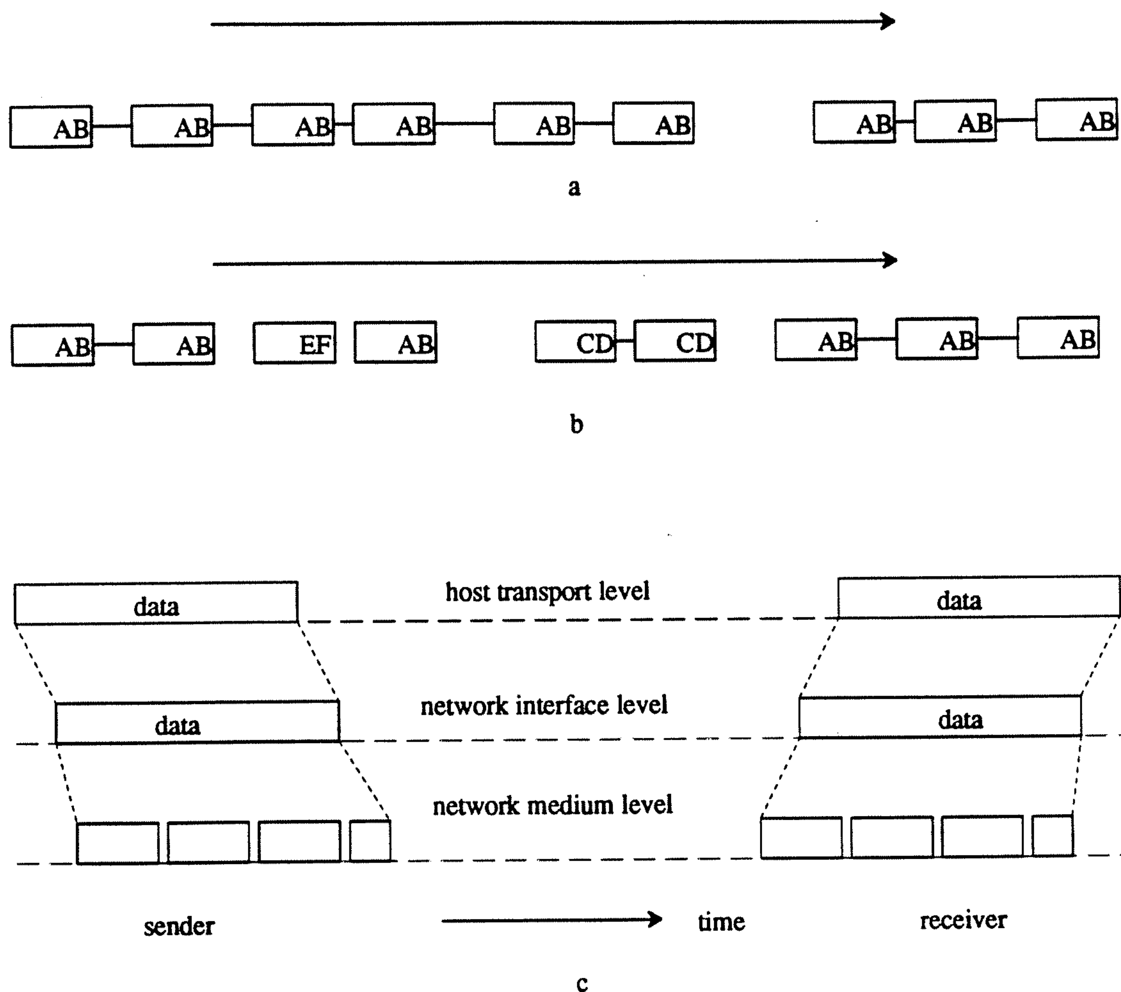


Figure 4.3

Although the term packet train has been used in the literature[20] to describe observed packet arrival patterns on a network (see Chapter 3), the following differences distinguish our packet train model from other notions of packet trains.

(1) The most important distinction is that our packet trains are not simply a model of network traffic. Generation of intensified bursts of logically inter-related packets as packet trains is one essence of our packet train model.

(2) In our model, packets in a train must belong to one transport layer entity. In other work[20], a burst of packets from two file transfers between the same sender and receiver nodes could be taken as one train. Our model does not allow such a view of packet trains, and would discourage multiplexing more than one logical data transfer activity onto one transport connection so that performance of a data transfer application can benefit from this model.

(3) Timing in packet arrival is critical in our model. Packets that have the same header control information but have a time gap between packets larger than the system defined maximum time gap, will not be considered as one train. In Raj Jain's[20] packet trains, there is no explicit maximum time gap requirement.

(4) Our packet train model includes components at the network medium, interface driver and transport layers, whereas in Raj Jain's notion, a packet train is only an entity existing at the network medium level.

4.3. Packet Train Mechanism Design Assumptions and Objectives

4.3.1. Traffic Characteristics

The goal of the packet train model is to achieve high speed bulk data transfer, while supporting the timely transfer of priority data. It is critical that this model perform well for current and future high bandwidth networks. Based on observations of current network traffic characteristics and features of newly designed data transfer protocols, we argue that the packet train is likely to be successful for future, as well as current networks.

Currently, average traffic load on LAN's is low. The average time gap (at least a few milliseconds to tens of milliseconds) between packets tends to be sufficiently large for a sequence of train packets to slip through with little chance to disturb other traffic. This means that transmission of packet trains large enough to benefit performance would not significantly affect the transmission delay of other traffic. Take an 80 Mbps token ring for example. Under average traffic loads as low as 3 to 5 percent of the network bandwidth (see Chapter 3), trains consisting of tens of packets can be readily accommodated, with substantial performance gains expected. On future networks with gigabits per second bandwidth, megabytes of data can be transmitted in just a few milliseconds. As bandwidth increases, it is expected that traffic load will also increase. However, traffic load on the average may still remain a small fraction of the available bandwidth[29]. As a result, sizes of packet trains that can be accommodated are likely to increase.

The bursty nature of network traffic further favors large train sizes, because clusters of packets tend to leave larger time gaps between packet bursts than if packets arrived more or less at some uniform interval. Future network traffic will likely be more bursty, because faster processors on higher bandwidth networks will be able to send packets at a higher rate, and communication applications will be more greedy in data transfer sizes.

New protocols are also designed with features that include sending large numbers of packets at a time. For example, the bulk data transfer protocol NETBLT[13] can send up to 50 packets at a time for a large block of data. VMTP[6] has a feature that allows a group of 8 packets to be sent at one time. It is not difficult to see that intentionally generated packet trains will fit with the traffic patterns resulting from the demands of future applications and protocols.

4.3.2. Network Characteristics

The packet train model will achieve its objectives only if per packet transmission time is substantially less than the per packet processing time. This establishes a minimum sufficient network bandwidth. For example, on a 10 Mbps Ethernet, a maximum size packet (1546 bytes) takes over 1.2 millisecond to transmit. On a powerful workstation, per packet software processing overhead is about 1 millisecond, making packet transmission a less critical concern for performance. Since the CSMA/CD medium access method is not suitable at higher bandwidths than 10 - 20 Mbps (because of collision detection requirements), we do

not consider an Ethernet as a suitable network environment for the packet train model.

Networks based on the token ring access method do not suffer from similar problems, because the circulating token provides the basis for network access mediation, eliminating the problem of packet collisions. Furthermore, this access method can guarantee an upper bound of medium access delay to all connected nodes, and the token priority mechanism provides prioritized accesses to the network medium. Because of these properties, token regulated network access methods are generally considered appropriate for high bandwidth. For instance, a version of the token ring access method, FDDI, is designed for 100 Mbps medium bandwidth. To exploit these properties, we have designed our packet train mechanism for token ring network environments, as specified by IEEE 802.5 and FDDI standards. For the rest of the thesis, a token ring network environment is assumed for the packet train model.

4.3.3. Design Objectives for the Packet Train Mechanism

We would like to achieve the following design objectives for the packet train mechanism.

(1) There should be no fundamental changes to the Token Ring medium access control method. This guarantees that a network interface device that does not support the packet train mechanism can communicate with an interface

device that does support the mechanism, at a performance level no worse than that achievable between two interface devices that do not support the packet train mechanism. Furthermore, extra hardware features needed for the packet train mechanism should be kept at a minimum.

(2) A sender network interface device must be able to send a large block of data as a sequence of train packets. Packet preparation and encapsulation must be within the real time of packet transmission so that train packets can be transmitted as back to back packets onto the network.

(3) A receiver network interface device must be able to recognize a packet train, perform data link layer functions and any offloaded protocol functions for train packets as they arrive. In particular, upon reception of the last packet in a train, the network interface device must be able to pass the whole train (not in packet frames) to the host for further processing in such a way that the host system can treat the whole train as if it has received one large packet.

(4) Urgent small packets should not be penalized by packet train transmission. Whenever a packet train is being transmitted on the medium, an urgent packet ready for transmission should be able to interrupt the ongoing train and obtain the network access right with minimal delay. Furthermore, normal token ring interface devices should require no modifications to interrupt an ongoing packet train.

(5) There should be no special mechanism for the resumption of transmission of an interrupted packet train. Such resumption should introduce minimal extra overhead.

(6) There should be no rigid packet train size requirement. A packet train can be as small as one packet and as large as the number of packets allowed by the per station token holding time. This requirement will enable a sender to start sending train packets as soon as possible, keeping the latency in delivering the initial data as low as possible.

4.3.4. Major Components of the Packet Train Mechanism

There are three major components, hardware and software, in our packet train mechanism.

First, a network interface device design, called the "network controller", that can support transmission and reception of packet trains. The structure of network controller must also support efficient execution of offloaded protocol functions. Such a network controller is the key component of our packet train mechanism. The prominent features of the network controller include a fast large packet buffer memory, a RAM for protocol execution and shared memory communication with the host, and separate data channels for packet data and for the on-board processor. Chapter 5 describes the details of our design of such a network controller.

Second, extensions to the token ring access rules that accommodate packet trains while still providing fair access for all network stations. In particular, we have proposed extensions to the IEEE802.5 and FDDI MAC protocol standards, and extensions to the functionality of the component of the network controller that interfaces with the network medium. Chapter 6 explains the details of these extensions. Appedix A and B give the train packet format and the finite-state machine description of the MAC protocol extensions.

Finally, the data structures, used to communicate between the network controller and the host operating system. In addition to this, the data structures also provide necessary control information for the execution of offloaded protocol functions at the network FEP level. Management of these data structures is the responsibility of the network controller's processor, but they are accessed by both the host processor and the network controller's processor. Chapter 7 describes these data structures and how they are used and maintained in the RAM of the proposed network controller.

Chapter 5

Network Controller Design Features

In this chapter, we describe our design for a network interface device which we call a "network controller". The design objectives for the network controller are to support transmission and reception of packet trains, as well as to provide efficient execution of protocol functions offloaded to the network front end processor. We first describe the network controller's overall architecture, and then its major components.

5.1. Network Controller Overall Architecture

As shown in Figure 5.1 below, the major components of the network controller include the following.

(1) The **MAC**, the medium access control component implements the token ring access method. This component is extended with logic functions for the transmission and reception of packet trains, implemented by microcode instructions stored in the **Microcode Store** component. In addition, the MAC generates interrupts to the on-board processor upon detection of events such as arrival of train or non-train packets. **MAC** has two ports to the **Buffer Memory** to provide for the full duplex functionality required by the token ring access method.

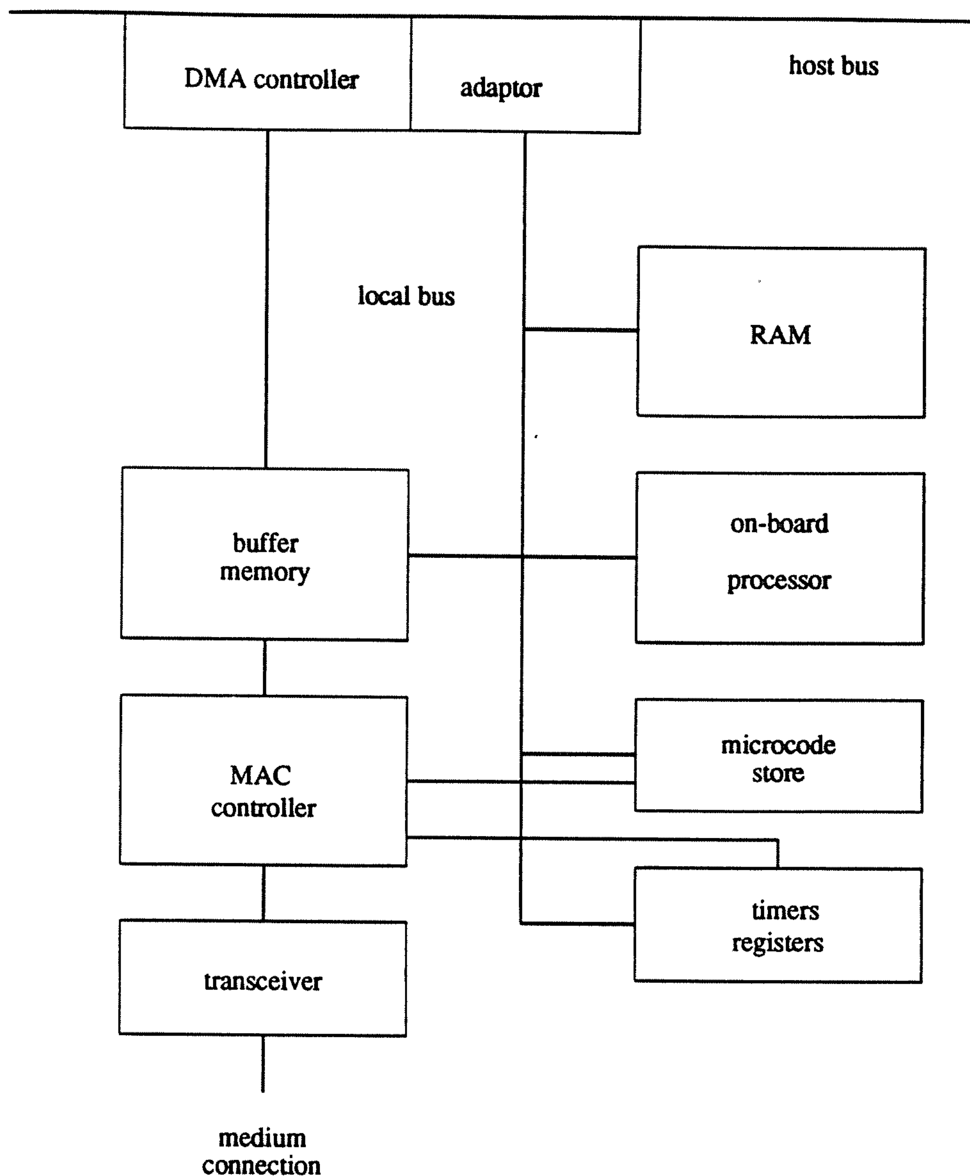


Figure 5.1: Network Controller

(2) The **Microcode Store** and **Timers** and **Registers**. The **Microcode Store** stores the microcode instructions that control the operation of the **MAC**. It implements the token ring access method as a state machine, that we will

describe in the next chapter. The **Timers** and **Registers** are used by the microcode program to recognize packet trains by monitoring the inter packet gap between successive packets, and recording and comparing header control information of successive packets.

(3) **Buffer Memory** component. This is a multi-port high speed random access memory. Its DMA port with the host bus has gather/scatter capability and interrupts the on-board processor when a DMA operation is finished. The buffer memory bandwidth is high enough to allow the on-board processor random access to the buffer memory during packet transmission or reception and while packet DMA transfer to/from host memory are in progress at the same time. This buffer memory only stores packets in transit between the host and the ring medium. It is controlled and managed by the on-board processor via the local bus.

(4) A **on-board general purpose processor**. This processor executes offloaded protocol code, manages the packet buffer memory, services packet and DMA event interrupts, and sets and updates the various timers and registers referred to above. The **adapter** between the host bus and the local controller bus allows the host CPU and the on-board processor to interrupt each other for communication.

(5) The **RAM** component is a random access memory that stores the offloaded protocol code, control data structures for packet processing and data

structures for communication between the host and the network controller. Since the on-board processor accesses this RAM more frequently than the host CPU does, the RAM is directly connected to the local bus and viewed by the processor as its private memory. It is also mapped into the host memory space via the adapter and thus appears to the host as one particular area of the host memory. This facilitates communication via shared memory between the host and the network controller.

As shown in the figure, the major components are connected to the network controller local bus. The bus is not meant for packet data transfers between the network controller and the host. Large packet data is transferred via a DMA interface between the buffer memory and the host bus. This avoids bus contention during block data transfers, and guarantees the processor access to other network controller components for packet processing. Communication between the host and the network controller other than packet transfers also takes place via this local bus.

Although the DMA interface is normally used to transfer packet data, the on-board processor can use the local bus to copy small packets between the buffer memory and the host memory. Copying small packets, for instance, 64 byte packets, can be faster than setting up the DMA registers, performing the DMA transfer and servicing a DMA completion interrupt. The on-board processor copies packets based on its load, and the expected performance benefits. The copy operation consumes a certain amount of processor cycles, but the performance benefits of a

faster data transfer is well worth the extra load on the processor. Because we limit the processor executed protocol functions to a carefully selected set of transport functions using the packet train protocol offloading strategy (see Chapter 7), the processor is not likely to be overloaded by performing such copy operations for small packets. Furthermore, the processor can still choose to use DMA transfers for small packets, when a heavy processing workload warrants it.

At system start-up time, offloaded protocol code is down loaded into the RAM from the host. The other parameters, such as timers and group addresses for the network controller, can also be down loaded at this start-up time.

The MAC component, the **on-board processor**, and the **Buffer Memory** are the most important components for our design objectives. We describe each of these components in further detail in the following subsections.

5.2. Medium Access Controller

The MAC component implements packet and packet train transmission and reception according to the token ring medium control protocol. must also follow the rules of the token ring medium access control protocol. In operation, the MAC is governed by the program in the Microcode store, driven by transceiver and timer events. The control program uses the following timers and registers.

(1) *SA* Register. This non-programmable register contains the header SA field of the last received packet, train or non-train. Before it is overwritten with

the new SA value, it is compared against the new value. The result is one of the four factors determining if the new packet is part of a packet train being received.

(2) *Maximum Packet Gap* register. Its value is a system parameter defining the maximum time gap allowed between two consecutive packets in a train. Its value is set by the on-board processor.

(3) *Train Recv* register is a boolean flag indicating the MAC is in packet train receiving mode. This register is set when the MAC component explicitly enters the packet train reception mode. It is cleared when one of the following conditions becomes true: (1) the *Last Packet Timer* expires, (2) a free token is seen (repeated or received), (3) a non train packet is received, or (4) a packet (train or non train) destined for this station is repeated (rewritten on the medium).

(4) *Last Packet Timer*. This timer keeps track of the elapsed time since the last octet of the previous packet arrived. The last octet of a packet is the Frame Status (FS) field of a packet frame for both the IEEE 802.5 and FDDI formats. The *Last Packet Timer* is initialized to the value of the *Maximum Packet Gap* and starts to count down when the MAC is in *Train Recv* mode, and when a packet FS field is seen during *Train Recv* mode. When the timer expires, MAC clears the *Train Recv* flag to indicate the end of packet train receiving mode. The timer is disabled when a packet SD (Starting

Delimiter) field is seen and when MAC is not in *Train Recv* mode.

(5) *Packet Descriptor List Pointers.* These registers point to the corresponding positions of packet descriptor lists in the **Buffer Memory**. There are two sets of *Current, Head, and Tail* registers, one for transmission and one for reception. The on-board processor manages allocation and deallocation of packet descriptors. The MAC component accesses the lists during packet transmission and reception. Details regarding their use will be given in the subsequent subsections.

5.3. Buffer Memory and Its Organization

The buffer memory is a large random access memory operating at the speed of the network medium bandwidth. The buffer memory has one serial port to the host DMA interfaces, one serial port to the MAC, and one random access port to the network controller local bus.

The DMA interface is programmable only to the on-board processor and it generates interrupts only to the on-board processor. The host CPU does not initiate DMA data transfers to/from the buffer memory. Rather, it passes to the on-board processor via RAM and interrupt information about the size and location of data or empty data buffers when it requests a data send or receive operation. The on-board processor uses this information to set up DMA transfer between the host memory and the buffer memory. The DMA interface has scatter/gather capability by which

data from a train can be collapsed into a consecutive block of host memory or scattered into separate *packet buffers* (to be discussed below) in the buffer memory from a consecutive block of host memory while performing DMA transfers. When transfer of a block of data is completed, an local bus interrupt is generated to inform the on-board processor, which can then inform the host via its interrupt to the host.

The DMA interface design includes a new feature that allows an "intermediate" interrupt to be generated each time a fixed number of bytes have been transferred regardless of whether the total transfer is completed. A counter register, part of the DMA interface, can be optionally set by the processor for generation of such interrupts. An intermediate interrupt notifies the processor data ready for packet encapsulation and lets packet transmission start without a long delay for the transfer of a complete large data block. Typically, the count is set to the size of one maximum length packet. Such intermediate interrupts have the lowest priority, and are ignored when the on-board processor is handling an interrupt for a more important event, such as packet arrival.

The size of the buffer memory should be balanced between its cost and ability to accommodate packet bursts. It should be chosen based on network transmission speed, host bus speed, host services and client population, statistical traffic peaks at the node, flow control strategy and buffer management schemes. Obviously, the optimal buffer size is different for different machines.

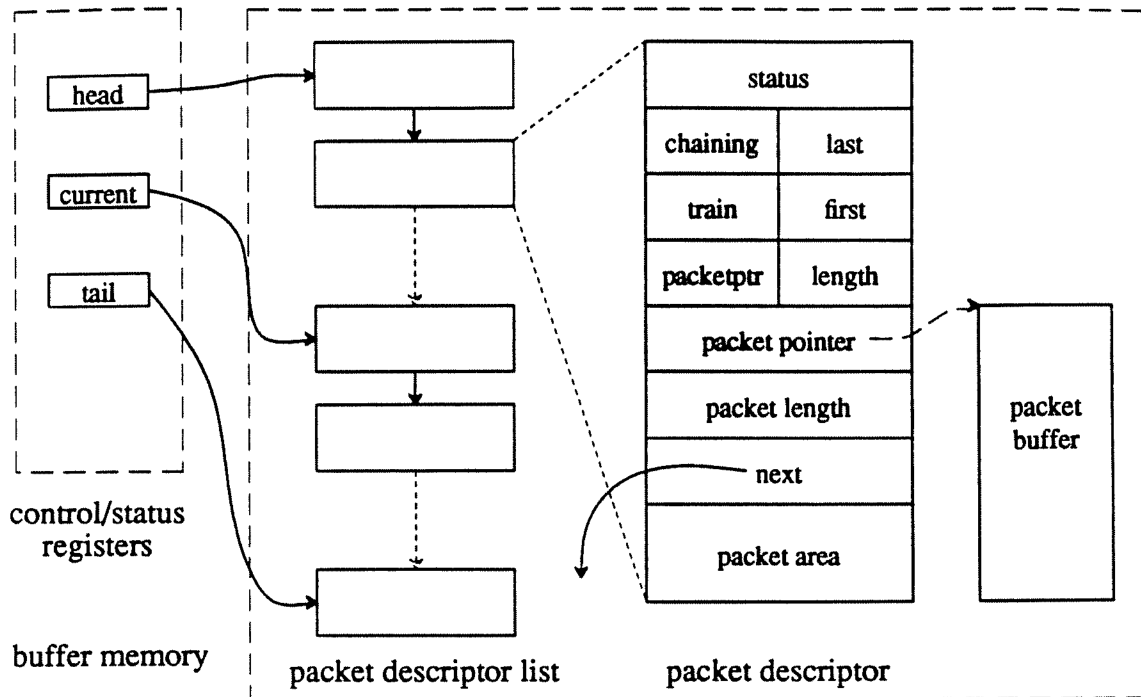


Figure 5.2: Packet Descriptors and Descriptor List

The buffer memory is organized into a set of packet descriptors and a pool of packet buffers, as shown in Figure 5.2. Packet descriptor data structure are constructed and managed by protocol programs running on the processor and accessed by the MAC during packet transmission or reception. There are two linked lists of packet descriptors, one for incoming data and one for outgoing data. There are three registers pointing to the head, tail and current positions of each list. The *head* and *tail* registers are used by the processor for packet processing, and allocation and deallocation of packet descriptor and buffers. The *current* register is used by the MAC to access a packet descriptor and packet associated for packet

transmission/reception.

Each packet descriptor is associated with at most one packet. Non-train small packets are stored inside the descriptor in a small data area (128 byte). If this area is too small, several descriptor are changed together. Each train packet is stored in a separate packet buffer, pointed to by a packet descriptor, whose size is the maximum allowed by the network medium access control protocol. Descriptor flags indicate where the packet data is stored, and if chaining is used.

The separate storage areas are based on a number of measurement[1,7,20] showing a strong bimodal pattern in which most packets are either short (less than 100 bytes) or long (maximum allowed packet size). Applications such as remote login and inter-process communication tend to generate short packets, while applications such as file transfer tend to generate maximum size packets. Packets whose sizes fall between these two extremes account for a very small percentage and these can be accommodated by the descriptor chaining mechanism. This design gives a reasonable balance between simplicity of buffer memory organization and efficiency of buffer space utilization.

The *status* field is used for synchronization between the on-board processor and the MAC, and to record transceiver status bits. The *MAC_Owner* bit in the *status* indicates if this packet descriptor can be used by the MAC component for packet transmission or reception. Other fields indicate normal or error cases for a packet transmission or reception operation. As each packet is transmitted or received, the

MAC copies status information from the transceiver logic (its status registers) into the *status* field. This lets a list of packet descriptors acts as a large set of status registers. This way, interrupt handling time becomes a less critical performance problem without having a large set of hardware MAC control/status registers.

5.4. Coordinating Accesses to Packet Descriptors and their List Registers

Since simultaneous updates and accesses to the packet descriptors by the MAC and the processor may compromise the integrity of descriptor field values, the following rules must be observed (by hardware and software) to prevent arbitrary orders of updates and accesses to the packet descriptors.

(1) Immediately after all packet descriptor field values are properly set and the descriptor is linked to the appropriate packet descriptor list, the on-board processor sets the *MAC_Owner* bit in the descriptor's status field.

(2) Although the processor can physically write the fields of a packet descriptor whose *MAC_Owner* bit is set (no hardware protection here), the processor should only do so in cases of error handling, in order to avoid possible data synchronization problems. Read only accesses to the fields are not harmful and allowed any time.

(3) The MAC is able to write only fields of a packet descriptor whose *MAC_Owner* bit is set (enforced by hardware protection). Read only accesses are allowed at any time.

(4) The MAC clears the *MAC_Owner* bit for a packet descriptor immediately after using it for packet transmission or reception.

The following rules summarize the timing requirements for the updates and accesses to the packet descriptor pointer registers in order to preserve the integrity and ensure the proper use of the packet descriptors.

(1) The *current* register is only updated by the MAC component. However, the processor is allowed read only access any time.

(2) The *head* and *tail* registers are only updated by the processor. However, the MAC component is allowed read only access any time.

5.5. Packet Train Interrupts

The MAC interrupts the on-board processor for packet transmission and reception, as well as some error conditions we will not discuss. Non-train packets are handled the same way as conventional token ring or FDDI packets. For packet train transmission and reception, additional interrupt handling is required.

A packet train completion interrupt is generated when transmission of the last packet in a packet train has finished and the packet has circulated back to the sending MAC. Only one interrupt is generated for a sequence of packets in a packet train, unlike conventional transmission, with 1 interrupt per packet. An additional pointer register called *Train Start* is used by interrupt handlers for train completion. This register is set to the packet descriptor for the first train packet, and, unlike the

current register, is not changed during transmission. When train transmission is completed, the interrupt handler uses this register to examine status of all transmitted packets. If any train packet generated an error, all subsequent packets must be treated as one logical train unit for retransmission. To preserve the original order of the train packets at the receiver, either all, or none, of the train packets following an error should be retransmitted or removed from the transmission list.

Packet train reception also generates a single interrupt, but this interrupt occurs at when the header of the first train packet is received. This interrupt placement is needed to reduce the latency at the receiver for a large packet train. If the interrupt is generated at the end of a whole train, processing of train packets will be delayed significantly.

Once the train reception interrupt handler is entered, the on-board processor stays within the handler to process subsequently arriving train packets, using polling. DMA transfer of packet data can be initiated on a per train packet basis. Transfer of the received train data to the host can be concurrent with reception of the rest of a packet train.

No further interrupt will be generated to the on-board processor when the end of a packet train is reached, i.e., the last train packet is received. The on-board processor exits the handler routine when it has processed the last received train packet and when the *Train Recv* register indicates that the end of a packet train has been reached. The processor uses the *current* packet descriptor register to determine

if the last received packet has been processed.

5.6. On-board processor

The on-board processor is responsible for the following functions.

(1) Servicing all interrupts (packet transmission and reception, DMA data transfer, interrupt from the host CPU).

(2) Management of the buffer memory. Allocation and deallocation of packet descriptors and packet buffers. Management of the packet descriptor lists.

(3) Data link layer functions such as packet encapsulation.

(4) Processing packet trains for transmission and reception, in particular, setting up train data DMA transfers, and locating and updating control data structures for packet trains.

(5) Management of Send/Receive Request/Status Control Records, the data structure in the RAM for the communication between the host and the on-board processor.

(6) Communication with the host CPU by hardware interrupts and by shared memory.

Because of the diversity and complexity of the above tasks, we choose to use a general purpose programmable processor as the on-board processor. Some new designs for network interface devices use special purpose hardware to speed up certain aspects of packet processing[9,22], such as process ID lookup and sequence

number check. Although this could improve performance to a certain extent, it fails to address the true performance problems, namely, the expensive host system operations invoked for protocol execution[11]. When higher layer protocol functions are offloaded to the network interface front end, use of special purpose hardware would not offer adequate flexibility and processing speedup to improve overall performance. Furthermore, using a general purpose instead of special purpose hardware can allow one to benefit from advances in microprocessor technology.

The on-board processor communicates with the host CPU via two modes: hardware interrupt and shared memory. The interface between the network controller and the host bus includes the logic for the host CPU and the on-board processor to interrupt each other. The on-board processor interrupts the host CPU to inform it of reception of a packet, or finish of a requested service, or some error condition that must be handled by the host. The host CPU interrupts the on-board processor whenever it needs immediate service attention. As will be discussed in Chapter 7, the host also has a choice not to be interrupted for certain events by indicating this choice in the *Service Request Record* passed to the network controller. The network controller can then use shared service request/status records to communicate with the host.

5.7. Implementation Consideration

Extension of the logic function for a token ring or FDDI MAC component to handle packet train transmission and reception is a straightforward process. Our

proposed registers are same in structure and logic as the existing timers and registers of the IEEE 802.5 and FDDI MAC protocols, such as THT and TRT timers. The state machine for MAC packet train operations, described in the next chapter, requires some extra states and tests of boolean expressions composed of various registers (see Appendix B). The additional hardware logic is less complex than that already required for the IEEE 802.5 or FDDI interface device MAC components.

For other major components, such as the RAM and on-board processor, commercially available hardware chips and devices can be used with few extensions. Implementation for the packet buffer memory, however, deserves some discussion here.

Several implementation choices exist for the packet buffer memory: a fast single port RAM, a multiport fast RAM, and a video RAM. Suppose the host bus is the 320 Mbps (block transfer mode) VME bus and the network is the 100 Mbps FDDI. In order to provide adequate bandwidth for host DMA and the network, and a 160 Mbps random access bandwidth for the on-board processor, a fast single port static RAM would require a read/write cycle time of 55 nanoseconds for a standard 32-bit wide memory. Although such memory is available, but it cost more and has less memory density than a video RAM. Use of a true multiport fast memory provides a straight forward implementation for the packet buffer memory, but at present, its cost may be several times that of a video RAM providing adequate bandwidth.

Currently, a video RAM seems to provide a reasonable implementation choice for our multi-port buffer memory. A video RAM consists of a set of dual-port static column RAM's. The dual ports are independently accessed, one for serial access and one for random access. The random access port, just like any single port RAM, requires address arbitration for each access which can be for any memory location. The serial access port connects to a shift register internal to the video RAM. The access to the serial port does not require address arbitration but can not be for any memory location (only the next serial location). The access to the serial port is actually to the shift register rather than the memory itself until the shift register is filled up or emptied. At this point, the shift register can be written to a row of video RAM cells, or a row of video RAM cells is fetched into the shift register, all in one memory cycle. When doing this, a memory row address is required and a memory cycle is "stolen" from the random access port. However, such "cycle stealing" has negligible effect on the effective bandwidth of the random access port. Nowadays, video RAM's with 32 bit word width, 40 nanosecond cycle time for the serial port and 200 nanosecond cycle time for the random port are available. Such a video RAM would provide a 800 Mbps serial access bandwidth and 160 Mbps random access bandwidth, and would be low in cost and large in size as compared to a multi port RAM providing the same bandwidth.

Figure 5.3 depicts how a video RAM is used to implement the buffer memory. The random access port is connected to the network controller's bus. Both host bus

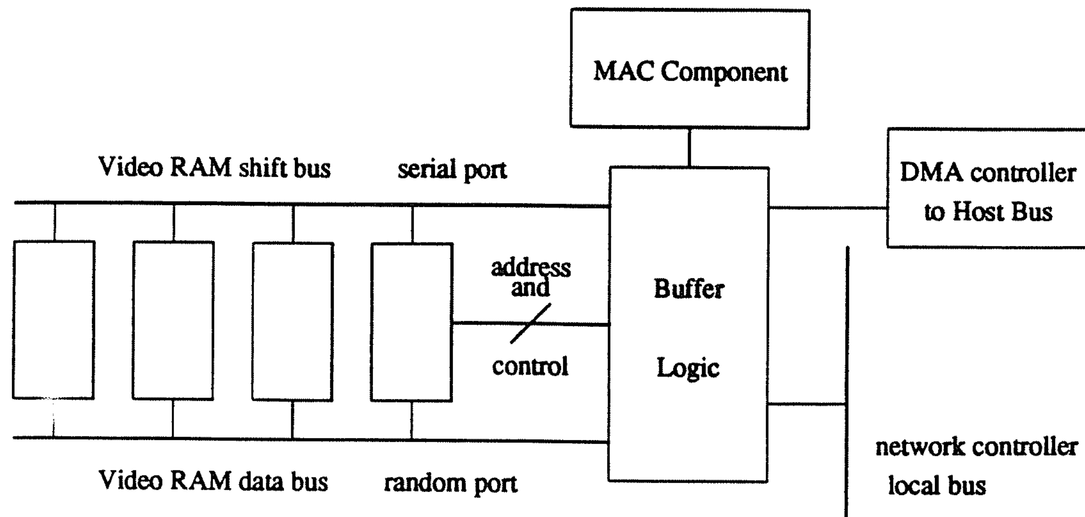


Figure 5.3: A Video RAM-based Buffer Memory

DMA and MAC component require only serial access. They are connected to two shift registers internal to the video RAM. The two shift registers share the one serial access port. In case of contention for the single serial port, 8 octet preamble and 1 octet Starting Delimiter field for IEEE 802.5 and FDDI packet format would allow sufficient buffering time at the MAC for resolving the contention without losing data from the network. Such a video RAM has been used by some network interface buffer memory[22].

Complicated data structures are maintained in the buffer memory. However, all is required for their management is the ability to manipulate pointers. The on-board processor can access any memory location for pointer manipulation via the random access port. Furthermore, such access through the random access port can

occur concurrently with DMA transfer and network transmission operations.

Chapter 6

Transmission and Reception of Packet Trains

In this chapter, we describe how packet trains are transmitted and received in the environment of token ring MAC protocols. We concentrate on how various MAC protocol operational parameters affect the timing, size, and transmission priority of packet trains. We also describe the functions that the network controller's MAC component must execute during packet train transmission and reception. We describe requirements for interrupt handling routines for packet train transmission completion, train transmission error and train reception.

6.1. Packet Train Transmission

In order to have a close succession of packets as a packet train on the medium, the sending station must be able to keep the token while transmitting a packet train until either the current train packets are exhausted or the token is requested for transmission of a higher prioritized packet. Not all token ring MAC protocols would allow this. For example, the Proteon Token Ring is a strictly non-exhaustive transmission method, that is, a station must release the token after transmitting each packet. An exhaustive transmission method is also not suitable because it does not provide for prioritized transmission. The two standard token ring MAC protocols, the IEEE 802.5 and FDDI standards, can support transmission of packet trains and

the packet priority mechanism. Although no structural extensions to these two protocols are needed, various operational parameters for these protocols have significant effects on packet trains in terms of their timing, size, and transmission priorities. The following two subsections describe packet train transmission with these two MAC protocols.

6.1.1. Packet Train Transmission and IEEE 802.5 MAC Protocol

The IEEE 802.5 Medium Access Protocol supports prioritized packet transmission. A station can only transmit a packet with a priority that is at least as high as the current token priority and the token reservation priority. Token reservation priority is provided for stations to compete for the token via the priority mechanism. For details, the reader should consult the IEEE 802.5 document[40]. This MAC protocol allows a station to keep transmitting packets with priorities no lower than the current token priority and the advertised token reservation priority. However, each station must limit the time it can hold the token to the value of the station's Token Holding Timer (THT). The specified default value for THT is 10 milliseconds. Packet train transmission can be readily accommodated by the IEEE 802.5 MAC protocol by keeping the token while transmitting a packet train until either the token must be released due to a higher priority token request, or until the transmitting station's THT timer expires. The priority advertised for a packet train by a station determines the medium access delay for a packet train. The length of time for which the token can be held determines the size of a packet train.

Depending on the urgency of the bulk data to be carried by a packet train, a transmitting station can choose an appropriate priority level for the request of the token, using the priority mechanism without any structural extensions to the MAC protocol. However, we impose a restriction for the use of the priority mechanism for packet train transmission: transmission priority must be the same for all packets in a packet train. The rational behind this restriction is that interruption of transmission of a packet train should be possible by any station that needs to transmit a higher priority packet. It may be tempting for a station to raise its priority during transmission of a train in order to maintain a large train size, but doing so would be detrimental to the MAC protocol's ability to provide urgent transmission for real time traffic. Furthermore, this restriction also maintains the data link layer service view that a packet train is a logic transmission unit having consistent service properties such as transmission priority. The restriction must be enforced by the MAC controller hardware, as described in the previous chapter, since the priority mechanism is based on a per packet basis and a sequence of train packets is still treated as separate PDU's by the MAC protocol.

The THT value sets an upper bound limit on packet train size. On an 80 Mbps ring, the 10 ms default THT value would allow a station to transmit a packet train of up to 50 2-Kbyte packets. Such a packet train size can enable high performance gains. In fact, our simulation study to be discussed in Chapter 8 shows that performance gains are upper bounded by large train sizes. Increasing train size

beyond a certain point achieves little extra performance gain.

Furthermore, the THT time-out value can not be arbitrarily increased so as to allow extremely large packet train sizes. The THT value is part of the time-out value of the TVX timer, which is used by an active ring monitor to detect the absence of valid transmission (for details consult[18,40]). Increasing the THT time-out value would increase the delay for such error detection. For token rings with low transmission bandwidth, packet train sizes would be severely limited. However, the packet train mechanism would not be useful for data transfer performance on such rings in the first place. For future high bandwidth rings, small THT values, for instance, in the neighborhood of 10 milliseconds, will not present problems for train sizes.

6.1.2. Packet Train Transmission and FDDI MAC Protocol

The FDDI standard[21] provides three modes of transmission: synchronous, asynchronous and restricted asynchronous. The synchronous mode bandwidth is used for packets that have real time deadlines, and ring stations demanding synchronous transmission are allocated a proportion of the ring bandwidth using a bandwidth allocation procedure (details not discussed here). The asynchronous mode is for less time critical traffic and no bandwidth is guaranteed for this mode. There is also an optional priority mechanism for the asynchronous transmission mode. The restricted asynchronous transmission is used to initiate an extended dialogue requiring substantially all of the the unallocated ring bandwidth (e.g., an extended

burst data transfer from a high speed device). The FDDI protocol requires each station to maintain a set of timers like THT and TRT (Token Rotation Timer). TRT is reinitialized to T_Opr each time it expires. T_Opr is a parameter negotiated among all stations on the ring and 2 * T_Opr is the minimum token rotation time that is guaranteed for all stations. If the token circulates to a station before the TRT expires, THT is initialized to the current value of TRT and TRT is reinitialized to T_Opr. A station can always transmit synchronous packets within its allocated synchronous time fraction, but is only allowed to transmit asynchronous packets until THT has expired (THT is enabled during asynchronous mode transmission). Obviously, if the token circulates back to a station late, the station is not allowed to transmit asynchronous traffic. The restricted asynchronous mode is entered when a token is marked as a restricted token and no other station can use a restricted token for asynchronous transmission. For the protocol details, consult ISO FDDI documents[18].

As with the IEEE 802.5 MAC protocol, no structural extensions are needed to accommodate packet train transmission. All three transmission modes can be used to transmit packet trains. However, each transmission mode has different requirements for train data properties and should be used under different traffic load conditions.

Use of synchronous transmission for packet trains requires train bulk data to have real time urgency. Packet train data should also exhibit regularity in its arrival

behavior from host users. Since there is overhead associated with bandwidth allocation for synchronous transmission (see [18]), an irregular arrival behavior would lead to either poor bandwidth utilization or frequent invocations of the bandwidth allocation procedure. When total bandwidth allocated for synchronous transmission is a small fraction of the medium bandwidth, use of asynchronous or restricted asynchronous transmission for packet trains should be encouraged, because it would actually provide sufficiently prompt network access.

In asynchronous mode, a station may keep the token for transmission of packet trains until either train packets are exhausted or THT expires. Although asynchronous transmission for trains can tolerate irregular train data arrival behavior, train size is limited to the proportion of bandwidth left from synchronous bandwidth allocation. If all the stations' synchronous bandwidth allocations add up to 100 percent of T_Opr[37], asynchronous traffic will be virtually blocked from the ring. This is a condition of network capacity saturation by traffic load, which should only occur rarely on a properly configured network. In general, the allocated bandwidth for synchronous traffic should be a small fraction of the capacity, making this asynchronous mode most appropriate for packet trains.

The THT value, initialized to TRT when the token arrives on time at a station, can also be severely limited in the case when asynchronous traffic is heavy, and evenly distributed on multiple stations. A smaller THT value limits transmitted trains to small sizes. To maintain reasonable packet train sizes, the restricted

asynchronous mode can be used. In this mode, the token is marked as restricted when transmitting train packets. Since no other stations except the initiating and destination stations can use the asynchronous bandwidth for packet transmission, virtually all the asynchronous bandwidth seized by the initiating and destination stations can be used for packet trains, allowing large size packet trains to be transmitted. After train transmission, the transmitting station restores the token to the unrestricted state.

If a priority mechanism is available for the asynchronous mode, the restriction that the same priority level be used for all packets in a train must also be observed for the FDDI MAC protocol for the same reasons mentioned in the previous subsection.

6.2. Interruption of a Packet Train for Urgent Packet Transmission

When an urgent packet needs to be transmitted, an ongoing packet train on the network medium can be interrupted by raising the reservation priority above the priority of the train packets. A train transmitting station must release the token upon seeing a reservation priority higher than the train packet priority. For FDDI, synchronous transmission bandwidth is guaranteed. Use of the priority mechanism to request immediate token release is only necessary for urgent asynchronous packets.

When releasing the token in the middle of transmitting a train, a train transmission completion interrupt is generated just as an interrupt would be generated when the whole train is transmitted. In order to preserve the sequence order of train packets, the interrupt handler routine must treat the rest of the train packets as one separate train. The interrupt handler has the option to either leave the rest of the untransmitted train packet descriptors on the transmission list or remove all of them from the transmission list. This can be done by manipulating the transmission packet descriptor list using the *Train Start* and *Current* registers. After handling this interrupt, resumption of transmission of the rest of the interrupted train is treated simply as transmission of a new packet train.

6.3. Error Handling for Packet Train Transmission

The following error cases can occur during transmission of a packet train:

(1) Corrupted data caused by the signal interference or by malfunctioning hardware. The FCS field is used to detect such errors and the E bit in the Frame Status field of a packet will be set by a repeating or receiving station upon detection of such an error.

(2) Packet not copied error, caused by either malfunctioning hardware or buffer overflow at the receiver. When this error occurs, The C bit in the FS field will not be set by the receiver.

Other error cases caused by hardware problems such as a broken ring usually need human intervention and can not be recovered by the MAC protocol. For the packet train mechanism, we only discuss error handling for the above two cases. When any one of these errors occurs, the receiver station stops receiving any train packets following the erroneous train packet by simply clearing the C bit in the FS field of the rejected packets.

At the sender station, when either the E bit is set or the C bit is cleared in the FS field of a train packet circulating back, the sender MAC immediately stops transmitting the rest of the train packets and generates a transmission error interrupt.

Note that on a large ring, signal propagation delay can be sufficiently long to allow more than one packet on the ring at any particular time. For example, FDDI allows the default of maximum ring delay (*D_Max*) to be 1.617 milliseconds, and a maximum size packet frame transmission time must be limited to 0.361 milliseconds (*D_Max*). Obviously, it is possible for the sender station to have transmitted a number of packets before an erroneous packet circulates back to it. In this case, if the receiver is allowed to receive subsequent packets following the erroneous one, retransmission of the erroneous packet at a later time will introduce an order of train packets different from the original train packet order. Although this out-of-sequence can be checked and corrected by higher layer protocols using the packet sequence number, doing so not only adds complexity to higher layer protocols but also violates the train transmission service property that a train of packets will not arrive

out of order at the receiver's network interface buffer. Therefore, the rule that a receiver station aborts receiving any train packets following detection of a packet error must be enforced to eliminate the possibility of out of sequence train packets.

The handling routine for the transmission error interrupt examines the transmission status of each packet starting with the packet descriptor pointed to by the *Train Start* register. All packets with C bit error as recorded in the status field in their packet descriptors, together with descriptors for untransmitted packets are treated as untransmitted packets of a new packet train. Usually, the train error handling routine will attempt retransmission of the untransmitted train packets if the error is data corruption type (E bit set). If the error is suspected to be buffer overflow, retransmission is delayed by removing the train packet descriptors off the transmission list and marking its corresponding service request record to this effect.

6.4. MAC Controller Operation for Packet Train Transmission

The network controller MAC starts its transmission of the first packet in a train the same way as it does for a non train packet, except that it does not generate an interrupt to the processor if the first train packet is not also the last train packet. After it starts transmission of the first train packet, the MAC component enters the *train transmission* mode, within which the network controller MAC component repeatedly performs the following function during transmission of each train packet. These functions, depicted in the following flow chart, must be executed within the real time of one packet transmission.

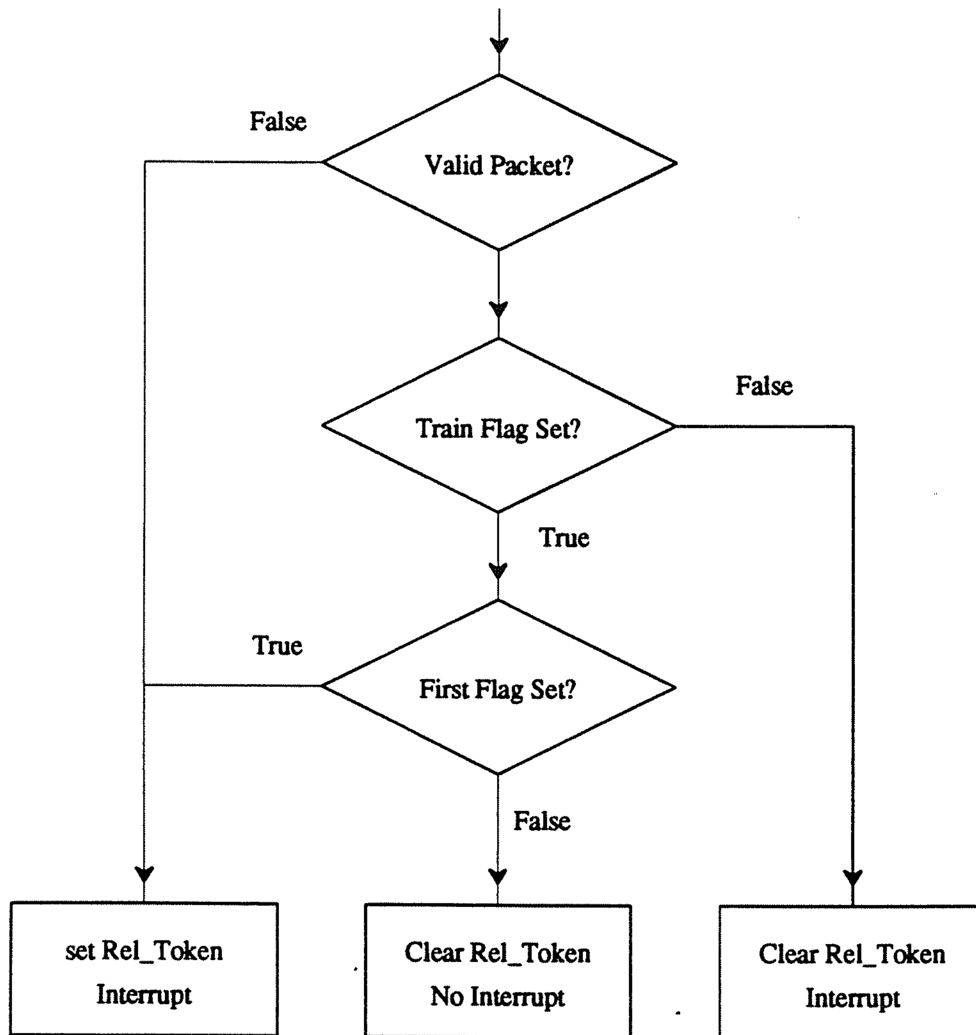


Figure 6.1: MAC Operation During Train Transmission

First, the MAC microcode program uses the *Next* field of the *Current* packet descriptor register to access the packet descriptor for the next packet to transmit. If the packet descriptor for the next packet is not available or not valid, the MAC sets Rel_Token flag so the token will be released and a packet train transmission

interrupt will be generated to the processor when transmission of the current train packet is completed. If it is valid, it accesses the *train* and *First* fields to determine if the next packet is a train packet. If it is not a train packet, the Rel_Token flag is cleared, and train transmission interrupt is generated. In this case, a station may transmit other non train packets without releasing the token depending on the packets' priorities. If it is a train packet but the *First* flag is not set, then there are more train packets to transmit. *Rel_Token* should be cleared and no interrupt should be generated. If the *Train* and the *First* flags are both set, the station has come to the start of new packet train. In this case, it sets *Rel_Token* and generates an interrupt to the on-board processor. Releasing the token prevents a receiver station from mistakenly receiving two different trains from a same sender station as one train.

6.5. MAC Controller Operation for Packet Train Reception

The following graph is the state transition diagram of the MAC component during reception of a packet train. States are numbered for the convenience of our discussion here.

The MAC component at a receiving station enters the RC_FR_CTRL state from the LISTEN state when the Starting Delimiter field of a packet is received. Within this state, a number of actions are taken by the MAC component for train or non train packets (see [18,40]). Upon examining the Frame Control (FC) header field which indicates whether the packet is in train format, the MAC microcode program will enter one of the three states based on the following conditions.

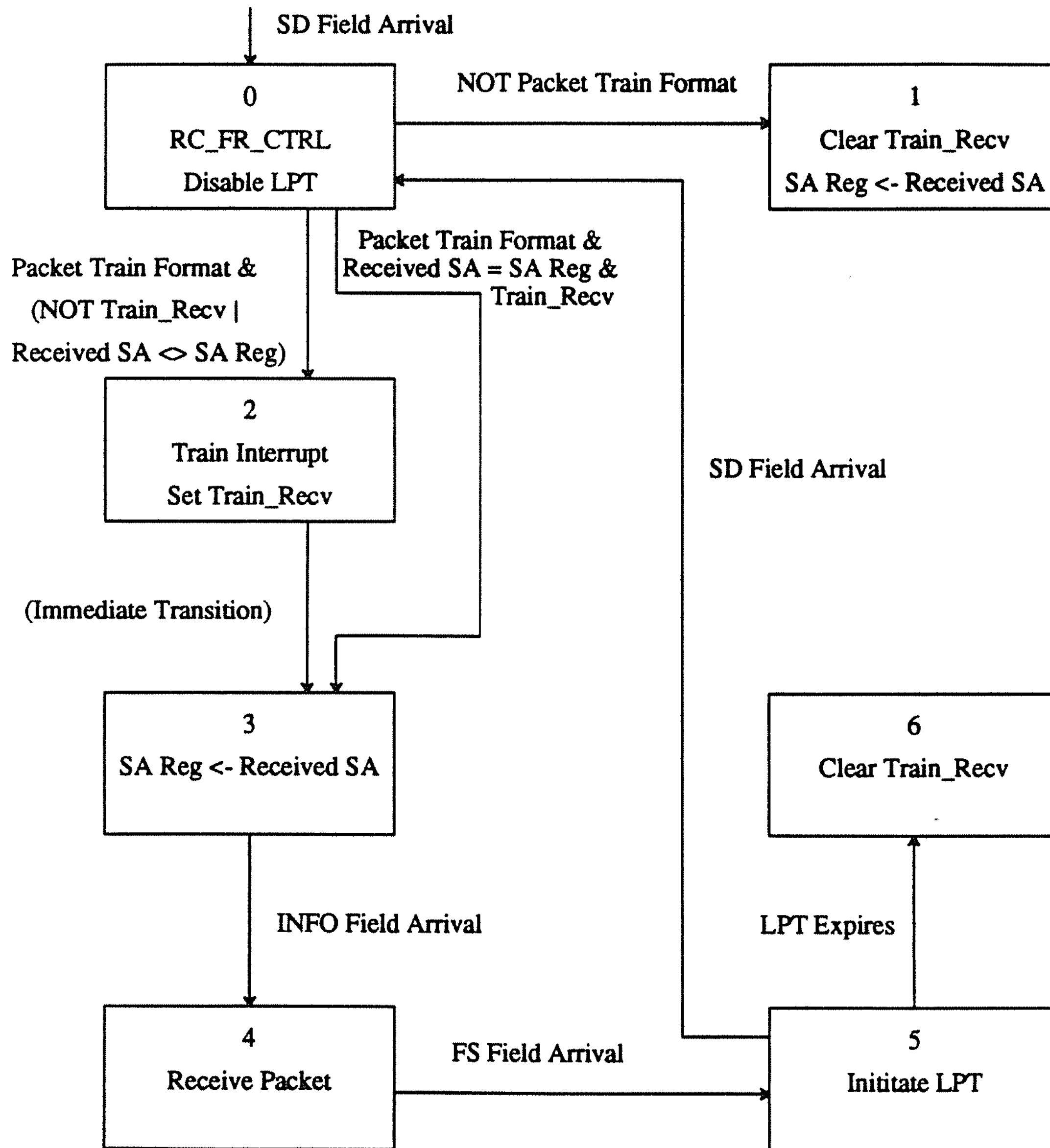


Figure 6.2: MAC State Transition for Train Reception

(1) If it is a non train packet, it enters State 1, in which it clears the *Train_Recv* flag and writes the received SA field into the *SA* register.

(2) If it is a train packet, but the *Train_Recv* is not set or the received SA field value is different from the *SA* register, it enters State 2, in which it sets the *Train_Recv* flag and generates a train arrival interrupt to the on-board processor. Then it unconditionally transits to State 3.

(3) If it is a train packet, and both the *Train_Recv* is set and the received SA field value is the same as the *SA* register, it enters State 3, in which it overwrites the *SA* register with the received SA value. This transition corresponds to the receiving subsequent train packets while the MAC is in the train reception mode. In this case, no train arrival interrupt is generated.

For every packet in train format, the newly received SA field must be compared against the stored SA field of the previous packet in order to distinguish the start of a new packet train. This is because the sender station may release the token upon exhaustion of train packets to send, while the receiver station is still in packet train reception mode. The released token may be seized by a downstream station for transmission of a packet train to the same receiver station whose position on the ring is further downstream. In this case, the receiver station will receive two packet trains from different senders with a time delay in between that may be shorter than *Maximum Packet Gap*. The receiving station can not rely only on the *Train_Recv* flag to distinguish between the two trains because of the short time delay. Both the *Train_Recv* and the received SA field must be checked for each train packet. In all three cases, the *SA* register is always refreshed with the received SA field.

State 4 is for reception of one train packet. The actions in this state are essentially the same as in the packet copy state for a non train packet, except that the *Train* Bit is set in the corresponding packet descriptor. The *First* bit is set in State 2.

State 5 is entered when the Frame Status is received, in which the *Last Packet timer* is reinitialized to the *Maximum Packet Gap* timer and starts to count down. If it expires before another packet Starting Delimiter field arrives, State 6 is entered in which *Train_Recv* is cleared to indicate the end of train reception mode. If another SD field is seen before the timer expires, State 0 is entered and the *Last Packet* timer is disabled. The state transition process described thus far is repeated for each subsequent train packet.

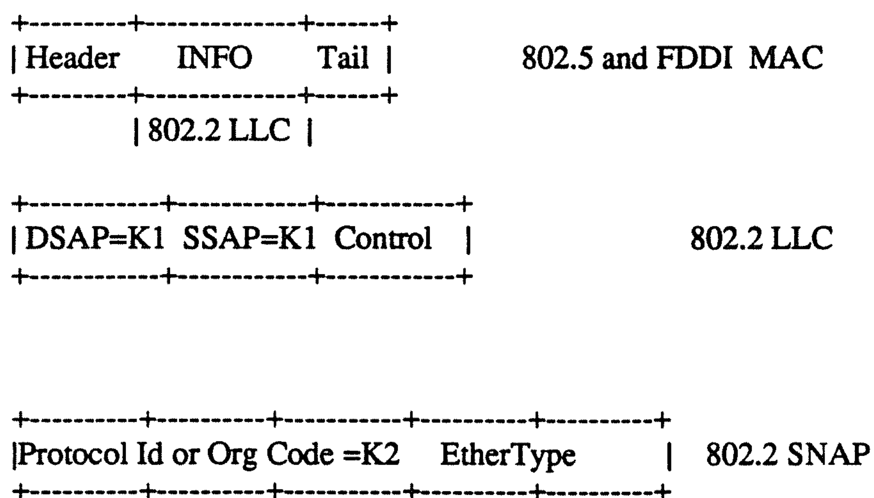
6.6. Packet Train Reception Interrupt Handling

Requirements for train transmission interrupt handling are relatively straightforward, and thus we do not describe its details here.

Packet train arrival interrupt handling is more complicated than packet train transmission interrupt handling. In addition to checking on the reception status of each train packet, the handler must locate the corresponding *Receive Request Record.* The on-board processor uses high level protocol information in the record to process the received packet train and host buffer information in the record to start DMA transfer for the received data as soon as possible. We describe certain details of this interrupt handling. The following is the pseudo code for the interrupt handler.

```
      pd <- current packet descriptor;
      busy wait on [NOT pd^.MAC_Owner Bit];

      /* get protocol type code at proper buffer offset */
first:  prot <- (int   pd^.packet_pointer + 6);

      /* prot type code used as an index to a data link layer hash table,
        call a previous registered routine to look up the service record
      */
      lookup_routine <- DataLinkTable[Hash(prot)].lookup;
      rec <- result of calling lookup_routine (pd^.packet_pointer);

loop:   /* process train packet in pd using rec  */
      IF NOT DMA_Control_Busy THEN
         Enable DMA with the following parameters
             rec^.Host_Data_Buff_Offset
             rec^.Host_Data_Buffer.length
             pd^.packet_pointer
             pd^.packet_length
         mark pd^.DMA_State as enabled
      ELSE
         mark pd^.DMA_State as pending
      END;

      /* set pd to the next packet descriptor and busy wait either for
        reception of next train packet, or for the end of packet train reception
      */
      pd <- pd^.Next;
      busy wait on [NOT pd^.MAC_Owner] OR [NOT Train_Recv];

      IF NOT Train_Recv THEN EXIT;

      IF pd^.First THEN GOTO first  /* if start of a new train */
                 ELSE GOTO loop;
```

A local variable, pd, is always set to point to the next unprocessed packet descriptor. First the processor busy waits on pd's MAC_Owner bit to clear, an indication that the first train packet is completely received into the current packet descriptor. Then the protocol type code is retrieved from the packet buffer which contains the INFO field of the received packet. The first 8 octets of the INFO field contains the LLC header and SNAP header as shown below[33].



```
+---------+-------------+------+
| Header     INFO     Tail |          802.5 and FDDI  MAC
+---------+-------------+------+
          | 802.2 LLC |

+------------+------------+------------+
| DSAP=K1  SSAP=K1  Control  |              802.2 LLC
+------------+------------+------------+


+----------+----------+-----------+----------+----------+
|Protocol Id or Org Code =K2    EtherType        |   802.2 SNAP
+----------+----------+-----------+----------+----------+
```

The total length of the LLC Header and the SNAP header is 8-octets, making the 802.2 protocol overhead come out on an nice boundary.

The K1 value is 170 (decimal).

The K2 value is 0 (zero).

The control value is 3 (Unnumbered Information).

The EtherType code, which is the protocol type code, is the last two of the 8 octets. This integer protocol code is used as an index to the DataLinkTable, and the proper lookup routine at the table entry thus indexed is then retrieved and called. This routine contains knowledge of header structure for the higher protocol layer corresponding to the integer protocol code, and therefore can search for the right *Receive Request Record* for the packet contained in pd. The returned record pointer rec is used in processing the packet in pd.

A host usually multiplexes different higher level protocols onto the same Logical Link Control layer. It is very difficult, if not impossible, to encode in this interrupt handling routine high level header structure information needed to find the appropriate *Receive Request Record* for the packets being received. To circumvent this difficulty, we register different lookup routines with different protocol type codes in an array type of data structure called DataLinkTable, and have the interrupt handler call upward the appropriate lookup routine to search for the correct record. This technique is similar to what is called up-call structure[12]. This way, the interrupt handling routine need not have specific knowledge about high level header structures.

Processing of each train packet is rather straightforward. DMA transfer for the packet data is set up if the DMA interface is not busy, using the host buffer information available in the found service request record. Otherwise, the packet is left in the packet descriptor and their packet descriptors are still a linked list. A field

in the service request record is set to point to the packet descriptor list containing the train packets.

After processing one packet, the processor will set pd to the next packet descriptor and busy wait again for another train packet to be received. It also waits on the Train_Recv flag and will exit this routine if this flag indicates that train reception is completed. If another packet train is received, it checks the *First* field in the packet descriptor to see if this packet is the start of another train. Different targets for GOTO are chosen for different cases.

Chapter 7

Data Structures and Transport Layer Functions

In this chapter, we describe the major data structures used by the network controller for the packet train mechanism and for executing offloaded transport layer functions. We also describe what functions are executed at the network controller in the context of the packet train model.

7.1. Major Data Structures for the Packet Train Mechanism

7.1.1. Service Request/Status Records

Previous chapters have alluded to the Send/Receive Request/Status Records (abbreviated as SRR, RRR, SSR, and RSR). Service request records are used by the host to pass a service request to the network controller. Service status records are used by the network controller to pass status information about a requested service to the host. All service request and status records have the same structure. Four different values for the *Op_code* field in a service request/status record are used to differentiate whether a record is a service request or status record and send or receive service record. The service request/status records are all stored in the network controller's RAM and used for all packet processing, both train and non train. The following describes their uses in detail.

(1) Control information for packet processing. When preparing packets to send, a send request record provides information as to how to construct packet header fields and from where in host memory to obtain (e.g., by DMA transfer) packet data. When processing received packets, a receive request record provides information as to how to check on packet headers and to where in host memory the packet data should be delivered.

(2) As shared data objects between the host and network controller, they provide for shared memory mode communication in addition to communication by interrupts. In certain cases, it is unnecessary, and advantageous in performance, to obtain immediate attention from the host or the network controller by using interrupts for service request or response. In the network controller RAM, proper lists of service records are maintained for the host or the network controller to examine to learn about service requests or status responses whenever it is appropriate time for them to do so.

(3) Providing information for processing offloaded protocol functions. For example, the packet sequence number field is used to check missing packets and to update receive or send state with respect to the progress of a data transfer activity. There are several fields used for data and ACK timeouts. The network controller uses these fields to handle data and ACK timeouts.

(4) Serving as part of the "firewall" mechanism for the host processor. Each RRR specifies what packets a particular higher layer protocol user at the host

will accept. Packets for which the network controller processor can not locate an RRR are not passed to the host and will be discarded, thus protecting the host processor from being overburdened by unwanted packets. Unwanted packets arise from situations such as packet flooding caused by errors in host software or network protocols. For packets that are only processed by the network controller, appropriate RRR's are constructed and registered at system startup time.

Certain protocols can be completely offloaded to the network controller. An example of these is the ARP protocol. The network controller handles all ARP requests from the network without having to involve the host at all. In order to receive ARP packets and direct these packets to the network controller, a receive request record for all ARP packets is pre-registered in the RAM by the network controller at the system startup time. The *Host_Process* and *Context _ptr* fields in this record are properly set so that ARP packets will be directed to the network controller for processing.

The following pseudo code describes the content of a service request/status record. Each record has three parts: the host specific part, the network controller work area part, and the management part. The first part is used to pass service request/status information from/to the host. It also has information to identify a record with a higher layer protocol process. The second part is used by the network controller for storing information relevant to packet processing. This information

mainly includes current sent (or received) packet sequence number, currently used host buffer offset for data fetch (or store), state of DMA operation, and pointers to packet descriptors in the buffer memory. The third part is for record management and is mainly pointers to other service request/status records.

The *ID* field is used to associate a service record with its user at the host whose service request caused the record to be created. In the case of TCP, for instance, the *Destination Port* field in a packet's TCP header is used to match the *ID* field during a search for the appropriate record for a packet. For different protocols, the *ID* may be interpreted differently. The Host_Owner boolean flag indicates which processor has both read and write access rights to the record. When it is set, the on-board processor has only read access but the host processor has both read and write access rights to the record. This flag is set when the network controller no longer needs the record for processing. When the network controller is using a service request record, it sets the *Using*, so that the host is warned of possible updates to certain fields that it may be reading at the same time. The *Op_code* indicates the type of service operation requested. The *Int_code* indicates the condition on which an interrupt is generated to the host by the network controller. The *Host_Process* is a host process ID, used by the host to quickly find the corresponding transport layer protocol process. *Context_Ptr* is used to find the process context block. Because a transport layer header prepared by the host may be in a buffer separate from that for the data, two buffer descriptors are needed. The *Int_Allowed* Boolean flag is the field used by

---

```
Service_Record {
    /* host specific part */
    id : int;  /* transport layer id, (e.g., port number for TCP or UDP) */
    Host_Owner : Boolean;   /* used for mutual exclusion */
    Using : Boolean;  /* set when network controller is using the record */
    Op_code : int;  /* send, receive, send_status, or receive_status */
    Int_code : int;  /*  condition on which an interrupt generated */
    Host_Process : int; /* id number for transport process at host */
    Context_ptr* : char; /* pointer to host process control block */
    Host_Data_Buffer  {
        addr* : char; /* data buffer address */
        length : int;  /* data buffer size */
    }
    Host_Header_Buffer  {
        addr* : char; /* header buffer address */
        length : int;  /* header buffer size */
    }
    Priority : int;   /* requested service priority */
    Int_Allowed : Boolean;  /* interrupt to the host allowed */
    Int_Size: int;  /* byte size for which interrupt allowed */
    Data_Time : real;  /* timeout value for data packets */
    ACK_Time : real;  /* timeout value for ACK packets */
    Timeout_Threshold : int; /* when its value reached, interrupt host */

    /* network controller work part */
    Host_Data_Buff_Offset : int; /* offset into host buffer where data start */
    Byte_Num : int;   /* byte sequence number sent or received so far */
    Pkt_Num : int;    /* packet sequence number sent or received so far */
    Header_ptr* : char      /* point to header in the buffer memory */
    pd_alloc_ptr* : char;  /* point to allocated packet descriptor list */
    pd_built_ptr* : char;  /* point to built packet descriptors */
    pd_head_ptr* : char; /* point to first packet descriptor on a pd list */
    pd_tail_ptr* : char; /* point to last packet descriptor on a pd list */
    DMA_State : int;       /* state of DMA operation */
    Last_Data_Time : real;  /* time when last data packet received */
    Last_ACK_Time : real;  /* time when last ACK packet received */
    Timeout_Count : int;

    /* service record management part */
```

```
    next* : char;          /* link to next service record */
}
```

---

the host to indicate to the network controller whether an interrupt should be generated when the requested service is finished. As discussed in the above, the network controller may not generate an interrupt to the host upon finish of a request service, if the host so desires for reasons of performance.

Fields in the second part are used as a per transfer activity "working area" for the network controller. The *Host_Data_Buff_Offset* always gives the current offset into the host data buffer from/to where the network controller should fetch/store data. This field is updated by the size of data fetched or stored. The *Byte_Num* field is the byte sequence number that has been received or built into outgoing packets so far. *Pkt_Num* is the corresponding packet sequence number that has been received or built (not necessarily already sent) so far. *Header_ptr* is a pointer to the header template built in the buffer memory. *pd_alloc_ptr* and *pd_built_ptr* are pointers to the lists of packet descriptors allocated in the buffer memory and already prepared for use respectively. *pd_head_ptr* and *pd_tail_ptr* are pointers to the head and tail of a list of packet descriptors put on the transmission (or reception) packet descriptor list (see 5.3) respectively. The other fields are in the second part are self explaining.

The third part mainly contains pointers to other service records for purpose of maintenance and manipulation of service request/status records.

Memory management for these service request/status records is straightforward. All the on-board processor executed code is memory resident. Some RAM space is allocated to a runtime stack. The rest of the RAM is allocated to these service request records (recall that all packet descriptors and packet buffers reside in the buffer memory). A free pool of these records is maintained as a linked list. A new record is taken from the front of the pool list for a new service request, and a free record is put at the front when no longer needed. A service record can be reclaimed when the *Host_Owner* is set and the *Op_code* indicates DONE condition. This simple allocation strategy makes RAM management trivial.

Note that all service request records are allocated and created within the network controller's RAM. Since this RAM is part of the host memory address space, the host can allocate a record from the free pool and set its fields according to its service needs before it "passes" the record to the network controller. Figure 7.1 depicts the sequence of actions (by marked numbers) taken when performing data transmission or reception operations using the service request/status records.

7.1.2. Protocol Lookup Tables

A protocol lookup table is a data structure used to facilitate the search for the appropriate service request/status record for a packet just received or to be prepared to send out. Recall the pseudo code for a packet train reception interrupt handler described in 6.6. That interrupt handler first uses the EtherType code in a received packet to obtain a network layer specific lookup routine. The routine is then called to

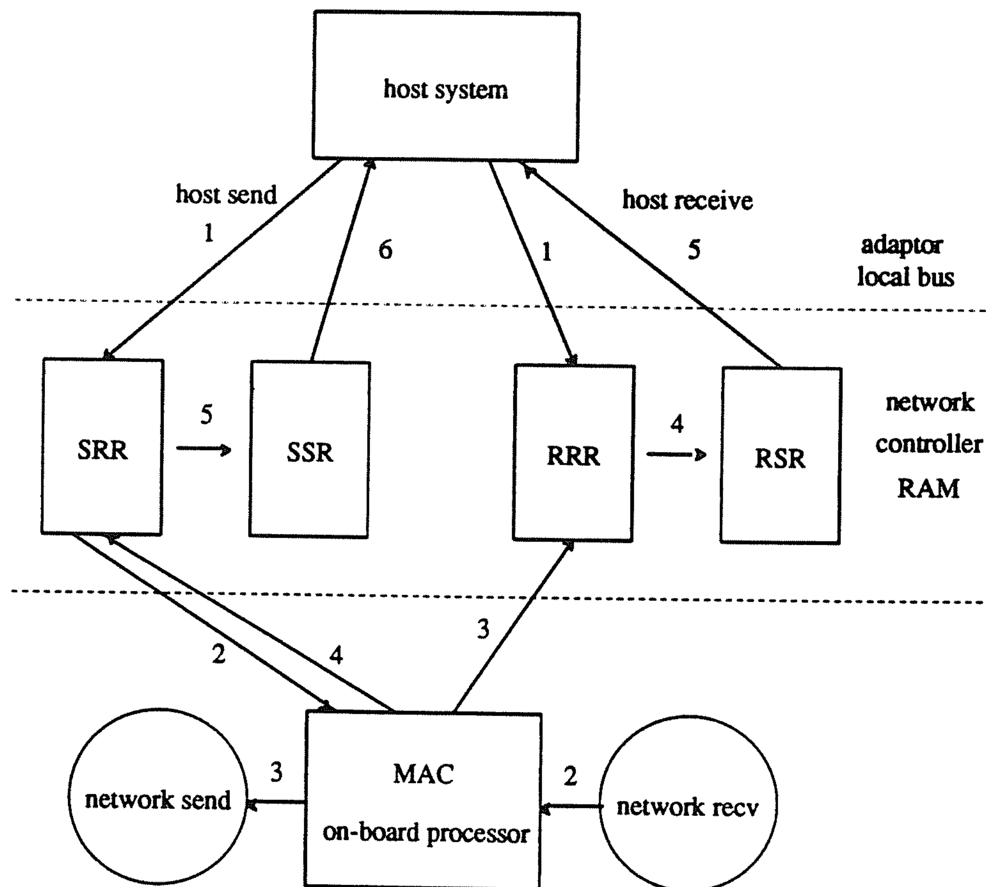


Figure 7.1: Use of Service Request/Status Records

look up in the protocol lookup table for the network layer protocol an entry corresponding to the transport layer protocol used by the received packet. A transport layer specific lookup routine (a pointer to it) found in the table entry is then called to locate the appropriate service request/status record on a list of such records linked from the table entry.

A different lookup table is used for each network layer protocol that is supported by the host. Figure 7.2 shows the structure and organization for one specific lookup table for the IP protocol, and how it relates to the service request/status records and packet descriptors. This table has an entry for each different protocol supported on top of IP. For example, there are table entries for TCP[35], UDP[34] and NETBLT[13] protocols. Each table entry is a record structure containing the following entries.

(1) protocol number, a value for the *protocol* field in the IP header;

(2) a pointer to a lookup routine which is stored in the network controller RAM but not shown in the figure. This lookup routine is called upwards from from the train interrupt handler to locate the right service request record;

(3) a pointer to a routine which is called during train packet encapsulation to replicate packet headers. The routine has knowledge of both network and transport header structures and updates header fields such as the packet size and sequence number when replicating a new packet header using the header template.

At system configuration time, these lookup and header replication routines are registered in the table entries for the higher layer protocols that use the IP protocol.

If a host supports multiple network layer protocols, more than one such protocol table is needed for the search and management of the Service Request/Status Records. Although all Service Request/Status Records have the

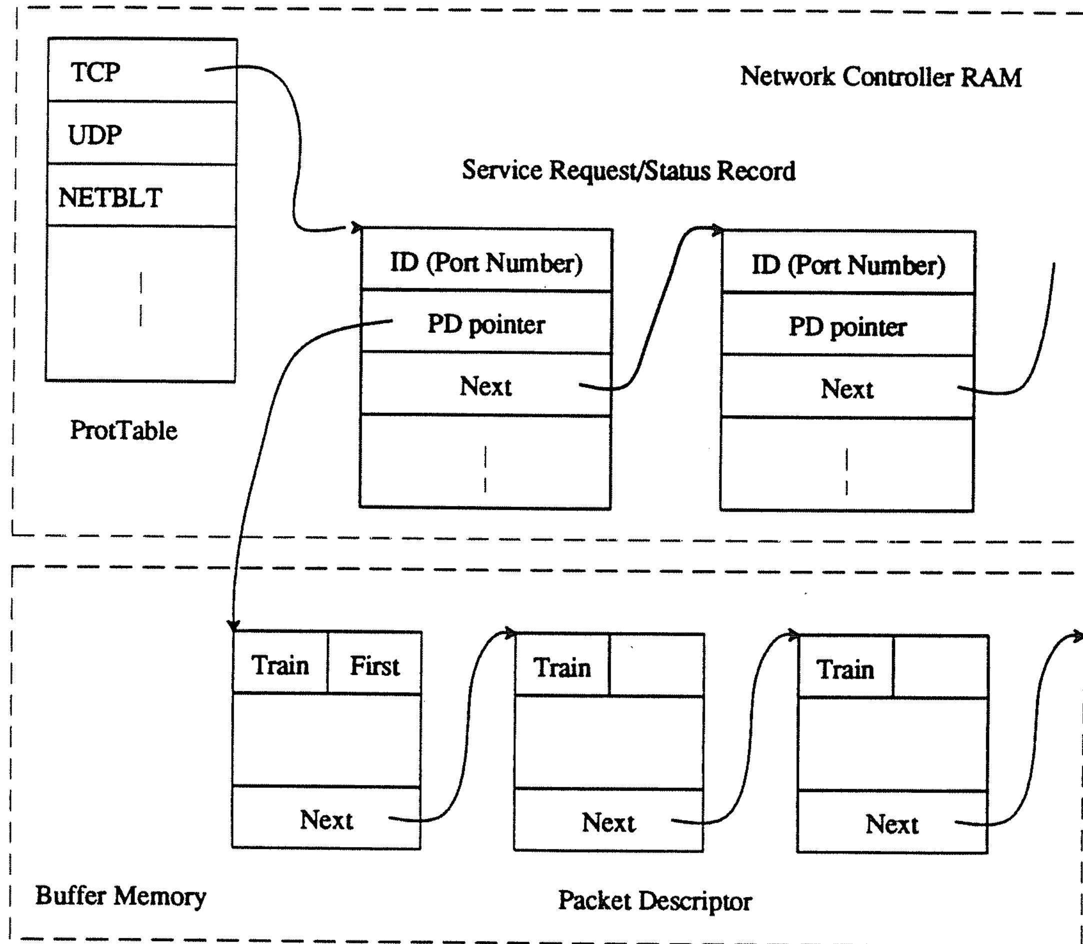


Figure 7.2: Protocol Lookup Tables and Service Request/Status Records

same structure, these tables may differ in structure and size, as dictated by different network layer protocols. However, the number of different network layer protocols will usually be small. For example, on many systems IP is the only supported network layer protocol.

7.2. Train Packet Encapsulation

Train packet encapsulation can take advantage of the structure of the packet descriptors to avoid data copy. When the host passes an SRR for sending a large block of data, the SRR contains memory buffer addresses for the data and packet header previously prepared by the host. This header thus passed contains transport or higher layer protocol information, and will be used by the network controller as part of the header template to construct all train packet headers.

The on-board processor copies the header from the host memory into the buffer memory. It allocates packet descriptors in the buffer memory. It then initiates DMA transfer to move data from host into allocated packet buffers linked from the allocated packet descriptors. The network controller combines a network layer header such as an IP header with the transport layer header copied from the host to form a header template. This header template will be replicated in the packet area of packet descriptors, with certain header fields updated on each replication. Because the transmit logic at the MAC component can extract packet data from both the *Packet Area* and the *Packet Buffer*, no data move is needed to collapse the header and data body into a contiguous block of memory before packet transmission.

Certain header fields must be updated when replicating the header template. Take TCP/IP for example. For the transport layer header, the *Sequence Number* needs to be incremented by the number of data bytes in the previous train packet. For the network layer header, the *Datagram Number* needs to be incremented by one

and the *Total Length* field may be shorter for the last train packet than for the other train packets.

Since these fields are protocol specific, each different header replication routine is needed for a different transport and network protocol combination. These routines are maintained in the protocol lookup tables described in the previous subsection.

Such train packet encapsulation allows a sequence of packets to be prepared with minimal processing overhead. Data copy of less than 100 bytes for header replication, plus the updating of a few fields, would take much less time than the DMA transfer of maximum size packets, the usual size for train packets. Further, DMA transfer of data from the host can proceed in parallel with header replication. The bottleneck is less likely to be software processing for packet preparation than the host bus DMA data transfer speed. This train packet encapsulation scheme provides an efficient means of constructing headers for a sequence of train packets.

7.3. Packet Train and Protocol Offloading

We now describe in detail those transport layer functions that are offloaded to the network controller processor in the context of the packet train model. Recall that in Chapter 4 we described our protocol offloading strategy as "vertical" offloading, different from the conventional offloading strategy of "horizontally" shifting the task of executing a whole protocol layer from the host to the network front end.

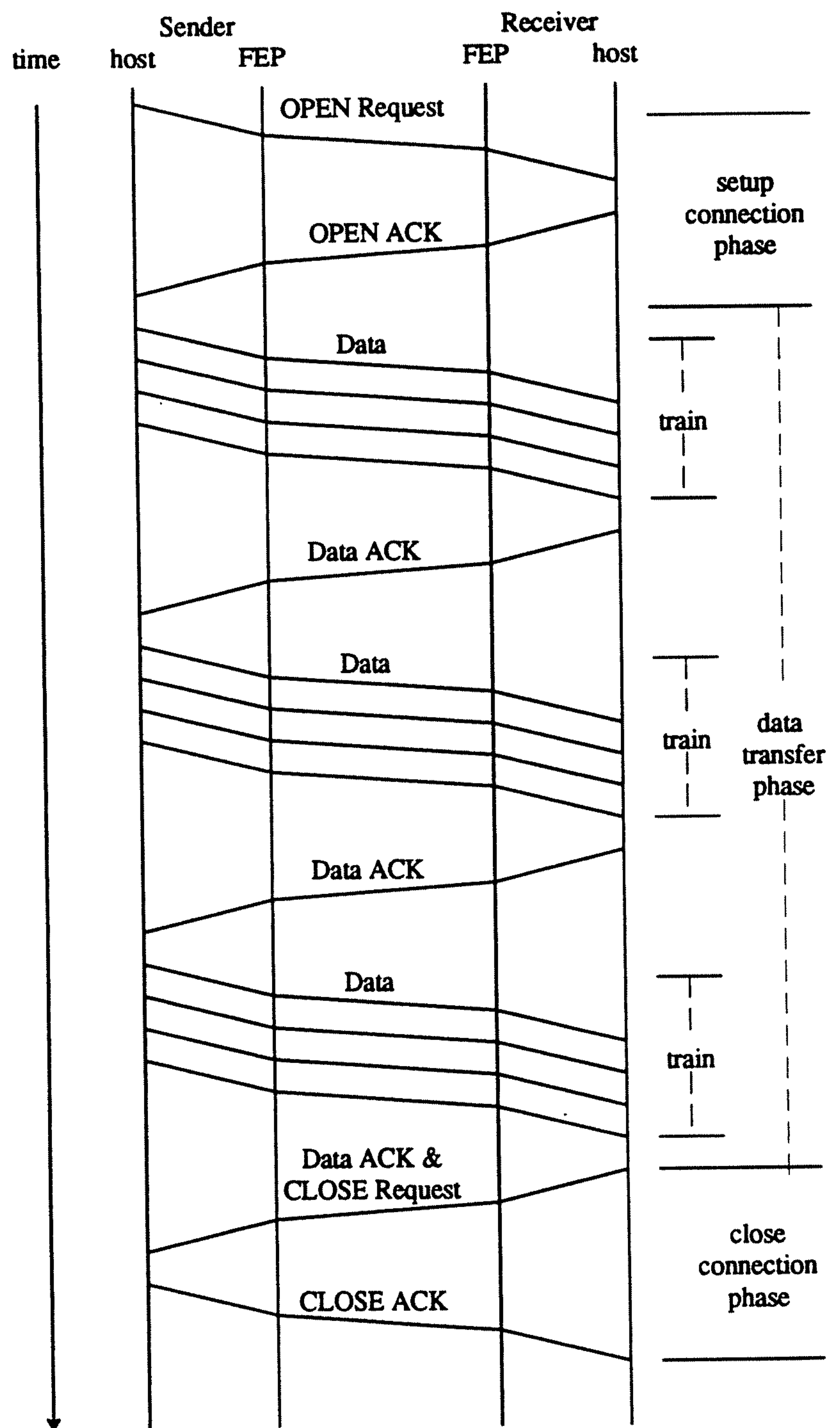


Figure 7.3: Protocol Offloading with Packet Trains

Figure 7.3 above depicts three phases involved in a bulk data transfer. In the first phase, the sending and receiving hosts pass several control packets back and forth to negotiate the setting up of a connection that will be used for the data transfer. Then follows the actual data transfer phase during which data is shipped from the sender to the receiver in chunks. In the last phase, the sender and receiver hosts inform each other of the end of data sent or received and finally close the connection. In the first and last phases, several control packets are passed back and forth between the two hosts, and functions are invoked at the hosts to negotiate and set up various parameters. In the data transfer phase, each data packet involves the same functions both at the network controller and at the host. Because of the large number of data packets, these functions will be repetitively executed and host operating system overhead must be paid each time a data packet is processed by the host.

In our offloading scheme with the packet train mechanism, no functions in the first and last phases are offloaded from the host. For the second phase, the actual data transfer, virtually all protocol functions invoked in the case of normal data packet processing and part of the functions invoked in the error cases are offloaded to the network controller. This second phase begins when the sending host passes a SRR to its network controller for transmitting the first batch of data, and the receiving host passes a RRR to its network controller for receiving the first batch of data packets.

the train transmission and reception level. This is due to the requirement specified in 5.3. Recall that train packet descriptors subsequent to a train packet descriptor for which any error is detected, must all be treated the same if they belong to the same transfer activity. Either all or none of these descriptors must be removed from the list. Furthermore, for any SRR, only one list of train packet descriptors is allowed on the transmission list at any time. These ensure that out of sequence errors will not occur at the packet train transmission level. Out of sequence error may occur at the transport level when received packet train in the buffer memory are destroyed before they are DMA transferred into the host memory, causing retransmission from the transport level. Retransmission at the transport level may introduce out of sequence errors at this level.

(2) Corrupted data in train packets. As described in Chapter 6, a corrupted packet is detected at the time of receiving the packet from the network medium and rejected right away. All subsequent packets in the same train, erroneous or not, are all rejected. The network controller must examine the error case and take needed actions such as retransmission of the train packets. Therefore, this type of error is handled by the network controller without intervention from the host.

(3) Inactivity for a long time. Both the sender and receiver may timeout on ACK and data respectively, indicating that the other side of the connection

may be either overloaded or dead. At the receiver side, timeout on data packets occurs at the network controller. The network controller periodically checks on RRR records and "timeout" when (current clock - *Last_Data_Time*) value is greater than the *Data_Time* value. The *Data_Time* value is passed in a RRR from the host when the data transfer phase begins. When a timeout occurs, the network controller increments the *Timeout_Count*. If the value is small, it sends out another ACK packet. Otherwise it informs the host by an interrupt. At the sender side, timeout on ACK packet triggers the network controller to send train packets with sequence numbers expected in the ACK packet. At the same time, the *Timeout_Count* is incremented. When its value is above *Timeout_Threshold*, it generates an interrut to the host. This means that the host must handle this error case.

(4) Host buffer overflow (flow control problems). The network controller will not DMA transfer to the host memory when the host buffer is full. It generates an interrupt with the *Int_code* indicating this condition to the host. Meanwhile, received data will be held in the buffer memory. The host can examine related fields in the RRR and decides what to do with this case.

When the last data packet is received by the network controller, it generates an interrupt to the host and the bulk data transfer enters the last phase, in which control packets will be accepted or sent from/to the network just as non-train packets.

Chapter 8

Simulation Study

In this chapter, we describe the simulation work performed to study the performance benefits of the packet train model. Our simulation program was coded in the DENET language[27].

Our goals of this simulation study were to understand the following.

(1) Under what system configurations the packet train mechanism will benefit bulk data transfer throughput performance. The relationship between some major system configuration parameters and the performance improvement by the packet train mechanism were studied.

(2) How the packet train mechanism affects urgent data transfer delay performance.

(3) A comparison between the packet train mechanism and the non-train traditional packet transmission scheme under different traffic intensities and system parameter assumptions.

Our simulation study was not meant to be an evaluation of the proposed network controller architecture, rather it was an evaluation of the performance gains by the packet train mechanism that can be supported by such network controller designs. Therefore, we assumed support for the packet train mechanism on a

network controller like our design, but did not simulate the details of such a network controller.

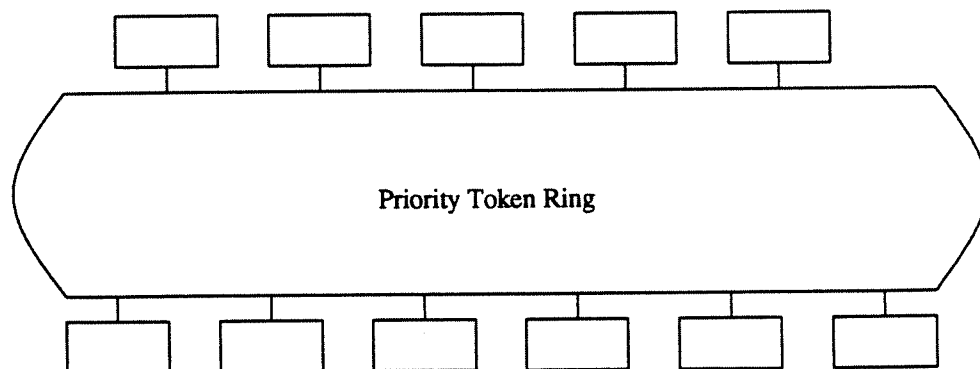
In the following, we describe the simulation model and parameters, and the major conclusions made from the simulation study.

8.1. Simulation Model

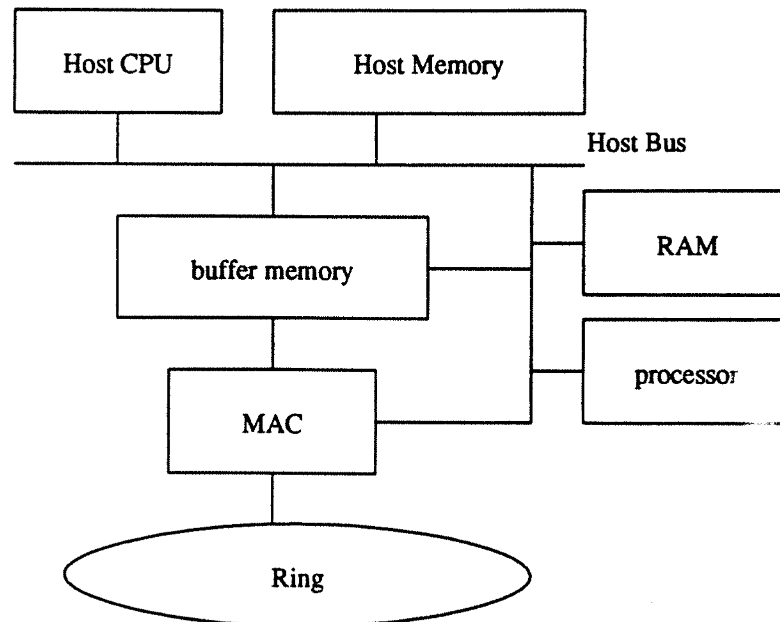
As shown in Figure 8.1 below, we simulated a priority token ring local area network of 80 Mbps bandwidth. The ring had 11 nodes and all were of the same node structure based on our network controller design. In our simulation runs, all 11 nodes were actively sending and receiving both bulk and urgent data. Because the actual number of nodes simultaneously performing such data transfers is rather small even on a large ring network with hundreds of connected nodes, we did not simulate a large ring network with most of the nodes being inactive. The token ring access method simulated is based on that of IEEE802.5. Each transmitted packet was removed from the ring by the source node. When transmitting a packet train, a transmitting station could hold the token as long as it had packets to transmit and there was no request for the token by higher priority urgent data. Although the simulated MAC protocol is strictly not an FDDI MAC protocol, our simulation results should also hold for an FDDI ring environment. This is because the FDDI's synchronous traffic transmission can be simply treated as more frequent token requests for urgent data as in the IEEE802.5 MAC protocol. For the priority mechanism, two priority levels were simulated, one for non urgent bulk data and one

for any urgent data. We felt no need to simulate 8 levels of priority because two levels of priority would give us the same effect of interrupting the transmission of packet trains as long as urgent data transmission was given high enough priority to interrupt an ongoing packet train. Transmission of an interrupted packet train would resume simply as a new packet train.

Figure 8.2 shows the structure of a node on the ring. Each node had a network controller similar in structure to the network controller described in the previous chapters. Each node's MAC component had one octet signal propagation delay. Packets were sent/received from/to the buffer memory on the network controller. Bulk data carried by train packets was DMA transferred from/to the host memory, via the system bus with its speed varied as one major parameter. The arrival of a non-train packet, or a train of packets, triggers only one receive interrupt to the on-board processor.



8.1: Simulated Token Ring



8.2: Simulated Token Ring Node Structure

In our simulation, bulk data transfer time was defined to be from the point when the transfer request arrived from a user at the sender host until an ACK was received by the sender host from the receiving user at the receiver host for the all the data packets. Urgent data transfer time was similarly defined. A simplified data transport protocol was simulated for both the train and non-train models. This transport protocol was first invoked when a bulk data request arrives at the host from a user. The transport protocol then passed a bulk data transfer request to its network controller. The network controller then transmitted a control packet (as a urgent packet) to the user at the receiver host, to set up a connection for the data transfer. At the receiver, if buffer memory space was available, the request would be granted

and a positive acknowledgement would be sent back to the sender (we always assume enough host memory for bulk data transfers). If no buffer space was available, such a positive acknowledgement would be delayed until some buffer space became available. Each time a DMA operation to transfer data to the host from the buffer memory was finished, or data packets were transmitted and buffer space vacated, the network controller processor would check on available buffer memory. At the sender's side, DMA transfer of data from the host into the buffer memory would not be initiated until a positive ACK was received from the receiver and a connection was set up between the users at the sender and the receiver hosts.

During the normal data transfer phase, the size of a packet train to be transmitted was limited to the number of prepared train packets at the moment when the token was seized. Packetization at the network controller could proceed in parallel with the DMA transfer of data from the host into the buffer memory. For the train model, protocol processing was done at the receiver host for a whole received packet train, and at the sender host for a large block of data for which enough buffer space was available in the buffer memory. For each packet train reception, interrupt handling was charged for a fixed amount of time, plus the time to receive (medium time) the packets in the train. This resembled the operation of the network controller processor that would keep processing each train packet while remaining in the train reception interrupt handler. For the non-train model, protocol processing at the host was performed on a per packet basis for both the sender and

receiver. Each non-train packet reception would incur one interrupt and was charged a fixed amount of interrupt handling time. Per packet interrupt handling time was the same for all single packets, both urgent and non-urgent control/data packets.

For urgent small packets, no DMA transfer was simulated to move data from/to the host. Instead, packet data was copied by the network controller. Urgent packets would always be inserted at the front of the transmission packet queue and non urgent packets were appended to the end of the queue. Network controller's buffer space was allocated on a first come first served basis. DMA transfer priority was given to received data to be transferred out of the buffer memory over data to be transferred into the buffer memory.

Corrupted data and packet loss due to broken links or malfunctioning hardware were not simulated. For the non-train model, packets rejected due to unavailability of buffers would automatically be retransmitted. For the train model, an interrupt would be generated to the on-board processor at both the sender and receiver when a train packet was rejected. The corresponding interrupt handling routine would then inform the transport protocol of this event. The sender would stop train packet transmission and wait for a control packet from the receiver to tell it to resume sending. The receiver would do so when some buffer space became available. Again, the network controller would check on available buffer memory space whenever a DMA data transfer or packet transmission was finished. We chose this simple flow control method instead of more complicated ones, in order to keep our

simulation results simple and straightforward for interpretation.

In our first simulations, we found out that if a "resume" packet was sent out as soon as some buffer space became available, a receiver's buffer would soon be congested again. Later in our simulation, sending of a "resume" packet was delayed until a number of packet buffers became available. This number was based on the ratio of medium to DMA speed. This simple delay rule effectively reduced the frequency of buffer congestion, but also occasionally kept senders waiting too long. This negative effect shows up in the results as the train model performing slightly worse than the non-train model for certain DMA speeds and packet sizes.

8.2. Simulation Parameters

Two types of traffic patterns were generated for each combination of parameters. The first was the single sender single receiver pattern, in which each bulk data sender randomly chose one of the other 10 nodes as the bulk data receiver. The second one was a multiple sender single receiver pattern in which all 10 nodes were sending bulk data to one single receiver. This second traffic pattern resembled the traffic characteristics among a server machine and a community of client machines commonly seen on local area networks.

The major parameter values are the following.

(1) Host bus DMA speed: 10, 20, 40, 80, 160 and 320 Mbps

(2) Bulk data packet size: 2, 4, 8, 16, 32, 64 and 128 Kbytes

(3) Protocol processing time: 1.5 and 6.0 ms

(4) Interrupt handling time: 0.4 ms

(5) Urgent packet size: 100 bytes

(6) Exponential distribution for bulk data and urgent packet request inter-arrival time

(7) Mean urgent packet transfer inter-arrival per node: 1 second and 100 ms.

Inter-arrival mean for bulk data transfer requests per node is varied to simulate different traffic workloads. Values used are 5 and 10 seconds.

Results from limited simulations using 100 ms as the inter-arrival mean for urgent data were essentially the same as the results with the 1 second mean, and hence are not reported here.

For most of the results reported below, bulk data transfer size was chosen to be 1 megabyte each. However, in order to better reveal the effect of different packet sizes on bulk data transfer time, we also used smaller bulk data transfer sizes ranging from 64 Kbytes to 256 Kbytes for each transfer.

A node is both a sender and receiver of bulk data and urgent data transfers. Each node had a two megabytes buffer memory. This large buffer size was chosen in order to avoid buffer overflow problems so that we could interpret our simulation results without being concerned too much with buffer overrun problems. However,

even with such a large buffer memory, due to the ability of the packet train mechanism to send a large amount of data at high rates, we still had buffer overrun problems in some cases as we will discuss later. Nevertheless, such problems were kept to the minimum.

We first validated the simulation programs by coding a model to imitate the structure of the Proteon interface and running this model with the parameters set to the measured values. Performance results from running this model are very close to the measurement results.

We obtained high confidence level for the simulation results by running simulations long enough until no difference in results would show up. Results reported in the rest of the chapter would not change by running their simulation any longer.

8.3. Simulation Results

8.3.1. Bulk Data Transfer Time Comparison

Figures 8.3 through Figure 8.7 compare train and non-train bulk data transfer times for different workloads and parameter values. Figure 8.8 compares train model bulk data transfer times for the two processing time values. The packet size used for these simulation runs is two kilobytes, the maximum allowed by a typical token ring network. Figures 8.3, 8.4 and 8.5 are for the single sender single receiver traffic pattern and Figures 8.6, 8.7 and 8.8 are for the multiple sender single receiver

pattern. Workload parameter values are noted on each graph. As can be seen from these graphs, for most system parameters and traffic workloads, the train model performs better than the non-train model.

(1) Figures 8.3 to 8.6 show that as host bus DMA speed increases, the difference in bulk data transfer time between the two models also increases. When host bus DMA speed is equal to or higher than the medium transmission speed, bulk data transfer time for the train model is at least one order of magnitude lower than that for the non-train model. With high DMA speeds, a sender can accumulate relatively large amounts of data while data previously DMA transferred into the buffer are processed and transmitted. This enables the sending of large packet trains which leads to large savings in protocol processing time at the receiver.

(2) For DMA speeds above the medium transmission speed, the latter becomes the limiting factor.

(3) For low host bus DMA speeds, improvement in bulk data transfer time for the train model is linear in proportion to the increase of DMA speed up to the medium transmission speed. The scale of improvement for the non-train model is less than linear because each packet incurs a fixed protocol processing overhead which begins to dominate as DMA speed increases. For example, Figure 8.5 shows that with a 6 millisecond overhead, there is virtually no improvement in bulk data transfer time when the host DMA

speed increases from 10 to 20 Mbps. Figure 8.6 shows that with a 1.5 millisecond overhead, no significant improvement can be seen from 20 to 40 Mbps. In the train model, DMA speed increases lead to increases in the size of packet trains which in turn leads to a reduction in protocol processing overhead.

(4) Figures 8.7 is for heavy workloads on a single node receiving from all other nodes (5 sec bulk data inter-arrival mean or 6 ms processing time). For the non-train model and for all DMA speeds, generated workloads saturate the capacity of the single receiver. Large numbers of transfer requests are queued up at the senders and the bulk data transfer times which include this queuing time become unbounded. Because of this, their corresponding curves are not shown in this graph. For the train model, we have only observed significant queuing of transfer requests at sender nodes for 10 Mbps DMA speed for the 5 second bulk data mean inter-arrival curve. This graph shows that for high workload conditions in which a non-train receiver node would be saturated, the train model can still perform well.

(5) Figure 8.8 shows the effect of software processing overhead on bulk data transfer time in the train model. For DMA speeds lower than the medium transmission speed, there is a significant difference in transfer time between the two curves for 1.5 and 6 ms processing times respectively. This is because a slower DMA speed does not enable the sending of large enough

packet trains to substantially reduce the processing overhead.

(6) Figures 8.3 and 8.6 show that at 10 Mbps DMA speed, the train model performs slightly worse than the non-train model. This slight increase in the train transfer time is caused by a sender being kept waiting for a "resume" packet from its receiver node. However, our results are conservative in the sense that we favored the non-train model by allowing it to simply keep retransmitting a rejected packet without being charged for the overhead incurred for each such retransmission. Note that for the 5 second mean transfer arrival interval, the train model performs better than the non-train model for all different DMA speeds.

8.3.2. Train Sizes

We have found that train sizes are directly affected by host bus DMA speeds. Table 8.1 shows average train sizes for different DMA speeds (both sender and receiver hosts) as received at the receiver network interface device. The table is for the single sender single receiver traffic pattern and the same workload parameters as in Figure 8.3.

DMA SPEED	TRAIN SIZE	
	AVERAGE	COEFF OF VAR
10Mbps	1.5	2.0
20Mbps	4.5	2.4
40Mbps	5.8	2.2
80Mbps	16	1.2
160Mbps	17	1.1
320Mbps	18.5	1.1

Table 8.1: Train Size vs. DMA Speeds

The increase in packet train size with DMA speed has a pattern similar to that of bulk data transfer time using packet trains. Train size increases substantially when the DMA speed is below the medium speed. Further increase is small when the DMA speed is above the medium speed. The DMA speed is critical to train size because it determines the amount of data accumulated at the network interface buffer at the time of train transmission. When the DMA speed is above the medium speed, arrival frequency and distribution of urgent traffic that affect train sizes become a more prominent factor than the DMA speed. In the above table, train sizes do not increase substantially for DMA speeds higher than 80 Mbps, the medium speed.

As indicated in Table 8.1, the coefficient of variation in train sizes are different for different size averages. A small average, i.e., at a low DMA speed, tends to have a large coefficient of variation. More size variation is likely for low DMA speed due to the fact that a large amount of data can be accumulated while a sender node is waiting for its turn to access the network medium. For high DMA speed, the

accumulated amount of train data is usually large, but its train size is often limited by urgent traffic interruptions. Because urgent traffic has the same statistical distribution in both cases, with high DMA speeds, train sizes tend to be large and less variant.

8.3.3. Urgent Packet Transfer Time

Results for all DMA speeds, processing times, workloads and bulk data packet sizes show that there is no significant difference between the train and non-train models with respect to maximum urgent data transfer time. In both models, an urgent packet can be inserted at the front of the queue of outgoing packets and both models used the same token ring priority transmission scheme. Hence, while significantly improving bulk data transfer performance, the train model supports prioritized transmission as well as the non-train model. A typical graph is given in Figure 8.9.

When an urgent packet is inserted at the front of the queue of outgoing packets, a bulk data packet from the same node may be in the middle of its transmission. When this bulk data packet circulates back to the sending node, the token is released, giving another node a chance to transmit at most one bulk data packet before an urgent packet is transmitted. Other nodes may be able to transmit urgent packets before the node in question can reclaim the token. Hence, the maximum waiting time to send an urgent packet can be twice the transmission time of a bulk data packet plus some number of urgent packet transmission times. The same is true for

the waiting time of the ACK packet from the urgent packet receiver back to its sender, making the total maximum time four times the transmission time of a bulk data packet plus the transmission time of a number of urgent packets.

The train model performs slightly better than the non-train model for urgent data transfers for large packet sizes. This is due to the fact that the train model generates less retransmitted packets and has lower processor utilization. In this case, an urgent packet is likely to be transmitted and processed more quickly, but these effects are not large.

8.3.4. Packet Size vs. Bulk Data Transfer Time

Figures 8.10 through 8.12 show the effect of packet size on bulk data transfer times for the two models with different combinations of workload and processing time. Only the multiple sender single receiver curves are given because the single sender single receiver results are very similar. All show curves for three DMA speeds, 20, 40 and 80 Mbps, and packet sizes varying from 4 to 128 Kbytes.

Earlier we saw that for the 10 Mbps DMA speed and 2 Kbyte packet size, the bulk data transfer time for the train model is slightly higher than that for the non-train model. We see this again in Figures 8.10 and 8.11 for 20 Mbps DMA speed and 4 Kbyte packets. The reason for this is the same as the one explained earlier.

We can make the following observations:

(1) The train model benefits with respect to performance by treating a train of packets as one large packet for protocol processing. Intuitively, use of large packets lessens this performance advantage for the train model, and the non-train model should perform equally well with sufficiently large packet sizes. As can be seen, the two models begin to perform equally well for packet sizes that are large enough to cause per packet processing to be overlapped in time with its DMA transfer. For example, for 1.5 ms processing time, the two schemes perform about the same for 16 Kbyte packets for DMA speeds above 80 Mbps, and for 8 Kbyte packets for DMA speed 40 Mbps.

(2) As expected, when the protocol processing time is increased 4 times to 6 ms, the packet size where the two schemes will perform equally well also increases by 4 times for most DMA speeds (Figure 8.12).

(3) Each DMA curve for the train model has an optimal packet size point with respect to the bulk data transfer time. For DMA speeds below the medium speed, this point occurs where the train and non-train models begin to have the same performance. As discussed earlier, slow DMA speeds lead to small size trains. Consequently, in this case, the train model performance advantage can not be fully exploited, leaving some potential for large packets to further improve throughput performance. For DMA speeds above the medium speed, large trains can be transmitted and received out of small packets. In this case, larger packet sizes do not improve throughput much.

Actually, small packets, 2 and 4 Kbytes, perform slightly better than large packets due to low end-to-end latency when buffering small packets at the interface buffer memory. This effect will be examined further in the next subsection. As is done in some systems, when the host bus speed is sufficiently high, packets could be directly transmitted from the host memory without buffering them at the interface. However, for the purpose of comparison under the same system structure and parameters, we have chosen to do the buffering at the interface device for all host bus speeds.

8.3.5. Effects of Large Packets

Looking at curves for DMA speeds over 80 Mbps in Figures 8.10 to 8.12, we can see that as packet sizes increase, bulk data transfer times also go up slightly. In order to reveal this more clearly, we have run a set of simulations with bulk data transfer sizes smaller than 1 megabyte. As we can see in Figure 8.13, when total transfer sizes are small, the increase of transfer time becomes more prominent. For the 1 megabyte transfer size, the increase in transfer time is about 10% when packet size is increased from 4 Kbytes to 128 Kbytes (Figures 8.10 and 8.12). However, for the same packet size increase, the transfer time more than doubled for the 128 Kbyte transfer size, and almost doubled for the 256 Kbyte transfer size. This increase in transfer time when packet size is increased can be explained by the problems pointed out in the previous chapters for large packets, namely, reduced concurrency (or overlapping) effects when large granularity is used for DMA and network

transmission operations. Consider for example that for the 128 Kbyte transfer size and 128Kbyte packet size, there would be no concurrency (or overlapping) effects. These adverse effects are specially severe at the beginning of a bulk data transfer. The smaller the total transfer size, the greater the effect on the total transfer time. This is one of the basic reasons why small packets should be favored over large packets, given that there are mechanisms, such as the packet train mechanism, to overcome problems of software processing overhead for small packets.

8.3.6. Summary

The train scheme improves bulk data transfer performance over the non-train scheme for a wide range of system configurations. The improvement is in the range of one order of magnitude when host bus DMA speeds are above the medium speed. For host bus speed lower than the medium speed, the improvement is linear with the increase of host bus speed. Performance improvement for the non-train model is much less significant in this case.

For small urgent data transfers, the train mechanism supports prioritized transmission as well as the traditional non-train scheme.

Using the train model, the optimal bulk data transfer throughput and small urgent data transfer delay performance occurs when host bus DMA speeds are above the medium transmission speed and packet sizes are no larger than 2 Kbytes. In certain cases (see 4.2.3), increasing packet size may improve bulk data transfer

throughput marginally. However, one must trade such marginal improvement against the problems incurred with large packets as described earlier.

Even when both models give comparable throughput performance in certain cases, the train model should still be preferred because this model reduces processing load on the processor performing protocol functions.

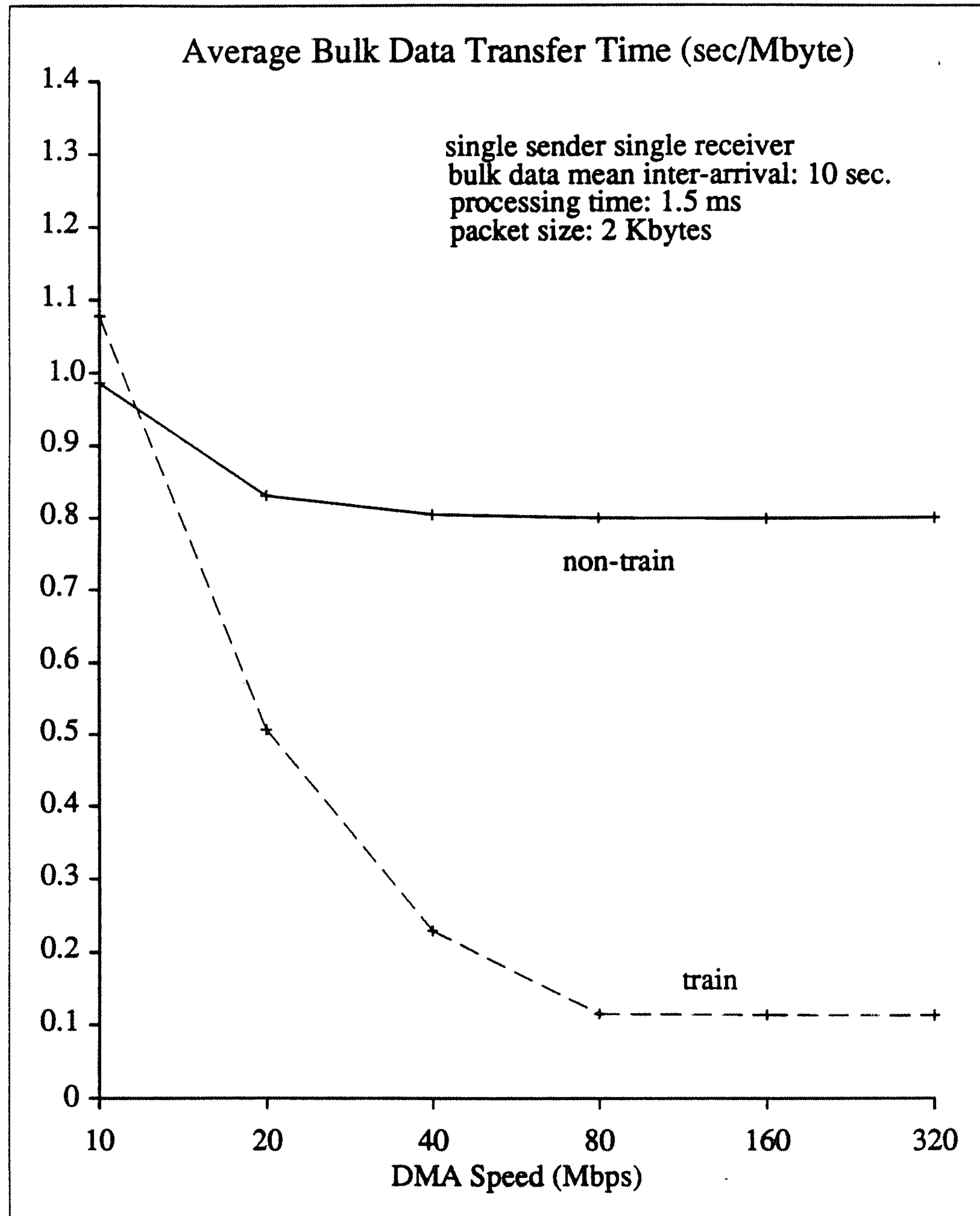


Figure 8.3

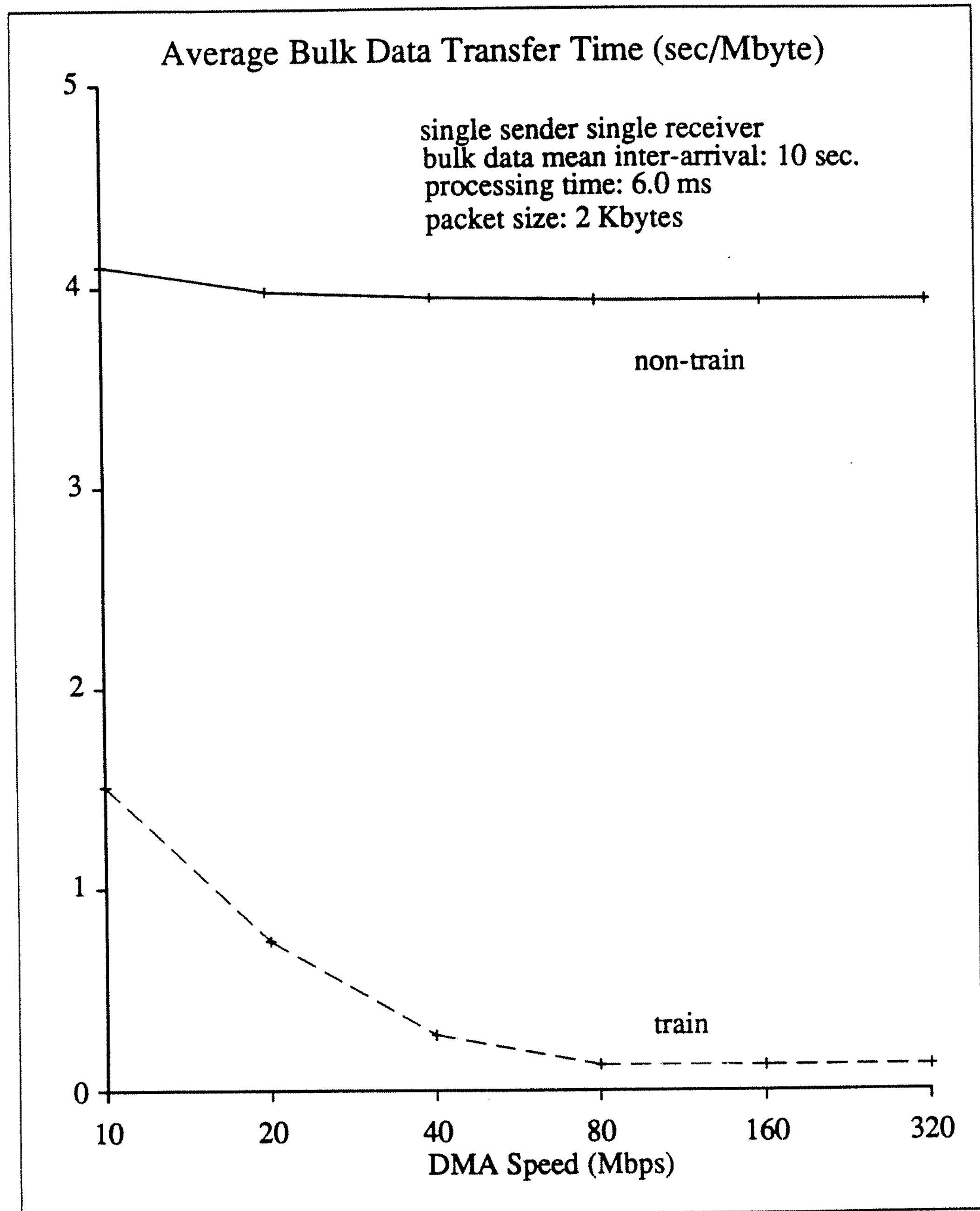


Figure 8.5

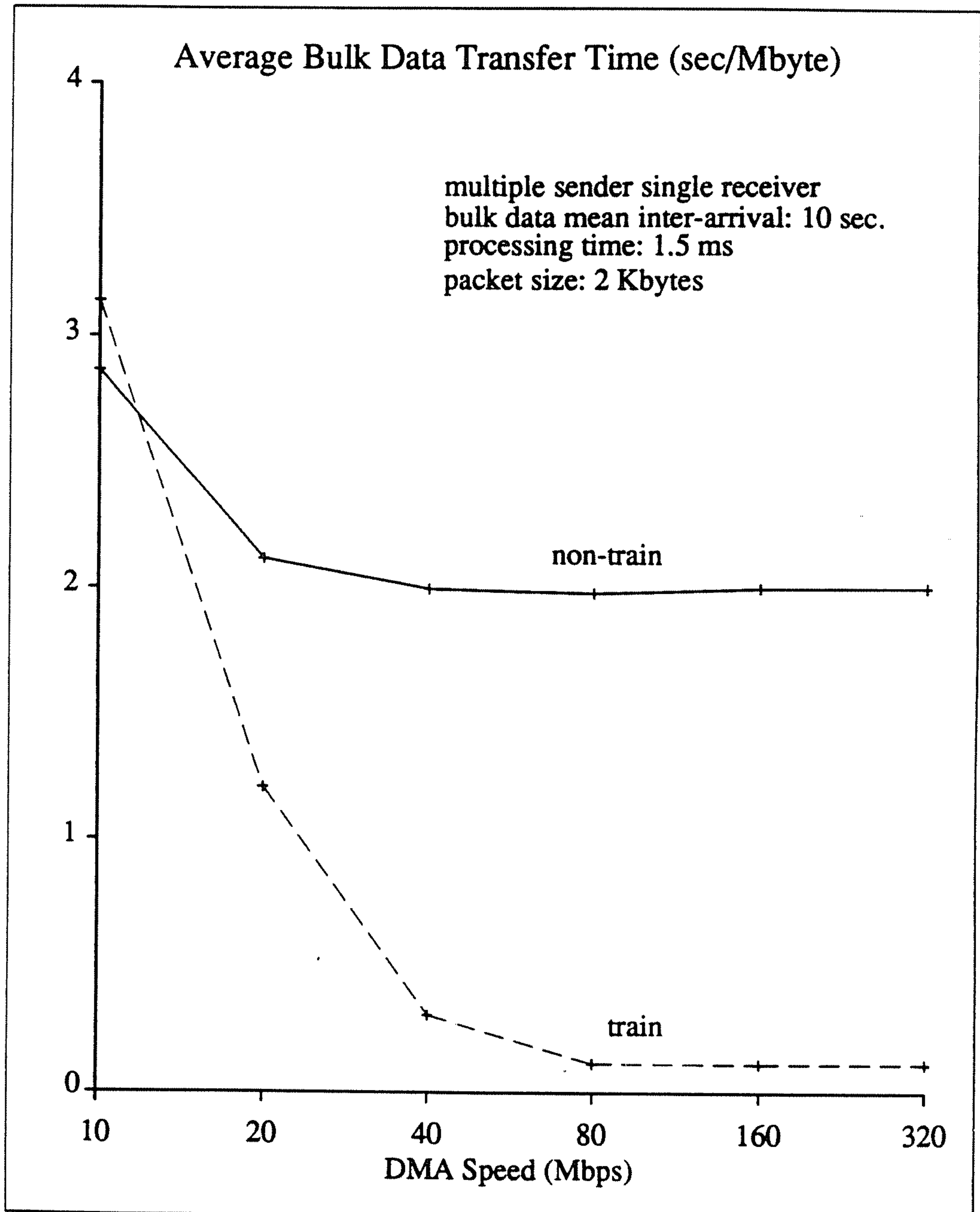


Figure 8.6

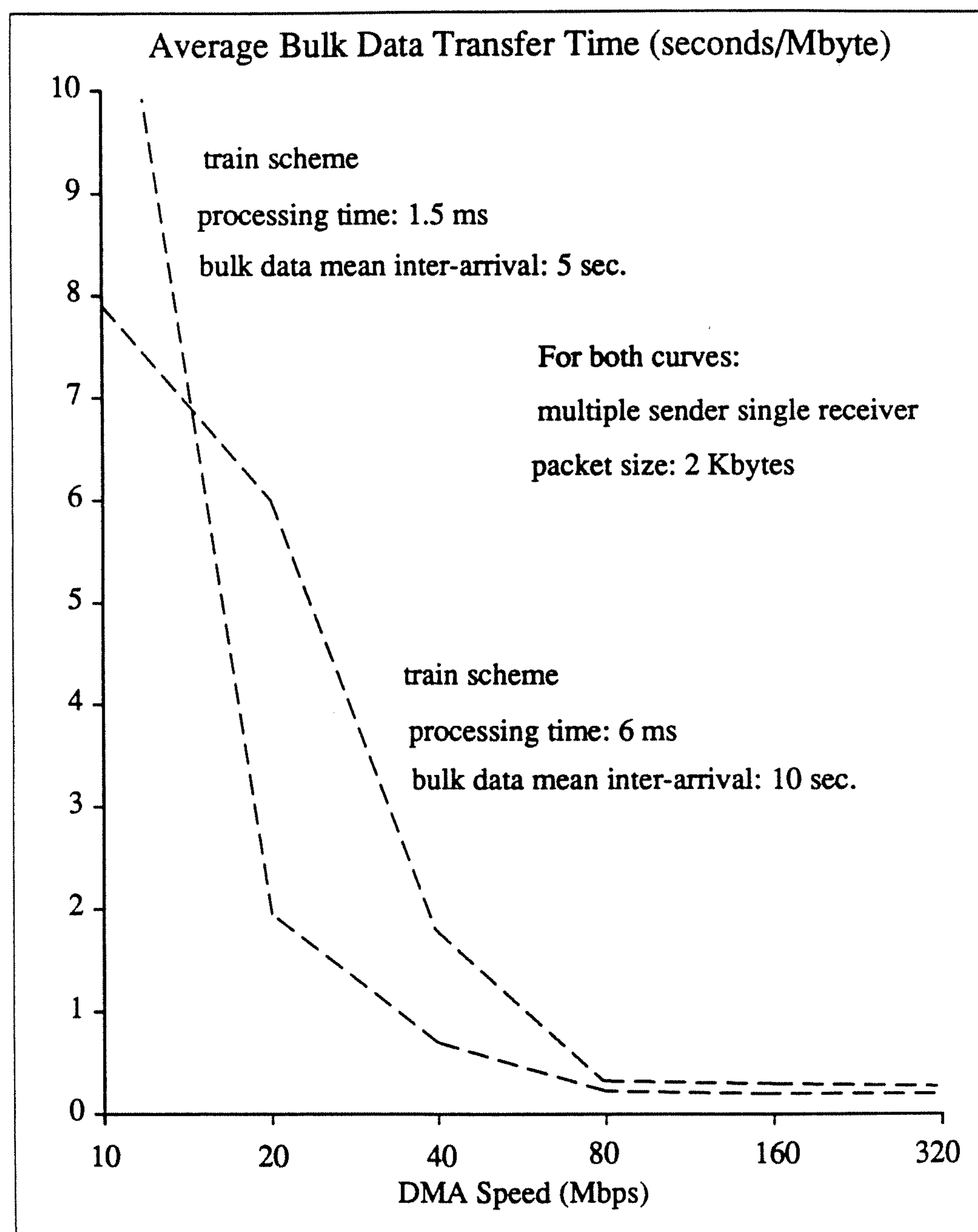


Figure 8.7

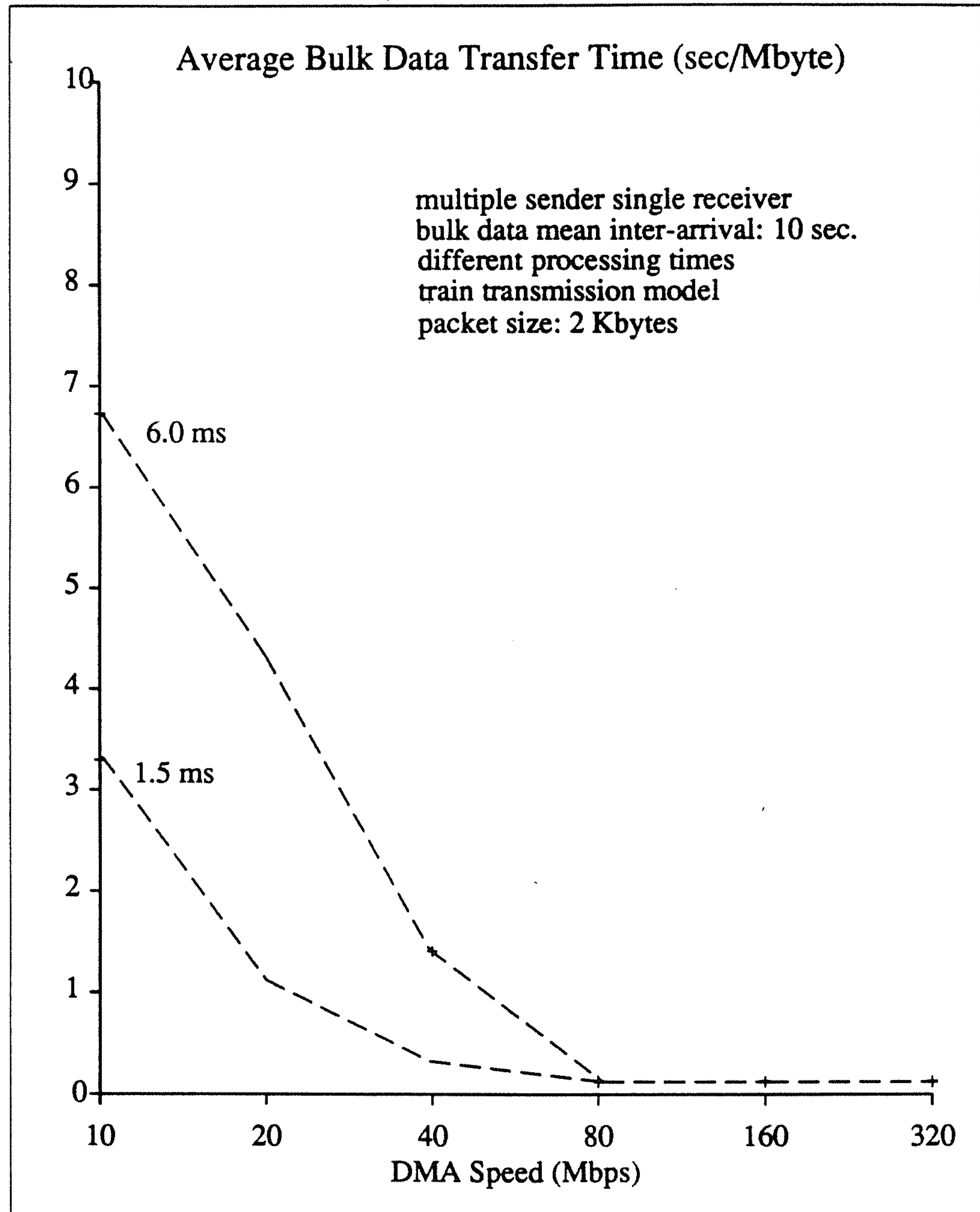


Figure 8.8

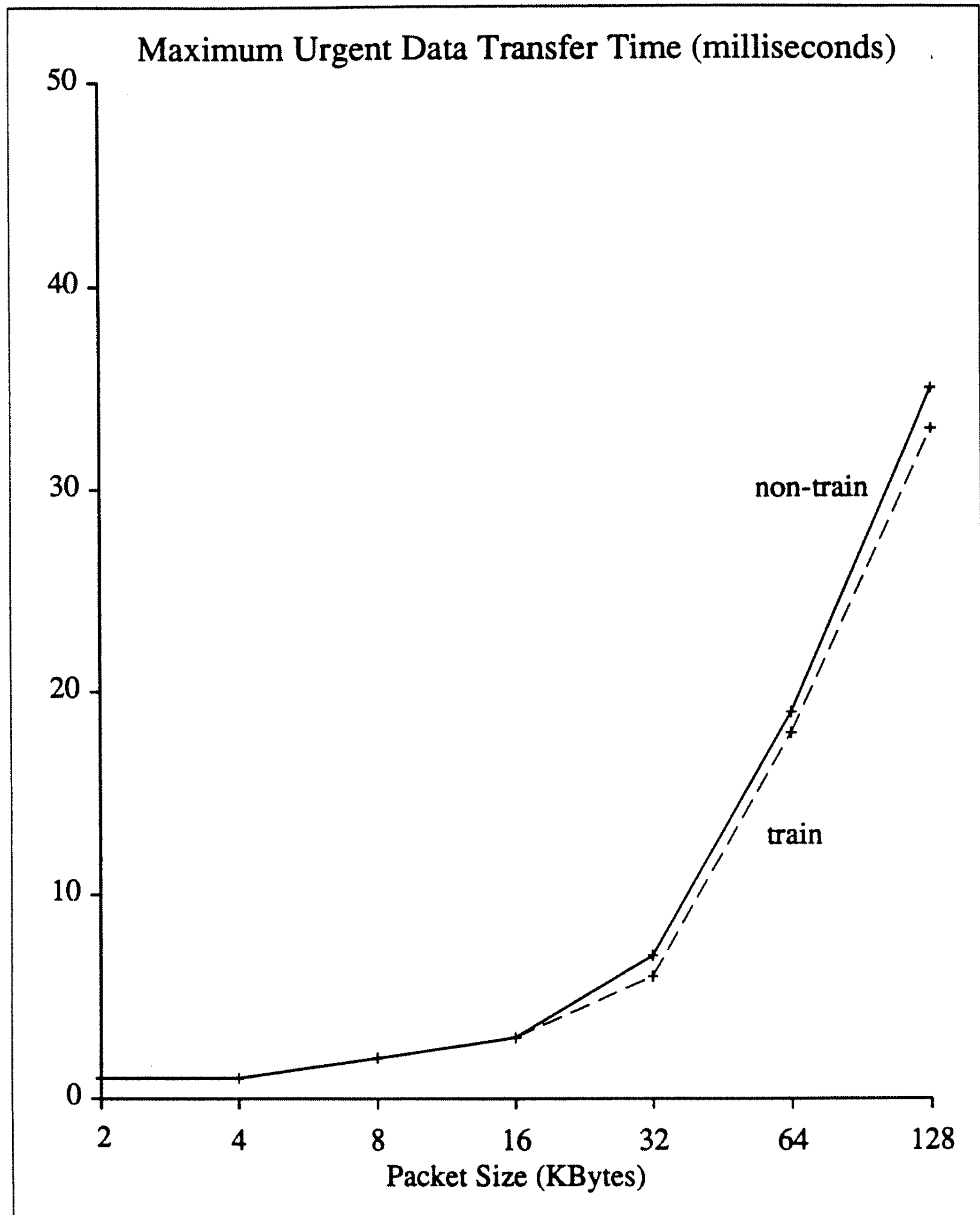


Figure 8.9

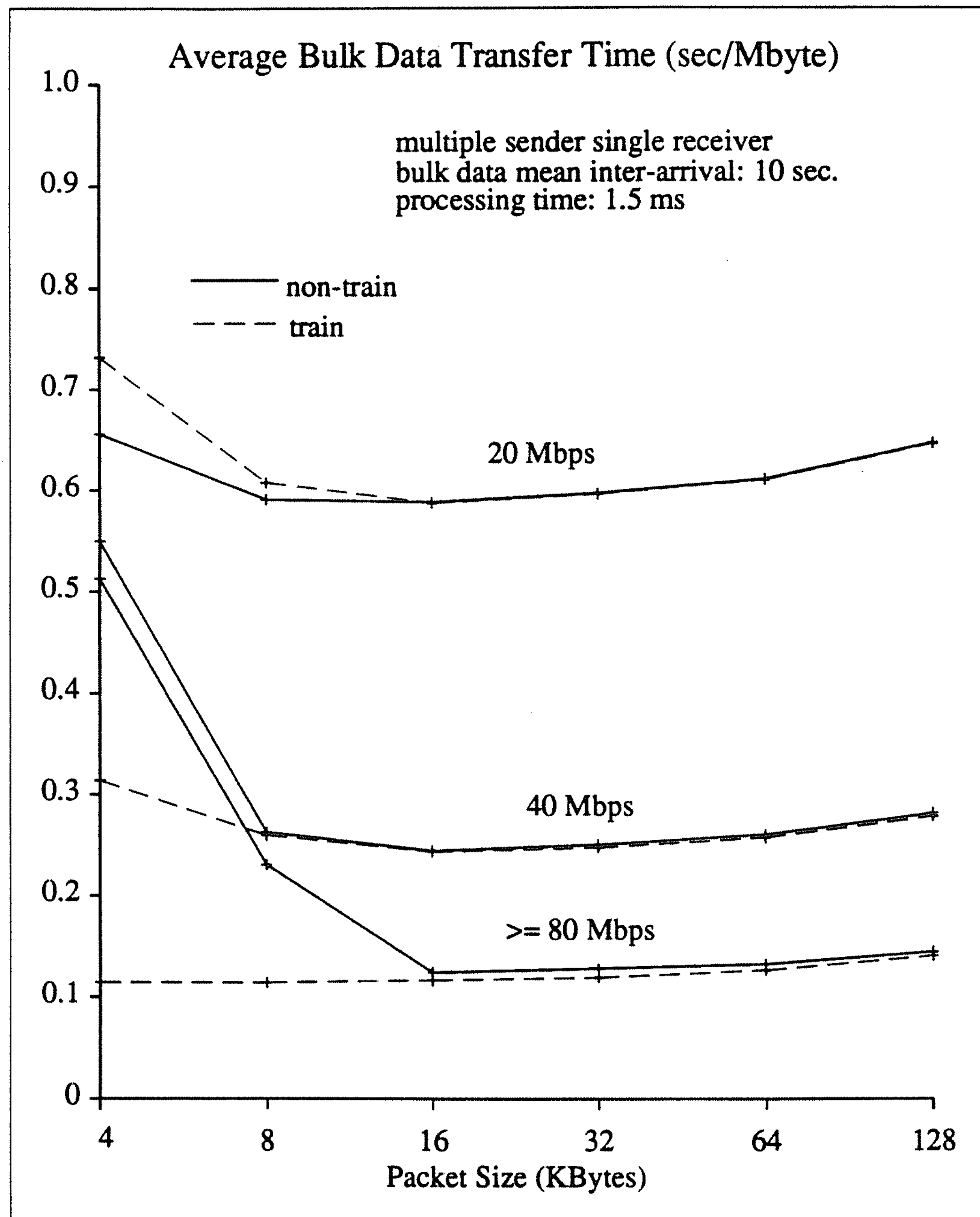


Figure 8.10

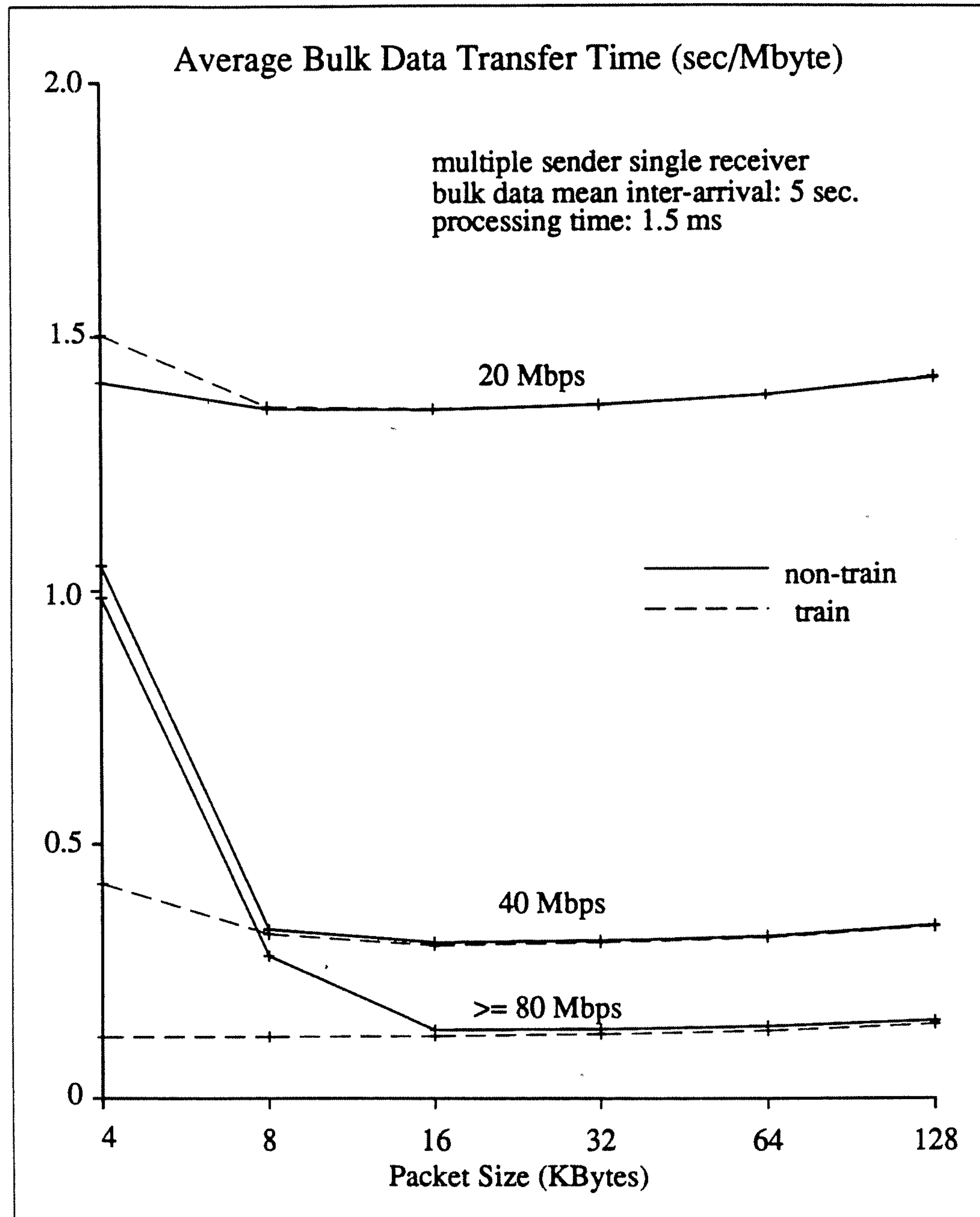


Figure 8.11

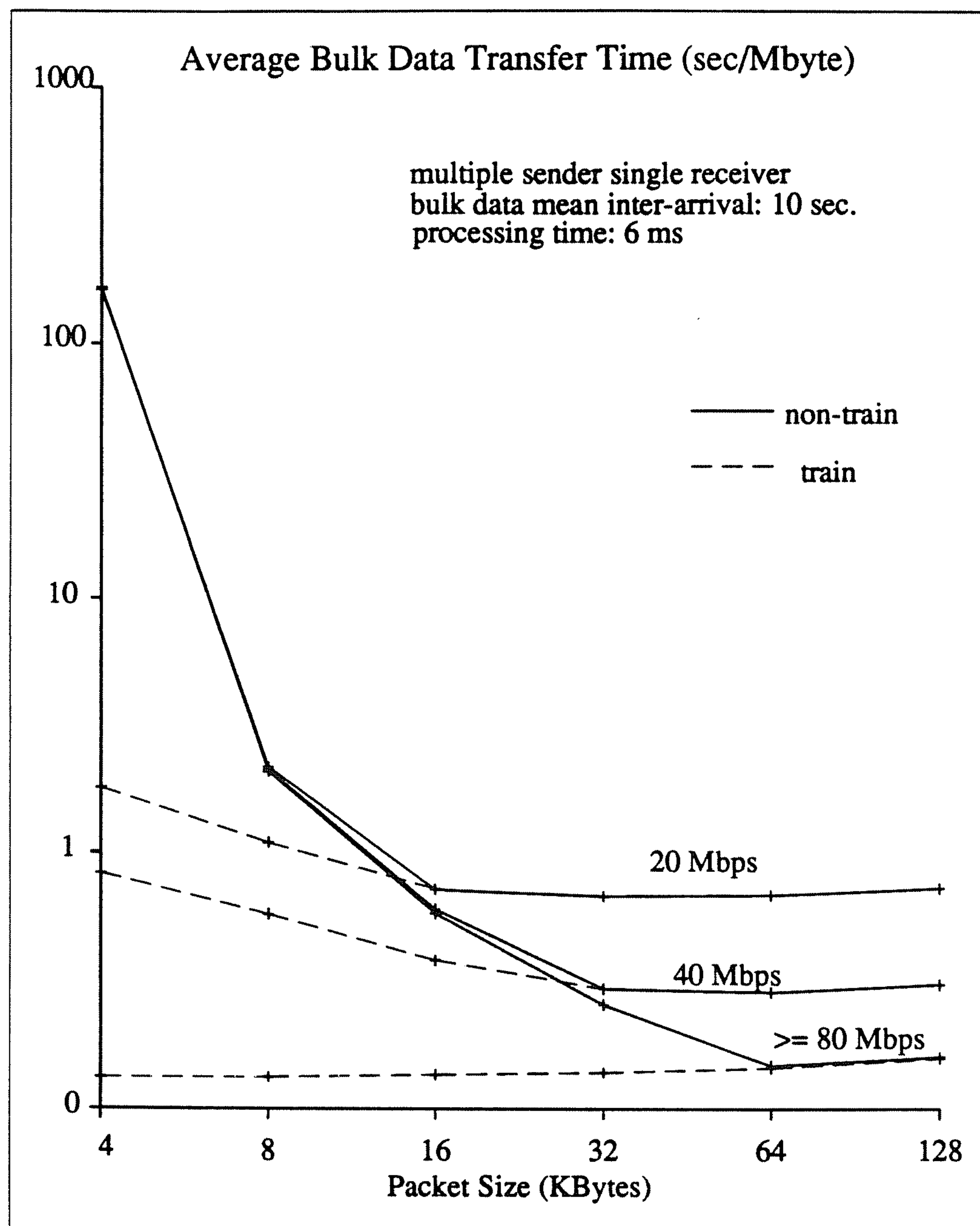


Figure 8.12

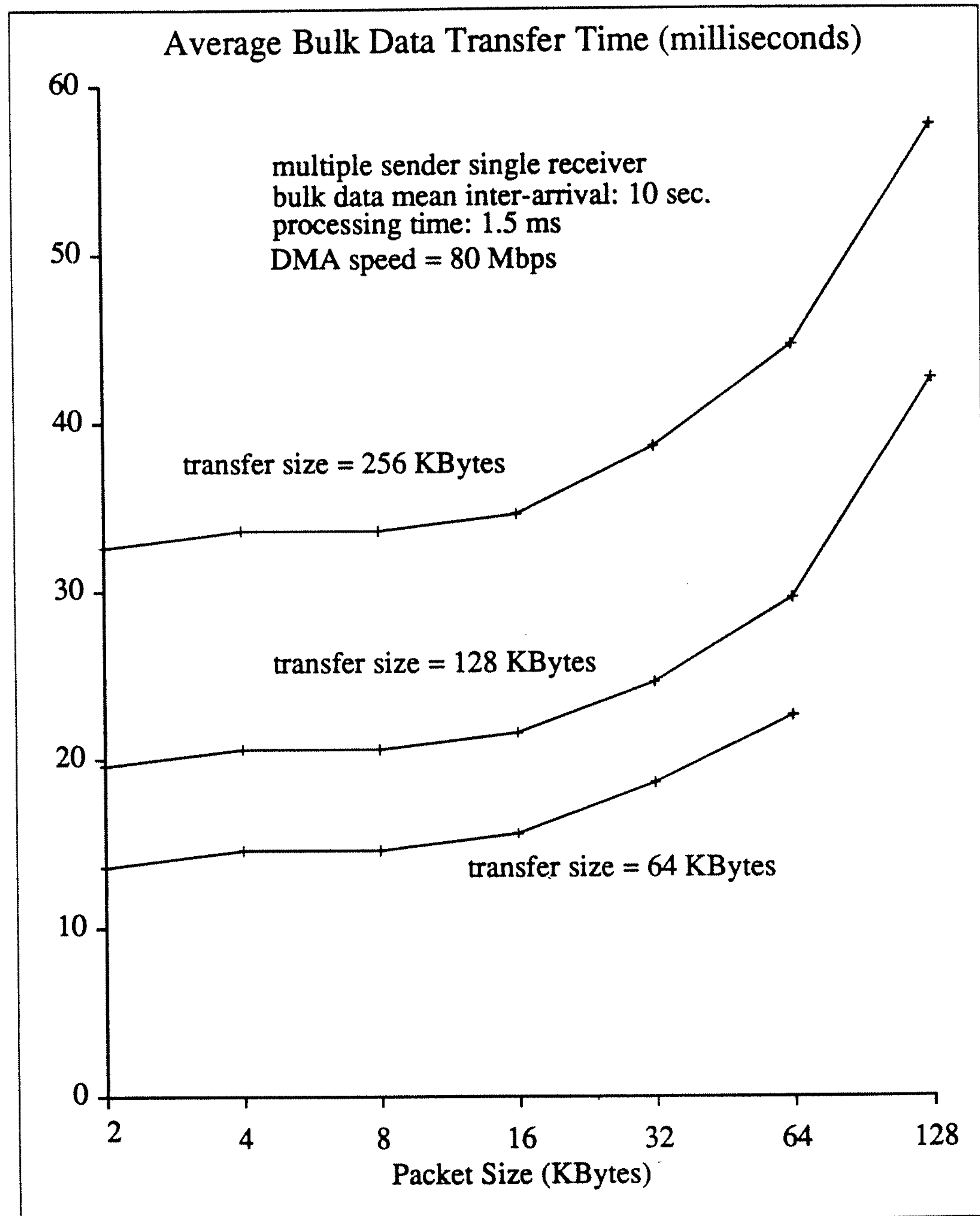


Figure 8.13

Chapter 9

Packet Train Approach in WAN Environments

The previous chapters have concentrated on the packet train approach in a local area network environment, in particular, a token ring network environment. In this chapter, we discuss how the packet train approach can be extended to a wide area network environment. In particular, we discuss WAN traffic characteristics, packet train performance advantages for WAN's and some WAN issues and problems requiring changes to the packet train approach.

9.1. WAN Traffic Characteristics

At present, transmission bandwidths for long haul networks range from 56 Kbps to 1.5 Mbps, much lower than those available for local area networks. When transmission bandwidths is the prominent performance constraining factor, the packet train approach can only benefit data transfer performance to a limited degree.

However, the current limitation on WAN transmission bandwidth will soon disappear. In the not too distant future, fiber optic transmission technology will offer long haul network link speeds in the range of gigabits per second. Although total traffic volume on future WAN's will increase, transmission bandwidth will no longer be a performance limiting factor as it is now. Along with this increase in bandwidth, computer processing power will also increase rapidly. Together, these two factors

will allow hosts and gateways to transmit packets at much higher rates than they can today. As computer memory is becoming large in size and moreover, because of price trends for computer memory, it will be possible to provide ample buffer space for packets will be provided at intermediate switching node and gateways. As a result, performance problems resulting from traffic congestion and buffer overrun will lessen even in the presence of a certain degree of packet burstiness.

At the same time, user communication applications will demand increasingly high performance. Larger and faster user to user data deliveries will become crucially important to the usefulness of a broad range of communication applications. Future communication protocols will be designed to take advantage of these technological advances, with features for sending large numbers of data packets at higher rates. Some protocols are already designed with features of sending large numbers of packets as bursts. For example, the VMTP protocol[6] has a feature of sending a packet group (of 8 packets) at a time. The NETBLT[13] protocol is designed to blast transfer up to 50 data packets within some short time period.

For both local area networks and wide area networks, traffic has been characterized by packet bursts. This will likely remain so for future WAN's, both because of the nature of computer communications and advances in networking and computer technology. It can be expected that traffic burstiness will intensify in terms of larger numbers of packets for bursts and closer time gaps between packets within bursts. In future WAN traffic environments, the idea of packet trains may

become one viable approach to improve data transfer performance.

9.2. Advantages of the Packet Trains

The main performance advantage of the packet trains, as discussed in the previous chapters, is to reduce packet software processing overhead at data sending and receiving end hosts. Because both LAN and WAN hosts use the same packet processing protocol and systems structure, it is easy to see that packet train performance advantage is applicable to data sending and receiving at end points in a WAN environment. In the following, we will mainly examine opportunities for performance improvement that the packet train approach may offer to intermediate switching nodes and gateways.

9.2.1. Avoidance of Packet Fragmentation and Reassembly

Moderate increases in packet sizes may be expected for networks with gigabits per second transmission bandwidths. However, transmission speed is not the only factor determining optimal maximum packet sizes. Switching techniques, computer memory structure, application needs and protocol architecture all play important roles.

Currently, WAN environments consist of interconnected networks with different transmission and switching speeds. In the future, a WAN will undoubtedly be an interconnection of a large number of networks with vastly different characteristics. Different networks, for their own optimal operation, will have

different packet size limits, i.e., MTUs (Maximum Transmission Unit). When packets traverse through several different networks, large packets will have to be fragmented for networks with smaller MTUs. Fragmented packets need to be reassembled either at destination hosts or at subsequent networks allowing larger MTUs. Packet fragmentation and reassembly causes high processing overhead and other performance problems[23].

To avoid packet fragmentation and reassembly, it has been suggested that the minimum of the MTUs of all traversed networks should be used for packets that will traverse different interconnected networks. However, this method may not always be practical under all circumstances. It also sacrifices potential performance benefits afforded by large packets that some traversed networks can accommodate.

The packet train idea seems to provide a solution towards this problem. When a large sequence of packets is sent in sizes that can be accommodated by all traversed networks, there will be no need for packet fragmentation and reassembly. At networks where larger sizes are preferred, packet trains can provide effects of large packets with respect to the following operations.

(1) Operations internal to a gateway, such as for buffer allocation, queue manipulation, and various resource scheduling. At present, most gateway machines use low end mini-computers. Software processing overhead is at least one of the performance limiting factors on packet switching throughput at such gateways. A train of packets, when dealt with as one data unit at a

time, serves as a large packet for the above operations. Repetitive invocations of these operations can be dramatically reduced. Therefore, packet train size could have a direct effect on gateway throughput performance.

(2) Packet transmission. On networks with high bandwidth links but long propagation delay, use of large transmission frames to send data is often desired for efficient bandwidth utilization. Each frame has some fixed overhead associated with it. Small frames would essentially magnify the overhead, reducing effective bandwidth. With packet trains, a large transmission frame can be used by containing a train of packets instead of a single packet. When doing this at the sender's side of such a transmission link, there is no need to alter the content of the train packet headers. Train packets can be simply concatenated in sequence, forming a special large packet format without any extra software overhead. Such packet "reassembly" is almost free in cost. At the receiver's side, if separation of a train of packets is desired, i.e., smaller MTUs have to be used, simple software processing is needed to recognize the original train packet boundaries, with the help of packet length information available in train packet headers. Such processing is far cheaper in processing overhead than packet fragmentation.

(3) Packet send/receive event interrupts. A train of packets can be exploited to reduce the number of interrupts from one interrupt per packet to one interrupt per packet train. To do this, hardware assistance at the link interface level is needed. Similar hardware features for a ring network interface device MAC component were described in Chapter 5. Such hardware features for a long haul network link interface should be inexpensive to implement.

However, packet trains may not achieve the full advantage of large packets for networks allowing large MTUs. As indicated in the above, "separation" of train packets will incur some overhead, although much lower than packet fragmentation. In addition, the packet train sizes may not map to large transmission frames in a convenient fashion. Nevertheless, considering the total processing overhead accumulated over a WAN path, the flexibility of packet trains has the potential to offer tremendous performance benefits when used to avoid packet fragmentation and reassembly.

9.2.2. Cut-through switching

Cut-through switching refers to switching a packet without completely buffering the packet. If several stages are involved in switching a packet, cut-through switching would allow a packet to proceed from one stage to the next once the header has been decoded, without being fully buffered. Cut-through switching offers significant performance gains. This technique is feasible mainly in hardware

based switching techniques[2]. One prerequisite for cut-through switching is that knowledge of which forward link to use can be readily obtained from packet headers or simple lookup procedures within the real time of packet reception.

On a typical "store and forward" WAN gateway, where software is responsible for much of the switching functionality, such cut-through switching is nearly impossible. Packets have to be received in full and buffered in memory while some routing function and possibly other protocol functions are being executed.

The property that a packet train is a sequence of data packets with essentially identical headers can be exploited to achieve an effect similar to cut-through switching on a WAN gateway. A feasible scheme would work as follows. When the first packet in a train arrives, an interrupt is generated to this effect, and routing and other needed software routines are invoked for this single packet as usual. Obviously, the first and perhaps some subsequent train packets must be buffered while the routing function is being executed. As soon as route and forward link information is returned by the invoked software functions, the previously received train packets can be scheduled for transmission on the identified link. The same route and link information is used for all subsequent train packets, which can be immediately queued for transmission. This way, transmission of previously received train packets proceeds while the rest of the packet train is still being received. The effect is similar to cut-through switching for packet trains whose reception time is longer than software function execution time.

This scheme requires some hardware support. A gateway must be able to recognize packet trains at the link interface level. A gateway processor must be able to set up a data move channel so that train packets can be directly received into the appropriate memory area used for packet transmission to an outgoing link. As discussed in 2.2 and 2.6, such hardware support is not difficult to implement. In addition, some limit on the size of trains allowed to be cut-through switched at a time must be enforced to prevent a large packet train from hogging an outgoing link. This limit should be a parameter, adjusted according to link speed and traffic statistics at a gateway.

9.3. Issues and Difficulties with the Packet Train Approach

In a local area network environment, where both sender and receiver hosts share the same transmission medium, it is relatively easy to maintain a packet train as a sequence of consecutive back to back packets, especially if the MAC protocol is based on an access regulation scheme like the token ring. In a WAN environment, however, this is difficult to achieve and often incurs undesirable delay effects. For instance, the following two cases present difficulties with packet trains.

First, consider that two adjacent networks interconnected by a gateway have different transmission speeds. To maintain a sequence of back to back train packets, arriving on the slow link and exiting on the fast link, the gateway must buffer some number of train packets before transmitting the received train packets. This extra buffering delay can result in significant additional packet transfer delay when a

packet train traverses several gateways with similar speed mismatches.

Second, consider the case of packets from different links being routed onto the same forward link. When packet trains arrive at the same time from different links and are routed on a first come first served per packet basis onto the same link, packets from different trains may be interleaved on the outgoing link, destroying train packet sequences. If buffering is used to delay all other packets while one train is being sent on a link, large delays may result.

The philosophy of the packet train approach is to take advantage of network traffic characteristics to benefit data transfer performance. Rigidly forcing packet trains in an environment with diversity and unpredictability would be contrary to such a philosophy. In order to benefit from the packet train approach in a WAN environment, there should be flexibility in how train packets are treated at gateways having different configurations and characteristics. Some guidelines can be followed to minimize buffering delay and maintain sequences of consecutive train packets as much as possible.

Buffering packet trains should be avoided in the following situations.

(1) When a gateway forwards train packets from a fast incoming link to a slow outgoing link.

(2) When a gateway forwards train packets from a slow link to a fast link and traffic load on the outgoing link is light.

Buffering on a train granularity basis should be used in the following situations.

(1) When traffic load is heavy and there are packets queued for transmission on an outgoing link.

(2) When traffic from several incoming links are forwarded to one outgoing link. If some scheduling scheme like cut-through switching is used, a reasonable size limit for each traffic flow should be enforced so that the outgoing link can not be monopolized and buffering delay for any traffic is kept small.

As packets in a train traverse through a WAN, they may be separated by large time gaps, interleaved with other packets, or regrouped into a consecutive sequence due to buffering, depending on configuration and traffic load at various gateways. Such dynamics will require extensions to the previously described mechanisms so that they still benefit data transfer performance.

Consider switching performance. Instead of using strictly cut-through switching, route and forward link information for multiple data packet flows can be cached in an associative memory to facilitate route look up. For each newly arrived packet, this cache is consulted first. If a match is found for the packet, route and link information is obtained from this cache without invoking normal expensive lookup procedures. If not found, normal route lookup procedure will be invoked and route and link information is entered into the cache. With train packets arriving in small clusters and occasionally long sequences, this scheme would achieve reasonable savings of gateway processing overhead. Furthermore, this scheme does not exclude

the use of cut-through switching whenever possible.

To avoid packet fragmentation and reassembly, small train packets should be used. At gateways where large packets are preferred, buffering may be required just to accumulate train packets to be scheduled as one large data unit. If processing load or traffic load on the outgoing link is heavy, such buffering delay is unavoidable anyway. In this case, a large packet effect is achieved without any extra cost. If traffic load or processing load is light, buffering should be limited to the extent that the link or the processor will be not be left idle.

Obviously, performance gains by the above extensions may be less than if consecutive back to back train packet sequences could be maintained. For a WAN environment, the relation between performance gains and the overhead (e.g., buffering delay) of maintaining packet trains will be complex to study. The complication of this relation is partially due to the large number of possible combinations of different network configurations and traffic load distributions, and partially due to the dynamics of network traffic, packet train sizes, and route paths taken by packets. Also, more potential performance optimization opportunities remain to be identified. These will be the focus of our future research with the packet train approach.

Chapter 10

Conclusion

10.1. Summary

The research described in this dissertation is concerned with improving throughput performance for bulk data transfers on local area networks with high transmission bandwidths. The goal is to achieve user level data transfer throughput close to what the network raw bandwidth can accommodate, under the constraint that fast response time for small urgent data transfers must be guaranteed. Towards this goal, we have described and evaluated a new approach, called the packet train Model, to overcome the bottleneck effects of packet software processing overhead on data transfer performance. This packet train mechanism is designed mainly to minimize the number of repetitive invocations of protocol functions and expensive host system support operations.

After presenting the "nuts and bolts" of the packet train mechanism in the thesis, we would like to offer two practical views of the packet train model as an informal summary of the details.

For the first view, the packet train mechanism can be simply considered as a practical way to send "large data packets" without forcing a large amount of data into one rigid data "container". Rather, data is carried in a sequence of flexible small

containers, that can be "chained" together to form a train, or to be separated as independent cars whenever a need for this occurs. The effects that the mechanism attempts to achieve are close to those of using large data packets on software processing, but the flexibility of small packets is retained. The second way to look at the packet train model is that it is merely a strategy to bring data transfer sessions down to the level of the network interface front-end, and the mechanism attempts to minimize host processing overhead by completing a bulk data transfer with a number of such "transfer sessions". Each such session is a "mini-blast" of data (see blast protocol [43]), with data sent as an interruptible sequence of back to back packets and processed as one logical unit at the sender and receiver hosts.

As has been demonstrated by our simulation study, the packet train mechanism can lead to substantial performance gains for bulk data transfers under a wide range of different traffic load and host configuration conditions. The performance gains are directly related to the per packet software processing overhead. In some cases, improvement of bulk data transfer throughput using the packet train mechanism can be as much as one order of magnitude over that using the non-train scheme. Because a packet train can be interrupted any time during its transmission on the network medium, the packet train scheme guarantees fast network access and hence small transfer delay for small urgent data under all traffic load circumstances.

A limitation of the packet train mechanism is its reduced performance benefit for bulk data transfer throughput under two circumstances: small urgent data transfer

workload is extremely high on the network causing frequent packet train interruption; and when the host bus (either the sender or the receiver) data transfer speed is much lower than the network medium speed. In both cases, the size of packet trains are small, rendering the packet train mechanism no better than the usual non-train mechanism.

10.2. Contributions

We view our research for this dissertation a contribution to computer network systems design in general, and to the subarea of bulk data transfer throughput performance in particular. Our research has a strong engineering-oriented flavor, but certain basic research issues were examined in a methodological way.

(1) Our method of reducing software processing overhead is new. Earlier research in this aspect concentrated mainly on simplification of communication protocols (or algorithms) or use of more advanced processor architectures to speed up protocol execution. Although the header prediction algorithm (see 2.6) is similar in approach to ours and its work parallels ours in time, it is limited only to a small section of the packet execution path and completely ignores host operating system support overhead.

(2) A new strategy to offload certain transport layer protocol functions to the network front-end is part of the packet train model. In contrast with the traditional way of offloading a complete protocol layer, this strategy has the

advantage of speeding up protocol execution without overloading a front-end processor. At present, when microprocessors are inexpensive in price but limited in power, our strategy is attractive because processing burden on the host CPU can be lessened and an interface device can still be managed by the host as a simple data send/receive I/O channel.

(3) The proposed extensions to the MAC two (IEEE802.5 and FDDI) token ring MAC protocols analyzed in the thesis are simple for implementation, easy to model and study, and most important, do not affect other basic aspects of these MAC protocols. These extensions allow a sequence of back to back data packets to be treated as a logical unit (a packet train) by sender and receiver hosts. Earlier work on MAC protocols for bulk data transfer performance, such as the Hyperchannel (see 2.1), achieved similar effects but were not applicable to a wide range of different applications.

(4) The proposed network controller allows packet transmission/reception event status to be registered in the packet buffer memory (in packet descriptors). This greatly reduces packet loss when interrupt handlers fall behind packet arrival rate, without resort to expensive multiple hardware interrupt logic. As one of our research criteria, we strove to achieve maximum possible performance gains with minimal hardware support extensions.

10.3. Future Work

Two major directions for future work with the packet train approach are the following.

(1) Build a network controller prototype with our designed extensions. With actual hardware support for the packet train mechanism, more experimental work can be done to further evaluate its performance gains under more realistic network traffic workload conditions.

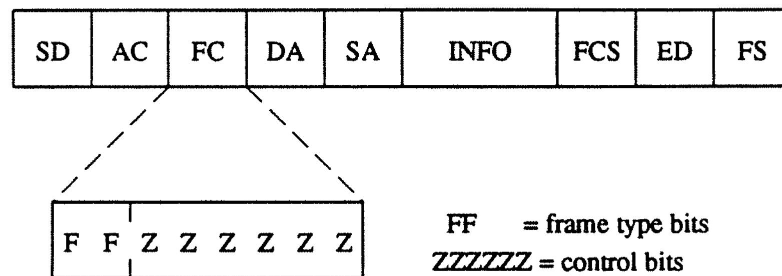
(2) In Chapter 9 we discussed various potential benefits for adapting the packet train concept to a WAN environment. We would like to study the extensions and modifications needed to do this and to perform more simulation studies to further understand the potentials and limitations for such an adaptation in WAN environments.

Appendix A

Train Packet Formats

Train Packet Frame Format for IEEE 802.5 Standard

The figure below shows IEEE 802.5 MAC frame format.



SD = Starting Delimiter (1 octet)

AC = Access Control (1 octet)

FC = Frame Control (1 octet)

DA = Destination Address (2 or 6 octets)

SA = Source Address (2 or 6 octets)

INFO = Information (0 or more octets)

FCS = Frame Check Sequence (4 octets)

ED = Ending Delimiter (1 octet)

FS = Frame Status (1 octet)

Figure A.1: IEEE 802.5 Standard Packet Frame Format

The frame format for a train packet is the same as shown in Figure A.1, with only

one difference from the regular non-train packet in the Frame Type bits in the FC field.

For the IEEE 802.5 standard, the Frame Type bits are defined as

00 = MAC frame (contains an MAC PDU).

01 = LLC frame (contains an LLC PDU).

1X = undefined format (reserved for future use)

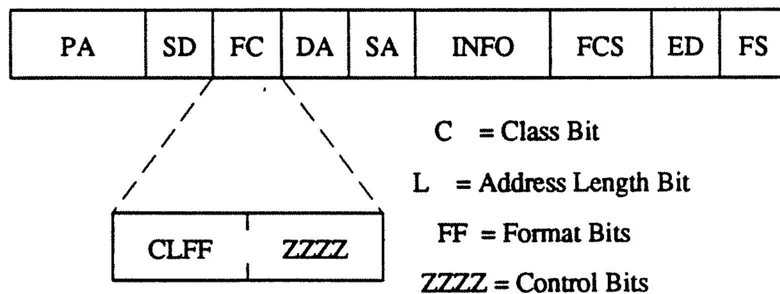
Using the reserved bit patterns for the Frame Type bits, our extension for a train packet is as defined below.

10 = train packet frame (contains an LLC PDU)

11 = undefined format (reserved for future use)

Train Packet Frame Format for FDDI Standard

Figure A.2 shows IEEE 802.5 MAC frame format.



PA = Preamble (16 or more symbols)

SD = Starting Delimiter (2 symbols)

FC = Frame Control (2 symbols)

DA = Destination Address (4 or 12 symbols)

SA = Source Address (4 or 12 symbols)

INFO = Information (0 or more symbol pairs)

FCS = Frame Check Sequence (8 symbols)

ED = Ending Delimiter (1 symbol)

FS = Frame Status (3 or more symbols)

Figure A.2: FDDI Standard Packet Frame Format

The frame format for train packets is the same as shown in Figure A.2, with only one extension that a bit pattern in the FC field that is normally reserved for implementer is now designated for train packets.

The FC bits are defined as below.

CLFF ZZZZ to ZZZZ

0X00 0000 Void frame

1000 0000 Nonrestricted Token

1100 0000 Restricted Token

0L00 0001 to 1111 Stations management frame

1L00 0001 to 1111 MAC frame

CL01 r000 to r111 LLC frame

CL10 r000 to r111 Reserved for implementer

CL11 rrrr Reserved for future standardization

Note: X is either 0 or 1 bit and r is reserved for future standardization and should be set to zero.

We tentatively designate the bit pattern CL10 r000 for the indication of train packets.

Appendix B

MAC Protocol Finite-State Machines with Extensions

In the following, we reproduce the finite-state machine diagrams for the IEEE 802.5 and FDDI MAC protocols with the needed extensions for packet train transmission and reception. Actually, packet train reception does not require any extensions to these two token access MAC protocols. The extensions are for an interface device's medium access controller component (see 6.5). To save space, we only reproduce the transmitter portion of the finite-state machine diagrams for these two MAC protocols. Furthermore, we do not give detailed explanations for each state and transition in these diagrams, as these are exactly the same as in[18,40]. An interested reader can consult these references directly.

Operational Finite-State Machine Diagram for IEEE 802.5 Standard

Figure B.2 gives the finit-state machine diagram for IEEE802.5 MAC protocol in[40]. The extension to accommodate packet train transmission is in bold face for the transition 12.

The addition of Rel_Token is for transmission of a packet train. When a sender station has packets for two different packet trains adjacent to one another, the Rel_Token flag will be set after the station is finished transmitting the first train and is ready to transmit the first packet of the second train. Chapter 6 described how this

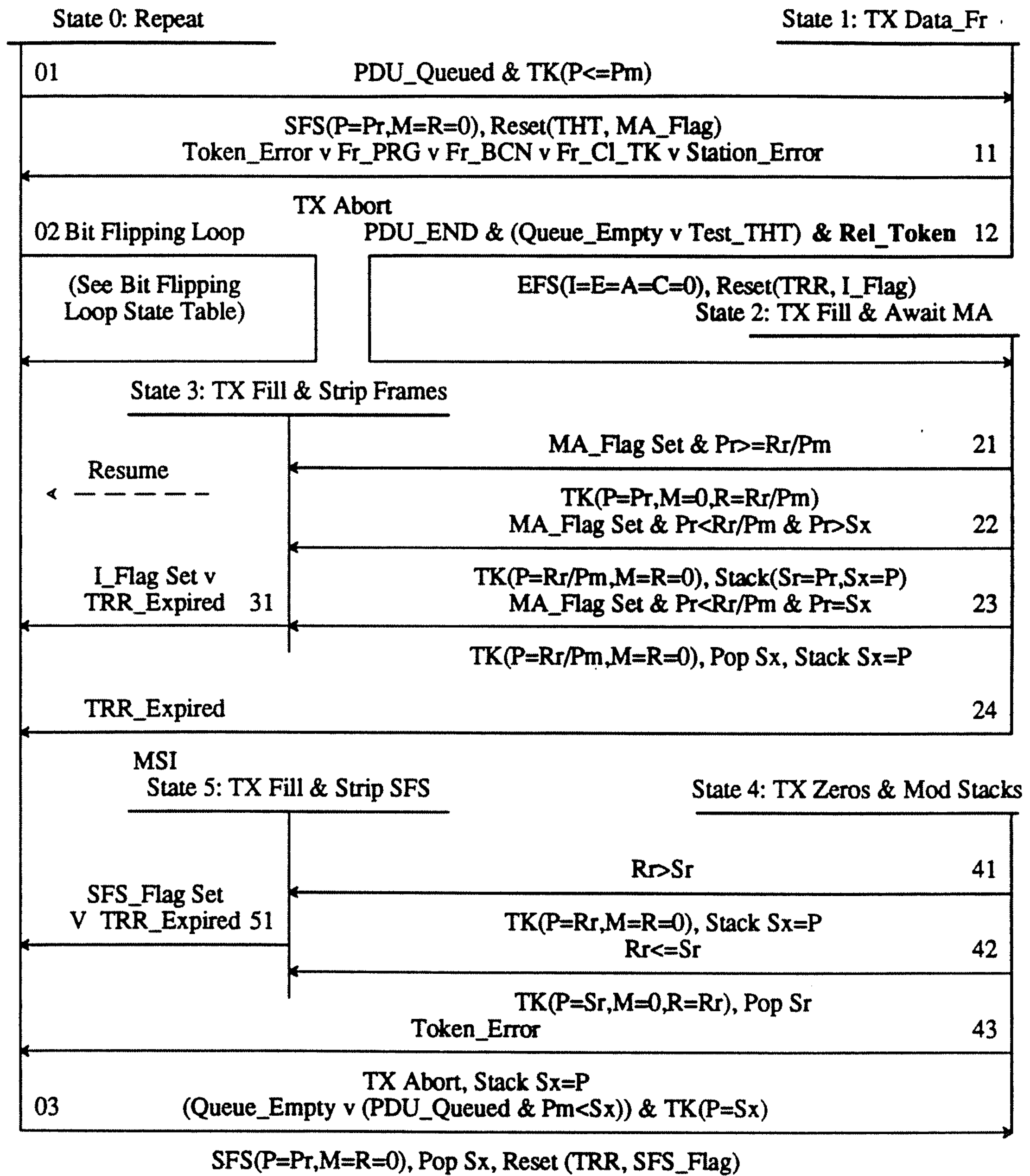


Figure B.1: Operational Finite-State Machine Diagram for IEEE 802.5

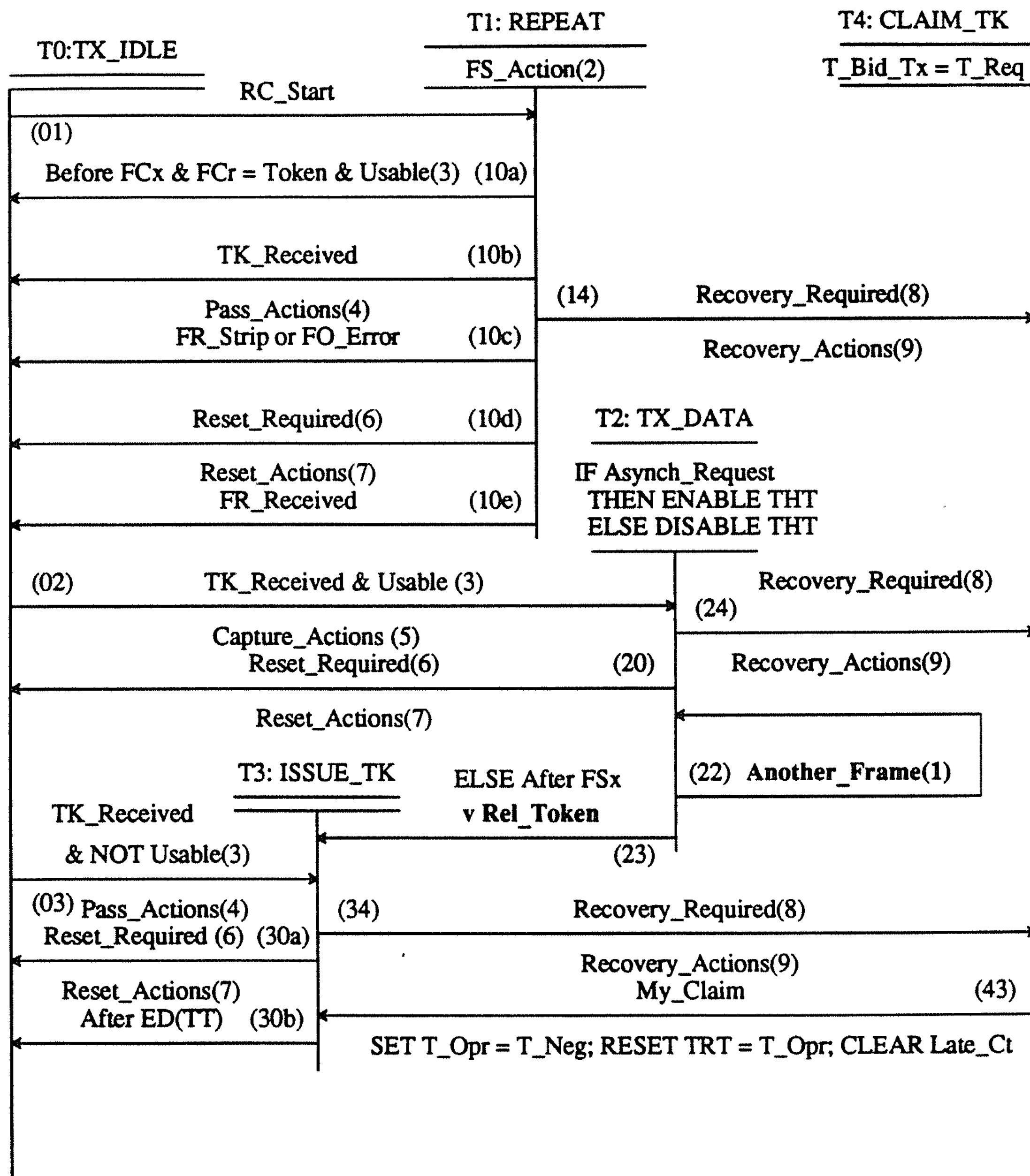


Figure B.2a: FDDI MAC Transmitter Finite-State Machine Diagram

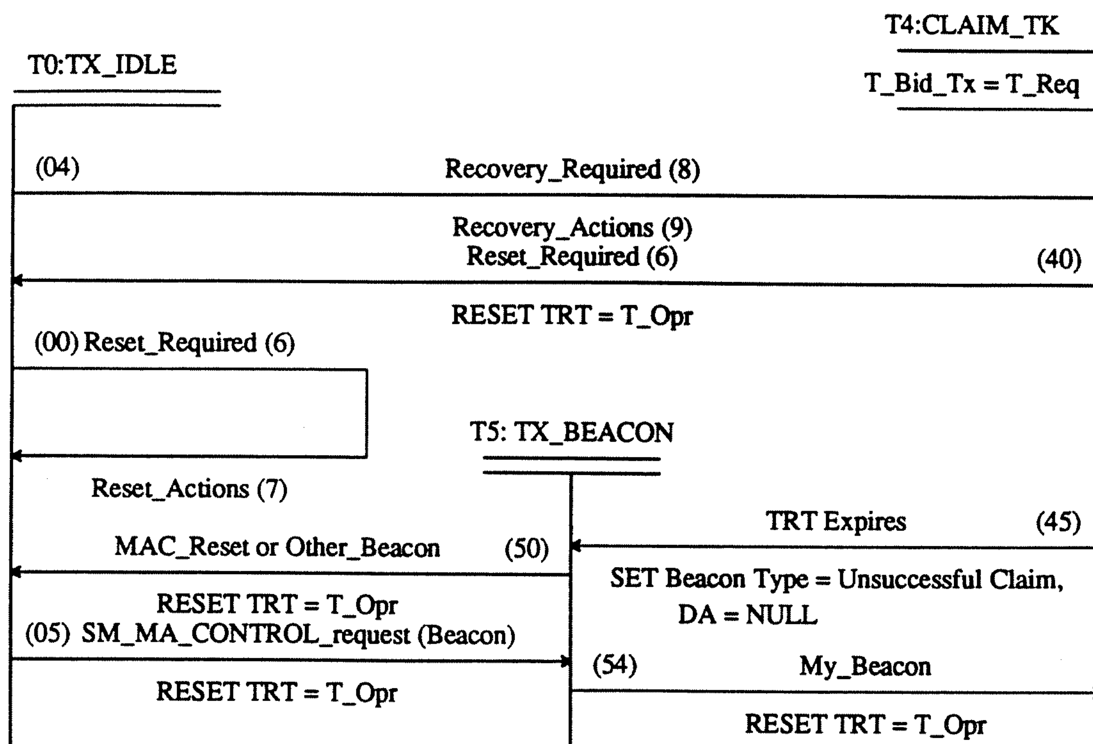


Figure B.2b: FDDI MAC Transmitter Finite-State Machine Diagram (Cont.)

flag is set during transmission of a packet train. When transmitting non-train packets or train packets that are not adjacent, this flag will remain unset.

Transmitter Finite-State Machine Diagram for FDDI MAC Standard

Figures B.2a and B.2b give the finite-state machine diagram for FDDI MAC standard[18] with the extensions to accommodate transmission of packet trains. The extensions are in bold face for the transitions 22 and 23.

Transition 22 is for a transmitting station to continue transmitting another packet frame after the Frame Status (FS) is transmitted. Originally, transition 22 is taken if TRT has not expired (Late_Ct=0), and if a synchronous request is queued

whose next frame transmission would not exceed the station's synchronous bandwidth allocation (not checked by MAC) or an asynchronous request is queued and the requested token class was received (R_Flag) and THT is to be ignored (Ignore_THT) or the Token Holding Timer (THT) is less than the requested priority threshold value (T_Pri(Request_Priority)). The needed extension is that in addition to the above conditions, the Rel_Token flag must also be clear for the transition to be taken.

Transition 23 is for a transmitting station to release the token. Originally, transition 23 is taken at the end of a completed packet frame if there are no more frames that may be transmitted as described in transition 22. This transition shall also be taken if TRT expires while the transmitter is waiting to transmit another frame (e.g., during frame setup time or while waiting for the Stream indicator). The needed extension to this transition is that this transition shall be taken when the Rel_Token flag is set, regardless whether the above conditions hold.

The Rel_Token flag is set under the same conditions as those for IEEE802.5, as described in 6.4.

References

1. Paul D. Amer, Ram N. Kumar, Rueybin Kao, Jeffrey T. Phillips and Lillian N. Cassel, "Local Area Broadcast Network Measurement: Traffic Characterization," *Tech. Rep. No.86-12*, (Jan., 1986).

2. Richard G. Bugenik and Jonathan S. Turner, "Performance of a Broadcaset Packet Switch," *IEEE Transaction on Communications* **37**(1)(1989).

3. John B. Carter, *Personal Communication,* Department of Computer Science, Rice University (1988).

4. D.R. Cheriton, "Local Networking and Internetworking in the V-System," *Proc. of 8th Data Communications Symposium* **13**(4)ACM SIGCOMM, (1983).

5. David R. Cheriton, "The V Kernel: A Softare Base for Distributed Systems," *IEEE Software*, pp. 19-42 (April, 1984).

6. David R. Cheriton, "VMTP: A Transport Protocol for the Next Generation of Communication Systems," *Proc. of Symposium on Communications Architectures and Protocols*, ACM SIGCOMM, (Aug., 1986).

7. David R.Cheriton and Carey L. Williamson, "Network Measurement of the VMTP Request-Response Protocol in the V Distributed Systems," *Technical Report No. STAN-CS-87-1145*, Dept. of Computer Science, Stanford University, (Feb., 1987).

8. Greg chesson, ''Procotol Engine Design,'' *Proceedings of INFORCOM 1987*, Silicon Graphics, Inc., (1987).

9. Greg chesson, ''XTP/PE Overview,'' *Proc. of 13th Conference on Local Computer Networks*, Silicon Graphics, Inc., (October, 1988).

10. Gary Christensen, *A Network Storage System.* 1979.

11. David Clark, John Romkey and Howard Salwen, ''An Analysis of TCP Processing Overhead,'' *Proc. of 13th Conference on Local Computer Networks*, (Oct. 10-12, 1988).

12. David Clark, ''The Structuring of Systems Using Upcalls,'' *Proceedings of the 10th Symposium on Operating Systems Principles*, pp. 171-180 Orcas Island, WA, (October 1985).

13. David Clark, March L. Lambert and Lixia Zhang, ''NETBLT: A High Throughput Transport Protocol,'' *SIGCOMM 1987 Workshop, Frontiers in Computer Communications Technology*, (Aug. 11-13, 1987).

14. DeWitt, D., Finkel, R., and Solomon, M., ''The Crystal Multicomputer: Design and Implementation Experience,'' *IEEE Transactions on Software Engineering* **SE-13**(8)(August, 1987).

15. D. C. Feldmeier, ''Empirical Analysis of a Token Ring Network,'' *Technical Memo. MIT/LCS/TM-254*, Massachusetts Inst. of Technology, Lab. of Computer Science, (January, 1984).

16. A. G. Fraser, "Towards a Universal Data Transport System," *IEEE JSAC* **SAC-1** pp. 803-816 IEEE, (Nov 1983).

17. Ricardo Gusella, "The Analysis of Diskless Worksation Traffic on an Ethernet," *Technical Report No. UCB/CSD 87/379*, University of California - Berkeley, (Nov., 1987).

18. International Organization for Standardization, "Information Processing Systems - Fibre Distributed Data Interface (FDDI), Part 2: Meda Access Control (MAC)," *DIS 9314-2*, (1987).

19. Van Jacobson, "Mail Message to comp.protocols.tcp-ip," *Message-ID: 8807200426.AA01221 for news group TCP/IP*, (July, 1988).

20. Rai Jain and Shawn Routhier, "Packet Trains: Measurements and A New Model for Computer Network Traffic," *Tech. Rep. No. MIT/LCS/TM-292*, (Nov., 1985).

21. M.J. Johnson, "Reliability Mechanisms of the FDDI High Bandwidth Token Ring Protocol," *Proc. of 10th Conference on Local Computer Networks*, (Oct. 1985).

22. H. Kanakia and D. Cheriton, "The VMP Network Adapter Board (NAB): High-Performance Network Communication for Multiprocessors," *Proc. of SIGCOMM. Symp. on Commun. Architectures & Protocols*, (Aug. 1988).

23. Chrisonpher A. Kent and Jeffrey C. Mogul, ''Fragmentation Considered Harmful,'' *Proc. of SIGCOMM '87 Workshop on Frontiers in Computer Communications*, (Aug. 1987).

24. Advanced Micro Devices, *Am 7990, Local Area Network Controller For Ethernet,* Advanced Micro Devices (Oct. 1986).

25. Keith A. Lantz, ''Rochester's Intelligent Gateway,'' *Computer*, (Oct. 1982).

26. Paul J. Leach, ''The Architecture of an Integrated Local Network,'' *IEEE Journal on Selected Areas in Communications* **SAC-1**(5)(Nov. 1983).

27. Miron Livny, *DENET Simulation Language Manual,* Computer Science Dept., University of Wisconsin-Madison (1986).

28. MERIT, Inc., *Management and Operation of the NSFNET Backbone Network,* MERIT, Inc., Ann Arbor, Michigan (November, 1987).

29. Anil Bhatia, James Sterbenz, and Guru Parulkar, ''Comments on Proposed Transport Protocols,'' *Tech. Report*, Department of Computer Science, Washington University (St. Louis), (1988).

30. G. Popek, ''Locus: A Network Transparent, High-Reliability Distributed System,'' *Proc. of 8th Symp. Operating Systems Principles* , ACM, (Dec. 1981).

31. Proteon Associates, *Operation and Maintenance Manual for the Pronet (TM) Local Area Communications Network,* Proteon Associates (1982).

32. Richard Rashid, *Accent: A communication oriented network operating system kernel,* Carnegie-Mellon University (1981). Draft Report

33. J. Postel and J. Teynolds, ''A standard for the Transmission of IP Datagrams over IEEE 802 Networks,'' *RFC 1042*, (February 1988).

34. J. B. Postel, ''User Datagram Protocol,'' *RFC 768*, (August 1980).

35. J. B. Postel, ''Transmission Control Protocol,'' *RFC 793*, (September 1981).

36. Floyd E. Ross and R. Kirk Moulton, ''FDDI Overview - A 100 Megabit per Second Solution,'' *WESCON* , p. 2/1 (1984).

37. K. C. Sevcik and M. J. Johnson, ''Cycle Time Properties of the FDDI Token Ring Protocol,'' *IEEE Transactions on Software Engineering* **SE-13**(1987).

38. J.F. Shoch and J.P. Hupp, ''Measured Performance of an Ethernet Local Network,'' *Comm. of the ACM 23(12)*, (Dec. 1980).

39. Alfred Z. Spector, ''Performing Remote Operations Efficiently on a Local Computer Network,'' *Communications of the ACM* **25**(4)(April, 1982).

40. ANSI/IEEE Std 802.5, *IEEE Standards for Local Area Networks: Token Ring Access Method and Physical Layer Specifications*, ISO Draft International Standard. 1985.

41. Richard Watson and John G. Fletcher, ''An Architecture for Support of Network Operating System Services,'' *Computer Networks* **4** pp. 33-49 (1980).

42. Richard W. Watson and Sandy A. Mamrak, ''Gaining Efficency in Transport Services by Appropriate Design and Implementation Choices,'' *ACM Transactions on Computer Systems* 5(2) pp. 97-120 (May 1987).

43. W. Zwaenepoel, ''Protocols for Large Data Transfers Over Local Networks,'' *ACM SIGCOMM 9th Data Communications Symposium*, (Sept, 1985).