

Performance Analysis of the IEEE
802.4 Token Bus

by

Catherine F. Summers
Alfred C. Weaver

Research Memo #RM-85-06

ABSTRACT

The use of computers in today's offices and factories is growing rapidly. With the expansion in the number of computers there is an expanding need for inter-computer communication. Local Area Networks can meet these needs.

There are many different network topologies and access protocols. The IEEE has proposed a new standard local area network that uses a token passing access protocol on a bus; this standard is known as 802.4.

This thesis studies 802.4 which is a complicated protocol with many features such as bounded delivery times and support of four message priority levels. The operation of the protocol is explained and an analytic model is developed that can provide estimates of network performance. A better understanding was gained and more comprehensive tests were performed through simulating representative network configurations. The number of possible network configurations precludes exhaustive testing; therefore care was taken so that the important parameters were identified and well tested. The experience and information gained in the simulation process illustrates several areas of importance when designing 802.4 networks.

ACKNOWLEDGEMENTS

This thesis would not have been possible without the guidance and assistance of Dr. Alfred C. Weaver, my advisor, and I would like to express special thanks to him for his support.

I would like to thank Alex Colvin for his explanations of 802.4 and the use of his simulator on which mine was based, and Mark Smith, for his help in mastering and maintaining the Apollo network. In addition I would like to thank the rest of my office mates who helped with the formatting and production of this thesis: Paul Ammann, Sue Brilliant, and Ray Lubinsky.

The research used in this thesis was funded by the General Electric Company of Charlottesville, Virginia. I would like to thank them for the support that made this project possible.

Table of Contents

ABSTRACT	i
ACKNOWLEDGEMENTS	ii
TABLE OF CONTENTS	iii
LIST OF FIGURES	vi
1 Introduction	1
1.1 Computer Networks	1
1.1.1 Long Haul Networks	1
1.1.2 Local Area Networks	2
1.1.2.1 Network Topologies	2
1.1.2.1.1 Star	2
1.1.2.1.2 Ring	3
1.1.2.1.3 Busses	4
1.1.2.2 Network Access Protocol	4
1.1.2.2.1 Centralized Control	5
1.1.2.2.2 Time and Frequency Division Multiplexing	5
1.1.2.2.3 Carrier Sense Multiple Access	6
1.1.2.2.4 Token Passing	6
1.2 The Study	7
2 The IEEE 802.4 Standard	9
2.1 IEEE Project 802	9
2.1.1 802.1	9
2.1.2 802.2 and Logical Link Control	9
2.1.3 Medium Access Control	10
2.1.4 802.3 - Contention Bus	11
2.1.5 802.5 - Token Ring	13
2.2 802.4	14
2.2.1 General Operation Characteristics	14
2.2.2 Priority and the Access_Classes	15
2.2.3 Logical Ring Membership	17
2.2.3.1 Joining the Token Passing Ring	18
2.2.3.2 Leaving the Token Passing Ring	20
2.2.4 Error Recovery	21
2.2.4.1 Lost or Multiple Tokens	22
2.2.4.2 Token-Pass Failure	24
2.2.4.3 Deaf Station	25
2.2.4.4 Duplicate Addresses	26
3 Analytic Modeling	27
3.1 Modeling	27

3.2	Definitions	27
3.3	A simple Model	31
3.3.1	Synchronous Only	32
3.3.2	Asynchronous Service	32
3.4	Throughput	33
3.5	Delay	33
3.5.1	Mean Delay	34
3.6	A real Model	36
4	The Simulator	38
4.1	Introduction to the Simulator	38
4.2	Simulator Design	39
4.2.1	Simulation Method	39
4.2.2	Time in the Simulator	40
4.2.3	User Interface	41
4.3	Simulator Functions and Capabilities	42
4.3.1	Network Configuration	42
4.3.1.1	Configuration Parameterization	42
4.3.1.2	Loading the Configuration	44
4.3.2	Simulating	45
4.3.2.1	Controlling the Simulator	46
4.3.2.1.1	Running the Simulator	47
4.3.2.1.2	Report Command	48
4.3.2.1.3	Altering the Simulator State	49
4.3.2.1.4	Simulator Interface	51
4.3.2.2	Managing the Events	53
4.3.3	Reports and Traces	54
4.3.3.1	Reports	55
4.3.3.2	Traces	58
4.4	Simulator Limits	59
4.4.1	Pascal Limitations	60
4.4.2	Simplification of 802.4	61
4.5	Testing Methods	63
5	The Performance of 802.4	65
5.1	The Performance Studies	65
5.2	Testing Methods	65
5.2.1	Designing the Tests	66
5.2.2	Test Organization	70
5.2.3	The Presentation of Results	73
5.3	Base Configuration Results	74
5.3.1	Throughput and Utilization	74
5.3.2	Delay vs. Offered Load	78
5.3.3	Token Cycle Time	79
5.3.4	Observations	82
5.4	Sensitivity to Packet Size and Number of Stations	84

5.4.1	Packet Size	84
5.4.2	Number of Stations	88
5.5	Access Classes and Timers	90
5.5.1	Even Load Distributions in the Access Classes	91
5.5.2	Uneven Load Distributions in the Access Classes	92
5.5.3	Timers	98
5.5.3.1	<i>High_Priority-Token_Hold_Time</i>	98
5.5.3.2	<i>Target_Rotation_Time</i> and Imbalanced Loads	100
5.6	Nonhomogeneous Stations	104
5.7	Transient Loadings	110
6	Conclusions	114
6.1	Analysis of 802.4	114
6.1.1	Protocol Overhead	114
6.1.2	Packet Sizes and the Number of Stations	115
6.1.3	Access_Classes and Timers	116
6.1.4	Nonhomogeneous and Transient Loads	116
6.2	Suggestions for Future Research	117
6.3	Summary	118
Appendix A	Manual for the IEEE 802.4 Simulator	120
Appendix B	Apollo Implementation of the Simulator	137
Appendix C	Simulator Test Cases and Results	141
Bibliography	144

List of Figures

Figure 1.1	Network Topologies	3
Figure 2.1	Relation of the ISO and 802 Models	10
Figure 4.1	Simulator Main Menu	46
Figure 4.2	Report Sub-system Menu	48
Figure 4.3	Display Sub-system Menu	51
Figure 4.4	Sample Simulator Display	52
Figure 4.5	Actual Bus Report	55
Figure 4.6	Actual Class Report	57
Figure 5.1	basecase as Displayed by Simulator's Editing Function	67
Figure 5.2	Simulator Shell Script	71
Figure 5.3	Simulator Input File	72
Figure 5.4	Base Configuration; Throughput vs. Offered Load	75
Figure 5.5	Base Configuration; Components of Bus Utilization	76
Figure 5.6	Base Configuration; Delay vs. Offered Load	80
Figure 5.7	Base Configuration; Delay Components and Offered Load	81
Figure 5.8	Base Configuration; Average Token Cycle Time	83
Figure 5.9	Delay vs. Offered Load for Varying Packet Size	85
Figure 5.10	Delay Components for Varying Packet Size	86
Figure 5.11	Average Delivery Time for Varying Number of Stations	89
Figure 5.12	Average Delivery Time for Evenly Distributed Loads	93
Figure 5.13	Delay Components for Evenly Distributed Loads	94
Figure 5.14	Delay Components for a 90/10 Load Distribution	96
Figure 5.15	Delay Components for a 10/90 Load Distribution	97
Figure 5.16	Varying the <i>High_Priority-Token_Hold_Time</i>	101
Figure 5.17	Varying the <i>Target_Rotation_Time</i>	102
Figure 5.18	One Station with Fifty Percent of the Offered Load	106
Figure 5.19	Two Stations with Fifty Percent of the Offered Load	107
Figure 5.20	A Station with Constant Load	109
Figure 5.21	Transient Loads and Token Cycle Time	111

CHAPTER 1

Introduction

1.1. Computer Networks

There is a growing trend towards computer automation of today's offices and factories. With this growth in the numbers of computers, there is a growing need for intercomputer communication. Computers that need to communicate with other computers across cities, states, countries, or even around the world are members of **Long Haul Networks**. Computers that share information and communicate over short distances like a building, a university campus, or a factory are members of **Local Area Networks**.

1.1.1. Long Haul Networks

There are many significant applications of long haul networks. They can be used to provide workers in a variety of geographically separate areas access to common databases as in an airline reservation system. They can be used to quickly transmit files between separate computer facilities. They can be used by a corporation to gather information from separate factories and offices for central data processing and evaluation.

Because of the long distances involved, long haul networks generally cannot use special high capacity communication media, like dedicated fiber optic or coaxial cables; instead they generally employ existing communication facilities such as the telephone lines or satellite communication. The problems and challenges to long haul network design are ones of routing and flow control.

1.1.2. Local Area Networks

A local area network allows communication between separate data processing devices located within a small area. Example devices are computers, terminals, sensors, programmable controllers, or peripheral devices such as disk drives or printers. The devices generally communicate over some high speed, low error rate, external medium installed expressly for the use of the network.

Some of the major advantages of local area networks (LANs) are the sharing of important resources, data and devices, the distribution of control processes to ensure reliability, and the ability to increase the power of a system incrementally. Some of the problems introduced with LANs are data integrity with distributed data bases, data security on the network, and incompatible hardware or software.

The issues in network design are network topology and network access protocols. The topology determines the method of station interconnection. The access protocols determine how an individual station gains access to the network.

1.1.2.1. Network Topologies

There are many ways to configure the stations on a network. Figure 1.1 illustrates the most common LAN topologies: star, ring, and bus. Each of these topologies has advantages and disadvantages.

1.1.2.1.1. Star

In a star configuration there is a central node to which every other station is directly connected. All communication is routed from the source station to the central node and then to the destination station.

There are two advantages to the star topology. First, the individual stations do not need to have any complicated networking hardware or software; most of the network functions can be assumed by the central node. Second, the maximum

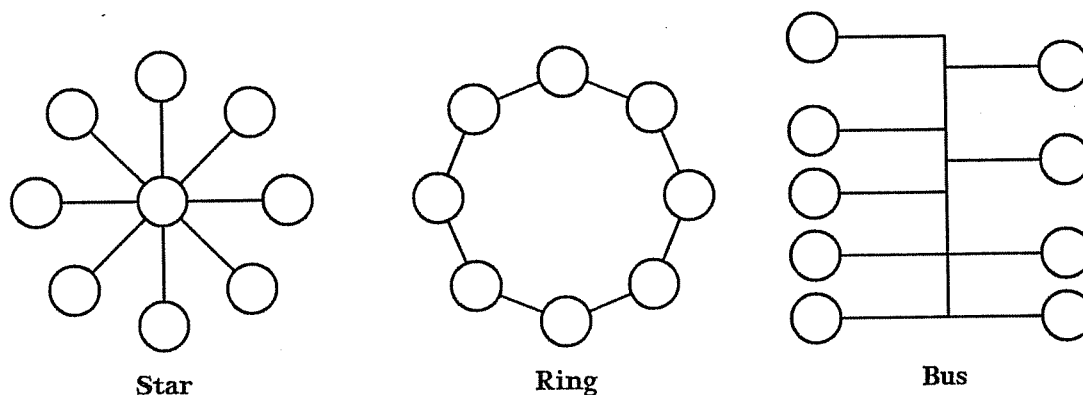


Figure 1.1 Network Topologies

delay of a message is the sum of the transmission time from the sender to the central node, the switching and buffering time of the central node, and the transmission time from the central node to the receiving station.

The main disadvantage of the star topology is that the central node represents a single point of failure. If the central node goes down, no stations can talk to each other.

1.1.2.1.2. Ring

In a ring each station has a point-to-point link with its two nearest neighbors. Generally these are unidirectional links so a station always receives messages from one neighbor and transmits the messages to the other neighbor.

The main advantage to the ring topology stems from this point-to-point communication. Very high speed communication media can be used in the links. Another advantage of rings is that because the entire message is regenerated at each node, the chance of an error due to attenuation of the signal is reduced.

The disadvantages to the ring topology are that a break in a single link can destroy the whole network, and because every station must read every bit of every message as it retransmits the message there is a minimum delay of one bit time per station added to a message's propagation delay.

1.1.2.1.3. Busses

In the bus topology, all the stations are directly connected to a shared transmission medium with no closed loops. When a message is placed on the bus it is heard by all of the stations connected to the bus.

One advantage of the bus is that all stations share a common perception of the state of the network. Another advantage is that breaks in the transmission medium may leave a large portion of the network operating.

Disadvantages to bus topology networks are that a break in the medium can isolate a station, and more complex network access hardware and software are often needed.

1.1.2.2. Network Access Protocol

There is a need to control a station's access to the network. Without access control, there is a chance that two stations will decide to broadcast messages at the same time, causing a collision which would result in the loss of both messages. The chance of a collision increases as the network load increases. It is the primary goal of the network access protocol to reduce or eliminate the chance of a collision and to recover if and when a collision occurs. Other goals of the protocol may be to share the network bandwidth between the stations on the network, and to provide maximum bounds on delivery times for messages.

There are several types of network access protocols. Some of the types are centralized control, time or frequency division multiplexing, carrier sense multiple

access, and token passing.

1.1.2.2.1. Centralized Control

In centralized control protocols, a single station is the master; it delegates the right to transmit to subordinate stations. The master station must maintain a list of all the stations in the network. Periodically, it must query the other stations to determine if they have traffic pending, or it must delegate the right to transmit to the stations. It is more efficient to ask the station if it has messages to send rather than simply giving it the right to transmit when it might not have any messages pending. A serious problem with the centralized control approach is that the network is vulnerable to collapse if the master station fails.

1.1.2.2.2. Time and Frequency Division Multiplexing

In time or frequency multiplexing, the communication medium is divided into discrete time slots or frequencies. In time division multiplexing, each station has one or more time slots allotted for its sole use. If a station does not transmit during its slot then the slot is wasted because no other station may transmit during that slot. In frequency division multiplexing, each station is allotted a portion of the the network's frequency spectrum for its sole use. Frequencies allotted to idle stations are wasted. The advantage to the multiplexing approach is that each station is guaranteed access to the network. The disadvantages are that precious bandwidth can be wasted by idle stations, the medium is partitioned to serve the maximum number of stations, even when only a few are active, and the individual messages of a single busy station can suffer large delays while the rest of the network bandwidth is not consumed.

1.1.2.2.3. Carrier Sense Multiple Access

In carrier sense multiple access (CSMA) protocols, all station listens to the state of the network before attempting to transmit. If the medium is idle and a station has a message to send the station will transmit the message; otherwise it waits until the medium becomes idle. There is a problem with the CSMA approach; station A at one end of a bus may sense that the network is idle and begin transmitting. One bit time before A's message gets to station B at the other end of the bus, B may sense an idle medium and begin transmitting its message. The messages from A and B will collide, interfering with each other. The details of a particular carrier sense protocol determine how quickly a station will attempt to transmit after the medium becomes idle, and what actions are taken when two transmissions from two separate stations overlap.

The most widely known and used variation of the CSMA protocols is Ethernet® [Xerox 82] which uses carrier sense multiple access with collision detection and binary exponential backoff. When a station on an Ethernet network detects a collision it aborts its transmission and sends a short jam message to ensure that all stations on the bus know that there has been a collision. Aborting the transmission of the corrupted message reduces the network bandwidth wasted by erroneous messages. Both stations involved in the collision then employ a binary exponential backoff mechanism which delays their next attempts to rebroadcast the collided messages.

1.1.2.2.4. Token Passing

Token passing access protocols attempt to remedy the problems of station starvation, long message delays, and static ring configuration. The station that possesses the token is allowed to transmit. When the token holder has finished

Ethernet is a registered trademark of the Xerox, Intel, and Digital Equipment Corporations.

transmitting or a timer has elapsed it passes the token to its successor. A logical ring is formed when the last station passes the token back to the first station.

There are two types of token access protocols. In the first type, the token is an explicit message with a unique bit pattern. In the second type, called Implicit Token Passing (ITP) [Weaver 84], there is no explicit token; station F "knows" that it comes after station C, and that when station C is finished transmitting it will be station F's turn to transmit. Explicit tokens are advantageous when the network configuration changes frequently, under high load conditions when the overhead of passing the token is small, and in situations when the token holder might wish to delegate the right to transmit to another station. The disadvantages of the explicit token passing schemes are that: (1) there is a distinct message (the token) that if lost requires expensive reinitialization of the network, (2) if the token is duplicated it would render the network unusable due to collisions until the duplicate token is destroyed, and (3) the token consumes network bandwidth as it is transmitted. Implicit token passing schemes have the advantages that no bit stuffing or other special processing is required to distinguish the token, there is no token that can be lost or duplicated, and there is no wasted overhead due to the token transmissions. Disadvantages to implicit token passing are the extra complexity needed to handle dynamic ring reformations.

1.2. The Study

This thesis describes the efforts involved in the study of the Institute of Electrical and Electronic Engineers' standard for Local Area Computer Networks that employ a token passing network access protocol with a bus topology. This standard is known as IEEE 802.4.

In Chapter 2 the IEEE 802 project is discussed. The full family of standards proposed by the 802 committees is described and the relation between the 802

standards and the International Standards Organization's Open Systems Interconnection model is explored. The majority of Chapter 2 is devoted to describing the design and functioning of 802.4.

In Chapter 3, an analytic model for the idealized operation of simple 802.4 networks is developed. The model provides insight into the steady state behavior of 802.4 and it provides a set of formulas that are useful when setting network parameters.

To study the performance of 802.4, I created a program that simulates the operation of 802.4 networks. Chapter 4 describes the program and the development process used to create and test the simulator. There are two approaches to the study the performance of proposed networks: simulation and analytic modeling. I chose to use a simulator for studying the performance of 802.4 for the following reasons. Simulation allows a study of the performance of the protocol for specific cases while analytic modeling only provides expected values for the general operation of 802.4 networks. A simulator also can be much less susceptible to the simplifying assumptions used in the analytic modeling process. The simulator can test nondeterministic actions in the protocol that cannot be captured analytically.

Using the simulator, I studied the effects on network performance by varying the values of network parameters. The simulations were designed not to test all possible configurations, an impossibly large task, but rather to tests specific features of 802.4 described in Chapter 2 and analytically modeled in Chapter 3. The tests and their results are described and analyzed in Chapter 5.

Chapter 6 relates the results obtained from the simulation studies to expected values generated by the analytic models. The final chapter summarizes the results obtained from this study and offers a judgement about the performance of 802.4.

CHAPTER 2

The IEEE 802.4 Standard

2.1. IEEE Project 802

The Institute of Electrical and Electronic Engineers has established a series of standards for Local Area Networks. These standards are being established by the IEEE's committee 802 [IEEE 802, 82]. There are six portions to the 802 standard.

2.1.1. 802.1

The first part, 802.1, describes the relationship between the other portions of the standard and the International Standards Organization's Open Systems Interconnection model (ISO OSI) [Tanenbaum 81]. The OSI model establishes a hierarchical decomposition of network structure by defining seven layers of protocol for inter-machine communication. In the ISO model, layer 1 is the most primitive layer of inter-machine communication and layer 7 is the most abstract. Figure 2.1 illustrates the mapping of the three lowest layers defined by the 802 standards to the bottom two layers of the ISO model. 802.1 also explains the relationship between the other 802 standards and higher level network protocols.

2.1.2. 802.2 and Logical Link Control

The standard 802.2 specifies the protocols at the Logical Link layer. The Logical Link Control layer (LLC) is the highest level of the network protocol specified by the 802 committee; this layer corresponds to the top of the OSI Data Link layer as shown in Figure 2. The LLC is responsible for the formation of two types of services for the higher level protocols. The first type is unacknowledged connectionless service. Interstation messages are datagrams, and the LLC provides

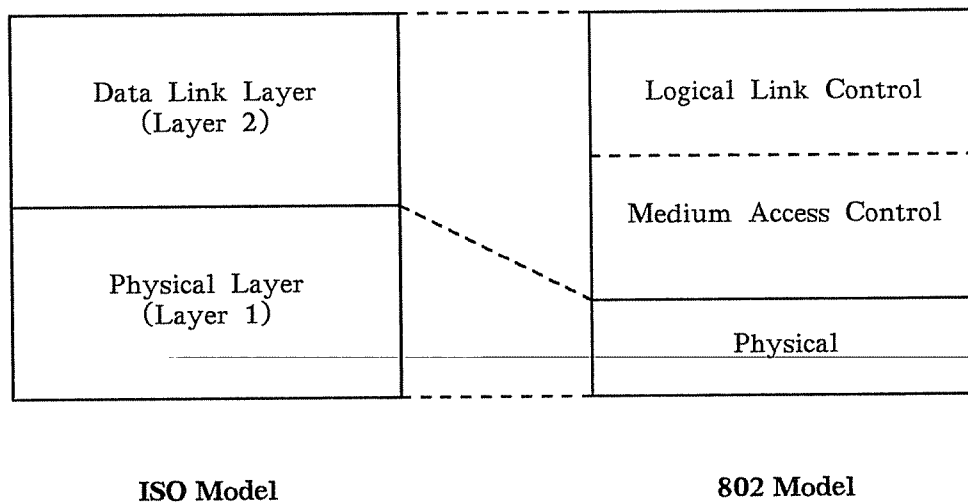


Figure 2.1 Relation of the ISO and 802 Models

services for message encapsulation, transmission and reception. The second type of service is connection oriented with the LLC providing virtual circuits between stations. The connection oriented service also provides flow control, sequencing, and error recovery services.

2.1.3. Medium Access Control

The standards 802.3, 802.4 and 802.5 define the Medium Access Control layer for CSMA/CD busses, token passing busses, and token passing rings respectively. The MAC layer is responsible for managing the access to the network through the Physical Layer; it is responsible for frame encapsulation, address and error recognition, and node failure detection and recovery. The Medium Access Control (MAC) layer extends the bottom portion of the OSI Data Link layer to provide services tailored for local area networks not provided in the ISO model which has been optimized for long haul networks. The extensions involve the MAC's actions in maintaining the network and the increased address and error recognition capabilities.

The three standards (802.3, 802.4 and 802.5) have some common features. In all three standards, addresses are specified to be either 16 or 48 bits long. The first bit of both 16 and 48 bit addresses is used to determine whether a message is addressed to an individual station or a group of stations; the special destination address of all ones is used for broadcast messages. The second bit of a 48 bit address can be used to determine whether an address is locally or globally administered; globally administered addresses are unique for all stations across distinct LANs. On any given network, all stations must use the same address size. Each standard requires that all messages be constructed of eight bit blocks known as octets. All three also use a 32 bit checksum for all data messages to guarantee that an invalid message is not erroneously accepted. Depending upon the address size selected, 16 or 48 bits, the smallest possible messages in an 802 network are either 96 or 160 bits respectively.

2.1.4. 802.3 - Contention Bus

It is the goal of the IEEE standard 802.3 to define the actions of a network in the presence and absence of collisions during message transmissions on a CSMA/CD bus. The standard also specifies the various media that can be used for the physical layer, the minimum and maximum spacing of stations, and other physical specifications which we will not discuss further.

802.3 employs 1-persistent CSMA/CD [Tanenbaum 81]. When a station has a message to transmit it enters a loop testing for an idle bus; when the station senses an idle bus, it will wait a short period of time for any previous message to have propagated to all the stations on the bus and then it will begin transmitting. The delay from the sensing of an idle bus to the beginning of transmission is to allow all stations to have a common sense of the bus state, and to allow the station that received the previous message time to process the message. If two or

more messages do collide on the bus, 802.3 specifies that the transmitting stations will send a jam signal that will inform all of the stations on the bus that there has been a collision, and the stations will then use a binary exponential backoff mechanism to reschedule the transmission of the collided messages. A counter of message transmission attempts maintained in each MAC is used to determine the delay before the MAC reenters the loop waiting for an idle bus. Equation 2.1 is used to compute the delay. No other messages can be transmitted by these stations until the collided messages have been successfully transmitted or the retransmission process has failed and the MAC has notified the LLC layer that the transmission failed.

From the above discussion it can be seen why 802.3 does not offer any guarantees on message transmission and cannot offer any priority transmission facilities. 802.3 was not designed to be used in real time environments where guaranteed message delivery or bounded delivery times are required; it was designed for use in an office environment with the associated bursty asynchronous loads of file and message transfer.

By choosing appropriate operating specifications, an 802.3 network can be configured as an Ethernet local area network; therefore it was the first of the 802 standards to be implemented since existing Ethernet chips, like the intel® 82586,

Let

n = number of collisions suffered by this message
 s = slottime, fixed by the standard to be 512 bit times
 $m = \min(n, 10)$

$$delay = \text{random}(0, 2^m) \times s$$

Equation 2.1

could be used.

2.1.5. 802.5 - Token Ring

The IEEE standard 802.5 describes a token passing ring local area network [IEEE 802.5 84]. 802.5 uses an explicit token passing scheme and offers the LLC eight levels of priority for message transmissions. 802.5 also offers immediate message arrival acknowledgements by appending a Frame Status field to the message which has two bits to be set by a station that recognizes the frame as being addressed to it and two bits to be set by the station if it successfully copies the message. Every station connected to the ring performs two functions; it provides the LLC with the facilities to transmit messages through the Physical Layer, and each station monitors the network state. One station is the active monitor and the rest are operating as standby monitors.

The token is an explicit message with the following characteristics: a starting delimiter, an access control field, and an ending delimiter, each of length 1 octet (8 bits). The first three bits of the access control field indicate the priority of the token; only stations with messages queued at priorities greater than or equal to the token's priority can claim the token, ensuring that the transmission of a low priority message cannot delay the delivery of a higher priority message existing anywhere in the network. If the fourth bit is a 0 the frame is a token, otherwise it is a data frame. The fifth bit is set by the monitor station to prevent a message or high priority token from continuously circulating on the ring. The last three bits are the reservation bits; a station with a high priority message queued can request that the next token be issued at that priority level.

The station holding the token has the right to transmit messages. The token holder can transmit any number of message frames until the transmission of the

intel is the registered trademark of the Intel Corporation.

next message would not complete before the expiration of the station's Token Holding Timer. When a station has exhausted its queue or its timer is about to expire it generates a new token of the appropriate priority and passes it to its neighbor. Any station that increases the priority of the token is responsible for restoring the priority to its current level. If the station should fail to reduce the priority, the active monitor will reduce the priority after the same high priority token has circulated the ring once.

2.2. 802.4

IEEE 802.4 [IEEE 802.4, 83] is a standard for bus topology local area networks using an explicit token passing scheme for network access. While the physical topology is a bus, 802.4 creates a logical ring of active stations through the token passing process. 802.4 offers four priority classes for message transmissions. The individual stations in an 802.4 network are free to leave and join the token passing ring as dictated by their traffic or station management decisions. The standard describes a robust protocol able to withstand the loss or duplication of tokens, and the failure or disconnection of stations.

2.2.1. General Operation Characteristics

The token in an 802.4 network is an explicit message of at least 96 bits for networks using sixteen bit addresses and at least 160 bits for networks using forty-eight bit addresses. Each station maintains the address of its successor in the logical ring, and that address is loaded into the token's destination address field. The token consists of one octet for the start delimiter followed by one octet of control information identifying the message as a token; two or six octets for each of the destination and source addresses; four octets for the frame check sequence; and one octet for the end delimiter. A station can optionally add octets of data

between the source address and the frame check sequence. The maximum size of any message in an 802.4 network is 8191 octets, excluding the start and end delimiters.

The station holding the token becomes the temporary master of the network. A series of timers, described below, limit the amount of time that a station can hold the token. During that interval, the token holder can use the network in any manner it wishes; it can even use its interval to implement other network access protocols such as polled response, or it can delegate its right to transmit to another station as long as that delegation obeys the time constraints. The option of imposing another access protocol on the network is not specified in the 802.4 standard except for the above mentioned time constraints. If the station has no use for the token, it will forward the token to its successor.

The token is passed from station to station in descending address order. The lowest addressed station passes the token back up to the active station with the largest address to close the logical ring. A station does not have to be a part of the token passing ring to receive transmissions or to respond to queries directed to it by the current token holder.

2.2.2. Priority and the Access_Classes

802.4 offers four *access_classes* of service for the eight LLC message priority levels. The highest LLC priority levels, seven and six, are mapped into *access_class* six, the *Synchronous access_class*. Levels five and four, three and two, and one and zero are respectively mapped into class four, the *Urgent_Asynchronous access_class*, class two, the *Normal_Asynchronous access_class*, and class zero, the *Time_Available access_class*. All 802.4 stations must offer all of the *access_classes*, or all transmissions must be routed through the *Synchronous access_class*. A station that does use the priority option must implement a queue for messages of each of the individual

access_classes, and it must provide three *token_rotation_timers* for the *Urgent_Asynchronous*, *Normal_Asynchronous*, and *Time_Available access_classes*. The timers are count-down timers loaded with station-dependent values, the *Target_Rotation_Time* for each *access_class*, every time the station receives the token. The use of these timers and the reload values will be explained below.

Whenever a station receives the token, it is guaranteed a certain amount of time for serving messages at the *Synchronous access_class*. This interval is a system wide parameter known as the *High_Priority-Token_Hold_Time*. When a station receives the token it enters the *Use-Token* state and loads its *token_hold_timer* with the *High_Priority-Token_Hold_Time*. If there are any messages enqueued for the *Synchronous access_class*, it begins to transmit those messages. If, after the completion of a transmission, the *token_hold_timer* has expired or the queue has been emptied, the station enters the *Check_Access_Class* state. A station in the *Check_Access_Class* state performs the following procedure: if the current *access_class* is the *Time_Available access_class* or the station is not implementing the priority option, the station will enter the *Pass-Token* state; otherwise it will (1) decrement the current *access_class*; (2) reload the *token_hold_timer* with the *token_rotation_timer* for the new *access_class*; (3) reload the *token_rotation_timer* with the *Target_Rotation_Time* for this class; (4) return to the *Use-Token* state.

The amount of time left in the *token_rotation_timer* that is loaded into the *token_hold_timer* determines how long the station will be able to serve the queues of the lower priority *access_classes*. To guarantee service at the *Urgent_Asynchronous access_class* Equation 2.2 must be true.

The *Target_Rotation_Time(Urgent_Asynchronous)* must also be large enough to allow for expiration of the *token_hold_time* during message transmission, and the possibility of other protocol traffic due to ring reconfiguration. The

Let

$TRT_4 = Target_Rotation_Time(Urgent_Asynchronous),$

$HPHT = High_Priority_Token_Hold_Time,$

$n =$ the number of stations that are members of the token passing ring,

$X_T =$ time to transmit a token to successor.

$$TRT_4 > (HPHT + X_T) \times n$$

Equation 2.2

Target_Rotation_Time for the both of the lower priority *access_classes* should be larger than the value of the next higher priority *Target_Rotation_Time* if service at the *access_class* needs to be guaranteed. To correctly use the priority feature of 802.4, the transmission of *Time_Available* messages should not interfere with the transmission of *Normal_Asynchronous* messages whose transmissions should be secondary to that of *Urgent_Asynchronous* messages, the timers should be set with

$$TRT_4 > TRT_2 > TRT_0$$

With this relationship there is more time available for serving *Urgent_Asynchronous* traffic than traffic at the lower priority *access_classes*, and as the token cycle time increases, a station will be able to transmit *Urgent_Asynchronous* messages after service at the lower classes is stopped. The relationships of the timers are discussed further in section 5.5.3.

2.2.3. Logical Ring Membership

An 802.4 network usually consists of two or more stations connected in a logical ring. Using the ability to delegate the right to transmit, a network could consist of one active station and a series of inactive stations only capable of responding to messages and unable to initiate "conversations". Generally, if a station believes itself to be the only member of a token passing ring, 802.4 assumes that the station is in error. Stations can leave and join the ring dynamically, and

any station must be able to patch the ring around a failed neighbor. This section discusses dynamic station membership leaving the error conditions mentioned above for the next section.

There are station variables used for maintaining the logical ring. The *inter_solicit_count* is a counter that is decremented each time the station passes the token; when it expires, the station will check for the existence of neighboring stations waiting to join the token passing ring. Each station saves its own address and the addresses of the preceding and following stations in the variables *TS*, *PS* and *NS* respectively. The boolean variable *NS_Known* is set to true when a station knows the address of its successor. *Lowest_Station* is true if the station has the lowest address of any active station. The *Bus_Idle_Timer* determines how long a station will listen to an idle bus before it will attempt to claim the token and initialize or reinitialize the token passing ring.

2.2.3.1. Joining the Token Passing Ring

Each time a station receives the token, it decrements the *inter_solicit_count*. If the value equals zero the station will open a response window by transmitting a *Solicit_Successor* frame and entering the *Await_Response* state. The *Solicit_Successor* frame is a MAC management frame with the destination address equal to this station's *NS*. After transmitting the *Solicit_Successor* frame, the transmitting station will open a response window, an interval of one network slottime, during which the station listens for the transmission of *Set_Successor* frames from stations waiting to join the ring. Only stations whose addresses lie between the destination address and the source address can respond to the *Solicit_Successor* frame. As a special case, if the token holder is the lowest station it will open two response windows; the first to enable stations with addresses less than its address to join the token passing ring, and the second window to allow stations with addresses greater than

the highest station a chance to join.

If the token holder does not receive any responses to the *Solicit_Successor* frame, it will reload the *inter_solicit_count* with another station management parameter, the *Max_Inter_Solicit_Count*, and it will enter the *Pass-Token* state. To avoid the situation of all stations on the network opening their response windows on the same token rotation cycle, 802.4 specifies that the least significant two bits of the *Max_Inter_Solicit_Count* must be randomized every 50 milliseconds or after every use of the value. The *Max_Inter_Solicit_Count* must be an integer in the range 16 to 255.

If one station responds to the *Solicit_Successor* frame, the token holder sets its *NS* to the source address of the *Set_Successor* frame. The token holder does not reload its *inter_solicit_count* when a successful *Set_Successor* frame is received. The token is then passed to the new ring member which sets its *PS* to the token's source address and must have set its *NS* to the *Solicit_Successor* frame's destination address.

If more than one station responds to the *Solicit_Successor* frame the responding stations enter the *Demand_In* state to contend for the single ring addition for this token cycle. When the responding stations attempt to transmit their *Set_Successor* frames, the transmissions will overlap and the collision will be noted by the token holder. The token holder will enter a loop transmitting *Resolve_Contention* frames and opening four response windows until it receives a valid *Set_Successor* frame. To resolve the contention, every station in the *Demand_In* or *Demand_Delay* states will enter a loop where a pair of bits from the station's address is used to determine which window to use for transmitting the next *Set_Successor* frame. If the pair of bits is (0,0) the station responds during the first response window; if (0,1), then during the second; if (1,0), then during the third; if (1,1), then during the

fourth window. The pair of bits is selected by the *contend_pass_count* which is incremented on each iteration of the contention algorithm. If a station hears a transmission during an earlier response window it drops from the contention process and returns to the *Idle* state. If the *contend_pass_count* is equal to the *max_pass_count* the stations have exhausted all of their address bits which means that two stations share the same address; a last try at resolving the contention uses a random number between 1 and 4 to determine which response window to use. The station (or stations) that loses the contention reports its duplicate address to the station management procedures and takes itself off line. As above, when the token holder receives a valid *Set_Successor* frame, it records its new successor and passes it the token. On the next token rotation the token holder will open another series of response windows to allow stations to join that lost the contention process on this token cycle.

2.2.3.2. Leaving the Token Passing Ring

Each station has two boolean variables, *in_ring_desired* and *any_send_pending*, which are used to determine if the station should leave the token passing ring. The *in_ring_desired* variable is set by the station management and as long as it remains true a station will remain an active participant of the token passing ring. The second variable, *any_send_pending*, is true whenever the MAC has any message in any of the *access_class* queues. A station that is a member of the token passing ring with *in_ring_desired* false but *any_send_pending* true will remain in the ring until it has emptied its queues.

When a station decides to leave the token passing ring there are two possible methods to effect the departure. The most drastic method is for the station to ignore the token when it is passed to the station on the next token rotation. This method uses the error recovery mechanism to patch the departing station out of

the ring. The more graceful method requires the station to wait until it receives its next token; as the token holder, the station transmits a *Set_Successor* frame to the station's *PS* with the station's *NS* as the new next station field of the message.

An ambiguity in the protocol concerns the actions to be taken when two or more neighboring stations patch themselves out of the ring on the same token rotation. For example, station 21 tells station 25 that station 18 will be 25's new next station, and then station 21 passes the token to station 18. Upon receipt of the token, station 18 tells station 21 that station 17 will be 21's next station, and passes the token to station 17. On the next token cycle, station 25 will attempt to pass the token to station 18 and fail. Through the use of the error recovery mechanism station 18 will be patched out of the network with a minimum of additional overhead. Two things should be noted: station 18's *Set_Successor* transmission to station 21 was a wasted transmission, and if more than two stations had left the ring, the error recovery mechanism will be more complex as will be shown in the next section. A solution to this situation would be to have all stations listen for any *Set_Successor* frames transmitted by their successor and to use the destination address of the *Set_Successor* frame as their predecessor's address, but this conflicts with page 7-7 of the Revision E 802.4 Standard which states that the *PS* "is set to the value of the source address of the last token addressed to the station".

2.2.4. Error Recovery

The 802.4 standard lists the errors that a MAC must be able to handle. The errors are lost or multiple tokens, a token-pass failure, a deaf station, and stations with duplicate addresses. One obvious type of error, the corruption of a data message, is not handled by the MAC layer; the reception of a bad message is reported to the LLC layer, and it is the responsibility of the LLC layer to deal with the

error by requesting retransmission or whatever other algorithm it wishes to employ.

2.2.4.1. Lost or Multiple Tokens

For the correct operation of the token passing protocol there has to be one and only one token in the network. If there is no token then no station would be able to transmit. If there is more than one token multiple stations could attempt to transmit at the same time, losing messages through collisions.

The token could be lost in several ways. The token holder could fail before passing the token, the token could be lost in a noisy environment that convinced the token holder that even though it is not receiving valid frames some other station must be transmitting something (see the discussion below on token pass failure), or the whole network has just powered up and no token has been created.

Each station starts its *bus_idle_timer* whenever it is in the *Idle* station state and it senses an idle bus. If the *bus_idle_timer* expires, the station is in the *Idle* state, the bus is idle, and the station has messages to send or it wants to be a ring member and is not the sole active station, the station will transmit a *Claim-Token* frame and enter the *Claim-Token* state. A station in the *Claim-Token* state will, if the bus is quiet and the *claim_pass_count* is less than the *max_pass_count*, increment the *claim_pass_count* transmit a *Claim-Token* frame with a data unit 0, 2, 4, or 6 slottimes long, and wait 1 slottime before repeating the process. If the station senses another station transmitting when the *claim_timer* expires it has lost the contention for the token and returns to the *Idle* state. If the *claim_pass_count* equals the *max_pass_count* the station has won the contention process and enters the *Use-Token* state.

The station with the lowest address sets the *bus_idle_timer* to six slottimes; all other stations wait seven slottimes. Since waiting stations drop out of the contention process when they detect other transmissions, the lowest addressed station

should win the contention for the token on network initialization.

Multiple tokens could be generated on an 802.4 network by a station with a faulty receiver. The station with a faulty receiver could erroneously timeout thinking that the bus was idle. The station would then generate a new token through the method described above. A duplicate token could be generated through the following scenario; H passes the token to F and F, having no messages to send, passes the token to C. When H's *token_pass_timer*, (see below) times out, H could sense an idle bus if C has not begun transmitting and pass the token to F, again creating two tokens. Another method of creating a duplicate token is to join two separate busses, each with its own token.

Regardless of the manner in which a duplicate token is generated, 802.4 specifies that the duplicate tokens will be removed from the network. A station in the *Idle* state will proceed to the *Use_token* state upon reception of a token. A station in the *Use-Token* state will transmit its messages even if there is other traffic on the bus. If the station does not have any messages queued (station F in the example above) it will check the variable *just_had_token* before it attempts to pass the token. *just_had_token* is set to true when the station passes the token, and it is reset when the station hears a valid frame from another station. If the variable is set then the station assumes that it has received a duplicate token and does not pass the token. A station in the *Check-Token-Pass* state will defer to other traffic on the bus assuming that either its successor has received the token and is transmitting or that there are duplicate tokens and it should not continue attempting to pass the token. If all of the stations with duplicate tokens decide that their tokens' are duplicates, then the token can be lost if all stations go to the *Idle* state.

2.2.4.2. Token-Pass Failure

When a station transmits a token to its successor, it starts the *token_pass_timer* and enters the *Check-Token-Pass* state. If the station hears a transmission from another station before the timer expires, the station assumes that its successor has received the token, and the station enters the *Idle* state. If the timer expires and the station is still in the *Check-Token-Pass* state, it will assume that the token pass has failed. If the failed attempt was the first attempt at passing the token the station will transmit another token to its successor, restart the *token_pass_timer*, and reenter the *Check-Token-Pass* state. When the second attempt at passing the token has failed the station will enter the *Pass-Token* state and try to reconnect the token passing ring.

After two attempts at passing the token to the successor station, the token holder first must evaluate the state of the bus to determine the course of action needed to repair the ring. A station seeking to repair the ring after two failed token passes has not heard any valid messages from any other stations since it first started the token passing process; if it had it would not be in this state. If the station senses a non-idle bus, then it assumes that its receiver is at fault (valid messages are being transmitted but not received by this station), and enters the *Idle* state.

A station that has had two token passes fail and that senses an idle bus will first assume that its successor has failed or has left the token passing ring without notifying its predecessor. The station will transmit a *Who_Follows* frame with its failed successor's address, open three *response_windows*, and enter the *Await_Response* state. If the successor to the failed station responds with a *Set_Successor* frame the token holder sets its *NS* to the source address of the received *Set_Successor* frame and passes the token to the failed station's successor, thereby patching the failed

station out of the ring. If the token holder does not get a response to the *Who_Follows* frame it will retransmit the frame.

When two attempts to elicit a response from the successor to the failed station have failed, the token holder will transmit a *Solicit_Any* frame to allow any other active station to respond with a *Set_Successor* frame and reconnect the ring. If a single valid response to the *Solicit_Any* frame is received the token holder will pass the token to the responder. If more than one station responds to the frame the resolution process described previously in section 2.2.3.1 is used to resolve the contention. Stations which were ring members between the token holder and its new successor will have to rejoin the ring as response windows are opened. If no response to the *Solicit_Successor* frame is received the token holder assumes that it has a faulty receiver; the faulty receiver is reported to the LLC layer, and the station enters the *Offline* state.

2.2.4.3. Deaf Station

A station with a broken receiver can pollute the network because it cannot hear the transmissions of other stations. Each station maintains several variables which will limit the duration of this error condition.

A deaf station in the *Idle* state will timeout believing the token has been lost. Because it cannot hear the transmissions of the other stations it will progress through the entire process of claiming the token. Once a deaf station has claimed the token it will begin to transmit its messages. Once it has emptied its queues or its timers have expired, it will attempt to find a successor to which it may pass the token. The station will not be able to hear the responses of other stations. When it does not receive a response the station will check to see if there are any messages enqueued, and if so it will "silently" pass the token to itself and resume transmitting. If the queues are empty the station will set *in_ring* to false, set

sole_active_station to true, increment the *transmitter_fault_count*, and return to the *Idle* state. If more messages for transmission arrive at the station's queues the station will repeat the process of claiming and using the token, otherwise it will remain idle.

2.2.4.4. Duplicate Addresses

Addresses must be distinct in an 802.4 network. If two or more stations have the same address, a token addressed to a single station would be received by more than one station, each of which would become the token holder and begin transmitting. Duplicate addresses are detected through the contention process and by an idle station monitoring network traffic. The duplicate addresses are detected by the MAC layer and reported to the LLC layer, but it is the duty of the LLC or higher layers to change the station's address.

A station which is contending for admission to the token passing ring uses the bits of its address to sequence through the contention process. As described in section 2.2.3.1, the station uses a pair of bits from its address to determine which window to use in responding to *Resolve_Contention* frames. If a station cycles through its complete address and is still involved in the contention process, then at least one other station has the same address. The station or stations that lose the contention process report their duplicate addresses to the LLC layer and go to the *Offline* state.

Stations in the *Idle* state listen to all bus traffic. If a station hears a valid frame with a source address equal to the stations address, it reports the duplicate address and goes to the *Offline* state.

CHAPTER 3

Analytic Modeling

3.1. Modeling

An analytic model of the operation of simplified 802.4 networks is developed in this chapter. The model is useful for analyzing the steady-state behavior of networks. It is also useful for generating expected results to be compared with simulation results, and the formulas used to develop the model are useful when the network designer is setting network parameters.

The second section of this chapter defines the notation used to develop the model and the relationships between the variables. The third section discusses the assumptions made in the simple model and provides simplified expressions for some network parameters. The fourth section provides a general expression for the throughput, and the fifth section discusses and analyzes the components of message delays. In the final section of the chapter the deficiencies of the modeling process are discussed.

3.2. Definitions

The following notation is used:

$N \equiv$ number of distinct stations on the logical ring

$R \equiv$ set of distinct servers on the logical ring

$\lambda_r \equiv$ the message arrival rate at server $r \in R$ (messages/second)

$\mu_r \equiv$ the mean message service rate at server $r \in R$ (messages/second)

$\rho_r \equiv$ the traffic intensity at server r

$\bar{m}_r \equiv$ the mean message length at server $r \in R$ (in bits)

$C \equiv$ the bus capacity (bits per second)

$Q_r \equiv$ the number of messages enqueued at server $r \in R$

$u_r \equiv$ the bus utilization at server $r \in R$

$\sum_{r \in R} u_r = U \equiv$ the total bus utilization

$X_{m_r} \equiv$ duration of a message transmission from server $r \in R$ (seconds/message)

$X_T \equiv$ duration of a token transmission (seconds/token transmission)

$T_r \equiv$ time server r transmits per token cycle (seconds/token cycle)

$T_C \equiv$ token cycle time (seconds/token cycle)

$V \equiv$ protocol management overhead (seconds/token cycle)

N is the number of stations that participate in the token passing process and are therefore members of the logical ring.

R is the set of message servers in the ring. Each active *access_class* at each station is represented by a server. A server has a queue for messages awaiting transmission, and if the server is not a *Synchronous access_class* server it has a *token_rotation_timer* which determines the time available for message transmission by the server.

Each server r has a message arrival rate λ_r which is measured in messages per second.

The mean message service rate μ_r in messages per second at server r can be expressed as

$$\mu_r = \frac{1}{X_{m_r}} \cdot \frac{T_r}{T_C}$$

which is the product of the message service rate and the fraction of a token cycle spent transmitting at server r . If the queue is emptied on each token cycle

$$\mu_r = \frac{T_r}{X_{m_r} \cdot T_C} = \frac{\lambda_r \cdot T_C \cdot X_{m_r}}{X_{m_r} \cdot T_C} = \lambda_r$$

The actual service rate while r is transmitting is $\frac{1}{X_{m_r}}$ which can be used to determine the traffic intensity:

$$\rho_r = \lambda_r \cdot X_{m_r}$$

The number of messages enqueued at a station on token rotation i consists of two terms: the messages not transmitted on the previous token cycle, and the message arrivals during the current token cycle.

$$Q_{r_i} = \left(Q_{r_{i-1}} - \frac{T_{r_{i-1}}}{X_{m_r}} \right) + \lambda_r \cdot T_{C_{i-1}}$$

To prevent losing messages because of the finite length of the queues, the average number of messages transmitted by the server must equal the message arrival rate; so the average number of message arrivals \bar{Q}_r during an average token cycle T_C is

$$\bar{Q}_r = \lambda_r \cdot T_C$$

The utilization u_r is the ratio of how long r is served per token cycle. It can be expressed as

$$u_r = \frac{T_r}{T_C}$$

Using \bar{m}_r as the mean message length at server r , including the necessary framing and address bits, C as the capacity of the bus, and g as the interframe gap required between all message transmissions, then the message transmission time X_{m_r} is computed as

$$X_{m_r} = \frac{\bar{m}_r}{C} + g$$

The token transmission time X_T is the sum of the time needed to transmit the token, the time for the token transmission to propagate to its destination, and

the time needed for the recipient to switch to the *Use-Token* state. X_T is assumed to be constant for all token transmissions.

T_r , the time that server r transmits in seconds per token cycle, depends upon the number of messages enqueued at r and the maximum time r has available for message transmissions. The number of messages enqueued at a server r is equal to the number of messages enqueued but not transmitted on the last token cycle and the number of messages that arrived during the current token cycle.

If r is a *Synchronous access_class* server then

$$T_r = \min(\bar{Q}_r \cdot X_{m_r}, T_6)$$

where T_6 is the effective *High-Priority-Token-Hold-Time* (*HPHT*). The effective *HPHT* can be one message transmission time greater than the actual *HPHT* since the expiration of the *token_hold_timer* will not abort a message transmission.

Determining T_r for the non-synchronous *access_classes* is not as straightforward as determining T_r for synchronous servers. For T_r to be equal to $\lambda_r \cdot T_C \cdot X_{m_r}$, the *Target_Rotation_Time*, TRT_r , for server r must be greater than $T_C + \lambda_r \cdot T_C \cdot X_{m_r}$. As explained in Section 2.4.2, when r becomes the active *access_class* for a station in the *Use-Token* state the *token_rotation_timer* for server r will be loaded into the *token_hold_timer*, and the *token_rotation_timer* will be reloaded with the *Target_Rotation_Time*. Server r will be served only if there is time available in the *token_hold_timer*. T_r depends upon the current token cycle time as follows:

$$T_{r_i} \leq TRT_r - T_{C_i}$$

If $TRT_r = X$ seconds and the last token cycle took X seconds then the *token_rotation_timer* will expire just as server r attempts to begin service on the current token rotation. Assuming that the server rarely exceeds the *Target_Rotation_Time* the average time r transmits can be expressed as

$$\bar{T}_r = \min(\bar{Q}_r \cdot X_{m_r}, \max(0, TRT_r - \bar{T}_C))$$

The token cycle time T_C is measured from when a station receives the token on one token rotation until it receives the token on the next token cycle. It can be computed as

$$T_C = N \cdot X_T + \sum_{r \in R} T_r + V$$

If the overhead V is ignored and the queues are drained at each server on each cycle, T_r would equal $\lambda_r \cdot T_C \cdot X_{m_r}$ for every server r and

$$T_C = \frac{N \cdot X_T}{1 - \sum_{r \in R} \lambda_r \cdot X_{m_r}}$$

The protocol management overhead V consists of the *Solicit_Successor* frames transmitted by token holders to allow other stations a chance to join the token passing ring, the response windows associated with the *Solicit_Successor* frames and any *Resolve_Contention* and *Set_Successor* frames transmitted. As explained in section 2.2.3, the period between windows is controlled by the *Max_Inter_Solicit_Count* and the number of stations waiting to join. Also included in the overhead are the retransmissions of tokens which were not correctly received by destination stations. If the probability of errors is small the number of retransmissions will also be small. There are other factors that can contribute to the overhead value, but generally V will be very small if not actually zero.

3.3. A Simple Model

To derive a simple model of 802.4 the following assumptions will be used. First, the bus is error-free. Second, all the stations are permanent ring members; no *Solicit_Successor* frames are transmitted and no response windows are opened; the overhead V is zero. Third, all the stations are homogeneous, each offering the same *access_classes*, each with identical λ_r . Fourth, all timers are set such that the

queue at each server is emptied on each token rotation.

3.3.1. Synchronous Only

In the simple, *Synchronous*-only case, the token cycle time can be expressed as

$$T_C = N \cdot X_T + N \cdot \lambda \cdot T_C \cdot X_m$$

which reduces to

$$T_C = \frac{N \cdot X_T}{1 - N \cdot \lambda \cdot X_m}$$

From the above equation, it can be seen that the token cycle time is uniquely determined by the message arrival rate λ . T_C is bounded by

$$N \cdot X_T \leq T_C \leq N \cdot (X_T + T_6)$$

The total utilization of the bus, U , is bounded by

$$0 \leq U \leq \frac{T_6}{T_6 + X_T}$$

because no more than T_6 seconds of message transmissions can occur before a token has to be passed.

3.3.2. Asynchronous Service

A simple case will be used to determine the mean token cycle time and the bounds upon the token cycle time in the presence of asynchronous service. Assume that all non-token traffic belongs to a single asynchronous *access_class* and that all members of the token passing ring use the same *Target_Rotation_Time*. The general formula

$$T_C = N \cdot X_T + \sum_{r \in R} T_r$$

reduces to

$$T_C = \frac{N \cdot X_T}{1 - N \cdot \lambda_r \cdot X_m}$$

and T_C is bounded by

$$N \cdot X_T \leq \bar{T}_C \leq TRT_r$$

The total utilization of the bus, U is bounded by

$$0 \leq U \leq \frac{TRT_r - N \cdot X_T}{TRT_r}$$

where $TRT_r - N \cdot X_T$ is the portion of the token cycle time not consumed by token transmission.

3.4. Throughput

The throughput S is the ratio of the number of data bits transmitted per bit time. No distinction is drawn between the bits that are actual data and those address and framing bits necessary to create a data frame. Token and other protocol frame transmissions are not included in the throughput. If the queue at each server r is drained on each cycle the average throughput is

$$S = \frac{\sum_{r \in R} \bar{T}_C \cdot \lambda_r \cdot \bar{m}_r}{\bar{T}_C \cdot C}$$

where $\bar{T}_C \cdot \lambda_r$ is the number of messages transmitted by server r and \bar{m}_r is the mean message length.

3.5. Delay

Every message transmitted by a server is subject to a delay before it reaches its destination. There are three components to the delay. The first component is the queuing delay which measures the delay from when a message enters a server's transmit queue until it reaches the front of the queue. The second delay component is the acquisition delay; how long does the message wait at the front of the queue before it begins transmission. The third component is the transmission delay, which is the difference between the start of the message's transmission and its completion. It is possible that a message will suffer no queuing or access delays, but all messages will encounter a transmission delay.

The first message to arrive in the queue of an idle server has a queuing delay of zero and an acquisition delay equal to the time that will elapse before the server begins transmitting. The acquisition delay of all messages that arrive after the first will be equal to the interframe gap unless the message arrives at an empty queue just before the server relinquishes the token. The acquisition delay is bounded by

$$0 < D_A < T_C - T_r$$

No message can have a larger acquisition delay as long as every server can empty its queue on each token rotation.

Of the $\lambda_r \cdot \bar{T}_C - 1$ messages that arrive after the first message, $\lambda_r \cdot (\bar{T}_C - T_r) - 1$ messages arrive while server r is idle. The queuing delay for these messages is the sum of the delay until r begins service and the delay of transmitting the messages enqueued in front of the message. Each of the $\lambda_r \cdot T_r$ messages that arrive while r is transmitting messages have a queuing delay equal to the transmission time for the messages enqueued ahead of the message.

3.5.1. Mean Delay

To determine the mean delay, the distinction between the acquisition and queuing delays will be ignored. The M/G/1 Vacation Model described by Fuhrmann and Cooper [Fuhrmann 84] is used to determine the wait time for messages in a queue. The first model described in the Fuhrmann and Cooper paper describes a single server which switches service around a ring of queues, stopping to serve any queue with pending messages.

The M/G/1 vacation model differs from the normal M/G/1 queuing model because when a message arrives at an empty queue in the vacation model it is not immediately transmitted as in the regular M/G/1 model; the message must wait until the token reaches the server for this queue. The first element of the wait

time is the delay before r is served. The second element of the wait time is the time needed to transmit the messages enqueued before the average message. The following definitions are needed for the model: λ is a Poisson message arrival rate; τ is the mean service time, and σ is the variance in the service time. Fuhrmann and Cooper derive the equilibrium cycle time to be

$$E(T) = \frac{c}{1-\rho} \quad (\rho < 1)$$

where c is the constant length of time for a token cycle with no traffic and $\rho = \lambda\tau$ is the server utilization. They then derive the equilibrium waiting time to be

$$E(W_1) = \frac{1}{2}E(T) + E(W_0)$$

where $E(W_0)$ is the waiting time in the queue derived from the Pollaczek-Khintchine formula and defined as

$$E(W_0) = \frac{\rho\tau}{2(1-\rho)} \left(1 + \frac{\sigma^2}{\tau^2} \right)$$

By the following substitutions the expected wait time formula can give an expression for the mean delay defined in terms developed in this chapter.

The delay due to the transmission of the succeeding messages can be expressed as

$$E(W_0) = \frac{\rho \cdot \tau}{2(1-\rho)} = \frac{\rho_r \cdot X_{m_r}}{2 \cdot (1-\rho)} = D_Q$$

Since the message size is constant the variance in the message transmission delay σ^2 will be zero.

The expected token cycle time $E(T)$ can be rewritten

$$E(T) = \frac{c}{1-\rho} = \frac{N \cdot X_T}{1 - \sum_{r \in R} \lambda_r \cdot X_{m_r}} = \bar{T}_C$$

The transmission delay for a message is the sum of the time needed to transmit the message, and the time needed for the message to propagate to the receiver. Assuming the average message propagation time is approximately equal to the interframe gap, the average delay in delivering a message can be expressed as

$$\bar{D} = \frac{1}{2}T_C + \frac{\rho_r \cdot X_{m_r}}{2 \cdot (1 - \rho_r)} + X_{m_r}$$

This expression for the average delay in delivering a message can be used to make observations about the general behavior of 802.4 networks. The minimum delivery time is bounded by the time required to transmit the message. When the network has lightly offered loads, small λ , most of the average delay derives from the contribution of the token cycle time, and will be approximately equal to one half the token cycle time. As the load increases more of the average delivery time is based upon the queueing delays caused by the load in the station.

3.6. A Real Model

The simple model developed in the preceding two sections provides a good description of idealized 802.4 networks. The formulas and expressions developed are very useful for setting the *Target_Rotation_Times* and other network parameters. However, the simple model is quite limited in its description of actual 802.4 network operations. Some of the problems with the simple model are discussed in this section.

The simple model assumed that there were no station failures or lost messages; the robustness of 802.4 is one of its major advantages, and the model cannot provide statistics on how well 802.4 meets the robustness criteria. Dynamic station membership was also not dealt with, so the varying overhead V was ignored. In the simple model it was assumed that all messages enqueued at a server are transmitted on each token cycle, but as can be seen in the expression for the token

cycle time, increasing the load will increase the token cycle time, and there are situations where that increase would be unacceptable. For example in an industrial control application, critical monitors may need to transmit their information every tenth of a second. The model cannot be used to provide anything other than steady state information. The model shows that if the load is changed then the token cycle time will, with time, adjust to the new load level, but how does a network react to transient loads? All of the above problems concern how the network reacts over time instead of how it reacts in the steady state.

Modeling a network's functioning over time is very difficult. As Steven Lavenberg points out in *Computer Performance Modeling Handbook* [Lavenberg 83]:

A queuing model is a dynamic probabilistic model; i.e. it uses probabilities to represent the evolution over time of a system. It is very difficult to analyze the transient behavior of such a model, i.e. the behavior as a function of time. The transient behavior is most readily studied via simulation. However it is often possible to analyze the steady-state behavior of such a model.

Instead of attempting to derive an analytic model for a limited number of exceptional cases, this thesis effort concentrated on developing a simulator that would allow the user to compare results with the simple analytic model and to explore the time dependent functions of 802.4.

CHAPTER 4

The Simulator

4.1. Introduction to the Simulator

To study the performance of 802.4 networks, a Pascal program was created that simulates the major functions and station states specified in the 802.4 standard. The simulator allows the user to create a network configuration, to simulate the operation of the network, and to generate traces of network actions and reports of network statistics. This simulator provides the network designer with a tool he can use to quickly create and test varying network configurations. The simulator was based upon the work done by Alex Colvin [Colvin 84] which had its origins in the simulator of David Albrecht [Albrecht 82].

The simulator is a 5000 line Pascal program called **tokbus** running on a Digital Equipment Corporation VAX 11/780 and on Apollo DN-300 workstations. While there exist programming languages which are specifically designed for simulation, Pascal was chosen as the development language because of programmer familiarity and the availability of Pascal compilers on the machines used for running the simulations. Another advantage was the ability to use the work of Colvin as the basis for **tokbus**.

Colvin created a Pascal program that simulated a subset of 802.4. In that simulator, the number of stations was fixed at the start of the simulation; there were no station arrivals, departures or failures. The Colvin simulator operated in an error-free environment, therefore it did not incorporate any of 802.4's error correcting capabilities. The simulator was designed to generate results to be compared with an analytic model of Colvin's subset of 802.4. It was not designed to

be used by anyone else other than its author.

The simulator designed as a part of this thesis used some of the basic structure of the Colvin simulator to which was added error generation and correction capabilities, dynamic station ring membership, and a simpler user interface. Some portions of the Colvin program were included directly, some portions were used after modification, and in other cases the Colvin program was simply used as a reference.

The second section of this chapter details the simulator's design. Section three covers the simulator's functions and capabilities. The fourth section discusses the limits of the simulator. Section five describes the validation testing methods. An appendix to the thesis contains the simulator's instruction manual.

4.2. Simulator Design

The design of the simulator required three choices: the selection of a method for performing the simulation, the selection of a representation of simulation time and station timers, and the selection of a user interface. The simulation method determined how the program represented network states, and how it made the transitions between the states. The goal in designing the user interface was to select a simple method for creating and running simulations.

4.2.1. Simulation Method

There are several methods for simulating queuing networks. The three methods investigated for use in this simulator were a finite state representation of the possible network states, a concurrent execution approach, and event driven simulation.

The finite state representation approach was not selected because of the difficulty of trying to represent the very large number of states in which an 802.4

network could operate. Any reduction in the number of station states used would simplify the simulator such that the results would not correctly model the performance of 802.4.

In the concurrent execution approach, the simulator would loop through all of the stations involved in a simulation for each time increment. If a station needed to perform any action in this small time period it would be able to perform the action. On most cycles, the stations would simply decrement any of the active timers and perform very few other operations, which is very similar to the actual operation of an 802.4 station. The simulator overhead required eliminated this approach. A related problem is deciding how fine a time increment to use.

The method chosen for the simulator's operation was event driven simulation. In event driven simulation, the transitions in network states are represented as events. An event consists of an event type, a time in the future at which the event occurs, and the address of the station where the event occurs. Some typical **tokbus** events are the arrival of a message in a server's queue, the completion of a transmission by the token holder, or the arrival of the token at the next station on the logical ring. Each event is scheduled to occur at a specific time in the simulator's future. The events are maintained in a time ordered queue. A new event is not added to the end of the queue, but rather inserted in time order. The next event is always at the head of the queue. The simulator's global time is updated to the time of each event as the event is dequeued, skipping periods of inactivity and concentrating the program's processing in the simulation of network transitions.

4.2.2. Time in the Simulator

The simulator's global clock is initially zero. As the simulator operates and events are dequeued, the global clock is set to the time of the last event. Because

more than one event can occur at the same point in time the clock is set to the time of the event rather than incremented. The global clock is available to all stations.

The 802.4 standard specifies that the timers in each station are countdown timers without wraparound; when the timer reaches zero it remains zero until reset by the station management software. The simulator represents the countdown timer in software by comparing the value in the timer with the global time. If the difference between the two values is positive, that is the time left in the timer. Negative differences indicate an expired timer. Timers are reset by adding an offset to the current global time; for example, when station 25 accepts the token and begins to serve its *Synchronous access_class*, the *token_hold_timer* is reloaded with the sum of the global clock and the *High_Priority_Token_Hold_Time*.

4.2.3. User Interface

The simulator uses menus and interactive questioning to lead the user through the complete simulation process. Menus are used when the user has more than two options for the next action or selection. Questions are used for binary selections or when the simulator requires a specific type of input with variable values, e.g. numbers.

When the user is required to enter information, the simulator will cycle until the user provides an appropriate response. Invalid input is ignored in most cases. The exception to the above statement is the inability of the simulator to ignore character input when it is reading a numerical value. This problem is inherent in the Berkeley Pascal used in the VAX implementation of *tokbus*. This restriction does not apply to the Apollo implementation, and the reasons why are explained in Appendix B.

The menu selection and question and answer approaches were chosen because the simulator should be an easy-to-use tool for network designers who should not have to worry about the exact syntax or order of commands.

4.3. Simulator Functions and Capabilities

The simulator can be used to perform three tasks for the network designer. He can create and save network configurations. He can run a simulation based upon a configuration, and he can generate reports of network and station status and utilization.

4.3.1. Network Configuration

To run a simulation, the user has to create and load a network configuration. A network configuration is created as the user enters values for a variable number of system parameters. After the values have been entered, the simulator must build the actual network configuration.

4.3.1.1. Configuration Parameterization

The first function of the simulator is to enable the user to easily create and modify configurations. There are two types of information needed by the simulator to construct a network configuration to be simulated. A method of saving and reusing configuration information was developed to reduce the configuration effort.

The first type of information is the set of network-wide parameters such as the *High_Priority-Token_Hold_Time*, the bus rate and the address size. It is also the set of values needed by the simulator to determine the actions of the simulator itself including the random number seed and the number of station classes.

The second type of information needed is the parameterization of the station classes. Each "physical" station on the simulated network belongs to a station

class. All stations that are members of the same station class have the same number of active *access_classes* with the same message creation rate and distribution, mean length and length distribution. The *Target_Rotation_Time* and *Max_Inter_Solicit_Count* values are the same for all class members, and the type of ring membership is constant in a class. The simulator uses the class structure for generating reports on message delays and ring membership. There can be stations from more than one type of station class on a simulated network; this feature allows the user to study networks consisting of nonhomogeneous stations.

To aid the network designer in quickly creating simulations and to prevent duplication in the configuration effort, the simulator allows the user to save and edit the configuration information. The system wide parameters and other global information can be saved in a file known as **DfitVals**. The user is given the option of using the defaults or creating their own values. The simulator also maintains a library of station class information in the file **ClassLib** from which the user can select classes to be used.

The files **DfitVals** and **ClassLib** are special files that are required for the correct operation of the simulator. Therefore, when the user requests the simulator to write any output file the simulator will not let the user use either of these file names. When saving class information, the simulator will query the user if he is about to overwrite an existing class file, but it will allow him to overwrite the file if he desires. The class information files are also treated as reserved when the user tries to write out reports or traces, see section 4.3.3.

Without the ability to modify the saved information, the default values and library of classes would be of little value. The user could run the same simulations again and again, but he could not study the effects of adjusting network or station parameters without re-entering the majority of the information. The simu-

lator provides the user with the ability to modify the saved information and newly entered information. The modification of the information just entered by the user allows a user to change mis-entered information. The editing method is explained more fully in the simulator's user manual in Appendix A.

After reading the default file or querying the user for the system parameters, the simulator displays the values and allows the user to modify any value. The modified or created values can then be saved, overwriting the old default values. After the default values have been set, the simulator will loop, for the number of classes specified by the defaults, reading station class information from the library or the user and allowing the user to modify and save the information.

As the user constructs the network configuration, the simulator performs some limited range checking. The user is constrained to selecting either the sixteen or forty-eight bit address modes, and the size of the token is lower bounded by the address size selected. It checks to be sure that the number of stations is less than the simulator limit of 512. The simulator does not allow station address conflicts. The simulator will not function if the user specifies a network without at least two initial members; there must be a logical ring on network startup, and the logical ring must be maintained throughout the simulation. These constraints are not designed to limit the user, but rather to establish minimums needed for correct configurations.

4.3.1.2. Loading the Configuration

After the user has entered the information for a station class, the simulator loads the class. For each station, the simulator constructs a record with elements (a) to record message statistics and maintain the queues of messages awaiting transmission for each active *access_class*, (b) to save its own address and the addresses of its predecessor and successor, and (c) to represent the *token_hold_timer*

and other timers and counters required in the standard, plus the station state.

As the simulator creates stations, it links them together into three lists. The first list consists of all stations that are members of the same station class; this list is traversed when generating class statistics. The second list is a circular, address-ordered list consisting of all stations. The third list is also circular and address-ordered, and it represents those stations that are members of the logical token ring.

If the station is not an initial member of the token passing ring it is not linked into the third list. Stations that are supposed to join a simulation in the future have an event scheduled that will awaken them and set their *in_ring_desired* flags and allow them to contend for ring membership. Stations that leave the token passing ring after some period of time have their departures scheduled and added to the event queue while they are being loaded. The station state is set to *idle* for all of the stations that are initial members of the token passing ring. The state of other stations is set to *unpowered*.

The first message arrival event for each of the station's active *access_classes* is scheduled when the station is created. If any of the station's queues are to be pre-loaded, they are loaded with messages.

4.3.2. Simulating

To run a simulation, the simulator must present the user with the ability to control the actions of the simulator, and it needs to schedule and manage the events representing the actions of the stations and the state of the network. As mentioned above, the simulator starts with a configured token passing ring, and the event queue will contain the pending message arrivals for each station.

After the last station class has been loaded, the simulator creates an initial token and passes it from the lowest addressed station to the station at the top of the ring. The event of the token's reception at the next station will be added to the event queue.

4.3.2.1. Controlling the Simulator

Once the simulator startup has been completed, the simulator will display the menu of Figure 4.1, the top level of the command structure, and wait for the user to select a command. At this level, the user is given several options for the commands that control the simulator's actions. After each command selection, if the user has not turned on the simulator's display capabilities, the main menu is displayed again to remind the user of the available commands.

The commands can be grouped into four types. The first type of commands are for running the simulator; they allow the simulation to progress by dequeuing and processing events. The second type of command is the report command; enter-

Enter Command Character :

```

Menu of Simulator Commands
D - Display statistics and stations;
E - set an Error condition;
H - Help by displaying this menu;
J - a station Joins the token passing ring;
K - Kill a station;
L - a station Leaves the token passing ring;
R - Report statistics;
S - run the simulator for a number of Steps;
T - run the simulator for a period of Time;
X - eXit the simulator.

```

Figure 4.1 Simulator Main Menu

ing the **R** command will provide the user with a selection of reports that can be generated and simulator traces that can be turned on or off. Commands of the third type control the state of the network; the user can alter the network state without altering the current time. The fourth type of commands control the simulator interface; they provide the user with displays that monitor network activity and can recall the main menu.

4.3.2.1.1. Running the Simulator

The **S**, **T**, and **X** commands allow the user to control the progress of the simulations. The **S** and the **T** commands run the simulator, and the **X** command exits the simulator stopping the simulation.

The **S** command steps the simulator by dequeuing a user specified number of events; one event is dequeued for each step. When the user enters the **S** command, he will be prompted for the number of steps he wishes the simulator to perform. After the simulator has dequeued and processed the correct number of events it returns to the command level. Entering a number less than one is a no-op, and the user is returned to the higher level command menu. Because there is no restriction on the number of events that can occur at a single point in time, stepping the simulator through a number of events will not necessarily cause changes in the global time or the user's view of the simulator's displays. Stepping the simulator is very useful when the user wishes to observe or trace simulator actions at some point in the simulation.

The user can run a simulation for a period of time with the **T** command. The user will be queried for the duration of time for which to run the simulator. If the user enters a value greater than 0, a `stop_simulation` event is scheduled at a time equal to the sum of the user entered value and the current global time. Running the simulator for periods of time is very useful for generating the

necessary traffic to provide accurate reports of message delays, and traffic loads by overcoming the effects of the simulator startup.

It is possible for the simulator to stop simulating before the specified number of steps or period of time has elapsed if the user has specified that the simulator should stop when a pre-loaded queue is emptied. When such a breakpoint is encountered, the simulator will offer the user the options of continuing the simulation, exiting to the command level, or turning off the breakpoint trap. The empty queue breakpoint feature allows a user to determine how long it takes a server to transmit the specified number of messages.

4.3.2.1.2. The Report Command

Entering the R command will transfer the user to the report sub-system. Figure 4.2 shows the menu of possible actions the report sub-system offers the user. The user can generate reports, toggle simulator traces, or return to the command level. This section will not discuss the body of the reports and traces which is the subject of Section 4.3.3.

Enter Selection :

B - Report Bus Statistics;
C - Report Class Statistics;
E - Turn On Execution Trace;
M - Turn On # of Messages Transmitted Trace;
Q - Turn On Queue Level Trace;
T - Turn On Token Trace;
X - eXit to Command Level;

Figure 4.2 Report Sub-system Menu

The reports display cumulative network statistics. The bus report provides statistics on network usage; the class report provides statistics on a station class' service. If there is more than one type of station class in the simulated network the user is given the option of generating reports on individual classes or all of the classes. The reports can either be displayed on the user's terminal or written to a file. If the user selects to have the report displayed, the screen will be cleared; the report will be displayed; the simulator will pause until the user has finished reading the report, and then either the main menu or the screen display will be redisplayed. If the user selects to have the report written to a file, he will be prompted for the name of the file. The procedure that opens the output file prevents the user from overwriting the files **DfltVals**, **ClassLib**, or any member of the class library.

The trace commands allow the user to turn on or off a trace of the simulator's execution, the number of messages transmitted by a station on each token cycle, the number of messages enqueued at a station when the station receives the token, and the duration of each token cycle. The traces allow the user to study the actions of the simulator over a period of time. When the user turns one of these traces on, he will be prompted for the name of the file into which the trace information will be written. As described above, the user is also prevented from overwriting any of the reserved files with traces.

4.3.2.1.3. Altering the Simulator State

The **E**, **J**, **K**, and the **L** commands can be used to alter the simulator state without changing the current time. The **E** command enters an error sub-system. The other commands query the user for the address of the station that should wake up and join the ring, be killed by the user, or be told to leave the ring.

The user has three options in the error sub-system. He can either kill the current message, set a noise condition of user specified duration, or return to the command level. The robustness of the protocol can be tested by selectively killing the current message and observing the network's actions. The noise bursts can also be used to study robustness. A noise burst lasts from the current time until the user specified delay has elapsed; no messages can be transmitted by any station during the noise burst. The noise burst is similar to the condition when a station with a faulty receiver pollutes the network with invalid or out of order transmissions.

When the user enters either a **J**, a **K**, or an **L** command, he will be asked to specify the address of the station he wishes to have join, be killed, or leave. If the user specifies a non-existent station, he will be asked for the address again. If he enters a zero for the station address the simulator will return to the command level.

The **J** command allows the user to specify that an unpowered station should power up and contend for ring membership when the next *Solicit_Successor* frame spanning the station's address is opened. If the station is already a ring member this command is ignored.

The **K** command kills the specified station. If the station is idle, then after it is killed it will be patched out of the token passing ring on the next token cycle. If the station is the current token holder, its death will cause a *bus_idle* event to be scheduled by its successor and cause a ring reconfiguration when the *bus_idle_timer* expires.

A station that is told to leave by the user will stop accepting new messages for its transmission queues and allow the queues to drain. As soon as the queues have been emptied the station will patch itself out of the token passing ring by

sending a *Set_Successor* frame to its predecessor before transmitting the token to its successor.

4.3.2.1.4. Simulator Interface

The user can modify the interface with the simulator by replacing the main menu with a display of the states of the stations on the bus, an in-depth display of a single station, the current elapsed simulator time, and the number of simulated token cycles. The command interface will question the user for the next command selection without displaying the menu. The user has the ability to turn the display on and off and to temporarily overwrite portions of the display with menus. The displays show the general state of the network and the specific state of a single station with its timers and message queues.

The user can control the display with the **D** command. Figure 4.3 shows the options available to the user after entering the **D** command. He can either turn the display on or off, specify that a particular station will be displayed in depth, specify that the in depth display will be the token holding station, or return to the command level. When the display is turned on, the main menu, Figure 4.1, is not displayed. If the user does not remember the command characters, the main menu can be temporarily displayed without turning off the display by entering the

Display Commands; Please Enter Selection :

O - turn On Station Display;
 S - give the full display for a Station;
 T - give the full display for the Token Holder;
 X - eXit to Command Level;

Figure 4.3 Display Sub-system Menu

H command. If the user selects any command with a menu of selections (the **D**, **E**, and **R** commands) the menu will overwrite the top portion of the display while the user makes his selection; the display will be refreshed after the selection has been made.

Figure 4.4 shows that there are four components to the display. On the top line there is the command prompt where the user will enter the next command character. The second line displays the elapsed simulation time and the number of token cycles that have been completed. The third part of the display starts on line 3 and extends to line 17; it shows the addresses of the bottom forty stations on the bus and their station states. When the token is at one of these stations an @ symbol is displayed to the left of the station's address and the state will be displayed as *usetokn*. The fourth part of the display is the in depth station display, which displays the station's address and station state. It displays the time

```

Enter Command Character :
elapsed time = 0.01050512          # of token cycles = 30
  20      21      22      23      24  @  25      26      27      28
idle     idle  joining  idle     idle  usetokn  idle     idle     idle
  29      30      31      32      33      34      35      36      37
idle     dead   idle     idle     idle  unpowrd  joining  idle     idle
...
      address = 25          state = use token
      0.00030871 seconds left for Synchronous service
Access Class      # Messages Queued      Token Rotation Timer
Synchronous              2
Urgent Asynchronous      4              0.01067810
Normal Asynchronous      2              0.01050327
Time Available           inactive

```

Figure 4.4 Sample Simulator Display

remaining in the *token_hold_timer* and the *access_class* currently being served, and it displays the number of messages enqueued at each active *access_class* and the *token_rotation_timer* for the non-synchronous classes. If an *access_class* is inactive that message is displayed. By default this element of the display is assigned to the token holder, but the user can specify the station which he wishes to have displayed if he wishes to observe the station while the token is elsewhere in the network.

4.3.2.2. Managing the Events

The user runs simulations by requesting the simulator to run for a period of time or a series of steps. The simulator performs this function for the user in a loop where events are removed from the event queue and processed by the correct event handlers. When the event handler or some other procedure requires some action to be performed in the future, it creates a new event and schedules it for the appropriate time.

As described in section 4.2.1, each event occurs at a specific time. Whenever an event is removed from the event queue, the simulator updates the global time to the time of the event and, after determining the type of event, calls the necessary event handler. When the simulator is being run for a series of steps or a period of time it loops through this process for the specified number of steps or until a stop simulation event sets the loop exit condition.

For the simulator to faithfully represent 802.4 networks it needs to create events that represent transitions in the state of the network being simulated. There are two types of events in the simulator. The first type of events represents non-802.4 specified events; this type includes message arrivals at a station, the delayed powering up of a station, or a stop simulation event specified by the user. The second type of events represents the 802.4 specified actions of a

station that is in one of the simulated 802.4 station states.

Whenever a message arrival event occurs, it specifies a station and an *access_class*. The message arrival event causes a new data message to be generated and added to the *access_class*' queue, and it causes the generation and scheduling of the next message arrival event for the station and *access_class*. There will always be pending message arrival events unless the station is scheduled to leave the token passing ring in which case it will not generate the new events. A message arrival can trigger changes in the state of the simulated network if the message arrives at a station that is powered up but not a member of the token passing ring; the arrival will cause the *in_ring_desired* flag to be set to true. The other non-802.4 specific events do not schedule other events; they are under user control.

The 802.4-specific events represent transitions in the state of the simulated network. Some of the transitions from one state to another require only a very small switching time; transitions like the changing of the current *access_class* or the powering up of a station are not modeled as events. The 802.4 events model transitions in the network state that occur over time. Typical 802.4 events are the completion of a message or token transmission or the expiration of the *bus_idle* or *token_pass* timers. The simulator will schedule the completion of these events and return to the main simulation loop. The uninteresting period from the start of a message transmission or timer until the end of the event is skipped, and the simulator can process any other events or perform needed actions that occur during the interval.

4.3.3. Reports and Traces

The simulator provides the user with the ability to generate reports of simulation results and traces of simulator actions. The details of how the user selects reports and traces were discussed in section 4.3.2.1.2. There are two different

reports that the user can select and four types of traces.

4.3.3.1. Reports

The two reports are the bus report and the class report. The bus report, Figure 4.5, provides the user with global data representing the network usage. The class report, Figure 4.6, can be used to determine statistics about the service obtained by the stations that are members of a particular simulator station class. The reports provide the user with cumulative, average, minimum, and maximum values from which information about the steady state operation of the network can be drawn.

The bus report shown in Figure 4.5 is an actual output file from the simulator. The version of the bus report displayed on the user's screen is identical to the figure except for the line giving the title of the report.

busout.6

Elapsed Time = 16.00000000

Token Cycles = 6023.

Average Token Cycle Time = 0.00265606

Minimum Token Cycle Time = 0.00111300

Maximum Token Cycle Time = 0.00446460

Total Traffic of 761333. messages for a total of 132981728. bits transmitted

Number of Ring Reconfigurations = 0

Type Of Message	Number Transmitted	Total Bits	%BandWidth
Data	374336.	95830016.	59.89
Overhead		35936256.	22.46
Actual Data		59893760.	37.43
Token	385525.	37010400.	23.13
Protocol	1472.	141312.	0.09
Corrupted	0.	0.	0.00
Propagation Delay			16.77
Bus Idle			0.11

Figure 4.5 Actual Bus Report

This simulation configuration was a variation of the basecase used in the performance testing. It consisted of 64 stations that were constant members of the token passing ring; 16 bit addressing was used, and the token was 96 bits; λ was exponential with a mean value equal to 366; the message size was a constant 160 bits. The offered load in this simulation was sixty percent.

The first line of the report is the title of the report and the file name under which it was saved. By having the name of the file visible in the report the user can easily determine which results belong to which file. The second line reports the elapsed time from the start of simulation until the report was generated. The third line shows that in the 16 seconds of simulation there were 6023 token cycles. The next three lines report the average, minimum, and maximum values of the token cycle time seen during the simulation. The seventh line of the report displays the total traffic in both messages and bits. If stations had left or joined the ring or the token had been lost then the number of ring reconfigurations would reflect those events. The remaining lines detail the network utilization during the simulation.

The utilization figures are based upon the user-selected bus rate of 10 megabits per second, and the duration of the simulation; in 16 seconds, 16×10^7 bits could have been transmitted. The "data", "token", "protocol", and "corrupted" categories list the actual number of bits transmitted by those types of messages, and the percentage of the total bandwidth they consumed. The propagation delay figure records how much of the bandwidth was consumed in message propagation delays. The bus idle figure is how often the bus was idle while stations opened response windows, and possibly the difference between the start of a message transmission and the generation of this report.

```

clsout.6
Class basecase
  Stations have joined the Token Passing ring      64 times.
  Synchronous Service Access Class
  Total Number of Messages sent from these queues =      374337.
  Average Queue Delay =                                0.00035376
  Minimum Queue Delay =                                0.00000000
  Maximum Queue Delay =                                0.00411018
  Average Access Delay =                                0.00098775
  Minimum Access Delay =                                0.00000000
  Maximum Access Delay =                                0.00411018
  Average Transmission Time =                          0.00002760
  Minimum Transmission Time =                          0.00002760
  Maximum Transmission Time =                          0.00002760
  Average Delivery Time =                              0.00136911
  Minimum Delivery Time =                              0.00002762
  Maximum Delivery Time =                              0.00436294

  Urgent Asynchronous Service Access Class
  Access Class Inactive

  Normal Asynchronous Service Access Class
  Access Class Inactive

  Time Available Service Access Class
  Access Class Inactive

```

Figure 4.6 Actual Class Report

The class report pictured in Figure 4.6 is another output file generated by the same simulation as described above in the description of the bus report. Because of the limits of most terminals, the screen version of the class report is much more condensed with short identifiers to the left of four columns of the delay statistics instead of one statistic per line.

Like the bus report the class report prints the name of the output file as the first line of the report. The second line displays the name of the class being reported; if there is more than one class involved in the simulation, the user can determine which statistics belong to which class. The third line tells the user how

many times stations of this class joined the token passing ring. In this case each station in the class joined once, but the user can determine how many times stations with dynamic ring membership joined and left the ring.

After reporting the membership statistics, the class report displays statistics on the delays experienced by the messages transmitted by the stations of this class. For each of the active *access_classes*, the name of the *access_class*, the number of messages transmitted, and the two delays and two times are reported. Inactive *access_classes* are denoted with the "Access Class Inactive" message. The Queue delay reports the delay from when a station entered the queue until it reached the front of the queue. The Access delay is the delay from when a message reaches the front of the queue until the message's transmission begins. The Transmission time reports how long it takes the message to be transmitted, and the Delivery time reports the total delay from message arrival in a queue until the completion of its transmission.

4.3.3.2. Traces

The four types of traces are the execution, number of messages transmitted, queue level, and token cycle traces. Traces are very useful for studying the transitions in a network during a simulation. The study is useful for the naive user who wishes to gain an understanding of the operation of 802.4, and it is useful for the experienced user who wishes to study particular network actions in response to certain events.

The execution trace records the times of message transmissions and token receptions. It records the time that an inactive station powers up or the time at which an idle station leaves the ring. It records all error conditions and ring reconfigurations. It records the time at which each active *access_class* is checked for queued messages. Following an execution trace while an identical simulation is

being run can allow a novice user to gain a better grasp of the operation of 802.4. The user is warned about turning the execution trace on and requesting a large simulation time; the resulting file can grow quite large, quite quickly.

The simulator will report the number of data messages transmitted by a station each time it is the token holder if the message trace is turned on. The trace also reports the address of the station and the time at which the token arrived at the station. The number of token cycles is reported each time the token is transmitted by the station with the lowest address. This trace can be used by the network designer who wishes to determine the number of messages transmitted with limited *High_Priority-Token_Hold_Time* and *Target_Rotation_Times*.

If the network designer wishes to trace the queue lengths, he can use the queue trace. The queue trace reports the station's address, the time at which the token arrived at the station, and for each *access_class* the number of messages enqueued for active classes and an "inactive" message for inactive classes. The token cycle count is reported on each token cycle. This trace can be used to monitor the queue levels in the individual stations to check for service starvation that might not be reported if the other stations are lightly loaded.

The token cycle trace records the token cycle number, the duration of the last token cycle, and the time at which the lowest addressed station passed the token back to the top of the logical ring. It can be used to determine the effects of transient loads by observing how long it takes the simulator to return to an average token cycle time after the application of a transient load.

4.4. Simulator Limits

The simulator is a powerful tool for the network designer, but it does have some limits. There are two types of limits with the simulator **tokbus**. Limits of the first type stem from the limitations of the Pascal programming language. The

second type of limits is due to the simplifications to 802.4 made in designing the simulator.

4.4.1. Pascal Limitations

As mentioned in section 4.1 there exist programming languages that have been designed for use in simulation, but Pascal was still chosen as the development language for the simulator. The limits Pascal imposed on the simulator are input and output and string size limits with the VAX implementation and numeric precision and the lack of a random number generator in Apollo's DOMAIN Pascal®.

The Berkeley Pascal used to implement the simulator on the VAX has several limitations to its Input/Output system. Whenever an exceptional condition is raised in the I/O system the program ceases execution; there is no method of trapping and handling these errors within the program itself. The I/O exceptions that can cause the premature termination of **tokbus** are a request to open a non-existent file, and the reading of non-numeric data during a numeric read. To satisfy the requirement that a file must exist before it can be read the simulator must have a file with the name **ClassLib** in its current working directory before attempting to read or save station class information. The first time the user creates a simulation he must make and save the default values because an attempt to open the file **DfltVals** will cause the simulator to terminate. The problem of dealing with character input while reading numeric values limits the simulator because an unwary user could cause the simulator to terminate with a misplaced keystroke or the common substitution of a letter 'o' for the numeral '0'. Unlike the requirement that dummy files be present in the working directory before program execution, the numeric problem requires user awareness and handling of the problem.

Apollo and DOMAIN are trademarks of the Apollo Computer Inc.

The string size limitation due to the Berkeley Pascal definition of type `alfa` to be ten characters long prevents the simulator from using long file names that could be more descriptive or could include pathnames that would allow the simulator to reference or write to files in other directories. The type `alfa` is required for the reset and rewrite functions needed to open files for input and output.

The implementation of Pascal on the Apollos provided strings of arbitrary length and the ability to trap errors when opening files. What the DOMAIN Pascal lacked was a random number generator, and numeric precision in its simple real and integer types.

DOMAIN C does provide an integer random number generator which was used to generate real random numbers, and DOMAIN C routines can be linked to DOMAIN Pascal to provide a random number generator for real numbers. One problem with the implementation of this random number generator is that the DOMAIN C random number function will return a value of zero, and `tokbus` requires that all random numbers be in the range $0 < X \leq 1$. To counter this problem a value of 10^{-15} is added to each real random number generated. Therefore the distributions are always biased by this small amount.

The limited precision of the real and integer types is due to the processor used in the Apollos. To obtain the necessary precision for real values the type `double` was used for all floating point variables. The limit of 32767 as a maximum integer value was an inconvenience that could have been circumvented through the use of `long_integers` but was not because the precision of the integers was determined to be sufficient.

4.4.2. Simplification of 802.4

In the discussion of the analytic model, section 3.5, it was stated that simulation allowed the network designer to analyze systems that were too complex to be

modeled analytically because the simulator would be free of the simplifying assumptions needed to create an analytic model. To simulate 802.4 some simplifying assumptions have been made which reduce the number of situations that can occur but do not reduce the functionality of the simulator.

The assumption that all addresses are unique has already been mentioned in section 4.3.1.1, and it is indicative of the types of simplifications made; errors or states that have no effect on the operation of the simulator are either not allowed at startup, ignored, or collected into a single state. Duplicate addresses in 802.4 can be a problem at most once in a network's operation; two stations would attempt to join the ring with identical addresses, and the one that lost the resolution process would take itself *Offline* and report the duplicate address to the station management software. One time, the resolution process would consume all of the maximum time due to sequencing through all the windows after which that situation would never occur again and is therefore of little interest in observing the behavior of networks.

Another state that is not allowed in the simulator but is allowed in the 802.4 standard is a ring with one member. The simulator requires that there are at least two stations that are part of the token passing ring so that there will always be events to process and time can progress.

A related state that is ignored is the formation of the initial token passing ring. When *tokbus* begins the simulation process it starts with a configured ring of stations with empty message queues and expired *token_rotation_timers* (unless the user preloads the queues when creating the configuration). The initial token is generated at the lowest addresses station and is passed to the highest station before the simulation begins.

The loss of the token in a configured token passing ring is an example of a series of events that have been collected into one process. The actual action of an 802.4 station when the token is lost was described in section 2.2.4.1. The simulator does not model the transmission of *Claim-Token* frames or the detection of those frames from other stations and the changing from the *Claim-Token* to the *Idle* state in all of the contending stations as the contention process proceeds. Instead the simulator schedules a *ring_reconfiguration* event for a time in the future equal to the time needed for the full contention process, increments the bus protocol frame statistics for the equivalent number of messages, and modifies the pointers to reflect the reconfigured ring. The same reconfiguration process is used when more than one station could respond to a *Solicit_Successor* frame.

The simplifications described above were made because the ability to handle the situations was deemed unnecessary for the use of the simulator as a tool to test the performance of 802.4. The situations described above should be rarely encountered in actual network implementations.

4.5. Testing Methods

Two methods were used to test the program and its validity. A series of tests was run on the program testing its abilities to handle extreme values and to generate results matching the results predicted for certain network configurations. It was also tested through constant use in generating the results of the performance tests that are the subject of Chapter 5.

The extreme value testing was conducted through the use of input files that were created with the knowledge that they contained invalid inputs followed by the correct inputs, and some files that generated degenerate network configurations where the results of executing the simulator reading from the specified file should be failure. The execution trace was used extensively to guarantee that the

simulator was correctly modeling the actions of 802.4 and that specific inputs were producing the correct actions and outputs. Through the use of shell scripts that could feed the simulator the correct series of input files, the testing process could be repeated as new versions of the program were created or updated. The Unix `diff` program was used to highlight differences in the versions of output files.

The values that were tested were known to be difficult cases for the simulator. This knowledge was derived through the development process and preliminary testing. The testing literature is undecided about whether it is better to target tests for situations the designer knows to be troublesome or to simply exercise the code in general. Because of the limited time available for this project the former method was used in the validation testing, and the general exercising of the code came during the performance testing. Any problems or deficiencies noted during the performance testing were corrected; the validation tests were supplemented by the new cases, and the supplemented validation tests were run again.

CHAPTER 5

The Performance of 802.4

5.1. The Performance Studies

The major focus of this thesis was to study the performance of various possible 802.4 network configurations by observing the sensitivity of the protocol to various parameters such as the number of active stations, the distribution of the load among the *access_classes* and the timer values. Another test of great interest was the performance of 802.4 networks under conditions of high transient loads and in error prone environments. The tests were performed using the simulator discussed in Chapter 4.

The second section of this chapter discusses the testing methods used to generate the reports discussed in the following sections. In the third section results for the base configuration are presented and compared with the results of the analytic model developed in Chapter 3. The fourth section discusses the testing of sensitivity of the protocol to the number of stations and the size of packets. The fifth section of this chapter describes the effects of dividing the load among the *access_classes* and the effects of changing the values of the *High_Priority-Token_Hold_Timer* and the *Target_Rotation_Timers*. The sixth section describes the studies of nonhomogeneous loading of the stations. The effects of transient loadings and error conditions are discussed in the seventh section.

5.2. Testing Methods

To accomplish the performance studies, the simulators on both the VAX and the Apollos were used to generate reports and traces that were then processed to

provide the data used in evaluating the performance of the protocol. Limited use was made of the VAX because of the high loads under which it normally runs. The Apollos were used much more extensively because two nodes were dedicated to this use and other nodes were also frequently available.

There were several issues in determining the testing method. The main issues were the design of the tests, the organization of the tests and testing process, and the presentation of the results.

5.2.1. Designing the Tests

There were two parts to designing the tests. The first part was deciding what information or series of commands the simulator would need to perform the desired tests. Secondly, what factors must be taken into account to ensure the validity and usefulness of the generated results.

The simulator requires the user to provide certain information that is used in configuring and running the simulations. The details of this information are presented in Chapter 4 and in Appendix A and the discussion here assumes the reader is familiar with the terms. Chapter 3 provides the network designer with useful formulas to employ in setting configuration values. In addition to establishing configurations that correctly test protocol sensitivity to the desired parameters, the type of results had to be selected. To study queueing delays the class report of Figure 4.6 was used. To study bus utilization the bus report of Figure 4.5 was used. The studies of the effects of transient loads used the token traces described in section 4.3.3.2.

To test the effects of changing a configuration parameter, as many factors as possible should remain constant between simulation runs, so it is advisable to use the simulator's ability to save and modify configuration information (see section 4.3.1.1 and Appendix A section 3.1). For example, to test the effect of the offered

load in messages per second on the average message delays, for each of the percentages of offered load from 10 to 100 percent, the basecase file, shown in Figure 5.1, had the mean message creation rate modified by replacing the base mean message creation rate (10% load) with the base mean message creation rate multiplied by 2 for a twenty percent load, 3 for a thirty percent load, etc. This method of varying the offered load was used in all simulation tests.

As another example to test the effect of varying the number of stations, section 5.4.2, new station classes were created by modifying the basecase station class. In the modified station classes instead of having 64 stations with a mean message creation rate of 61 messages per second the new classes had number of stations

```

      Editing the Class Information
1 - Name of Class = basecase
2 - Number of Stations = 64
3 - Priority Option is Not Used.

      Synchronous      Urg Asynch      Nml Asynch      Time Avail
      4 - active        9 - inactive    15 - inactive    21 - inactive
Mean Msg Crt Rate : 5 - 61.00      10 -             16 -             22 -
Msg Crtn Distrib  : 6 - expont      11 -             17 -             23 -
Mean Msg Length   : 7 - 160         12 -             18 -             24 -
Msg Lngth Dist    : 8 - constnt      13 -             19 -             25 -
Target Rotation Time :              14 -             20 -             26 -

27 - Stations are Always Members of the Token Passing Ring.

29 - Stations open response windows after every 255 token cycles
30 - Low Address = 1              31 - High Address = 64
32 - The Queues are not PreLoaded.

```

Enter the Number of the Field to be Changed (0 to stop) :

Figure 5.1 basecase as Displayed by Simulator's Editing Function

ranging from 8 to 256 with corresponding mean message creation rates ranging from 488 to 15.25 messages per second.

To obtain valid results, the tests had to simulate realistic network configurations, the tests had to run for sufficient periods of time to compensate for transient effects of simulator startup and shutdown and representative data should be generated.

The simulator will not prevent the user from simulating infeasible or nonsensical network configurations; simulating the action of a two station network where the stations are always members but have no messages arriving and no messages preloaded could be performed, however the results of the test would have limited value. The user must be aware of the limits under which successful networks could be configured. The offered load in bits per second should not exceed the capacity of the transmission medium. The *Target_Rotation_Time* values should not be less than the time required for passing the token around the logical ring. The formulas of Chapter 3 can be very useful to the network designer in aiding the determination of the limits to values such as the *High_Priority_Token_Hold_Time* or the *Target_Rotation_Time* values.

There are two transient effects in the simulator, both dependent upon the queues of messages awaiting transmission. The first effect arises when the simulator is started. At startup, the *token_rotation_timers* have expired and the queues are empty unless the user has specified that they should be preloaded. If the queues are not preloaded then usually several token cycles will pass before messages begin to arrive in the queues. If the queues are preloaded then the access and queueing delays of the preloaded messages in the stations at the top of the ring will be shorter than the delays for the messages at other stations. The second transient effect of the enqueued messages is due to the messages that are waiting

transmission when the simulator is stopped. If the traffic on the network is bursty, then the failure to transmit these messages may reduce the reported bus utilization.

To overcome the transient effects, simulations should run for long periods of time so that the averages reported in the bus and class reports will be more indicative of actual network operation. All simulations that were used to generate the data used in these performance reports were run for 16 seconds of simulated time.

The value of 16 seconds was chosen after a series of simulations of the base network configuration (see section 5.3, Appendix C) were performed for simulation times of 1, 2, 4, 8, 16, and 32 seconds. The results obtained for the series of simulations were compared to determine how much simulation time was required to overcome the transient effects; the statistics observed were the percentages of bus utilization, the average delivery times and the components of the average delivery times for messages. The mean values of the above statistics were computed. The system was assumed to be stable when the differences between the mean values and the reported statistics produced by increasing the simulation time were less than one percent. The 16 second duration was chosen because it met the criteria above for all cases except when the offered load was greater than 95% of the bus capacity, and because of infinite queue growth, those systems will never stabilize.

The performance studies test the sensitivity of the protocol to changing network parameters. Before the values generated by the simulator could be used to analyze protocol performance, the sensitivity of the simulator itself had to be examined. The two areas of sensitivity that were examined are sensitivity to the duration of simulation time and the sensitivity to the random number seed. The sensitivity to simulation time was discussed above.

The sensitivity to the effects of different random number seeds was determined by running a series of simulations where the only parameter that was varied was the random number seed. Fifteen random number seeds varying from 1 to 120258 that were themselves selected randomly were tested. Simulations were run for networks operating at offered loads of 10, 40, 50, and 90% of the bus capacity. Again, the percentages of bus utilization and the average delivery times were compared to the mean values. The differences between reported values and the means were less than one tenth of a percent in all cases. These small differences show that the dependence upon the random number seed is very small and that the results achieved in the performance studies are representative values from the distribution of possible results for different seeds.

5.2.2. Test Organization

A method was needed to organize the tests and testing process such that the testing process would be as automated as possible and the results of a series of simulations could be grouped together for latter processing. It was desirable to automate the simulation process because of the very large number of tests needed by the performance study. The large number of tests with the resulting large number of output files made result organization necessary.

While the simulator **tokbus** was designed to be used interactively, the simulation process was performed in a batch style through the use of shell scripts like the one in Figure 5.2 and simulator command files, Figure 5.3. The program **tokbus** is reading from standard input which is being fed from the files **sim1** through **sim10**. As discussed below, the **sim** files contain the information to drive the simulator. The standard output of the simulator is being dumped to the file **sout**; this output is not important in this application of **tokbus**, so it can be discarded. Multiple runs of the series of simulations can be invoked by executing the

```
tokbus <sim1 >sout
tokbus <sim2 >sout
tokbus <sim3 >sout
tokbus <sim4 >sout
tokbus <sim5 >sout
tokbus <sim6 >sout
tokbus <sim7 >sout
tokbus <sim8 >sout
tokbus <sim9 >sout
tokbus <sim10 >sout
```

Figure 5.2 Simulator Shell Script

shell script.

The simulator command file shown in Figure 5.3 consists of the same series of commands that the user would enter from the keyboard if the simulator was being used interactively. The C style comments have been added to clarify the inputs. The simulation being run by this file will use the system parameters saved in the file **DfltVals**. The only station class is **basecase**, and it will be used with the mean message creation rate modified to 366 (60% offered load). The modified file is not saved. After the sixteen seconds of simulation, a bus report is generated and written to the file **busout.6** and a class report is written to the file **clsout.6**. This simulation input file would produce the reports of Figures 4.5 and 4.6.

The shell script of Figure 5.2 and the simulator input file of Figure 5.3 are examples of VAX scripts and command files; the different formats of Apollo scripts and input files and the reasons for the differences are explained in Appendix B.

The hierarchical file structuring of the UNIX® file system was used to organize the simulation process and results; the results of each test were saved under a single directory which was the child of a directory that described the common type of tests. To simplify the actual testing process and to take advantage of the

```

y          /* use default values          */
0          /* no changes                  */
r          /* read a file                 */
basecase   /* select basecase from library   */
5          /* change field 5, see Fig 5.1   */
366        /* change to 366                 */
0          /* no more changes                 */
n          /* do not save modified file     */
t          /* run simulator for a period of */
16.0       /* 16 seconds                     */
r          /* select report sub-system        */
b          /* select bus report                 */
w          /* write report to file           */
busout.6   /*
r          /* select report sub-system        */
c          /* select a class report             */
w          /* write report to file           */
clsout.6   /*
x          /* exit the simulator            */

```

Figure 5.3 Simulator Input File

distributed processors of the Apollo network, each directory contained links to the pertinent class library and default values files and a link to the simulator itself. The simulator could be run simultaneously on several nodes simulating several variations of network configurations.

For example the tests of the number of stations are grouped under a directory named **nstations** and the results of each simulation are grouped under **8stations**, **16stations**, **32stations**, etc. The directory **8stations** contained the links to the files **base8**, **ClassLib**, **DfltVals**, and **tokbus**, and it contained copies of the simulation input files and a submit script. At the shell level the directory was changed to **8stations** and the submit script was executed.

5.2.3. The Presentation of Results

The reports and traces generated by the simulator provide the user with data from a single simulation. The reports provide informative legends for each statistic reported. There are two ways to study the results. In the first approach, the user could save copies of simulator reports and compare the data by reading the pertinent sections of a series of reports. A better approach, which was used in this thesis, was to employ UNIX tools such as `awk` and `leroy` to process the information and present it in a more concise, often graphical, form.

The `awk` editor allows the user to process a stream of input searching for patterns or conditions and performing actions based upon the pattern recognized. To generate the delay curves like the ones in Figures 5.6 and 5.9, the class report files were fed through `awk` which was instructed to filter the input and to produce as output the numerical value of the average delivery time reported in the class reports.

The `leroy` plotting package was used to plot the test results. It was used to draw the legends and scales of the result figures, and it generated the curves based on the simulator output. The `awk` processed data could be used as points to be graphed with the `leroy` plotting command of `graphxy`. Graphical presentations of the test results allows the reader to gain a more complete understanding of the performance of 802.4 than would be possible with tabular data.

As stated above, simulations were run for offered loads varying from 10 to 100 percent of the bus capacity; the offered load was incremented by 5 percent to create 19 test cases. In many of the graphs all 19 points are not displayed because attempts to display the delay values at some of the higher loadings would distort the scale of the graphs and obscure variations in the displayed values of the majority of the results.

5.3. Base Configuration Results

The base configuration was chosen to recreate the base configuration used by Alex Colvin in his modeling of 802.4 [Colvin 84]. The details of the configuration are in Appendix C, but the reader should be aware of several facets of the configuration. It consisted of a single class of stations with 64 stations that were always members of the token passing ring. The addresses were 16 bits, and the token was 96 bits long. The message creation distribution was exponential, and the message length was a constant 160 bits.

In studying the base configuration, there are three items of interest. The first item is the relationship of throughput, which measures the number of bits transmitted per unit time, to the offered load. The second relationship discussed concerns the effect of offered load upon the average delay in delivering a message; the effect of the varying load on the three components of this delay is also discussed. The third part of this discussion is the relationship of the offered load and the average token cycle time.

5.3.1. Throughput and Utilization

Figure 5.4 illustrates that the throughput or utilization achieved in the test, the solid black line, was almost exactly equal to the theoretical maximum values, the dashed diagonal line running from the lower left to upper right corner of the graph. At ten percent offered load at most ten percent of the network capacity can be consumed by the data transmissions. As the offered load increases to one hundred percent of the bus capacity, the throughput does not also go to one hundred percent. The maximum throughput must be less than one because of the need to pass the token and the time consumed in message propagations and other protocol traffic.

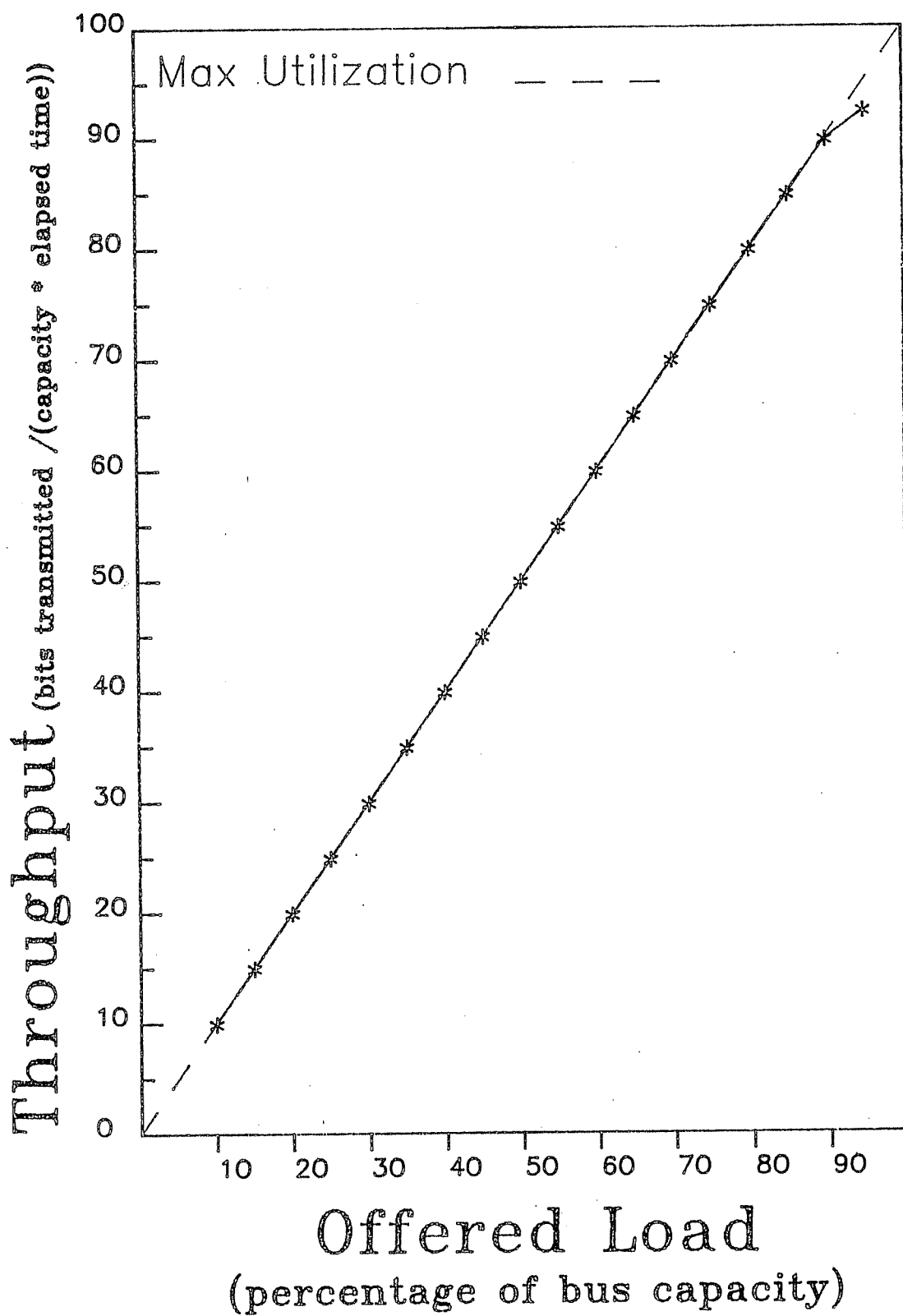


Figure 5.4 Base Configuration; Throughput vs. Offered Load

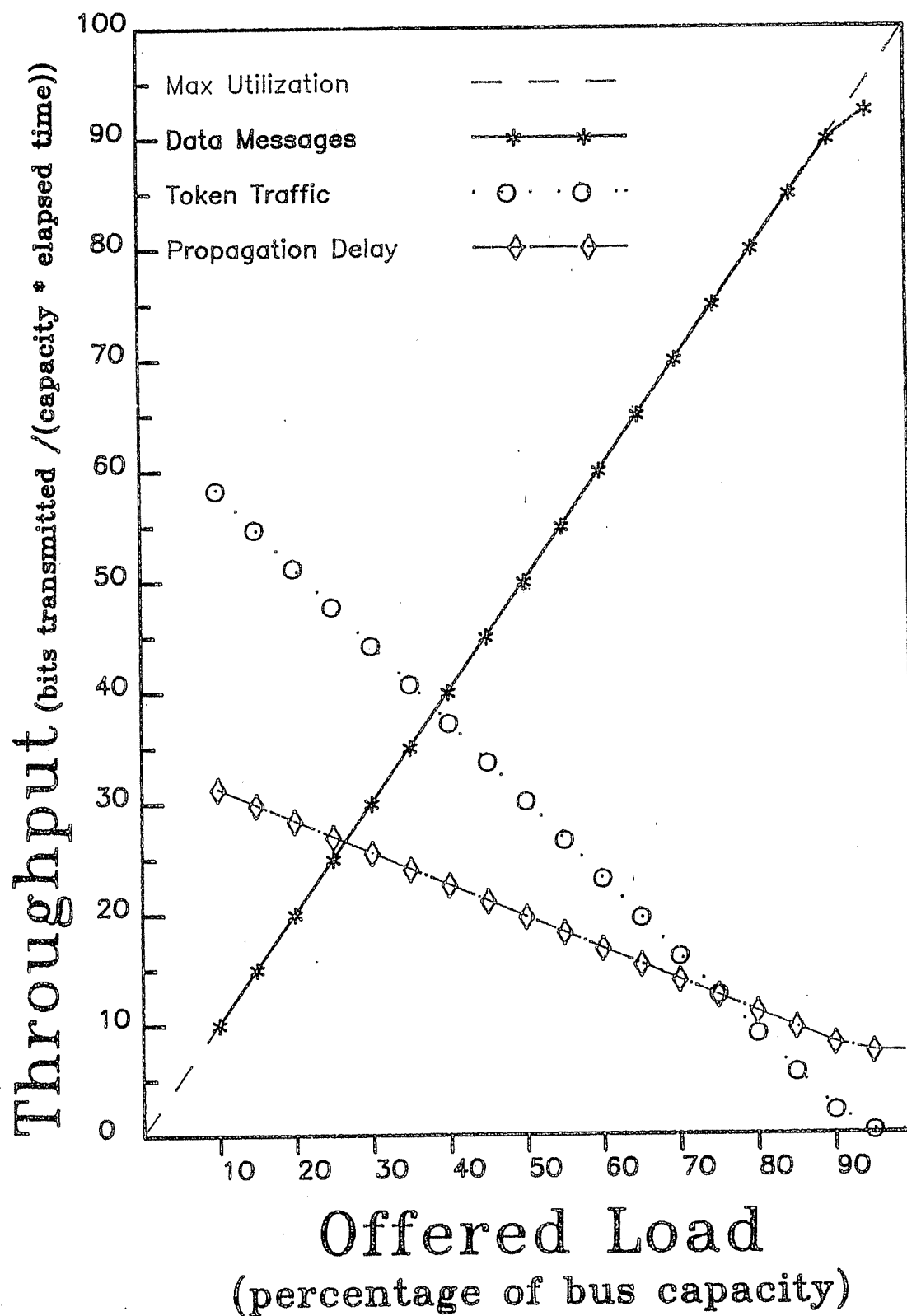


Figure 5.5 Base Configuration: Components of Bus Utilization

In 802.4, whenever there are no messages to send the token should be passed to maintain the logical ring; except for brief response windows the bus should never be idle. Three components of the total utilization of the bus are shown in Figure 5.5. The components are the percentage of bus capacity consumed by data transmissions (including all message framing bits), token transmissions, and propagation delays. There are other components of bus utilization that are not presented in Figure 5.5, such as the percentage of bus capacity consumed by corrupted frames, other protocol transmissions, and idle time during response windows. These components do not consume significant portions of the total bus capacity.

At low offered loads, token passing and propagation delays consume most of the bus capacity. As the load increases, more of the bandwidth is consumed by message transmissions. At low loads many token transmissions may be required before the token is received at a station with an enqueued message. With moderate offered loads, fewer token transmissions are required before a station with an enqueued message receives the token. For very high offered loads, each station has several messages to transmit every time it receives the token. In all of the cases, as the load increases token traffic decreases.

The amount of bus capacity consumed by propagation delays also decreases as the offered load increases as seen in Figure 5.5. There are two factors responsible for this decrease. The first factor is the different sizes of the tokens and the data messages; when transmitting the larger data frames more of the bandwidth is consumed before transmission stops and the next propagation delay occurs. The second factor responsible for the decreasing amount of bandwidth wasted by propagation delays is the different possible values for the propagation delay itself; during a token transmission, no new message transmissions can commence until the token is received at the next station, and it begins transmitting. After a data message

transmission, the sender only has to wait two microseconds, the interframe gap, before beginning the next transmission.

5.3.2. Delay vs. Offered Load

The curve of Figure 5.6 displays the average time, in milliseconds, between a message arrival in a station's queue and the completion of the message transmission. This average delivery time plot was generated by the average delivery times reported by a series of simulations. Figure 5.7 decomposes the delivery time into its component parts.

The delay versus offered load curve of Figure 5.6 illustrates the ability of 802.4 to perform well under low to moderate loads, but when driven to the limits of the communication medium, performance declines as queueing delays increase exponentially. The rapid increase in the average delivery time shown in Figure 5.6 corresponds to the increases predicted in section 3.5.1; as the load increases the average token cycle time increases contributing to an increase in the delivery time, and as the token cycle time increases more messages arrive which requires more time for message transmissions increasing the delay for the average message.

Figure 5.7 illustrates two of the three components of the average delivery time and the average delivery time itself. The two components illustrated are the average queue delay and the average access delay. The average transmission time is not shown for two reasons; it depends upon the size of the messages, not the offered load, and in this case with the small, constant message sizes it is a trivial contributor to the delivery time.

The average queue delay measures how long messages wait in the transmitting station's queue before reaching the front of the queue. At low loads, most messages are the only ones awaiting transmission at the station when it receives the token. Therefore the average queue delay is very small. At high loads, there are

two factors increasing the average queueing delay. The first factor contributing to the increase is that as the load increases, the average message is less likely to be the only message enqueued at a station, and it will therefore have to wait longer to be served even when the station is holding the token. The second cause of the increase is the increased delay before the station receives the token and begins to serve the queue. The queueing delay can be seen to be increasing dramatically once the offered load passes seventy percent of the bus capacity.

The average access delay measures the average of the time from when a message reached the front of the station's queue until the message's transmission began. At low loads most of the delay incurred by a message is due to the access delay; the message may arrive when the token is elsewhere in the logical ring, and it will remain enqueued at least until the station receives the token. Comparing Figures 5.7 and 5.8 illustrates how closely the average access delay curve matches the average delivery time curve at low offered loads. The average access delay is approximately equal to one half of the average token cycle time for low loads, as was predicted in the description of mean delay in section 3.5.1.

It is interesting to note that as the load increases, the contribution of the access delay to the total delay in delivering a message does not increase as dramatically as the queueing delay. This small increase can be attributed to the fact that while the message at the front of the queue has a larger access delay than a message in a lightly loaded network, the messages that follow the first message have access delays equal to the interframe gap which is small. Averaging the few large values with the many smaller values produces a moderate value.

5.3.3. Token Cycle Time

The graph of the average token cycle time displayed in Figure 5.8 shows that as the load was increased the token cycle time also increased. Because the value

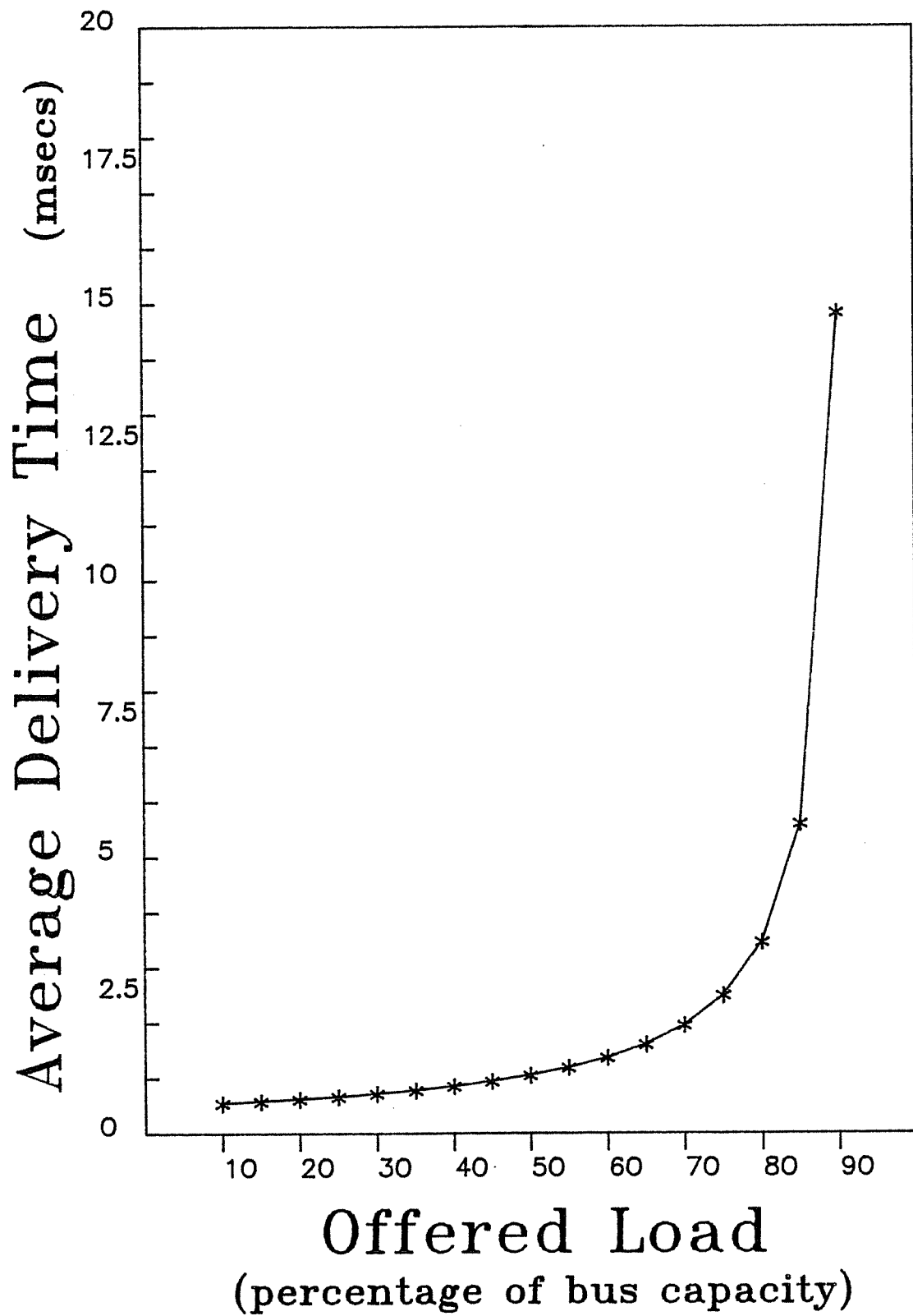


Figure 5.6 Base Configuration: Delay vs. Offered Load

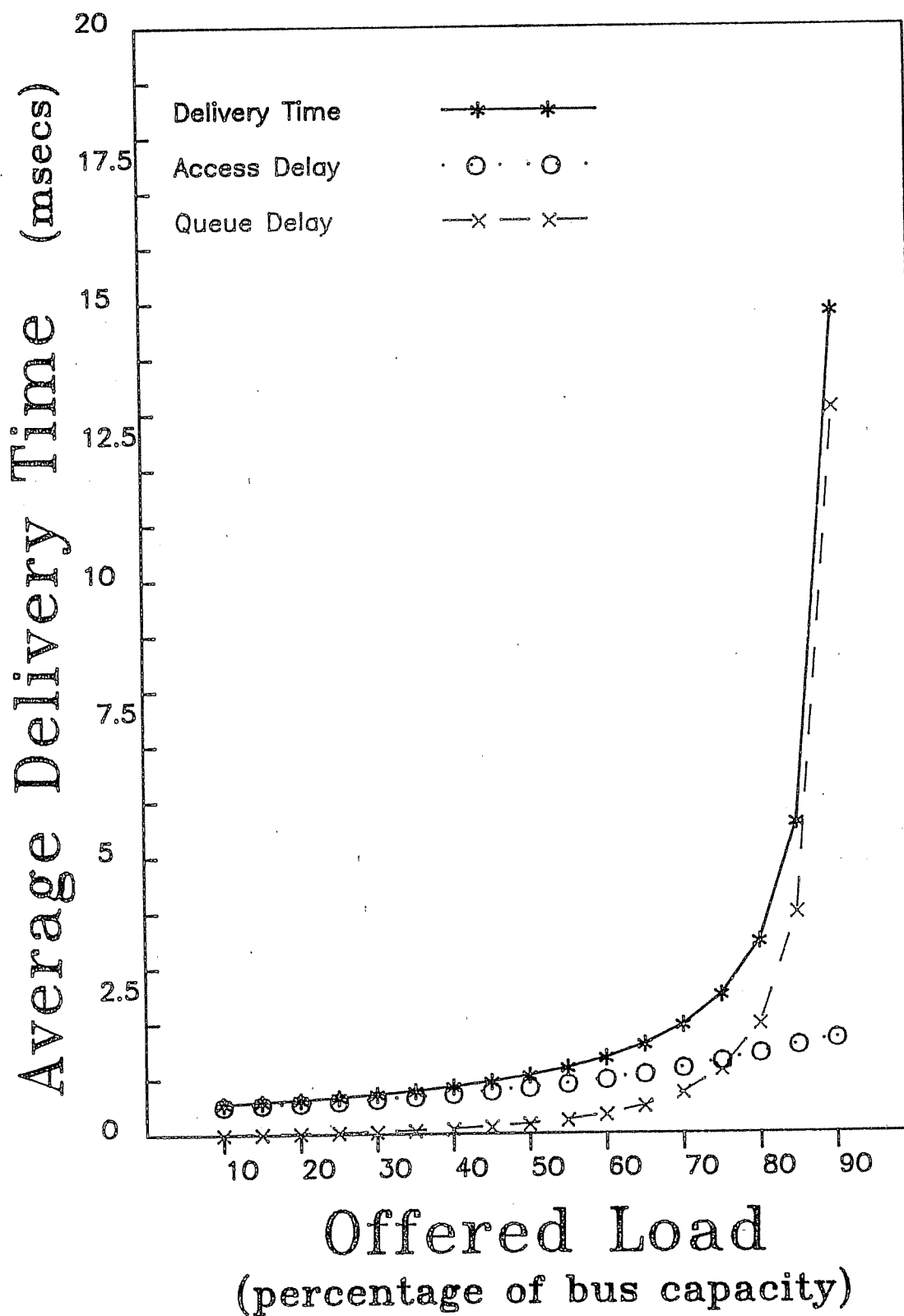


Figure 5.7 Base Configuration: Delay Components and Offered Load

of the *High_Priority-Token_Hold_Time* was set to an effective infinity (ten seconds) in this series of simulations the token cycle time was not (effectively) bounded from above. Each station would hold the token until its queues had been emptied. As the load was increased, each station held the token longer, increasing the average token cycle time. The token cycle time is obviously bounded from below by the time required for the 64 token transmissions needed to pass the token around the logical ring.

5.3.4. Observations

With the information displayed and discussed above, it is possible to make some simple observations about the performance of 802.4. Observations can be made about throughput, utilization and message delays.

As expected, throughput is very nearly equal to the maximum possible throughput until very high offered loads are reached. The penalty paid for this nearly perfect throughput is the exponential growth in the average delivery and token cycle times as the load increases in the absence of any token timer restrictions.

Because of the large values set for the *Max_Inter_Solicit* count and the constant ring membership of all of the stations, very little of the available bandwidth is consumed by protocol frames. Most of the bandwidth is available for any message which needs to be transmitted. At low loads most of the bandwidth is consumed by token transmissions and propagation delays but most of the bandwidth is available for message transmissions.

At low loads stations have to wait, on the average, half of the token cycle time before they can transmit a message on a bus that is mostly used just for token passing. In these situations a simpler access protocol could reduce the average delivery time to the minimum of message transmission time plus the

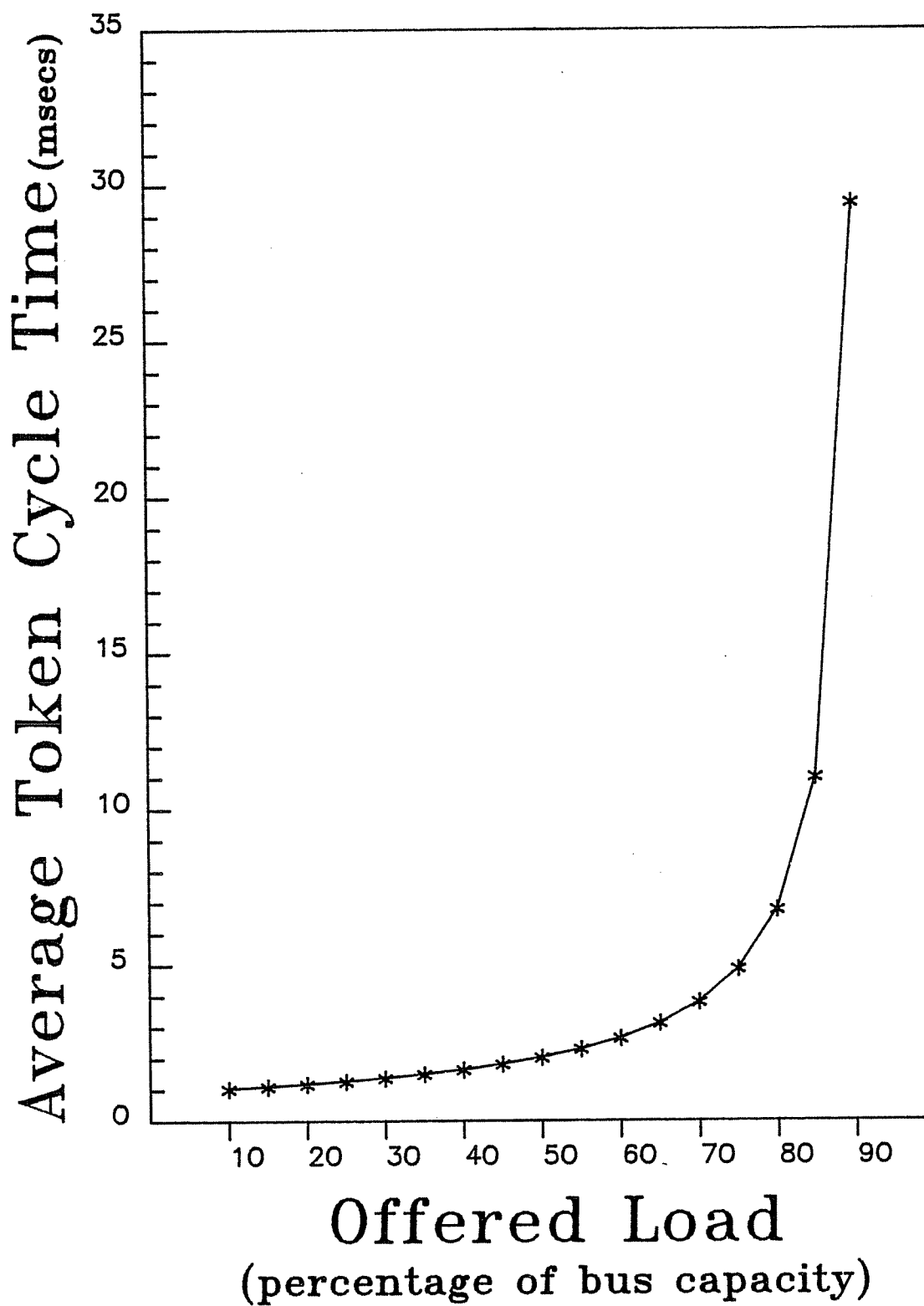


Figure 5.8 Base Configuration; Average Token Cycle Time

propagation delay. However, as the load increases 802.4 will still provide short, bounded delivery times that other protocols might not be able to provide, and the percentage of the network bandwidth consumed by 802.4 protocol frames decreases as the load increases.

5.4. Sensitivity to Packet Size and Number of Stations

In the base configuration used to generate the graphs displayed above the packet size of 160 bits of actual data could be classified as small data packets. The 64 stations that constituted the token passing ring can be viewed as a medium size network. The effects of varying the packet sizes from 160 to 5120 bits and the effects of varying the number of stations from 8 to 256 are analyzed below.

5.4.1. Packet Size

Intuitively, as the size of the data increases the average delivery time also increases; it takes longer to transmit a large packet versus a small packet. Figure 5.9 shows this to be true for lightly loaded networks, but as the offered load increases the average delivery time for small data packets increases at a much faster rate than the increase for large data packets. (It should be noted that the average delivery time scale in Figures 5.9 and 5.10 is much larger than the scale in Figures 5.6 and 5.7; the change in scale was necessary to illustrate the differences in the delivery times.)

The reasons for the smaller increases in the average delivery time for the networks with the larger data sizes are illustrated by Figure 5.10 which compares two of the components of average delivery time for the 160 bit and the 5120 bit data packets. Similar to the discussion in section 5.3.2, the average transmission delay is not shown in Figure 5.10; the values of the delay are obviously different, but the differences are constant with respect to the offered load. The average queuing

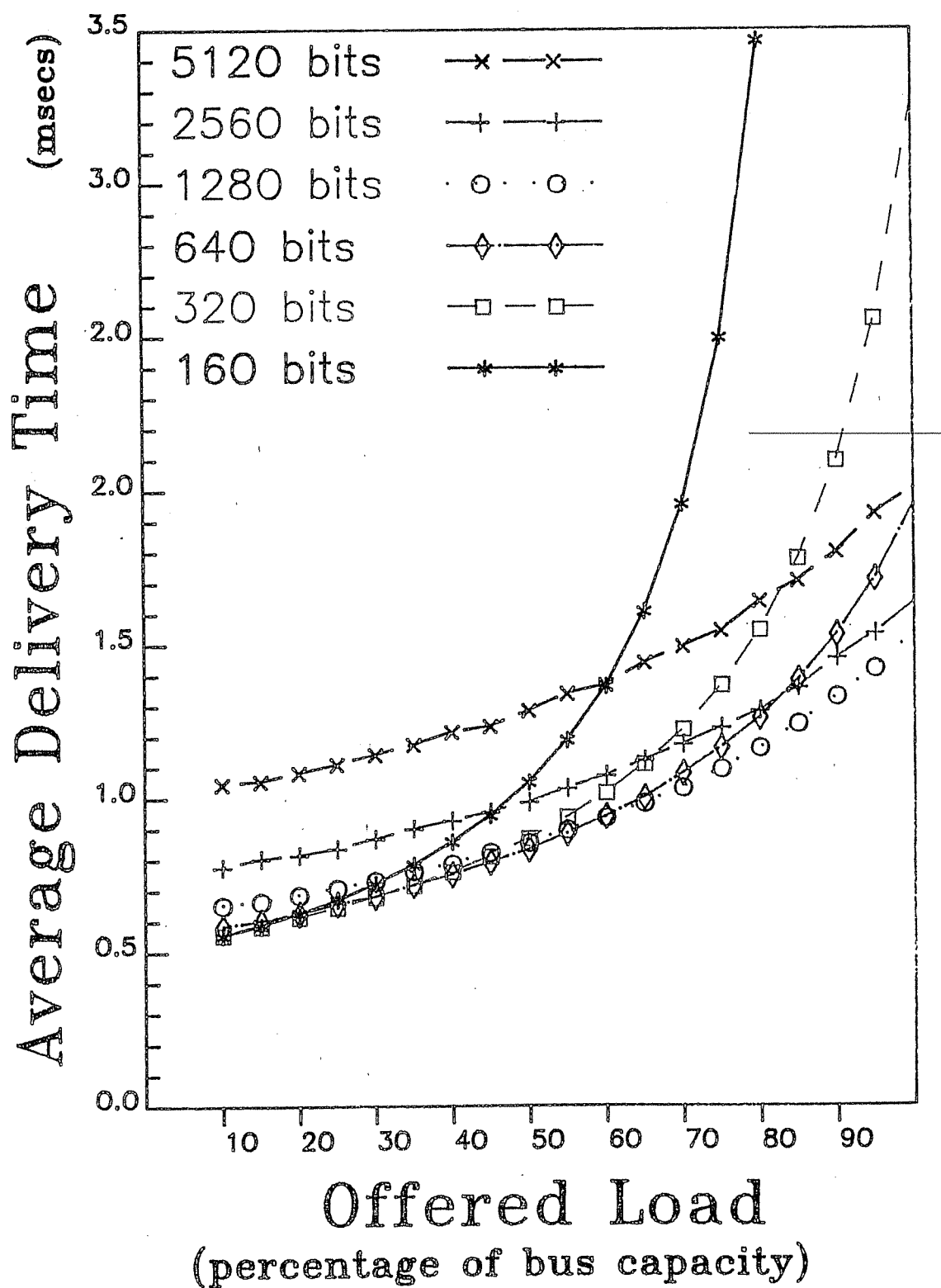


Figure 5.9 Delay vs. Offered Load for Varying Packet Size

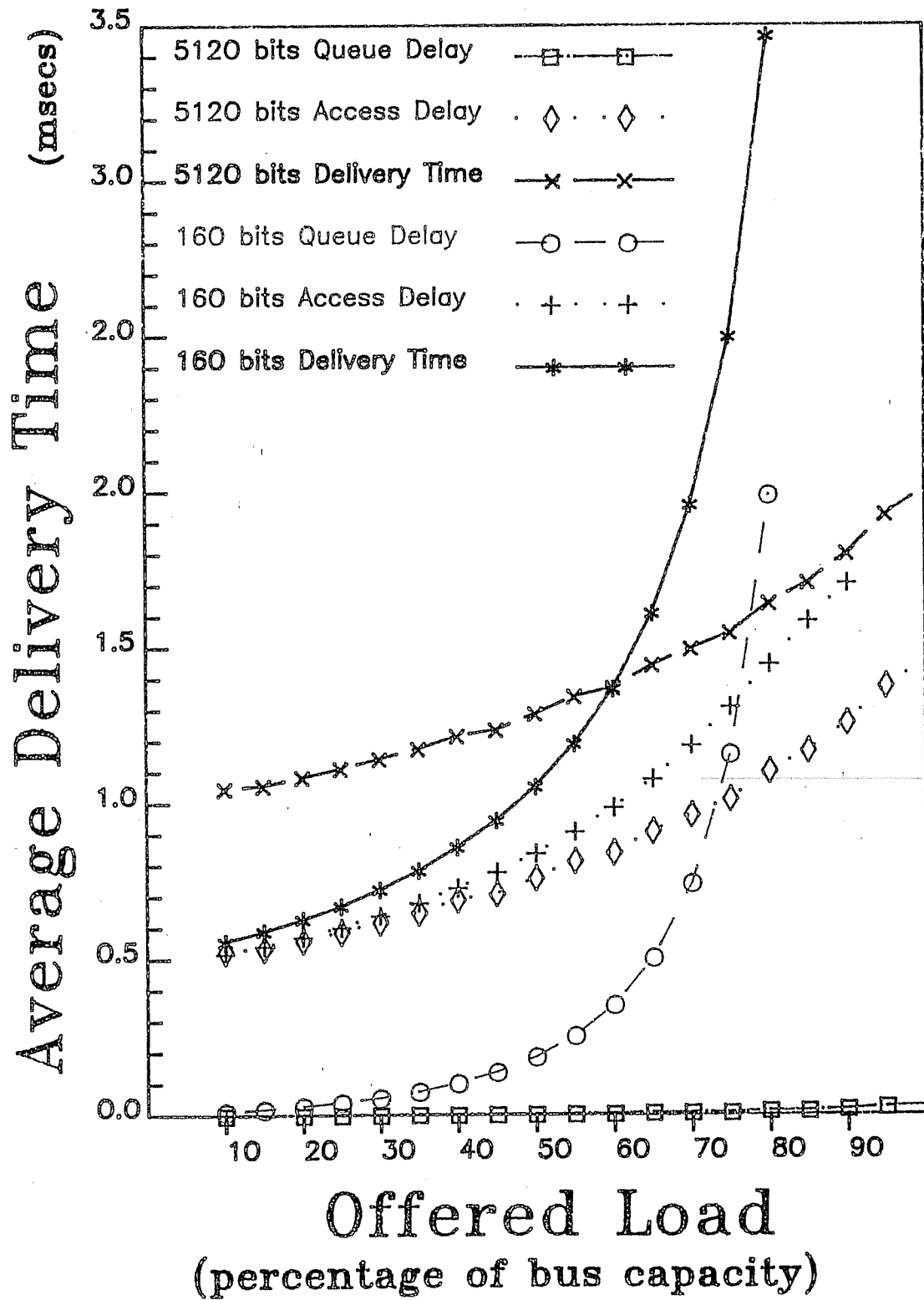


Figure 5.10 Delay Components for Varying Packet Size

and access delays show the two reasons for the different rates of increase.

As seen before in Figure 5.7 and as shown by the dashed line of Figure 5.10, the queuing delay for the 160 bit messages is very small for lightly loaded networks, but it increases exponentially as the load increases. At higher loads it is more likely that the average message must wait a longer period of time before reaching the front of the queue. For the network transmitting 5120 bit messages, the queueing delay remains small even when the offered load increases towards the total bus capacity. Given the low message arrival rate of the large messages, even at high offered loads it is rare that several of the large messages are simultaneously enqueued at a station.

The dotted curves of Figure 5.10 illustrate the effect of offered load on the average access delays. The access delays for the smaller packets are a large portion of the average delivery times for lightly loaded networks. As the load increases the average access delay increases because the token cycle time has increased, but it does not increase as rapidly as the average queueing delay. For the large data packets the average access delay increases at approximately the same rate as the average delivery time. As mentioned above rarely is there more than a single message enqueued at a station so the increase in the average delivery time depends more upon the traffic in the rest of the network than on the traffic at a particular station.

There are advantages to both types of packet sizes. The advantages to smaller packets come from the reduced average delivery time in lightly to moderately loaded networks; at loads less than sixty percent the small packet has a shorter average delivery time than a large packet. At higher loads, as the small packet message delivery time increases exponentially, it can take less than two milliseconds to deliver the average large packet. If the application was the transfer of general

messages to all machines or a single machine, it would be advisable for the station management software to encapsulate many small messages into one big message for a single transmission on a heavily loaded network. If the individual stations are too busy or too limited to filter out their messages from the large data packets, or if time critical information needs to be reported, then smaller packets would have to be used and care should be taken to ensure that loads did not increase and drive the average delivery time into the exponential portion of the delay curve.

Another advantage to the larger packet size is the increase in the throughput of actual data when compared to the smaller packet sizes. As mentioned in sections 2.1.3 and 3.2, all data messages are framed with either 96 or 160 bits of addresses, frame identifiers and checksums. The ratio of actual data to protocol overhead is much larger for large messages than for small ones. More bits of data are transmitted for every protocol bit and more data is transmitted before the station must wait an interframe gap and resume transmitting.

5.4.2. Number of Stations

A network consisting of a small number of stations should have better performance than a network with a larger number of stations operating under the same loads because less time is required to pass the token around the logical ring. Figure 5.11 illustrates the average delivery times for five network configurations where the number of stations varies from 8 to 256. To maintain even loading across the series of simulations the message creation rates at each station were modified such that, whereas a 64 station configuration has a mean creation rate equal to 61 messages per second, the rate for the 8 station configuration was 488 messages per second.

As can be seen in Figure 5.11 the network consisting of 8 stations has the best performance in this test. It has a very short delivery time for all messages

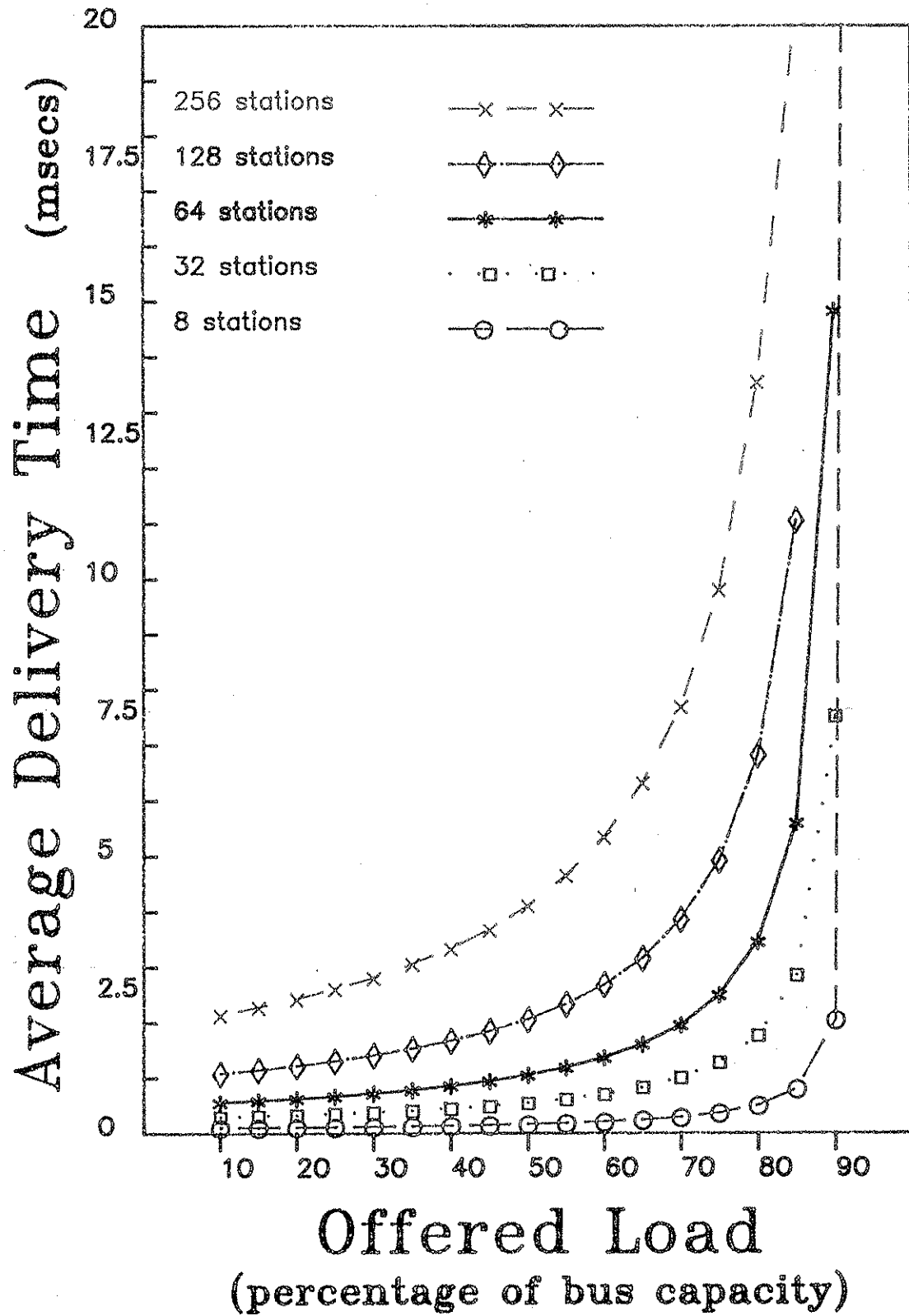


Figure 5.11 Average Delivery Time for Varying Number of Stations

until the average delivery time becomes exponential as the offered load passes eighty percent. The reasons for this short delay are the short time required for a token cycle and the very high message creation rate even at low loads; very few tokens are passed before a station with an enqueued message receives the token.

The network consisting of 256 stations has much poorer average delivery times because of the increased number of token passes required before a queued message is encountered. Whereas the base configuration has acceptable delay characteristics until it reaches the definite knee of the delay curve, the average delivery time for the 256 station network is increasing rapidly under moderate loads.

From the discussion above, the fewer the number of stations that are members of the token passing ring the better the performance of the network will be. Networks with a small number of stations are not as quickly affected by changes in the offered load; however, when they do approach the limits of the bus capacity the average delivery time grows exponentially. Stations with a larger number of active stations have poorer performance that decreases more rapidly as the load increases, and because of the increased bandwidth consumed by the token passing process, less of the bus is available for message transmissions.

5.5. Access Classes and Timers

In the tests performed above all message traffic was at the *Synchronous access_class*, and the *High_Priority-Token_Hold_Time* was set to an effective infinity of ten seconds; every station could drain its queue of all messages on each token cycle. The effects of distributing the load across several *access_classes* and of varying the timer values are explored in this section.

The first tests described below analyze the effect on average delivery time when the messages are evenly distributed across the access classes. The second tests describe the average delivery times when the loading is not evenly distributed.

In the third set of tests the effect of changing the timer values is analyzed for the base configuration and for distributed loads.

5.5.1. Even Load Distributions in the Access Classes

In the absence of any timer restrictions, all of the queues at all of the stations should be drained on each token cycle as in the base configuration. Similarly, at low loads most of the average delivery time is caused by the access delays. As the offered load increases, the contribution of the average queueing delay to the average delivery time increases and becomes the predominant factor.

Figure 5.12 shows the results of a set of simulations where both the *Synchronous* and *Urgent Asynchronous* are active with identical message creation rates and no time restrictions. The average delivery times for both *access_classes* are almost exactly equal; the variation was always less than four tenths of one percent. The agreement between the average delivery times for both *access_classes* can be seen in both Figures 5.12 and 5.13. Both *access_classes* also share identical average access and queueing delays as shown in Figure 5.13.

It is interesting to note that the average delivery times for both *access_classes* are equal to the average delivery time for the base configuration. Again, there is less than four tenths of one percent difference between the results for the base configuration which has all of the load in the *Synchronous* queue and this test which has the load equally split between the *Synchronous* and the *Urgent_Asynchronous access_classes*. In the absence of any time restrictions, all queues should be served equally, and that was observed for tests where the load was evenly distributed across all of the *access_classes*.

The even distribution of the load across separate *access_classes* without any timer restrictions does not use the *access_classes* to control the loading or provide priority service, both of which are the functions that the separate queues were

designed to perform. This distribution would provide a user with several queues in which to place messages if the physical size of each queue was limited.

5.5.2. Uneven Load Distributions in the Access Classes

The next test was to examine the effects of uneven load distributions. Two types of uneven loading were examined. In the first set of simulations, ninety percent of the offered load was at the *Synchronous access_class* and the other ten percent was in the *Urgent_Asynchronous access_class*, and in the other test the loadings were reversed. Both *access_classes* had exponential message creation distributions and constant 160 bit data messages.

In contrast to the negligible differences between all of the delay components that were observed in the cases where the load was evenly distributed, the average delays shown in Figure 5.14 possess very different characteristics than the average values in the base configuration. Both the *Synchronous access_class*, which is responsible for ninety percent of the offered load, and the *Urgent_Asynchronous access_class*, which has the remaining ten percent of the offered load, experience very similar average delivery times. The differences lie in the two components of the delivery time.

The most remarkable differences are between the average access delays for the *access_classes*. The average access delay for the *Synchronous* class illustrates the same behavior seen in the previous tests; at low loads the access delay is the major portion of the delay, but as the load increases the rate of increase in the average access delay is not as great as the overall rate of increase. The average access delay for the *Urgent_Asynchronous* class behaves quite differently. As the load increases, the access delay also increases rapidly, much more rapidly than the increase for the other *access_class*. The different queue loads and resulting queue lengths are the causes of this difference. A message that arrives in an empty

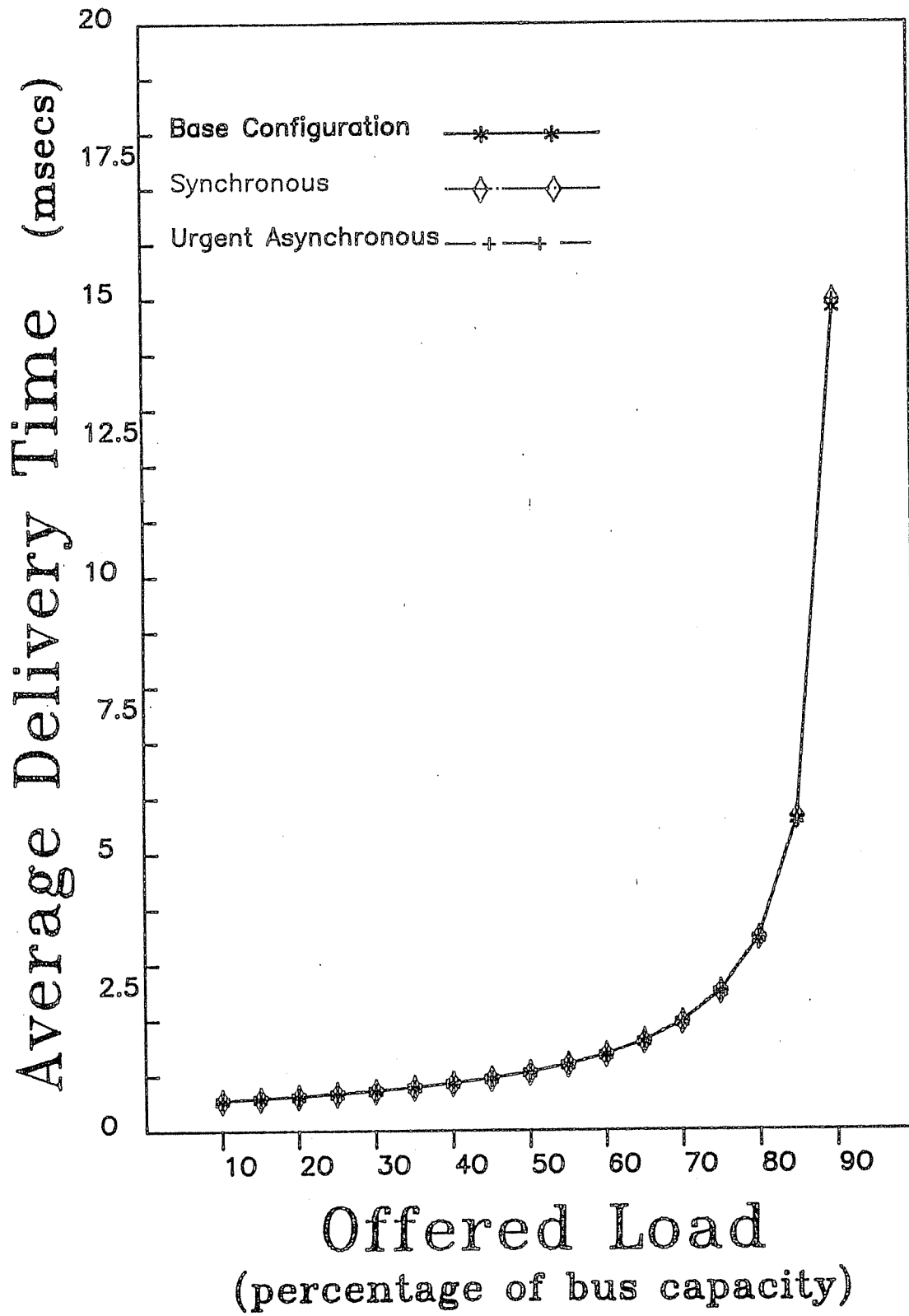


Figure 5.12 Average Delivery Time for Evenly Distributed Loads

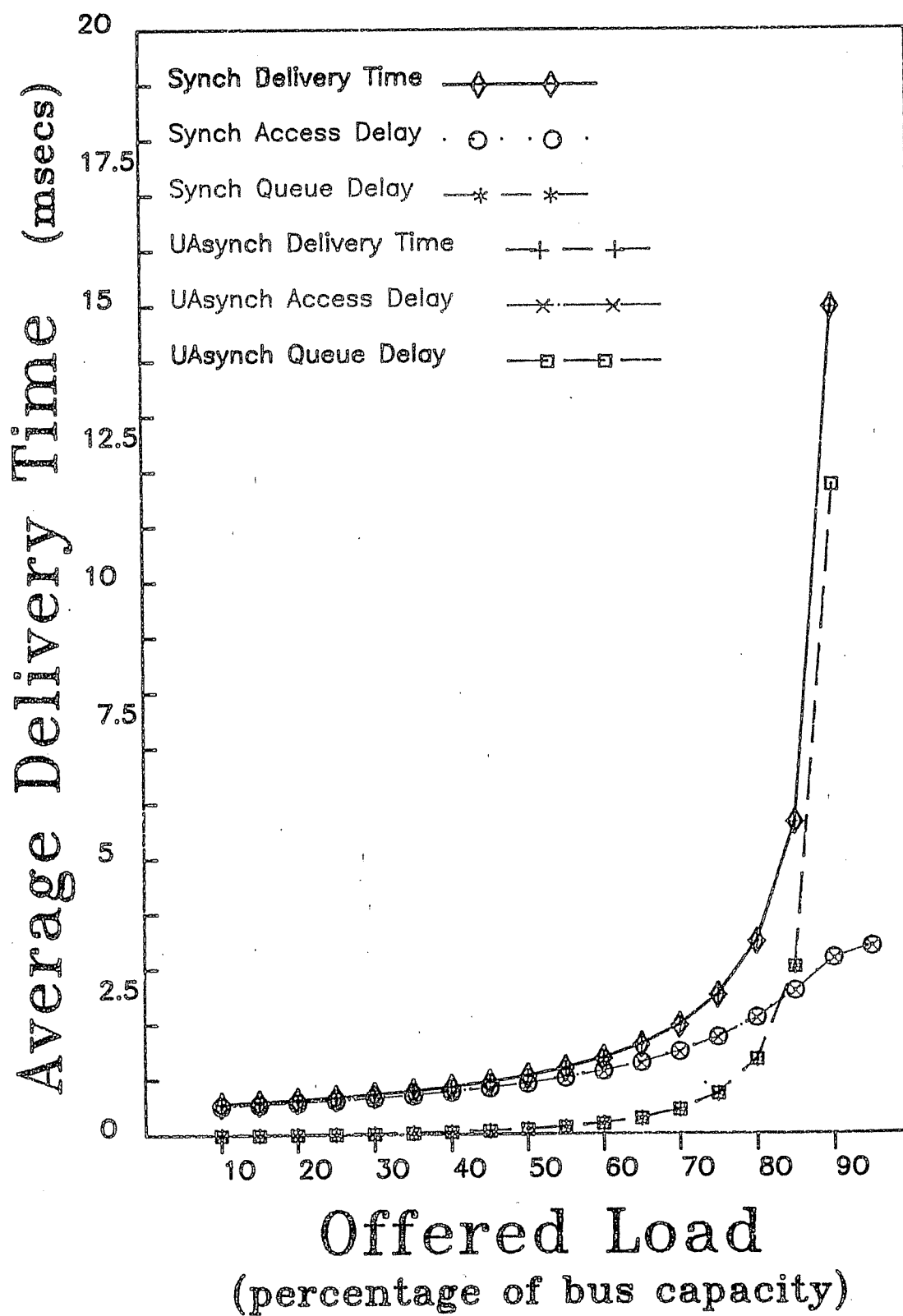


Figure 5.13 Delay Components for Evenly Distributed Loads

queue for either *access_class* will have a larger access delay if the total network traffic is high than it would have with low network loads. The average access delay will be smaller for the station with more messages in its queue because all of the succeeding messages transmitted on this token cycle will have access delays equal to the interframe gap. The message in the lightly loaded queue will not have as many other messages with short access delays to compensate for the large delay incurred by the initial message.

For the same reasons, light load in a queue while the network load is much higher, the average queueing delay is substantially smaller for the lightly loaded class; most messages are enqueued in empty queues and most of the delay is from the average access delay. The *Urgent_Asynchronous* messages experience very little queue delay until the network load passes seventy percent. As the total offered load grows past seventy percent of the bus capacity and as the load for *Urgent_Asynchronous* queues approaches ten percent of the capacity, the average queueing delay grows exponentially. This rapid growth has two interdependent causes based upon the token cycle time. Given the longer token cycle time, more messages can arrive at a queue during the token cycle. Because the token cycle is so long the larger number of messages will all experience larger queueing delays and drive the average queueing delay up exponentially.

The second test performed upon stations with highly imbalanced load distributions divided the offered load such that ten percent came from the *Synchronous access_class* and the other ninety percent was processed by the *Urgent_Asynchronous access_class*. Similar to the observed effects for the reversed load distribution described above, the lightly loaded *Synchronous* queue experiences very small average queueing delays because there are rarely multiple messages awaiting transmission. The *Synchronous* messages experience proportionally much larger delays than

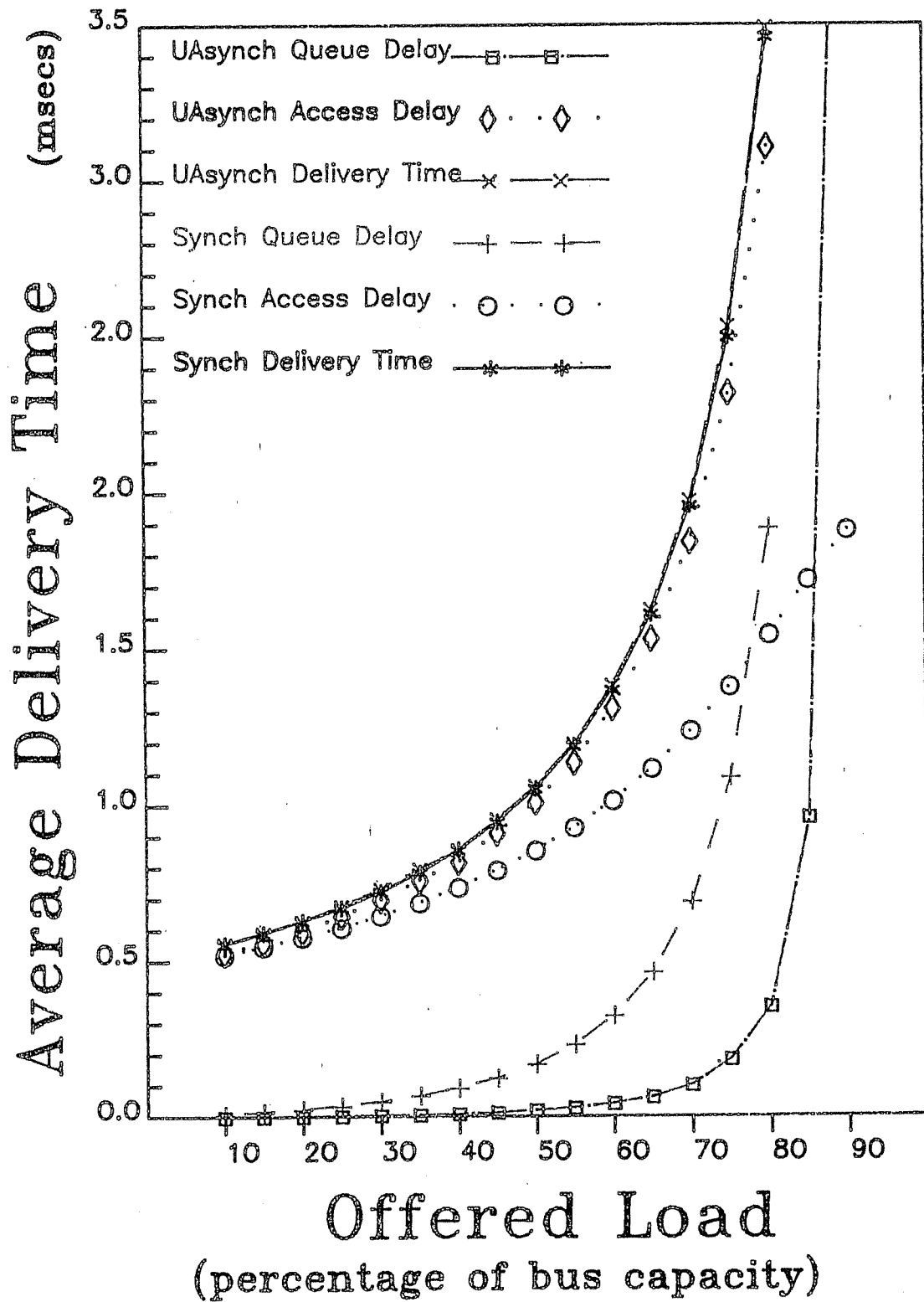


Figure 5.14 Delay Components for a 90/10 Load Distribution

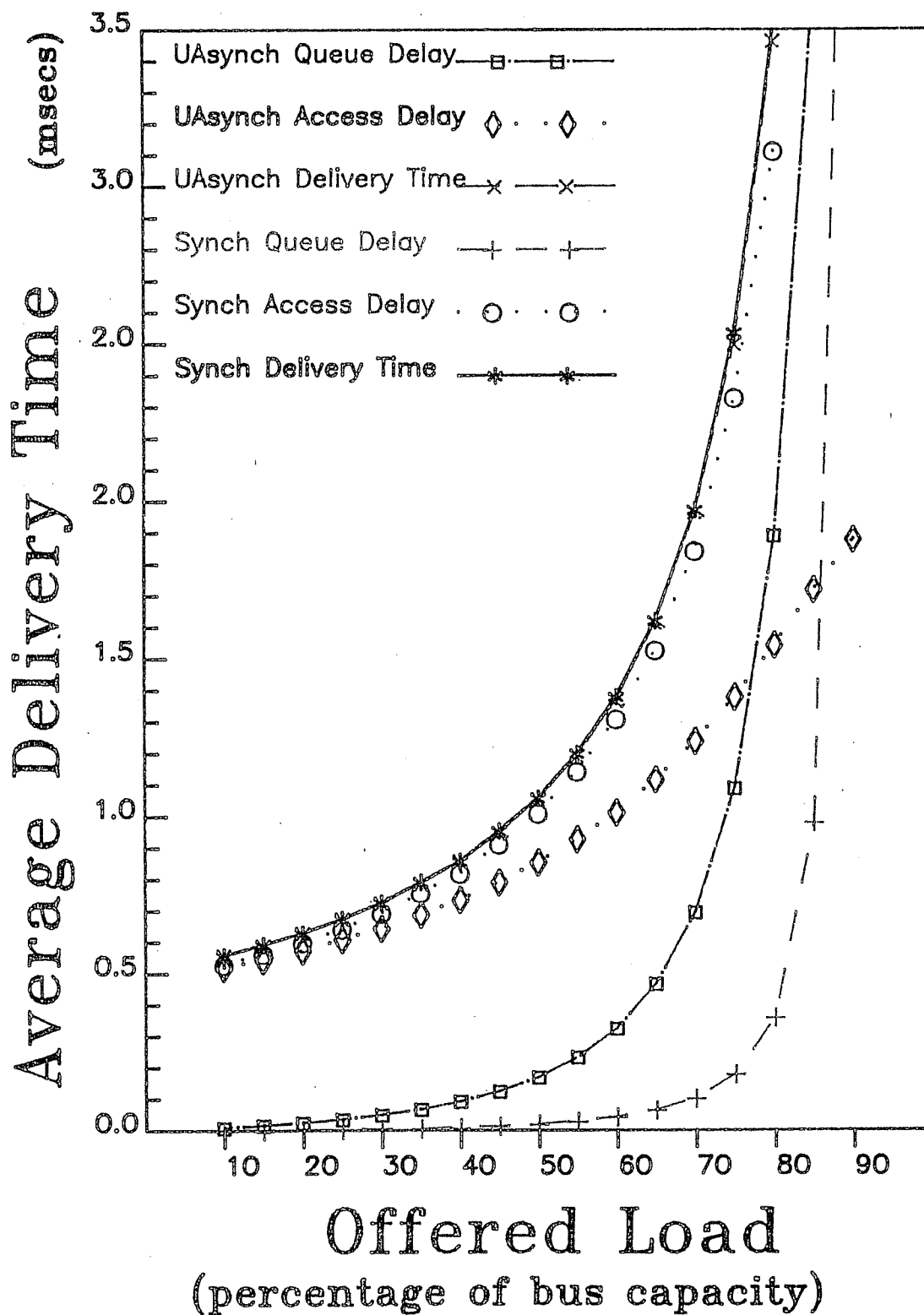


Figure 5.15 Delay Components for a 10/90 Load Distribution

the messages in the *Urgent_Asynchronous* queues, but both types of messages experience the same average delivery times.

It should be noted that there were no time restrictions on either the *Synchronous* or the *Urgent_Asynchronous* service in the tests described above. Very different results are possible if timers are used to partition the time available for either or both of the *access_classes* as discussed below.

5.5.3. Timers

The function of the *token_hold_timer* is to guarantee that no station should hold the token forever and prevent other stations from transmitting. As discussed in section 2.2.2, the *token_hold_timer* is reloaded every time a station begins to serve an *access_class*; if the timer has not expired then messages from the current *access_class*' queue can be transmitted until the timer has expired.

In the tests in this section, limits on the *High_Priority-Token_Hold_Time* and the *Target_Rotation_Time* values are set, and the effects of various settings upon some of the configurations tested earlier are examined. The first test examined the effect of changing the *High_Priority-Token_Hold_Time* for the base configuration. The second test examined the effects of changing the value of the *Target_Rotation_Time* for the imbalanced load configuration of Figure 5.15. Also tested was the effect of identical, ascending, and descending *Target_Rotation_Times* for networks with identical loadings at all of the *access_classes*.

5.5.3.1. High_Priority-Token_Hold_Time

The sensitivity of message delays and throughput to changing values of the *High_Priority-Token_Hold_Time* was tested by varying the time allotment from one to ten packet times where a packet time is defined as the time required to transmit a single packet. Using the message transmission time formula of section 3.2 the

packet time value in this experiment was defined as 27.5 microseconds. The effect of using values of one, two, four and ten packet times on network performance is shown in Figure 5.16.

When the *High_Priority-Token_Hold_Time* is equivalent to the transmission time of a single message, the arrival of more than one message per token cycle will make the queues grow to infinity. The maximum token cycle time is given by

$$T_C = N \cdot (X_T + X_m)$$

and in this case with $N = 64$, $X_T = 14.6$ microseconds, and $X_m = 27.6$ microseconds,

$$T_C = 0.0027 \text{ seconds/token cycle}$$

At 2.7 milliseconds per token cycle the offered load must be less than 370 messages per second at each station; the throughput is limited to be less than sixty percent of the bus capacity. This limit can be seen in Figure 5.16 where loads above fifty percent cause exponentially increasing average delivery times when the *High_Priority-Token_Hold_Time* is equal to one packet time. The other curves on Figure 5.16 also illustrate rapid growths in the average delivery time for all of the configurations once the maximum token cycle time limit for the configuration is reached.

In the case where at most one message can be transmitted by a station on a token cycle, 802.4 becomes the much simpler protocol of round robin service; each station has one chance to send one message once every token cycle. With the bus capacity consumed by the token passing process a simpler protocol would be advisable in this application to avoid the complexity of 802.4. In the other cases, the use of 802.4 is justifiable, depending upon the value chosen; loads from fifty to eighty percent of the bus capacity can be handled without incurring exponentially increasing message delivery times, and the token cycle time is bounded in the

absence of any ring reconfiguration or token losses.

5.5.3.2. Target_Rotation_Time and Imbalanced Loads

The configuration with ten percent of the offered load in the *Synchronous access_class* and ninety percent in the *Urgent_Asynchronous access_class* was used in the second timer test. As noted in the discussion in section 5.5.2 and as shown in Figure 5.15, the messages in the *Synchronous* queue have very large average access delays as the load increases. The transmission of *Synchronous* messages is being delayed by the transmission of messages at the lower priority *Urgent_Asynchronous access_class*. If the *Synchronous* traffic consisted of important, time-dependent information, it would be desirable to transmit those messages in favor of the lower priority messages.

To achieve better service for the *Synchronous access_class* the *Target_Rotation_Time* values could be set to provide bounded token cycle time. If the token cycle time is bounded, each station is guaranteed to receive at least *High_Priority-Token_Hold_Time* seconds of service at the *Synchronous access_class* on each token cycle. The *High_Priority-Token_Hold_Time* was not restricted because of the light *Synchronous* loads. The values of the *Target_Rotation_Time* that were tested were one, two, four, and ten milliseconds.

A value of one millisecond limits the bus to transmitting at most three messages per token cycle from the *Urgent_Asynchronous* queues because 934 microseconds are consumed by the token passing process. If the value is doubled to two milliseconds, 39 messages could be transmitted from the *Urgent_Asynchronous* queue on each cycle; a four millisecond *Target_Rotation_Time* would allow the transmission of 111 messages, and ten milliseconds would allow 329 messages per cycle.

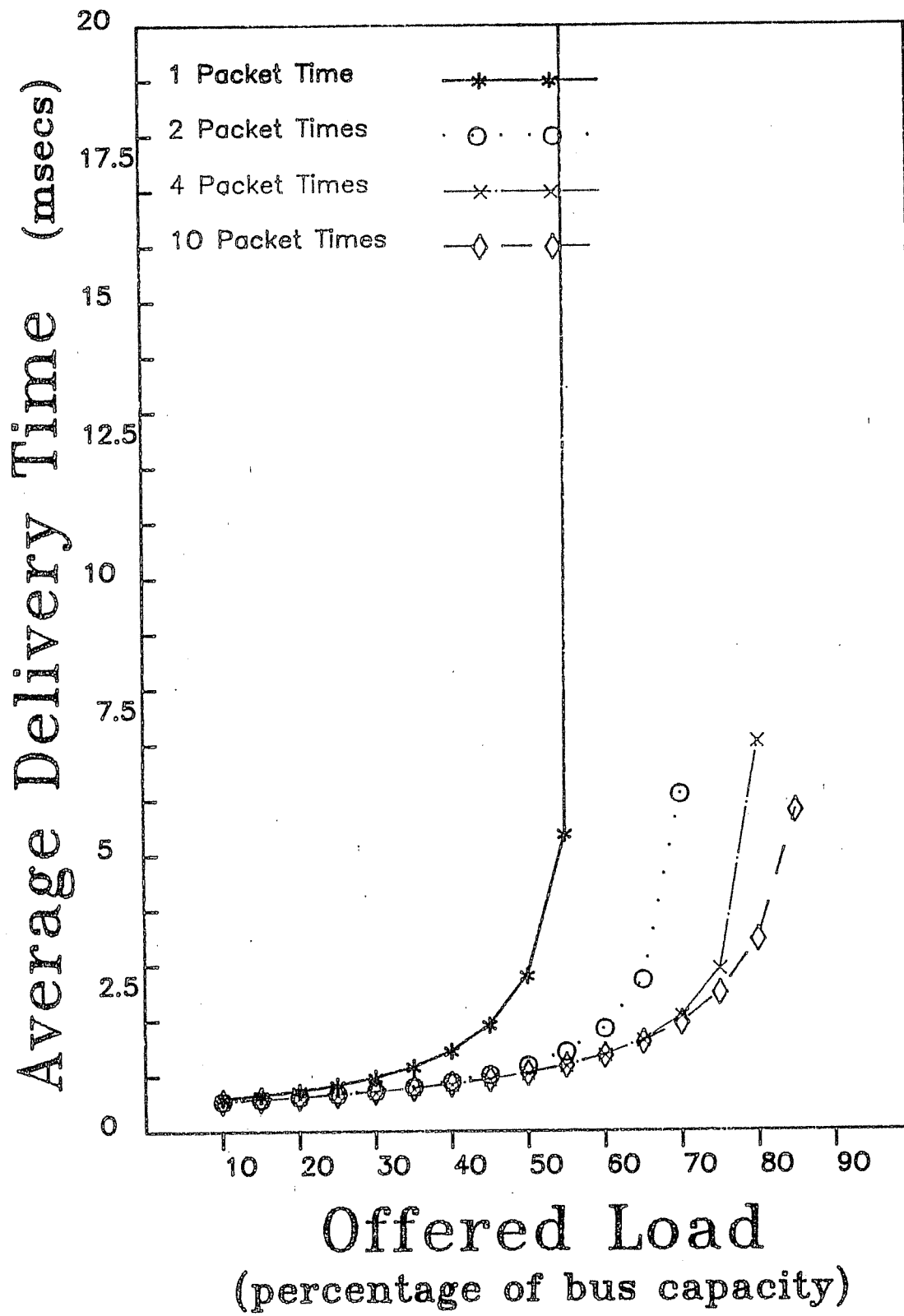


Figure 5.16 Varying the *High_Priority-Token_Hold_Time*

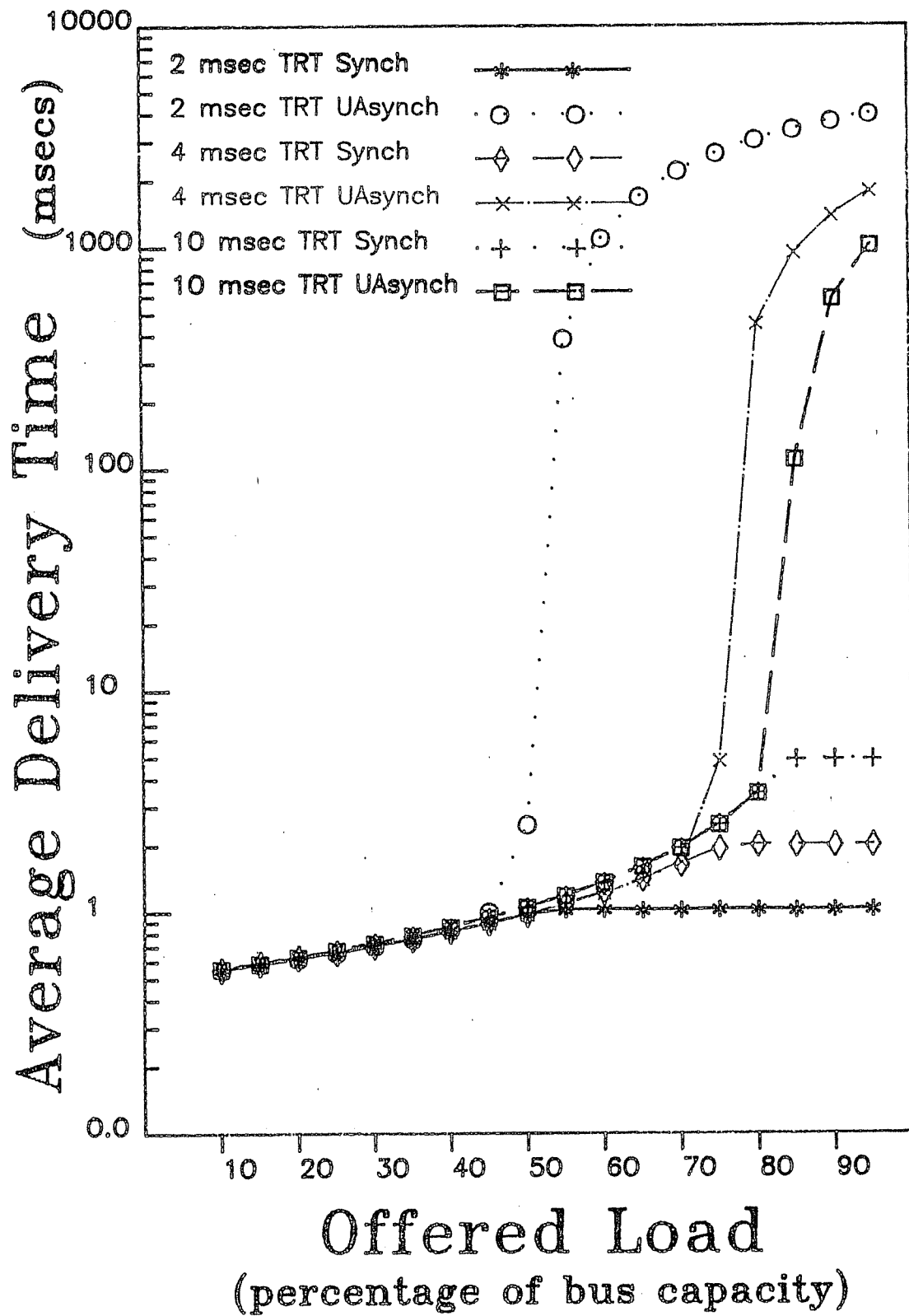
Figure 5.17 Varying the *Target_Rotation_Time*

Figure 5.17 illustrates the results of the simulations with *Target_Rotation_Times* of two, four, and ten milliseconds. The results of the one millisecond test are not shown because even at very low offered loads the average delivery time for the *Urgent_Asynchronous access_class* was experiencing exponential growth and there was too little *Synchronous* traffic to observe changes in the average delivery time.

A *Target_Rotation_Time* value of 10 milliseconds has very little effect in this simulation until the offered load passes eighty percent of the bus capacity. Until that threshold is passed both *access_classes* are experiencing the same increases in the average delivery time. At offered loads greater than eighty percent of the average delivery time for the *Urgent_Asynchronous* class increases from 3 to 500 milliseconds while the *Synchronous* average delivery time only increases to 4.9 milliseconds, approximately one half of the desired token cycle time of 10 milliseconds.

The simulations with a *Target_Rotation_Time* of 4 milliseconds also show very few differences between the *Synchronous* and *Urgent_Asynchronous* average delivery times at low loads. The threshold in this series of simulations is seventy percent of bus capacity. As the load passes seventy percent the average delivery time for the *Urgent_Asynchronous access_class* increases rapidly as the *Synchronous* delivery time again is approximately equal to one half the desired token cycle time of 4 milliseconds.

The threshold for the 2 millisecond simulations was fifty percent of the bus capacity as can be seen in Figure 5.17, after which the *Synchronous* average delivery time approaches one half of the token cycle time. Another interesting result of the 2 millisecond test is the flattening of the average delivery time curve for the *Urgent_Asynchronous access_class* as the offered load passes sixty percent. The average delivery time has grown quite large but the queues still have finite

length; messages may have long delays but they do get transmitted. The 4 millisecond curve also is starting to show the same behavior but it is too close to the bus capacity where all delays become infinite.

From the above discussion, it can be seen that it is possible to bound token cycle time and thus *Synchronous* delivery times through the use of the *Target_Rotation_Time*. Small differences in the *Target_Rotation_Time* values can cause large differences in the token cycle time and the delay of the restricted *access_class*.

5.6. Nonhomogeneous Stations

It is quite probable that all of the stations on a network will not share the same loading characteristics. In factory automation applications some machines may need only occasional messages or may generate messages infrequently while others could have many messages to send at almost all times or require constant control information. This section of the thesis examines several network configurations with nonhomogeneous members of the token passing ring.

Several types of nonhomogeneous network configurations were studied. The first variation from the base configuration is a network where fifty percent of the total offered load originates in a single station. In the next set of simulations, two stations shared fifty percent of the offered load. The third test describes the effects of varying the load in all but one station.

The station which is responsible for fifty percent of the offered load is identified as the "Monster" in the graph of Figure 5.18. The other 63 stations are referred to as the "Base". The "Base" stations had a message creation rate of 31 messages per second for a ten percent offered load. The "Monster" had a message creation rate of 1951 messages per second. Both types of stations were transmitting the standard 160 data bit message.

Unlike the previous cases where the queuing delay in the highly loaded station varied little from the queueing delay in the base configuration, the average queuing delay for the "Monster" station becomes the dominant factor in the average delivery time much more quickly. The more rapid increase in the contribution of the queueing delay to the total average delivery time is caused by the extreme concentration of messages in the single station. In the test reported in Figure 5.15, ninety percent of the load was concentrated in one *access_class* but distributed over 64 stations.

Figure 5.18 shows the "Monster's" average access delay actually decreasing as the load increases. As the load increases, the number of messages transmitted by the "Monster" before it passes the token also increases. The first message that is transmitted on a token cycle has an access delay equal to the time that the token was elsewhere in the logical ring. The remaining messages have access delays equal to the interframe gap. Therefore as the load increases more messages are transmitted by the "Monster" with the short interframe gap rather than the longer cycle time.

From the comparison of the average delivery times for the "Monster" and the "Base" stations, it appears that to achieve better performance a station should attempt to be as busy as possible with respect to the rest of the network. The "Monster" could not receive such favorable treatment in the presence of token holding time restrictions because it would be forced to pass the token even though its queues are not empty and even if there are no other messages in the system.

The second type of nonhomogeneous station loading separated the offered load into three sources. One station was the source of twenty five percent of the offered load, another was also responsible for twenty five percent of the offered load, and the remaining 62 stations were responsible for the remaining fifty percent

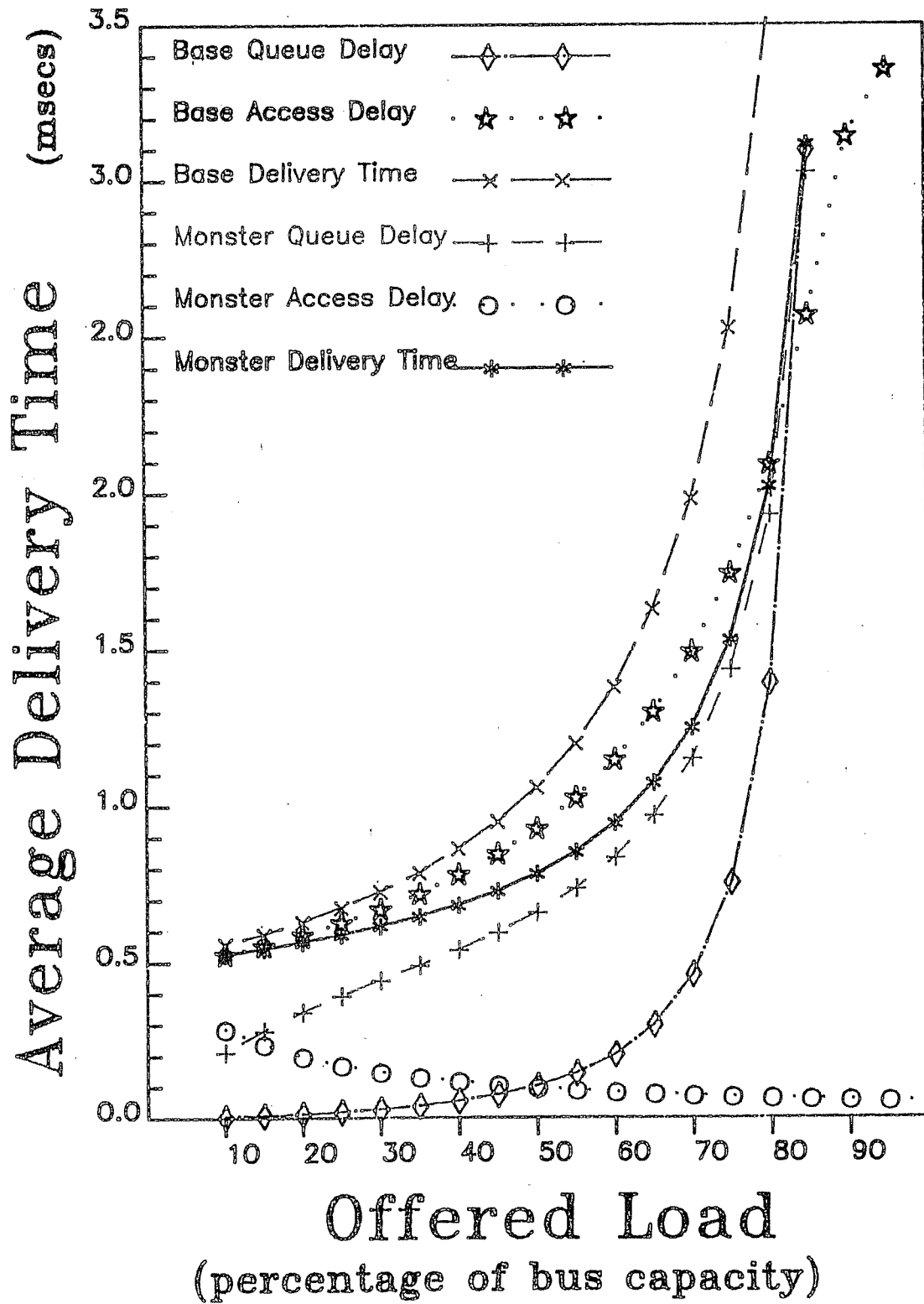


Figure 5.18 One Station with Fifty Percent of the Offered Load

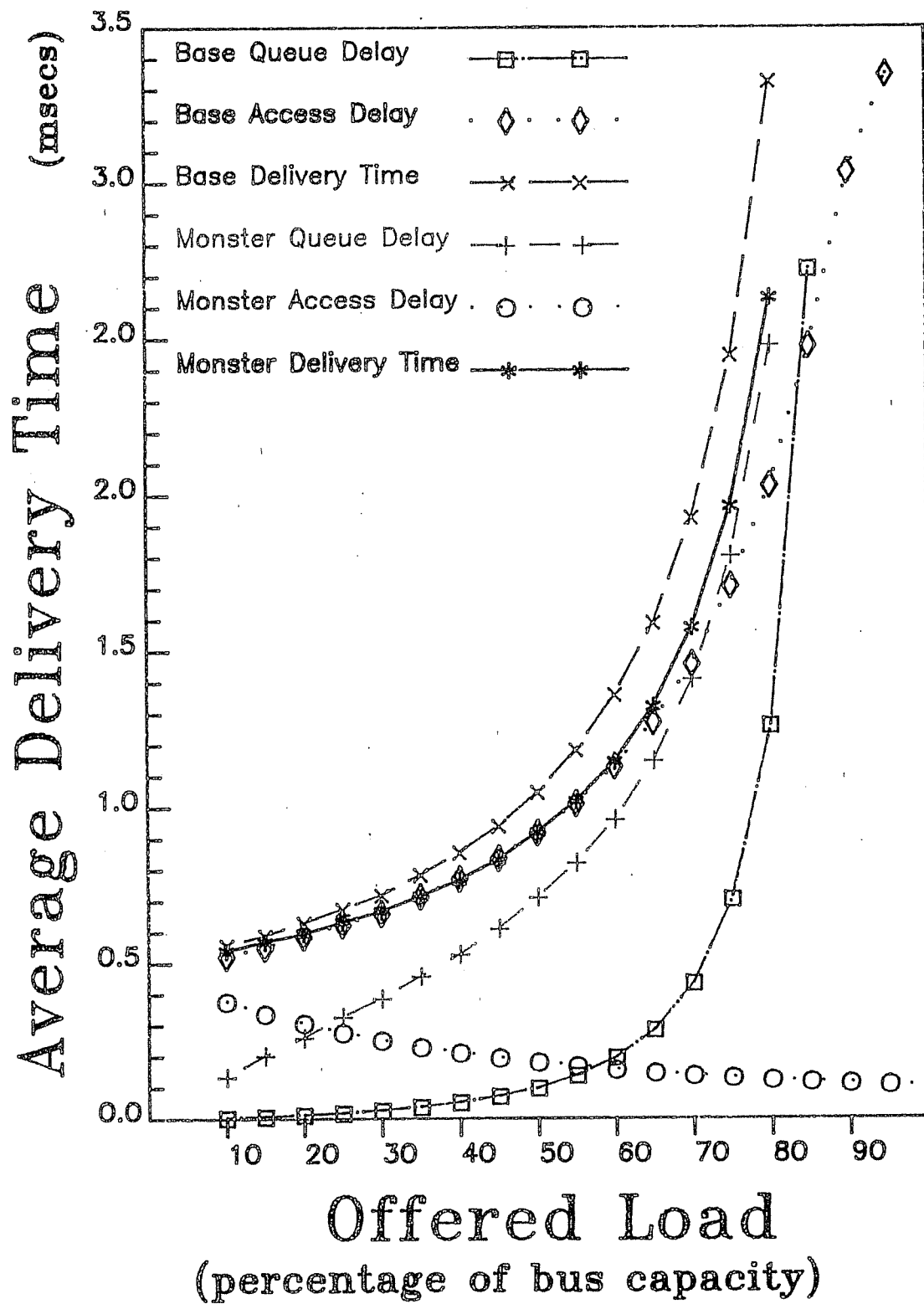


Figure 5.19 Two Stations with Fifty Percent of the Offered Load

of the load. Two versions of this configuration were tested: once with the two highly loaded stations as logical neighbors and again with the stations equally separated on the token passing ring. There were no appreciable differences in the results from either version, or in the values reported by both stations. In Figure 5.19 the average values of only one of the "Monster" stations are reported for reasons of clarity.

As in the previous report, the average access delays for the monsters decrease as the total load in the network increases. The base's average access delay is very large and follows the average delivery time curve until high loads are encountered. The monsters have large and rapidly growing average queue delays caused by the large amount of traffic in these queues. The base's average queueing delay is small because of the light loading in these stations.

Despite the similarities between the delay components of the two overloaded stations and the single overloaded station, the average delivery time for the two stations is much closer to the average delivery time for the base stations than in the previous case. Because the average delivery time depends heavily on the traffic offered by the rest of the network, each of the separate monsters does not have the same performance as the one individual monster. High traffic stations in networks with multiple large servers will not achieve as high a performance as a single high traffic station.

The third type of nonhomogeneous loading explored was to test the effect of a varying load at all but a single station. The single station with a constant message arrival rate is the "SteadyUser" in Figure 5.20. The "SteadyUser's" constant message arrival rate was 305 messages per second. The remaining stations had message arrival rates which varied from 61 to 610.

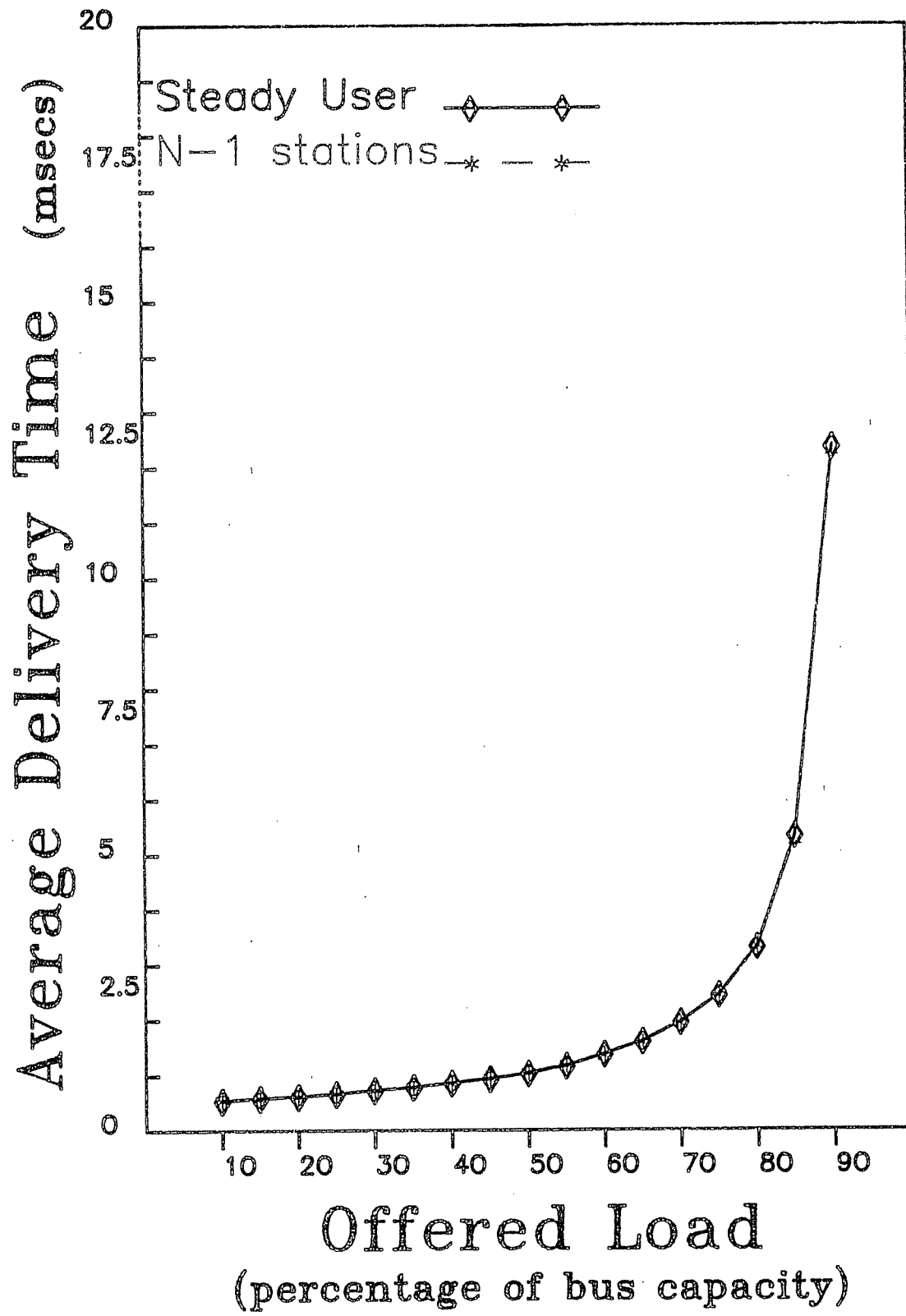


Figure 5.20 A Station with Constant Load

In contrast to the results observed earlier, the steady user's average delivery time does not differ from the average delivery time for the $N-1$ other stations as is shown in Figure 5.20. While at low loads the steady user has larger queuing delays than the other stations, that is to be expected since the other stations have very few messages. The average access delays in the steady user's queue increase with the increase in offered load by the other stations. The queueing delay also increases as the total offered load increases. These increases in the average delivery time and its components as the network load increases are logical; with more messages in the rest of the network it takes more time for the token to return to the steady user, creating larger access and queueing delays.

The results of all three tests of nonhomogeneous station loadings demonstrate quite clearly that a station's performance depends heavily on the performance of the rest of the network. In the absence of timer restrictions lightly loaded stations can incur large delays while waiting for the token, and overloaded stations can achieve high performance at the expense of the other stations. If there are several stations with above average loads, they lose their advantage relative to the rest of the network because of the presence of the other highly loaded stations.

5.7. Transient Loadings

To study the effects of transient loadings, a series of simulations was run with active token traces. The transient loading was achieved by creating a station class consisting of one station that transmitted four 32 kilobit messages once every second. The other stations in the network had the usual base configuration.

The purpose of this experiment was to determine how quickly the transient load could be absorbed and how long it would take the network to return to the state it was in before the loads were applied. This was accomplished by charting the token cycle times for the periods just before and just after the imposition of

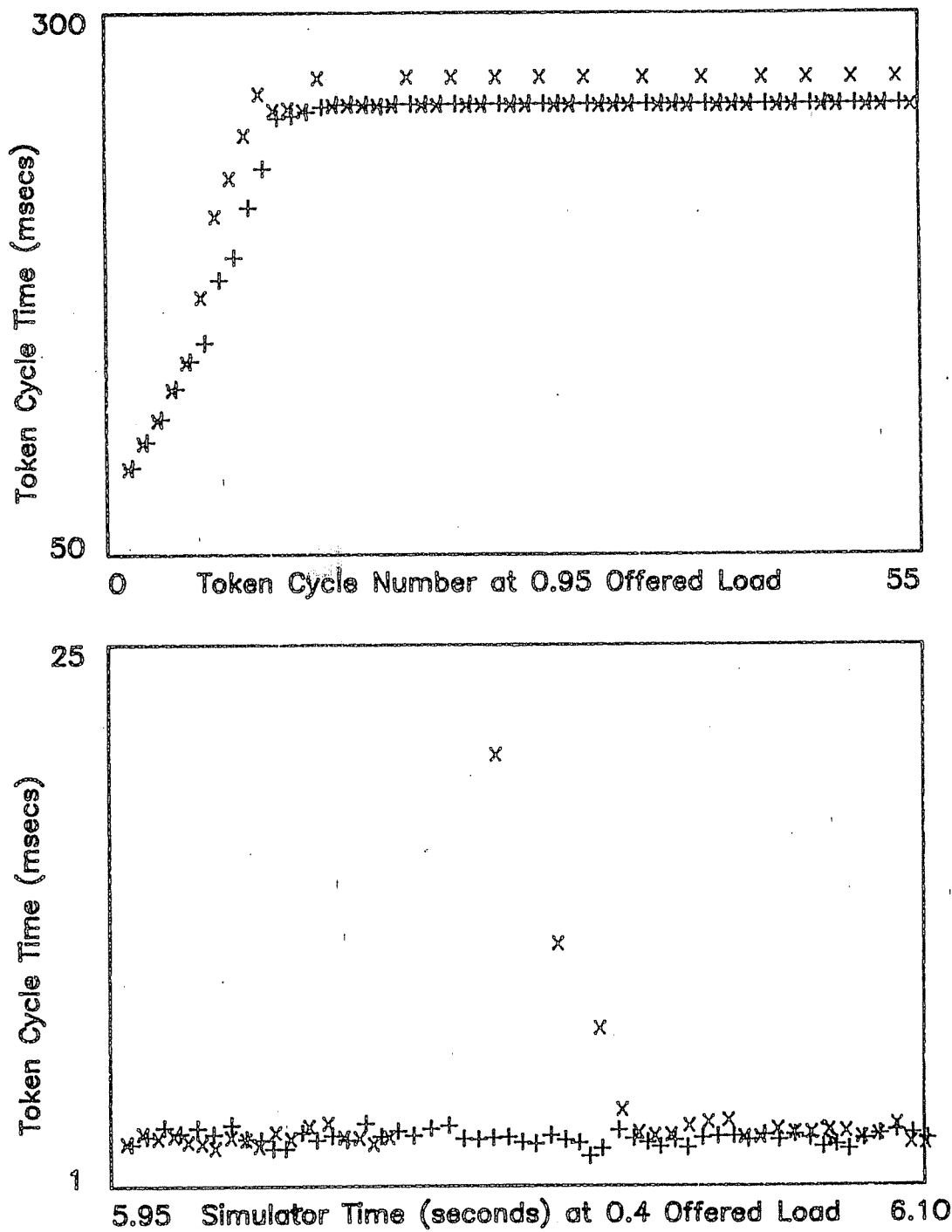


Figure 5.21 Transient Loads and Token Cycle Time

the transient load, Figure 5.21. This figure also charts the token cycle time for a configuration without the transient loads. The "x"s are token cycle times in the network subject to the transient loads and the "+"s are the normal token cycle times.

Because there were no timer restrictions, all stations could dump all of their messages on all token cycles. During the time interval when the four large messages are being transmitted the other stations will receive a larger number of messages than they would receive on the normal token cycle. It will take longer to transmit the extra messages, thus increasing the current token cycle time by more than just the transmission time of the transient load. Succeeding token cycles will also be slightly longer because of the increased number of messages that arrived in the previous token cycle. If the token cycle time never decreases then a dangerous situation could develop as the cycle time increases on each imposition of the transient load and the network reaches an overloaded state with queues overflowing.

Figure 5.21 illustrates the results seen in these tests. The lower graph is for the network operating at forty percent capacity. The effects of the transient load quickly dissipate over the following few token cycles. Within five token cycles the token cycle time has returned to the pre-transient load level. The simulations for offered loads of sixty and ten percent show similar response.

The addition of a transient load to the ninety five percent loaded network does not cause as drastic a change because the size of the load is not substantially larger than the rest of the network traffic as in the other cases. The transient load does drive the network to its high average delivery time more rapidly than in the unloaded case. At the high level the transient load produces relatively small perturbations in the average token cycle times.

The ability of 802.4 to handle high transient loads has been shown in these tests. With infinite timers, the stations are quickly able to transmit the extra messages and return to normal operation. The possible effect of limiting the timers both of the overloaded station and the other ring members was not explored.

CHAPTER 6

Conclusions

6.1. Analysis of 802.4

The main goal of this thesis was to gain an understanding of 802.4. The large number of possible configurations prevented an exhaustive test of all 802.4 system parameters, but the ones tested and analyzed in Chapter 5 have provided a large base from which to analyze the important parameters and functions of the protocol.

6.1.1. Protocol Overhead

At offered loads below the capacity of the transmission medium many network access protocols can provide acceptable performance. 802.4 suffers in comparison with many simpler access protocols at low loads because of the overhead associated with the frame encapsulation method and the token passing process.

Every 802.4 message has at least 96 bits of addressing and framing bits encapsulating the actual data. In a sustained conversation between two stations these extra bits increase the overhead. While the framing bits can be a large overhead for small messages, they provide the station with a high probability that erroneous messages will not be accepted as valid because of the thirty two bit checksum that is part of the framing sequence. This large checksum makes the chance of accepting an erroneous message very small.

Protocols without explicit tokens do not lose bandwidth to the token passing process. As discussed in section 1.1.2.2.4, implicit token passing frees that portion of the bandwidth that would have been consumed by token transmissions, but does

so at the expense of more complicated ring membership management. Protocols that do not use tokens either waste bandwidth through TDMA or FDMA, or they can not offer bounded delays.

The token passing process on a nearly idle ring maintains the logical ring and adds very little to the average delivery times of messages. The amount of bandwidth consumed by 802.4 token transmissions declines as network load increases, providing better utilization of the bus as the offered load increases. Throughput in 802.4 is limited by the token passing process, but in the absence of time restrictions, the throughput in bits transmitted is approximately equal to the offered load.

6.1.2. Packet Sizes and the Number of Stations

The performance in a network is very dependent upon the size of the messages being transmitted and the number of stations in the logical token passing ring. Larger messages and smaller logical rings provide better performance.

Larger messages offer better performance when compared to smaller messages because more bits are transmitted before the station has to pause and wait an interframe gap before it begins to transmit the next message or passing the token. The advantages of large messages are only evident at high offered loads; at lower loads it is better to transmit the messages individually rather than encapsulating them into single large messages because transmitting the messages individually could give each message better service than if its transmission was delayed until a large packet had been assembled.

Smaller token passing rings offer better performance than large rings. The advantage is due to the fewer number of token transmissions required before a token cycle is complete. Since the token cycle time is the major contributor to the delay in delivering a message, the shorter the token cycle time the smaller the

average delivery time.

6.1.3. Access_Classes and Timers

When there are no time restrictions on the *access_classes*, dividing the load among the classes does not offer any improvement in network performance. In the presence of time restrictions set by the *High_Priority-Token_Hold_Time* and the *Target_Rotation_Times* service can be more precisely tailored. The token cycle time can be controlled by setting both types of timers.

In general to provide the best service for the *Synchronous access_class* the *High_Priority-Token_Hold_Time* should be set large enough that a station can transmit all messages from the *Synchronous* queues on each token cycle. To implement the priority service of 802.4 the *Target_Rotation_Time* for the *Urgent_Asynchronous access_class* should be set to the maximum desired token cycle time. While this time can occasionally be exceeded by the transmission of *Synchronous* messages or ring maintenance traffic, *Urgent_Asynchronous* traffic will defer to *Synchronous* traffic. The *Target_Rotation_Times* for the lower *access_classes* should be smaller than the previous value if the priority is to be maintained for all servers.

6.1.4. Nonhomogeneous and Transient Loads

In the studies of nonhomogeneous loads, two facts were clear: a station that is incredibly overloaded compared to the other stations in the network will receive better performance than the other stations in the network; the performance at non_overloaded stations depends more upon the traffic in the rest of the network than traffic at the station.

The simple model for message delay is very close to actual performance. For most loads, delay depends more upon the token cycle time than upon the time it

takes to service the station. Stations that are highly overloaded do have a larger contribution to the components of their own average delays but still the average delivery time depends upon the total network traffic, to which an overloaded station is a large contributor.

Even while operating at high offered loads, 802.4 can accommodate the application of large transient loads. The transients do not cause permanent changes to the operating environment; within a few token cycles the backlog of messages can be transmitted and the token cycle time will return to its previous range. The ability to handle the large transients, coupled with the ability to bound delivery times and allow dynamic ring membership make 802.4 a good choice for many applications in local area networking.

6.2. Suggestions for Future Research

There were many facets of 802.4 that were not fully explored. Several areas in particular that would be of interest to study are the effects on delay of transient ring membership, combinations of number of stations and packet sizes to determine if concentration is desirable, and specific message worst case delivery time.

It was shown in the preceding chapter that smaller token passing rings provided better performance than larger rings under the same offered loads. Stations with light traffic could improve network performance by removing themselves from the token passing ring when they are idle, thereby reducing the size of the ring. When the lightly loaded stations have messages to send they could attempt to rejoin the ring. Questions are: (1) how much delay would the stations encounter when trying to rejoin the ring? (2) would the increased ring reconfiguration traffic consume more bandwidth than was saved by the departure of the lightly loaded stations? and (3) under what conditions should a station decide to leave the token

ring?

Smaller token passing rings have better performance than large rings, and larger messages yield better performance than small messages. Is it desirable to have single stations act as collectors, gathering information from many other machines and transmitting a collection of messages as a large packet instead of several small ones? In control applications where message delivery times are important this method may not be desirable, but it should be studied for other applications. A more exhaustive study may be able to find optimum ranges for both the number of stations and the size of packets.

802.4 is a deterministic protocol; bounds upon delivery times can be determined through the use of the *High_Priority-Token_Hold_Time* and the *Target_Rotation_Times*. In an industrial control application, it could be useful to test and see how closely the bounds can be set. As an example, a message advising all machines in a factory to shut down because of a failure in the manufacturing system should be transmitted as soon as possible to minimize any possible damage. Through the use of the empty queue flag on a preloaded message queue these times could be determined, and the designer could experiment with values that provided the best response for the special message while still providing acceptable performance under normal operating conditions.

6.3. Summary

There are, of course, many other areas of study that were not tested or mentioned above. More extensive tests of error handling, transient loading, and variable load distributions would increase our knowledge of 802.4's performance characteristics.

The work presented in this thesis does offer the reader an understanding of 802.4, its uses, operation, and performance. It also has described limitations with

the protocol such as the large overhead consumed by the token passing process on a lightly loaded network. The IEEE has designed a complicated but efficient protocol and one with many applications. With physical implementations of 802.4 being developed, it will soon be possible to validate the results obtained here through simulation by comparison with actual performance data.

APPENDIX A

Manual for the IEEE 802.4 Simulator

1. Notation

The following conventions are used in this manual:

Words that are *italicized* are 802.4 variable names or reserved words.

Words that are in **Bold Face** are simulator reserved words or command selections.

Words that are in Messenger Font are simulator prompts or menus.

The *Messenger Italic Font* is used for warnings.

2. Introduction

The simulator consists of four files; the executable binary code **tokbus**, the default network parameter settings file **DfltVals**, the class library file **ClassLib**, and a sample station class file basecase. The three files **tokbus**, **ClassLib**, and **DfltVals** are required to be in the user's current working directory for the correct operation of the simulator.

There are two versions of **tokbus**; the first runs on the University of Virginia's Department of Computer Science Vax 11-780; the second runs on the department's Apollo DN-300 workstations. If the user is on the departmental Vax he should be aware of the following restrictions:

(1) if the files **ClassLib** and **DfltVals** do not exist in the user's current working directory, attempts to read the defaults or to read or save a class will cause the program to terminate.

(2) file names for station classes and output files are limited to ten characters by Berkeley Pascal.

(3) if the user enters a character in response to a request for a numeric

value, the program will terminate.

- (4) when the user makes a menu selection or enters a single character response to a prompt, he must enter a carriage return to complete the response.

If the user is on one of the departmental Apollo workstations then he is no longer bound by the ten character file name limit or dependent upon the existence of the **classlib** or **dftvals** files, but must allow for the following restrictions:

- (1) file names must not contain uppercase letters and should not begin with a numeral or underscore.
- (2) the user must guarantee that the simulator is invoked in a window large enough to accommodate the simulators menus and displays; at least 25 rows and 80 columns.
- (3) the user can not use input redirection to drive the simulator because of the raw input and output mode used in the Apollo implementation.
- (4) characters entered as menu selections or in response to prompts do not require carriage returns, so the user must be careful when entering such responses.

In general, when the user is requested to enter a single character response to a prompt or as a menu selection either upper or lowercase letters will be accepted, but only characters matching the option or represented in the selections will be accepted.

The file names **tokbus**, **ClassLib** and **DftVals** are reserved; no class can be saved or report written with those names. The simulator also prevents the user from overwriting any of the user created station class files with reports or simulation traces. A station class file can be overwritten by another station class, but only after the user confirms his decision to do so.

3. Program Execution

The program is invoked by entering the command **tokbus** at the shell prompt. The first action of the program is the configuration of the network to be simulated.

The second function of **tokbus** is to provide the user with an environment to run the simulation and collect statistics.

3.1. Configuring the Simulator

There are two main steps in the configuration process. The first step is to determine the settings of system parameters such as the bus length, rate and propagation delay, the *High_Priority-Token_Hold_Time*, the use of 16 or 48 bit addresses, the size of the Token, and the *Token_Pass_Timer* and *Bus_Idle_Timer* values. The second step in the configuration process is the creation of the classes of stations in the network being tested; stations that are members of the same station class share common configuration parameters and exhibit common operating behavior.

3.1.1. System Parameterization

In setting the system parameters, the user has the option to use a set of default values saved in the file **DfltVals** or to enter the values in response to a series of prompts. The simulator needs the following information from the user or from the file **DfltVals**.

The random number seed.

The bus rate in bits per second.

The length of the bus in meters.

The propagation delay in nanoseconds per meter; which should also include the delays in the transmit machine and the receive machine.

The probability of a bit error; this is a constant error distribution which implies that if the probability is 10^{-6} then for every million bits transmitted the last bit will be in error.

The *High_Priority-Token_Hold_Time* which determines the maximum duration of *Synchronous access_class* Service at every station in the network.

The size of the address field which is either sixteen or forty eight bits. If

the address is sixteen bits, then ninety six bits of protocol addresses and overhead will be added to each message transmitted. If the address size is forty eight, then one hundred and sixty bits will be added to each message.

The size of the token. If the address is sixteen bits, the token must be at least ninety six bits. If the address is forty eight bits, the token must be at least one hundred and sixty bits. The token can be larger if the user is including data in the token frame.

The number of classes of stations involved in this simulation.

The *Token_Pass_Timer* delay which controls how long a station waits in the *Check-Token-Pass* state before it assumes that the token pass has failed. A station assumes that the token pass has failed when the network is silent after the expiration of the timer.

The *Bus_Idle_Timer* delay which controls how long a station that is in the *Idle* station state waits for any transmission before it enters the *Claim-Token* state.

Once the user has finished entering the parameters or if he has chosen to use the default values, he will be presented with the display of Figure 1 which consists of a number, a description of the variable, and the current value of each system parameter. He will be able to edit any of the values by entering the number displayed to the left of the description. For example to change the number of classes of stations:

The user enters a nine (9) in response to the field to change prompt.

The simulator overwrites the field to change prompt with **Enter the Number of classes of stations :** and reads the user's answer. The number of station classes must be greater than zero (0).

The simulator updates the display of Figure 1 and waits for the user's next selection.

By entering a zero (0) the user exits the editing of the system parameters.

After the user has entered a zero (0), he will be asked if he wishes to save the system parameters. If he responds with a Y, the current values will overwrite the old values in the default parameter file **DfltVals**. If the user selected to use the default values and did not change them there is no need to resave the default

```

1 - Random Number Seed : 25
2 - Bus Rate (in bits per second) : 1.0000000E7
3 - Bus Length (in meters) : 1.0000000E3
4 - Propagation Delay (in nsecs per meter) : 5.0000000E7
5 - Probability of a Bit Error : 0.0000000
6 - High Priority Token Hold Time(in seconds) : 0.0010
7 - Address Size : 16
8 - Size of the token : 96
9 - Number of Classes of Stations : 2

```

The Token Pass Delay Time : delay in the Check-Token-Pass state before the Token Holder assumes the Token pass failed.

Bus Idle Time Delay : how long a station listens to a quiet bus before entering the Claim-Token state.

```
10 - Token Pass Time Delay (in seconds) : 0.000100
```

```
11 - Bus Idle Time Delay (in seconds) : 0.000100
```

Number of Field to change (0 to stop) :

Figure 1

settings.

3.1.2. Station Class Creation

The next step in the configuration process is the creation of the classes of stations. A class of stations all have the same *access_classes* active, the same message arrival rates, sizes and length and creation distributions. They also have the same ring membership properties, (always members, initial members, or members who join after a specific time or join and leave depending upon their message queues) and *token_rotation_timers*. The user will have specified in the system parameterization portion of the configuration process how many station classes will be created.

tokbus allows the user to create and modify a library of saved classes which can be used in successive simulations. For each class of stations in the simulated

network, the user has the option of creating a new class or using an existing class. The names of the valid files that compose the library are saved in the file **ClassLib**. File names for station classes must adhere to the restrictions listed in section 2.

If the user selects to make a class, he will be asked to supply the following information.

The name of the Class.

The Number of Stations. There is a Maximum Number of Stations (512) for the simulator configuration.

Will the Priority Option for the *access_classes* be used.

If the Priority Option is not selected, the user will be asked to supply the *access_class* parameters for the *Synchronous access_class*, otherwise he will be asked to select the active *access_classes* and to enter their *access_class* parameters. The *access_class* parameters are:

the mean message creation rate (in messages per second),

the message creation distribution (Constant, Exponential, or Uniform),

the message length (in bits; not including the protocol overhead),

the message length distribution (Constant, Exponential, or Uniform),
and the *Target_Rotation_Time* value (except for the *Synchronous access_class*).

The type of Ring membership for stations of this class. The options are the stations will Always be members of the TokenPassing Ring, the stations will Join the ring after a user specified delay, the stations will be initial members of the Token Passing Ring, but they will Leave the ring after a user specified delay, or the stations can be dynamic members of the ring, entering or leaving as they have Traffic, messages to send. Stations which Join the ring can either remain as members or leave the ring after a further delay. Stations whose ring membership depends upon their Traffic can be initial members of the ring, and the user can specify a delay in token cycles or seconds from when the last message is sent to when the station will patch itself out of the ring. *Warning, at least two stations must be members of the token passing ring for the simulator to function.*

The *Max_Inter_Solicit* count. This value determines how often stations will open response windows to enable other stations to join the token passing ring. The integer entered must be between 16 and 255 and its value is randomized by plus or minus three digits each time it is accessed.

The Addresses of the stations. The user can specify a specific range of addresses for the stations, or he can allow the addresses to be assigned by the computer. There is a Maximum Address of 1024 ($2 * \text{Maximum Number of Stations}$).

The PreLoading of the station message queues. The user can select to preload the message queues for a class of stations. He can load all active queues (the *access_class* must be active), or he can select which of the active queues he would like preloaded. For each preloaded queue, the user will specify the number of messages to be loaded. The messages' length will be determined according to the message creation distribution and the mean message length. The user can set a flag such that when any of the queues are emptied the simulator will pause, and provide the user with the option of continuing the simulation, turning the empty queue flag off, or exiting to the command level.

After creating a new class or reading in an existing class, the user is presented with a display of the number of each field, the name of the field and its value, as shown in Figure 2. At the bottom of the screen the user is prompted to


```

      Editing the Class Information
1 - Name of Class = multiclass
2 - Number of Stations = 64
3 - Priority Option is Used.
      Synchronous      Urg Asynch      Nml Asynch      Time Avail
      4 - active        9 - active      15 - active      21 - inactive
Mean Msg Crt Rate : 5 - 244.00      10 - 61.00      16 - 61.00      22 -
Msg Crtn Distrib  : 6 - expont      11 - expont      17 - constnt      23 -
Mean Msg Length   : 7 - 160        12 - 160        18 - 160        24 -
Msg Lngth Dist    : 8 - constnt      13 - unifrm      19 - constnt      25 -
Target Rotation Time :              14 - 1.00e-2      20 - 1.10e-2      26 -

27 - Stations are Always Members of the Token Passing Ring.

29 - Stations open response windows after every 128 token cycles.
30 - Low Address = 1              31 - High Address = 64
32 - The Queues are not PreLoaded.

```



Enter the Number of the Field to be Changed (0 to stop) :

Figure 2.

enter the number of the field he would like to change or a zero (0) if the values are correct. If the user selects one of the numbers preceding a field, he will be allowed to change the value of the field. There are some instances where changing the value of one field will require the user to change other values as well. For example, using the class of Figure 2, if the user enters 32 he will be asked if he wishes the simulator to stop when the queues are emptied, and he will be prompted for which of the active *access_classes*' queues he wishes to preload. When the user enters a zero, if the class was either edited or created, he will be prompted to enter a Y if he wishes to save the class. If he elects to save a class it will be added to the class library under a name the user specifies. This editing allows the user to correct mistakes made while entering a class, and to modify an existing class for parametric studies.

3.2. Running the Simulator

When the user has completed the configuration process, **tokbus** presents the user with a menu of commands, Figure 3. The user can invoke the simulator

Enter Command Character :

Menu of Simulator Commands

- D - Display statistics and stations;
- E - set an Error condition;
- H - Help by displaying this menu;
- J - a station Joins the token passing ring;
- K - Kill a station;
- L - a station Leaves the token passing ring;
- R - Report statistics;
- S - run the simulator for a number of Steps;
- T - run the simulator for a period of Time;
- X - eXit the simulator.

Figure 3

commands by entering the letter to the left of the command description at the Enter Command Character : prompt. The simulator will always return to the main menu, unless through station departures there are not enough stations to maintain the token passing ring, or the user selects the X command.

3.2.1. Display Command

The Display command produces a menu of selections, Figure 4. The user will loop until he enters one of the correct character selections. The commands are :

O - toggle the screen display ON/OFF. When the display is ON the user's terminal will resemble Figure 5 otherwise it will show the main menu. The screen display is off by default. Turning the screen display on slows the simulator because of the extensive I/O involved with constantly refreshing the displays, but the display can provide a user who is unfamiliar with 802.4 a better understanding of the protocol as he watches the actions in the simulated network.

S - give the full station display to a specific station. The full station display shows the station's address, the current station state, the time left for service at this station, the priority level being served, and the number of messages and the *token_rotation_timer* values for the active *access_classes*. This display covers

Display Commands; Please Enter Selection :

```
O - turn On Station Display;
S - give the full display for a Station;
T - give the full display for the Token Holder;
X - eXit to Command Level;
```

Figure 4

```

Enter Command Character :
elapsed time = 0.01050512          # of token cycles = 30
 20      21      22      23      24  25      26      27      28
idle    idle    joining    idle    idle usetokn    idle    idle    idle
 29      30      31      32      33      34      35      36      37
idle    dead    idle    idle    idle unpowrd    joining    idle    idle

...

address = 25          state = use token
0.00030871 seconds left for Synchronous service
Access Class      # Messages Queued      Token Rotation Timer
Synchronous              2
Urgent Asynchronous      4              0.01067810
Normal Asynchronous      2              0.01050327
Time Available          inactive

```

Figure 5

the bottom seven lines of the users terminal. The user is prompted to enter the address of the station he would like to have displayed in full. If the user enters a zero (0), the station display will not be changed.

T - give the full station display to the current token holder. This is the default condition.

X - exit to the main menu without invoking any display commands.

3.2.2. Error Command

The Error Command presents the user with a menu of the following selections.

K - to kill the next message that is transmitted.

N - to set a bus noise condition. The user will be prompted to enter the duration in seconds of the bus noise condition. For the duration of the noise condition the bus will not be usable for transmission of data or protocol frames. If the duration of the noise spans the token passing process, the token

will be lost and the ring will reconfigure itself with the lowest addressed station claiming the token.

X - exit to main menu.

3.2.3. Help Command

The Help command will display the main menu, Figure 3, on the users terminal. If the user has turned the display on with the **D** command, the main menu will not be displayed between commands. If the user needs to see the list of commands and the corresponding command character, it can be recalled without turning off the screen display.

3.2.4. Join Command

The Join Command allows the user to select a station that is waiting to join the ring and power the station up setting the station's *in_ring_desired* flag to true. When the selected stations predecessor in the token passing ring opens a response window, the station will be able to join the token passing ring.

In the network displayed in Figure 5, station 34 could be powered up by:

Entering a **J** at the Enter Command Character : prompt.

The user is prompted to Enter Station Address (0 to quit) :

The user would enter **34** to select station 34. The simulator will loop prompting for the station address until the user enters a valid address or a zero (0) to abort the command.

Station 34 would be set to the joining state and the display would be updated.

3.2.5. Kill Command

The Kill Command allows the user to select a station and to kill the station. If the station is the token holder, its successor station will have its *bus_idle_timer* timeout, and will enter the *Claim-Token* station state. When the successor station

wins the contention for the token it will enter the *Use-Token* state and message service will resume.

In Figure 5, station 30 has been killed by the user.

3.2.6. Leave Command

The Leave command allows the user to select a station that will patch itself out of the token passing ring when it next receives the token.

3.2.7. Report Command

The Report command presents the user with a menu of options Figure 6. The options are of two types; make a report of simulator statistics or toggle traces of network states and transitions.

If the user selects a report, he can either have it displayed on the terminal or written to a file of his selection. If a report is displayed on the terminal, it will pause after it has printed the report and ask the user to enter a character to continue. *In the Vax implementation the user must enter a CHARACTER and a CARRIAGE RETURN.*

Enter Selection :

```

B - Report Bus Statistics;
C - Report Class Statistics;
E - Turn On Execution Trace;
M - Turn On # of Messages Transmitted Trace;
Q - Turn On Queue Level Trace;
T - Turn On Token Trace;
X - eXit to Command Level;

```

Figure 6

busout.6

```

Elapsed Time = 16.00000000
Token Cycles = 6023.
Average Token Cycle Time = 0.00265606
Minimum Token Cycle Time = 0.00111300
Maximum Token Cycle Time = 0.00446460
Total Traffic of 761333. messages for a total of 132981728. bits transmitted
Number of Ring Reconfigurations = 0

```

Type Of Message	Number Transmitted	Total Bits	%BandWidth
Data	374336.	95830016.	59.89
Overhead		35936256.	22.46
Actual Data		59893760.	37.43
Token	385525.	37010400.	23.13
Protocol	1472.	141312.	0.09
Corrupted	0.	0.	0.00
Propagation Delay			16.77
Bus Idle			0.11

Figure 7

The traces are all written to files. When the user selects the type of trace he wishes to have generated, he is asked to enter the name of the output file. The user will not be able to name his output file **DfltVals**, **ClassLib**, or the name of any file that is in the class library. The same restriction applies to written report files.

B - Report the Bus statistics. This generates the report of Figure 7 which provides the user with figures on total bus usage from which he can gauge network performance.

C - Report the Class statistics. If there are more than one classes of stations in the simulation the user will also be asked to select the class he would like to have reported or to have all classes reported. If the user selects all classes and has selected to have the report displayed on his terminal, the display will pause after each class. A selection of all classes and writing the report to a

file will create one file with each class reported separated by blank lines. Figure 8 shows a sample class report for a class of 64 stations that are only transmitting from the *Synchronous access_class*.

E - toggle the Execution trace. The execution trace writes a trace of all message transmissions and their sources, and all protocol events such as the passing or receipt of the token, the checking of the *access_classes*, the failure of token passes or stations, and the death, departure, joining, and powering up of a station. *The execution trace produces copious amounts of output; use it sparingly.*

```

clsout.6
Class basecase
  Stations have joined the Token Passing ring      64 times.
  Synchronous Service Access Class
  Total Number of Messages sent from these queues =      374337.
  Average Queue Delay =                                0.00035376
  Minimum Queue Delay =                                0.00000000
  Maximum Queue Delay =                                0.00411018
  Average Access Delay =                                0.00098775
  Minimum Access Delay =                                0.00000000
  Maximum Access Delay =                                0.00411018
  Average Transmission Time =                           0.00002760
  Minimum Transmission Time =                           0.00002760
  Maximum Transmission Time =                           0.00002760
  Average Delivery Time =                                0.00136911
  Minimum Delivery Time =                                0.00002762
  Maximum Delivery Time =                                0.00436294

  Urgent Asynchronous Service Access Class
  Access Class Inactive

  Normal Asynchronous Service Access Class
  Access Class Inactive

  Time Available Service Access Class
  Access Class Inactive

```

Figure 8

M - toggle the Message Transmission trace. The Message Transmission trace reports the address of the token holder, the time that the token arrived at the station, and the number of messages transmitted by the station while it held the token.

Q - toggle the Queue trace. The Queue trace reports the address of the new token holder, the number of messages enqueued when the token arrives, and the time of the token arrival.

T - toggle the Token Cycle Time trace. The token cycle count is incremented each time the token is passed from the lowest addressed station to its successor except for the initial token transmission on simulator startup. The Token Cycle Time trace reports the number of completed token rotations, the duration of the last token rotation, and the time when the cycle was completed.

X - eXits to the main menu.

3.2.8. Step Command

The Step command allows the user to enter the number of steps he wishes the simulator to take before returning to the command level. Each step is equivalent to removing the next event from the simulator's event queue and processing the event. If the user enters a number less than zero, this will be taken as a null operation and the simulator will return to the main menu. In the Apollo implementation the user can step the simulator one step by entering a Carriage Return at the Enter Command Character : prompt.

A simulator step can appear to have more than a single action by the simulator and can appear to have no visible action. For example, if the token holder has entered the *Pass-Token* state a single step could go as far as the receipt of the token at the next station, and the start of a transmission from the next station. A simulator step can also appear to have no action if the next event was the

arrival of a message to a station's queue.

3.2.9. Time Command

The Time Command allows the user to run the simulator for a period of seconds. Once the simulator has been started on a time command, control will return to the command level in three ways; time will advance the number of seconds the user specified, if a station with the EmptyQueue flag set empties one of its queues, if through station departures from the token passing ring there are not enough stations to form a ring (number of stations < 2).

If the user enters a value less than or equal to zero the simulator will return to the command level without processing any events.

3.2.10. Exit Command

The Exit command stops the simulator and returns the user to the operating system. Any trace files are automatically closed when the user exits the simulator.

Notes

There is an ambiguity in the standard about the action to be taken when successive stations patch themselves out of the token passing ring. This ambiguity can best be demonstrated with an example.

Consider five stations 10, 15, 20, 25, and 30.

30 passes the token to 25.

25 has no messages to send and to patch itself out of the ring it sends a Set_Successor frame to 30 telling 30 that its successor will be 20. After the Set_Successor transmission, 25 passes the token to 20 and leaves the ring.

20 receives the token from 25 and repeats the process of sending the Set_Successor frame and then passing the token. However, 20 has sent the Set_Successor frame to 25 not 30 so 30 still thinks 20 is its successor.

15 receives the token from 20 and also patches itself out of the ring.

When 30 tries to pass the token, it will attempt to pass the token to 20, a station that has left the ring. If 20 does not accept the token and then patch itself out again then station 30 will have to use the error recovery features to rebuild the token passing ring. If 20 does accept the token it might not have a valid view of the token passing ring.

tokbus handles the ambiguity by ignoring the receipt of the token by any station that is dead or has left the token passing ring.

APPENDIX B

Apollo Implementation of the Simulator

1. Introduction

The IEEE 802.4 token bus simulator **tokbus** was originally developed on a Digital Equipment Corporation VAX 11/780 under the 4.1 distribution of Berkeley UNIX. The program was ported to a network of Apollo DN-300 workstations. Several modifications to the original program were needed to accomplish the port.

Many of the modifications were minor mechanical changes caused by the different implementations of the Pascal programming language, differences such as different reserved words, a different syntax for include files, and a different convention for file access. The major modifications were caused by the radically different methods of screen control on the Apollos than on the VAX; the procedures for reading user input and displaying simulator output required major changes to provide the user with a common interface to both versions of the simulator. Some of the modifications resulted in improvements in the original version of the program.

2. Minor Modifications

The implementation of Apollo's DOMAIN Pascal required several small changes in the Apollo version of the simulator. DOMAIN Pascal has several different keywords; the simulator had to be changed to reflect these differences. For example, the phrase "next" is a keyword; "next" was used in many record declarations as a pointer to dynamic instantiations of the record. It was a simple procedure to edit the files and change all occurrences of "next" to "nxt". The modifications needed to

accommodate the other keyword and syntactic differences were also easily performed through the editing process.

Other modifications were because of the different conventions for file access. Berkeley Pascal has a very limited interface with the UNIX file system. Apollo's Pascal has a much more extensive interface with the file system. A peculiarity in the file system implementation on the Apollos does not allow Pascal programs to reference file names that begin with uppercase characters or digits. For this reason, the reserved file names **ClassLib** and **DfltVals** are **classlib** and **dfltvals** on the Apollos. The Apollo file interface does allow a Pascal program to test for the existence of files. This improvement, allows the program to recover in the absence of the files **dfltvals** and **classlib** when the user selects to use the defaults or a saved class entry; the simulator has values for each variable that will be used if the files do not exist, and the user will be presented with these values at the editing stages in the simulator's network configuration process.

Another limitation to the Pascal implementation on the Apollos was the lack of a Pascal random number generator. Random numbers are needed for the scheduling of events and the determination of non-constant message sizes. DOMAIN C does provide an integer random number function, and it was possible to use the C random number function to provide the real random numbers needed by simulator. Unfortunately and contrary to the documentation, the C random number function can return a value of zero which is an illegal value for the simulator's use. To circumvent this limitation a small value of 10^{-15} was added to all real number values returned.

3. Major Modifications

The major modifications to the program were caused by the Apollo's method of screen management. The simulator needs full control of the cursor so that it

can correctly update the screen displays during the network configuration and simulation phases. This control is straight forward in the VAX implementation of **tokbus**, but it was difficult to obtain and had large penalties associated with it in the Apollo implementation. In addition to the obvious and expected changes needed in the input-output procedures, all portions of the simulator dealing with the user interface unexpectedly had to be restructured.

After much experimentation it was found that to obtain the full screen addressing needed for the user interface, the standard files input and output had to be set to the "raw" mode. In "raw" mode full cursor control is available, but the only input procedure available is the read character routine. To read both integer and real numbers, the numbers had to be read a character at a time and then converted to the correct representation. Having to create numbers from characters did cause one improvement in the Apollo implementation, non-numeric input could be ignored when reading a number making the simulator more tolerant of user mistakes.

The major disadvantage to using the "raw" mode of input and output was that when in "raw" mode simulator input and output could not be redirected to either read from or write to files. As stated in Chapter 5, the performance studies were accomplished by using the simulator in a "batch" mode instead of interactively. The Apollo implementation of **tokbus** does have the ability to read simulator commands and user inputs from a file. DOMAIN Pascal allows a program to test for the existence of a file on the command line when the program is invoked. The simulator used this feature to test whether it was being run in batch or interactive mode. The disadvantage to this feature was that separate reads of every user input had to be encoded; the simulator needed the ability to read from both an input file and from standard input.

4. Conclusion

There are several limitations to the Apollo implementation of the Pascal programming language, and there are some advantages to it also. The requirement of using "raw" input and output to obtain full screen control is a serious drawback to using the Apollos in applications such as this simulator. Despite the problems, the Apollo version of **tokbus** was used extensively in the thesis' performance tests, and it performed very well.

APPENDIX C

Simulator Test Cases and Results

The two tables below represent the information used in the base simulations. The first table lists the system wide network and simulator parameters. The second table lists the class information.

Simulator Default Values	
Random Number Seed	0
Bus Rate in bits per second	1.00E+07
Bus Length in meters	1.00E+03
Propagation Delay in nanoseconds per meter	5.00E+00
Probability of a Bit Error	0.00E+00
High_Priority_Token_Hold_Time	1.00E+01
Address Size in bits	16
Token Size in bits	96
Number of Station Classes	1
Token_Pass_Time in seconds	1.00E-05
Bus Idle Time in seconds	1.00E-05

Table I.

Base Class Information	
Class Name	basecase
Number of Stations	64
Priority Option	not used
Synchronous Message Creation Rate	61
Message Creation Distribution	Exponential
Mean Message Length	160
Message Length Distribution	Constant
Stations are	Always Members
Station Addresses are	Random
Token Cycles Between Response Windows	255
Queues are	not Preloaded

Table II.

The following tables represent the results of the nineteen simulations used in testing the base configuration. Table III is a synopsis of the class reports; it presents entries for the offered load, the number of messages transmitted, and the

average values for the queueing and access delays and the transmission and delivery times. Tables IV and V list data from the bus reports. In Table IV the number of token cycles and the average token cycle time are presented in addition to the total number of messages transmitted and the decomposition into the types of messages transmitted. Table V list the percentages of bus utilization for data messages, the proportion of data messages that was overhead and actual data, tokens, protocol frames, the bandwidth consumed by propagation delays, and the time the bus was idle.

Class Report Statistics					
Offered Load	# Mssgs Xmitted	Ave Queue Delay	Ave Access Delay	Ave Xmit Time	Ave Deliv Time
0.10	62455	0.00001099	0.00051851	0.00002760	0.00055710
0.15	93738	0.00001871	0.00054347	0.00002760	0.00058979
0.20	124785	0.00002883	0.00057228	0.00002760	0.00062871
0.25	156010	0.00004072	0.00060357	0.00002760	0.00067189
0.30	187202	0.00005620	0.00064147	0.00002760	0.00072527
0.35	218504	0.00007618	0.00068278	0.00002760	0.00078656
0.40	249680	0.00010322	0.00073066	0.00002760	0.00086148
0.45	280924	0.00013860	0.00078151	0.00002760	0.00094771
0.50	312064	0.00018722	0.00084095	0.00002760	0.00105577
0.55	343227	0.00025549	0.00090995	0.00002760	0.00119304
0.60	374337	0.00035376	0.00098775	0.00002760	0.00136911
0.65	405591	0.00050506	0.00107878	0.00002760	0.00161144
0.70	436744	0.00074400	0.00118703	0.00002760	0.00195863
0.75	467918	0.00115999	0.00131071	0.00002760	0.00249829
0.80	499199	0.00198984	0.00144798	0.00002760	0.00346542
0.85	530276	0.00399250	0.00158738	0.00002760	0.00560749
0.90	561193	0.01310391	0.00170786	0.00002760	0.01483937
0.95	577434	0.22129283	0.00171388	0.00002760	0.22303431
1.00	578536	0.57830183	0.00163870	0.00002760	0.57996813

Table III.

Bus Report						
Offered Load	# Token Cycles	Ave Token Cycle Time	# Mssgs Xmitted	# Data Xmitted	# Token Xmitted	# Protocol Xmitted
0.10	15169	0.00105473	1037090	62454	970860	3776
0.15	14252	0.00112262	1009434	93738	912157	3539
0.20	13341	0.00119926	981968	124784	853856	3328
0.25	12426	0.00128756	954381	156010	795299	3072
0.30	11510	0.00139001	926770	187202	736688	2880
0.35	10593	0.00151036	899114	218504	677986	2624
0.40	9678	0.00165313	871528	249680	619429	2419
0.45	8762	0.00182593	843913	280924	560813	2176
0.50	7850	0.00203809	816403	312063	502420	1920
0.55	6935	0.00230701	788817	343226	443863	1728
0.60	6023	0.00265606	761333	374336	385525	1472
0.65	5106	0.00313344	733666	405591	326795	1280
0.70	4193	0.00381566	706143	436743	268376	1024
0.75	3279	0.00487820	678602	467917	209917	768
0.80	2361	0.00677503	650912	499198	151138	576
0.85	1450	0.01102720	623457	530275	92862	320
0.90	543	0.02945076	596089	561192	34769	128
0.95	67	0.23564788	581736	577433	4303	0
1.00	34	0.42279754	580755	578535	2220	0

Table IV.

Bus Utilization Percentages							
Offered Load	Data Traffic	Overhead	Actual Data	Token Traffic	Protocol Traffic	Propagation Delays	Bus Idle
0.10	9.99	3.75	6.25	58.25	0.23	31.24	0.29
0.15	15.00	5.62	9.37	54.73	0.21	29.79	0.27
0.20	19.97	7.49	12.48	51.23	0.20	28.35	0.26
0.25	24.96	9.36	15.60	47.72	0.18	26.90	0.24
0.30	29.95	11.23	18.72	44.20	0.17	25.45	0.22
0.35	34.96	13.11	21.85	40.68	0.16	24.00	0.20
0.40	39.95	14.98	24.97	37.17	0.15	22.55	0.19
0.45	44.95	16.86	28.09	33.65	0.13	21.10	0.17
0.50	49.93	18.72	31.21	30.15	0.12	19.66	0.15
0.55	54.92	20.59	34.32	26.63	0.10	18.22	0.13
0.60	59.89	22.46	37.43	23.13	0.09	16.77	0.11
0.65	64.89	24.34	40.56	19.61	0.08	15.32	0.10
0.70	69.88	26.20	43.67	16.10	0.06	13.88	0.08
0.75	74.87	28.08	46.79	12.60	0.05	12.43	0.06
0.80	79.87	29.95	49.92	9.07	0.03	10.98	0.04
0.85	84.84	31.82	53.03	5.57	0.02	9.54	0.02
0.90	89.79	33.67	56.12	2.09	0.01	8.11	0.01
0.95	92.39	34.65	57.74	0.26	0.00	7.35	0.00
1.00	92.57	34.71	57.85	0.13	0.00	7.30	0.00

Table V.

Bibliography

- [Albrecht 82]
David C. Albrecht, "Network Protocol Design: A Simulation Approach", Master's Thesis, DAMACS Report No. 82-09, University of Virginia, Department of Applied Mathematics and Computer Science (November 82).
- [Colvin 84]
M. Alex Colvin and Alfred C. Weaver, "Modeling the IEEE Token Bus", Master's Thesis, DAMACS Report No.84-03, University of Virginia, Department of Applied Mathematics and Computer Science, (June 1984).
- [Fuhrmann 84]
S. W. Fuhrmann and Robert B. Cooper, "Application of Decomposition Principle in M/G/1 Vacation Model to Two Continuum Cyclic Queueing Models (Especially Token Ring LANs)", Submitted for publication, (June 14, 1984).
- [IEEE 802, 82]
IEEE 802 Committee, *IEEE Project 802 Local Network Standards, Draft C*, IEEE (May 17, 1982).
- [IEEE 802.4, 83]
IEEE 802.4 Subcommittee, "Token-Passing Bus Access Method", in *IEEE Project 802 Local Network Standards, DRAFT E*, IEEE (July 1983).
- [Lavenberg 83]
Stephen S. Lavenberg and Charles H. Sauer in *Computer Performance Modeling Handbook*, Steven S. Lavenberg ed, Academic Press, New York, New York. pp 57 (1983).
- [Tannenbaum 81]
Andrew S. Tannenbaum, *Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ, (1981).
- [Weaver 84]
Alfred C. Weaver and David W. Butler, "A Fault Tolerant Network Protocol For Real Time Communications", Proceedings 1984 International Conference on Industrial Electronics, Control and Instrumentation, (October 1984).
- [Xerox 82]
Xerox Corporation, *Ethernet: A Local Area Network*, (November 1982).