Performance Comparison of DQDB and FDDI for Integrated Networks

Farzad Tari
Victor S. Frost

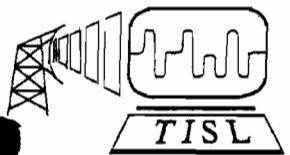
TISL Technical Report TISL-8580-3

Prepared for:
NCR Corporation
3718 N Rock Road
Wichita, Kansas 67226

and

Kansas Technology Enterprise Corporation (KTEC)
112 W 6th, Suite 400
Topeka, Kansas 66603

July 1990



Telecommunications and Information Sciences Laboratory
The University of Kansas Center for Research, Inc.
2291 Irving Hill Drive Lawrence, Kansas 66045

Abstract

The performance measures of two high speed network protocols, IEEE 802.6 Distributed Queue Dual Bus (DQDB) MAN and Fiber Distributed Data Interface (FDDI), were compared using computer simulation techniques. As an analysis tool, a simulation model was implemented to evaluate the steady state performance measures of the DQDB protocol. The performance measures of the simulation model were validated with those released by the IEEE 802.6 working group. The two protocols were then compared on the basis of their throughput, delay, network utilization, and fairness characteristics using an integrated network consisting of voice, video, and data stations. The performance measures of FDDI were obtained using the existing General LAN Analysis System (GLAS).

Table of Contents

Chapter 1

Introduction

Metropolitan Area Networks (MANs) are designed to carry different types of communications traffic simultaneously, allowing more stations to communicate over longer distances and at greater speeds than LANs with enhanced services. MANs can provide backbone networks by connecting homes, small businesses, and LANs. They are also designed to accommodate integration of voice, video, and data traffic.

Although the concept of metropolitan area networks seems to spring from the LAN, LAN technologies cannot be applied to MANs due to the inherent inefficiencies in most LAN Media Access Control (MAC) schemes. Two protocols that are currently being considered for metropolitan area networks are the Distributed Queue Dual Bus (DQDB) protocol proposed by the IEEE 802.6 working group and the Fiber Distributed Data Interface (FDDI) protocol of the American National Standard Institute (ANSI).

1.1 Problem Statement and Thesis Contributions

Although metropolitan area network protocols, in particular DQDB and FDDI, are designed to support different types of communications traffic simultaneously, the performance characteristics of these protocols for integrated traffic and multiple

priority asynchronous services need to be further studied. The current trend for supporting services like voice and video on computer networks seems to be centered around the idea of packetization and coding of these services in such a way that they can be supported using asynchronous data transmission schemes. However, the feasibility of such techniques has not been studied for the two protocols considered in this thesis. Also, it is important to know the effects of supporting high speed services like packet video using the asynchronous priority mechanism of FDDI and DQDB on the performance characteristics of other asynchronous service users on the network which have smaller service priorities. Finally, the performance comparison of the two protocols for such integrated networks has not been studied. DQDB and FDDI use fundamentally different schemes for supporting isochronous services (like PCM voice) and asynchronous services (like computer data). For integrated traffic on metropolitan area networks, the relative advantages and disadvantages of using one protocol over the other needs to be studied.

The contributions of this thesis to the above problems are the followings:

1. The feasibility of supporting packet video on the highest asynchronous class is studied for both protocols. The two protocols are compared on the basis of their throughput and delay performance.

2. The performance characteristics of the isochronous services as well as lower priority asynchronous services of the two protocols are compared. The performance characteristics considered here are delay, throughput, and network utilization.
3. In an integrated network, for lower priority asynchronous users (like computers), the two protocols are compared on the basis of their fairness in serving stations at different locations on the network.
4. A detailed simulation model of the IEEE 802.6 DQDB protocol is constructed using the discrete event scheduling techniques. This model is used to evaluate the performance of the DQDB protocol. The model is validated by making comparisons of the performance characteristics produced by the model to the ones released by the IEEE 802.6 committee.

1.2 Overview of the Project

To study the performance characteristics of the DQDB protocol [3-6] a simulation model of the protocol was constructed in C language. The model which is developed using discrete event scheduling provides the steady state performance measures of the DQDB network. The performance measures are given on both global and per-node basis. Isochronous services as well as single priority and multiple priority cases of asynchronous services can be

simulated using the DQDB model. However, operations such as network initialization functions and reconfiguration due to a bus failure which do not pertain to the steady state operation of the network are not included in the model. To help understand the modeling techniques and the capabilities and limitations of the model, the next chapter is devoted to the explanation of the DQDB protocol. The third chapter includes the modeling details.

The simulation model was validated by comparing the performance measures generated by the model to those obtained by Newman [9]. Both single priority and multiple priority cases were considered. The validation procedures and results are presented in Chapter 4.

The simulation model explained above was used to make comparisons between performance measures of the DQDB protocol to those of FDDI [10-13]. The FDDI protocol was modeled using the General LAN Analysis System (GLAS) package [14]. An integrated network consisted of voice, video, and data stations with line rate of 100 Mbps and length of 200 km was considered. The two protocols were compared on the basis of their throughput, delay, network utilization, and fairness characteristics. The results of this comparison are presented and analyzed in Chapter 5.

Conclusions and open issues for further research are discussed in Chapter 6.

Chapter 2

The DQDB MAN

The IEEE 802.6 which is known as the Disturbed Queue Dual Bus (DQDB) is the IEEE standard for Metropolitan Area Networks. Although the protocol uses the dual bus architecture which is similar to that of Fasnet [1], and Express net [2], the access method is completely different. Also, the DQDB MAN does not have a limited line rate or maximum network length. This chapter includes architectural perspectives as well as a description of the access method of the DQDB [3–6]. A simple example of the operation of the distributed queue is also presented.

2.1 Architectural Perspectives of the DQDB MAN

Some of the most important architectural aspects of the DQDB MAN are described in this section. The DQDB bus and protocol architecture are also discussed.

2.1.1 DQDB Bus Architecture

Each DQDB uses a pair of counter flowing unidirectional buses, known as a dual bus pair. One bus is denoted bus A and the other is denoted bus B. Because both buses are operational at all times, the capacity of the network is twice the capacity of a single bus. The transmission link that connects two adjacent nodes in a network is a full duplex connection between the nodes that carries both bus A and bus B. A given node, however, must know which bus

to use to communicate with another node. The transmission on each bus are formatted as fixed length entities called "slots." All slots are generated by the station at the head of the bus known as the "slot generator" and are terminated at the end of the bus. Figure 2.5 (section 2.2.1) shows the format of a slot. The fields within the slots that can be altered by a station are initialized to zero by the slot generator at the head of each bus. The stations are attached to both buses via read and write connections. As shown in Figure 2.1, the writing on the bus is by logical OR of data from upstream with the data from the station. The read connection is placed ahead of the write connection. Nodes observe the data passing on a bus, but never remove it and only alter it when permitted by the access protocol. The bus architecture of the DQDB MAN has been chosen for its reliability. As can be seen from Figure 2.1, data passes by the node and not through the node as is the case in ring networks. Hence, the DQDB nodes can fail or even be removed from the network with no consequence on the operation of the rest of the network.

DQDB networks can operate in either of the two topologies shown in Figure 2.2. The looped dual bus topology is an extension of the open bus topology with the only change being that the end points of the buses are co-located. Data do not flow through the head point of the loop. Thus, while the looped bus architecture may appear similar to a ring, it has no logical similarity to it.

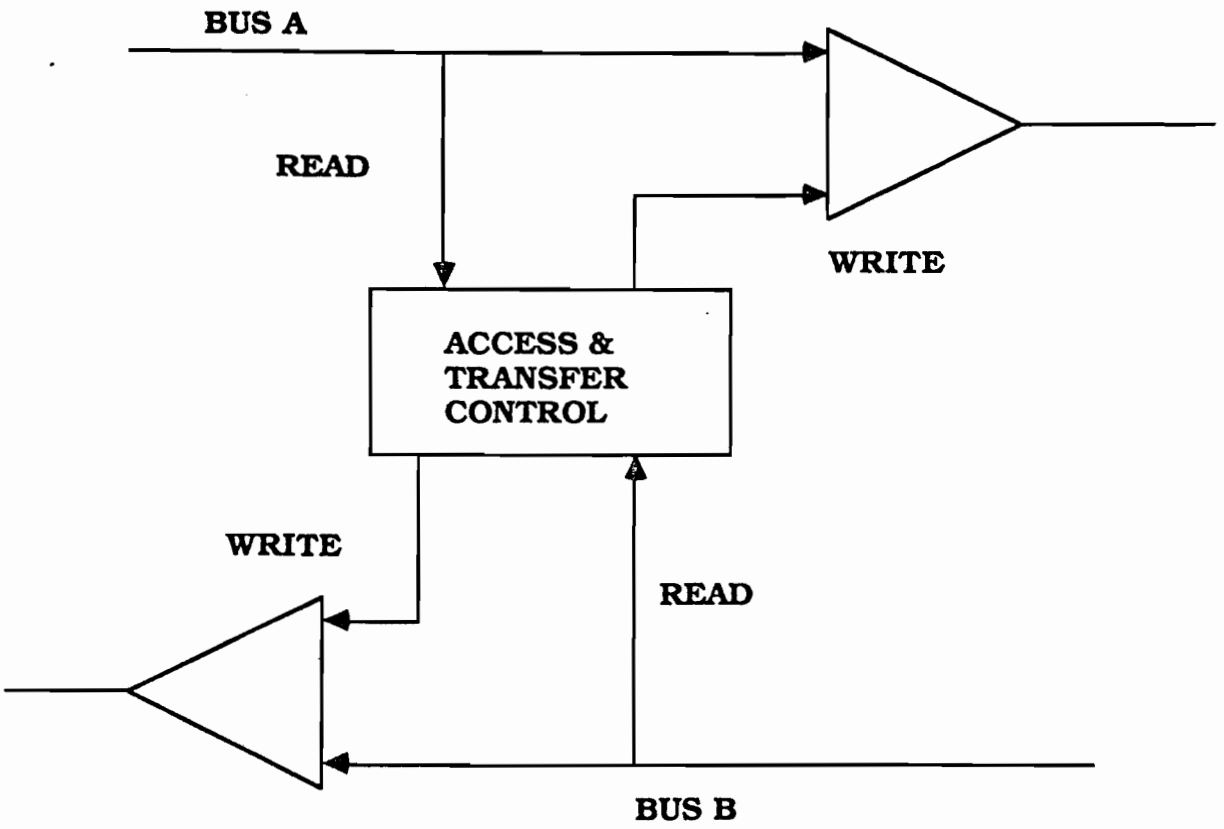
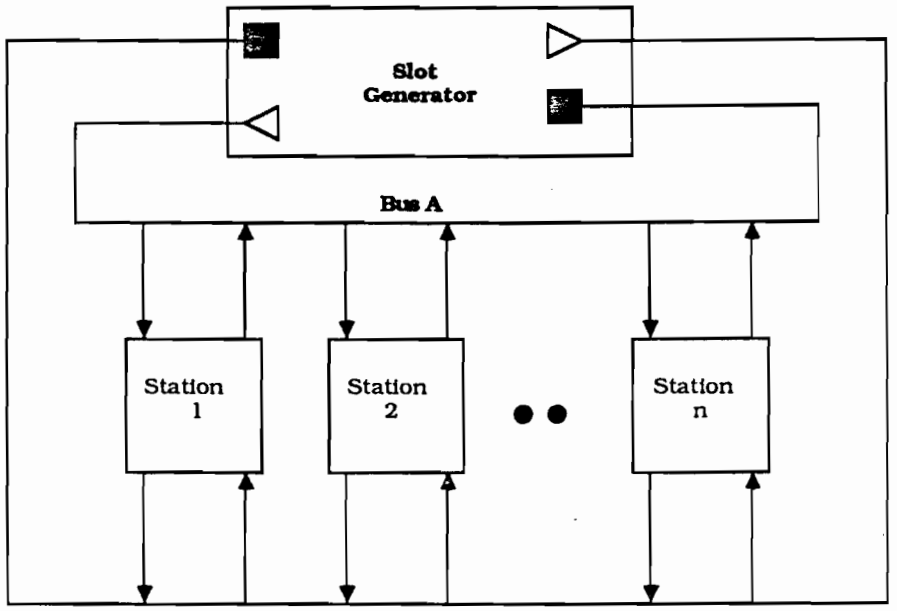
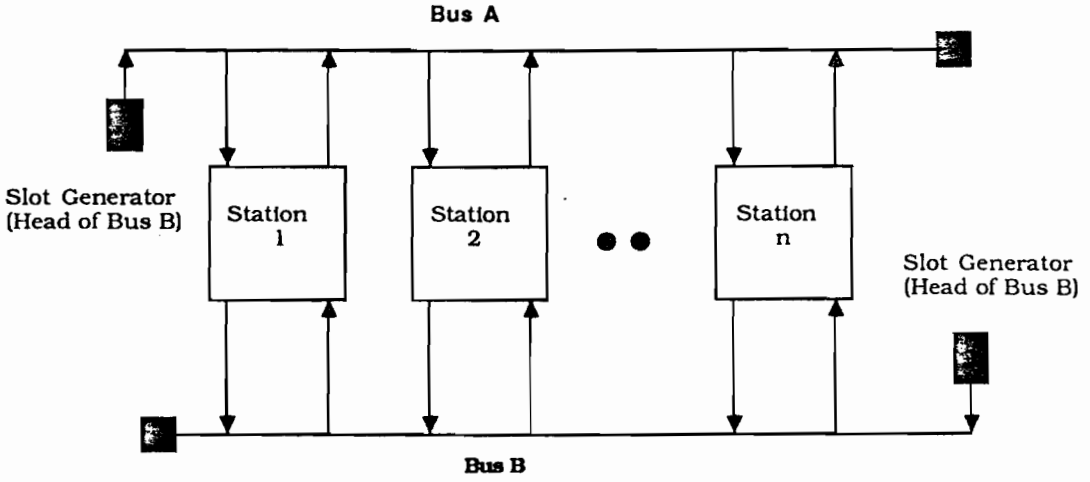


Figure 2.1: Bus Architecture



Looped Dual Bus Topology



Open Dual Bus Topology

Figure 2.2: Bus Topologies for DQDB

2.1.2 Protocol Architecture

The basic protocol architecture of the 802.6 protocol is shown in Figure 2.3. The 802.6 physical layer corresponds to the OSI physical layer and specifies how to use different underlying transmission media and speed. The Physical Layer Convergence Protocol (PLCP), part of the physical layer, adopts the capabilities of the transmission system to provide the service expected by the DQDB Layer. The PLCP will be different for every transmission system, but it is this part of the physical layer that allows the wide range of media and speed options supported by the network.

As shown in Figure 2.3, the DQDB protocol is designed to support three types of services: a) Connectionless (datagram) MAC service to the IEEE 802.2 Logical Link Control (LLC) sublayer; b) Connection-Oriented (Virtual Circuit) data service for the transfer of bursty data, such as signaling or packetized voice; and c) Connection-Oriented (Virtual Circuit) isochronous service where "isochronous" refers to the timing characteristics of event or signal recurring at known, periodic intervals, such as conventional digitized voice or video.

The Asynchronous Transfer Control (ATC) and the Isochronous Transfer Control (ITC) provide a connection-oriented data transfer service to all asynchronous transfer users and isochronous service users, respectively. They also perform all functions that are necessary to adapt the information provided by

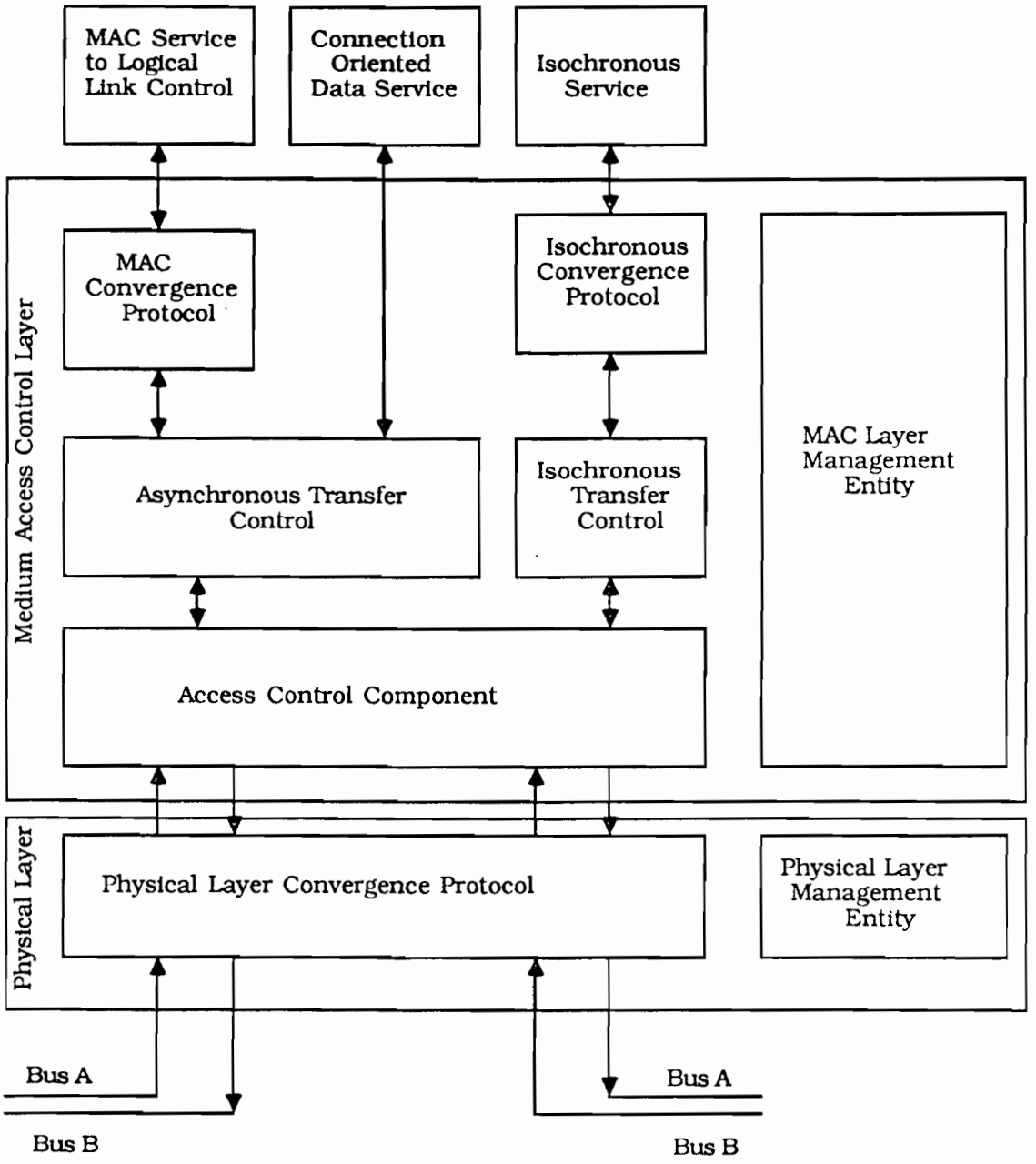


Figure 2.3: Protocol Architecture

the higher layers to the format required by the Access Control Components.

The MAC Convergence Protocol (MCP) supports a segmentation and reassembly function which is responsible for adapting the connection-oriented segment based service offered by the ATC entity to the standard MAC service required by the LLC sublayer. On the other hand, the Isochronous Convergence Protocol (ICP) provides a smoothing function to match the isochronous rate required by the isochronous service users to the actual transfer rate offered by the ITC entity. Because isochronous users of a single station might have different rate requirements, there is an ICP associated with each isochronous user.

The Layer Management Entities (LMEs) provide network management functions for each layer, compatible with the network management procedures of IEEE 802.1. The Access Control Component (ACC) performs all the read and write operations for the DQDB access protocol. It transforms the full duplex transmission connection presented by physical layer into the dual unidirectional buses required for operation of the DQDB access protocol. The internal structure of the ACC is shown in Figure 2.4. Each ACC contains a pair of Configuration Control and Slot Generator (CCSG) entities. The CCSG performs the timing operations. Also, if the node is the head of a bus, this entity generates correctly formatted slots and MAC management information. The medium access control for transfer of information

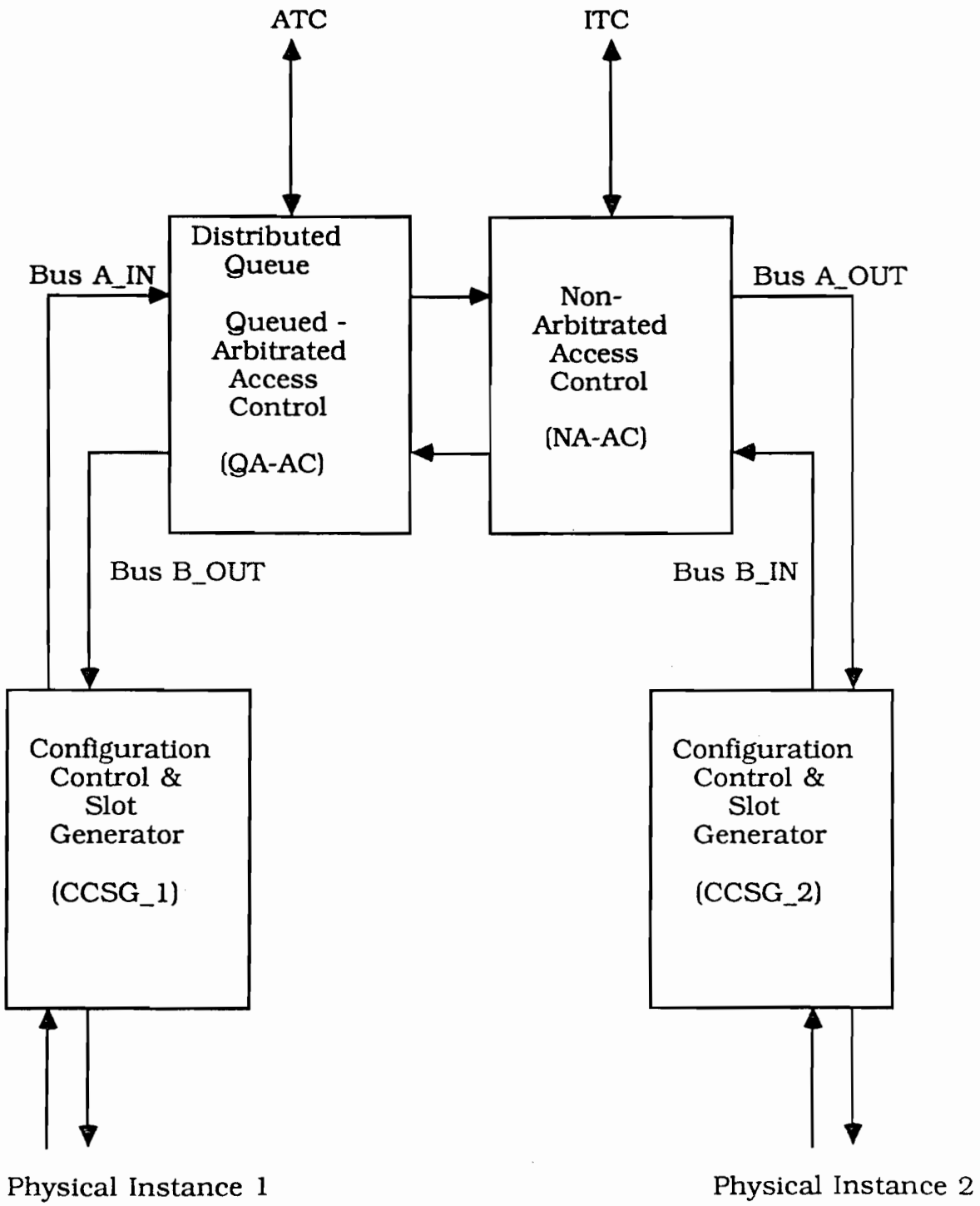


Figure 2.4: Access Control Component

between peer Asynchronous Transfer Control entities and peer Isochronous Transfer Control entities is done by Queued Arbitrated Access Control (QA-AC) and Non-Arbitrated Access Control (NA-AC), respectively. The operations of QA-AC and NA-AC are explained in the next subsection.

2.2 Access Control Protocol

This section includes the slot access scheme for both the Queued Arbitrated Access Control (QA-AC) and Non-Arbitrated Access Control (NA-AC).

2.2.1 Queued-Arbitrated Access Scheme

The QA access method supports services that are usually bursty in nature. The unit of transmission is called a "QA Slot" which has the format shown in Figure 2.5. Fixed length QA segments are written into empty QA slots under the control of the Distributed Queue. The distributed queue is a function which allows the formation and operation of a queue of QA segments which is distributed across the network. It gives performance similar to that of a centralized queue. Also, the QA-slot access protocol supports four priorities, from priority 0 to 3 with priority level 3 being the highest priority level.

Access Control Field	Segment
1 Octet	68 Octets

Slot Format

BUSY	SL_TYPE	RESERVED	PSR	REQUEST
1 Bit	1 Bit	1 Bit	1 Bit	4 Bits

Access Control Field

Figure 2.5: Slot Format

2.2.1.1 Operation of the Distributed Queue

The operation of the distributed queue is specified by the Distributed Queue State Machine (DQSM) and the Request Queue Machine (RQM). There are eight instances of the DQSM and RQM in each node: one for each possible combination of requested bus and access queue priority. This can be seen in Figure 2.6. Also, this figure shows that all instances of the DQSM for a particular bus pass information between themselves about requests for access made by the node itself. These requests are called self-requests to distinguish them from requests made by other nodes.

Operation of the Distributed Queue State Machine

A DQSM uses two sections of the Access Control Field (ACF) in each slot to control the QA access. These are the BUSY field and the Request bits (there is one request bit associated with each priority). When the BUSY bit is set, it indicates to the DQSM that the slot is filled with user data, and hence not available for access. The request field is used to send a request that the station has a packet queued for transmission at a particular priority.

Each DQSM keeps a current record which indicates the number of segments awaiting access to the bus. When a DQSM has a segment for transmission, it uses this count to determine its position in the distributed queue. If no segments are waiting, access is immediate, otherwise, deference is given to those

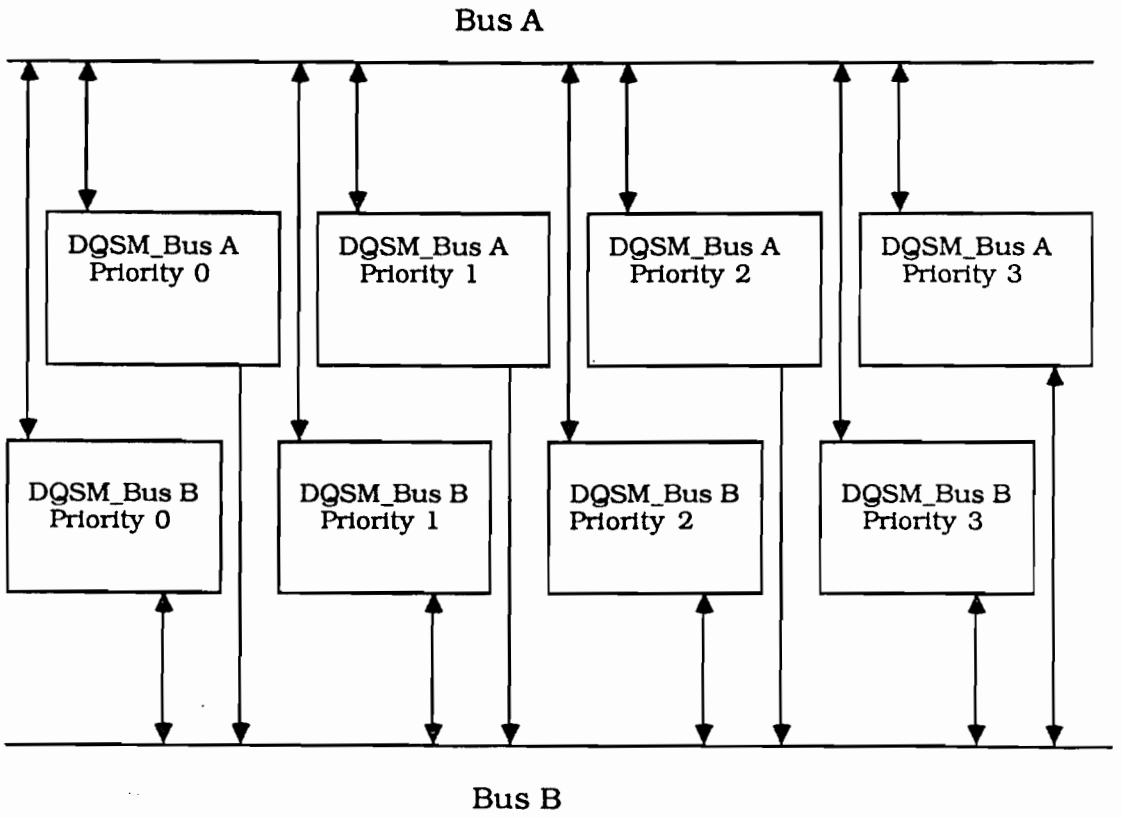


Figure 2.6: A Station's Model Showing the Distributed Queue State Machines (DQSMs) Associated with each Bus.

segments who were queued first. This is implemented by the use of two counters at each DQSM: a request counter denoted by REQ-I-CNTR-x and a countdown counter CD-I-CNTR-x. Note that "I" indicates the priority level and "x" indicates the access bus associated with the DQSM.

The generic DQSM for queued-arbitrated access to Bus x at priority I is shown in Figure 2.7. As shown in the figure, a DQSM can be in one of three states: Idle, Countdown and Standby.

The DQSM is in the Idle state when it has no QA segment to be transferred at priority level I on Bus x. While in this state the DQSM maintains a count of the results generated by nodes downstream on Bus x that are still outstanding. Note that downstream is defined with respect to the flow on the forward bus. The count also includes requests from the node itself for access to Bus x at higher priorities. This is accomplished by incrementing the REQ-I-CNTR-x each time there is a request, either on Bus y (the opposite Bus) or from another DQSM within the node, which has the same or higher priority. On the other hand, the REQ-I-CNTR-x is decremented each time an empty slot passes on Bus x. This is because the empty slot will be used by either a DQSM downstream or within the node which has a greater priority level. Upon the receipt of a QA segment to transmit, the DQSM checks the REQ-0-CNTR-x and examines the priority level 0 queue for Bus x at the node. Then, if REQ-0-CNTR-x is zero and there is no QA segment queued at the priority 0 queue, the DQSM informs other

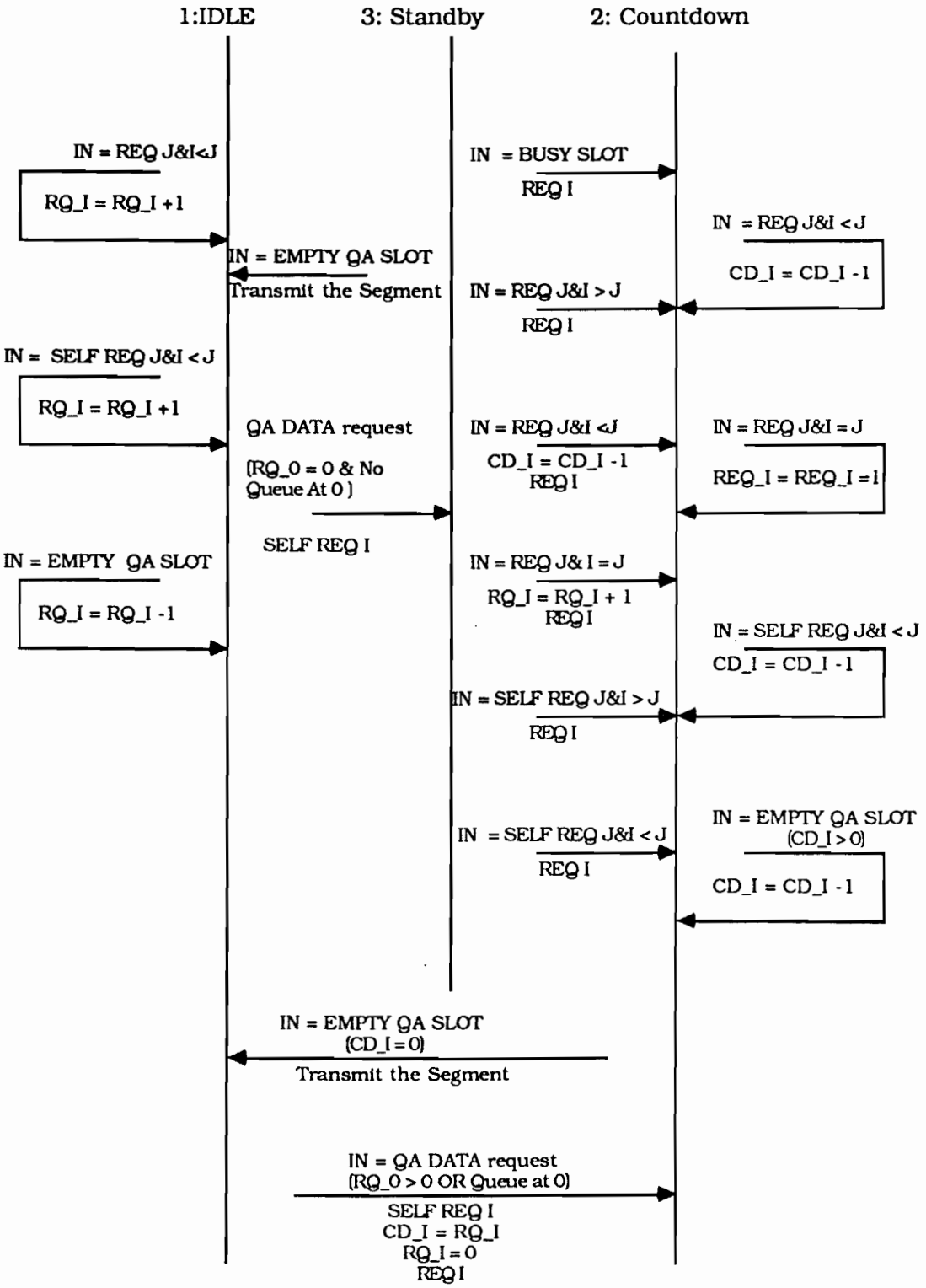


Fig 2.7: Distributed Queue State Machine (DQSM) for Priority Level I. RQ_I and CD_I represent the values of REQ_I_CNTR-x and CD_I_CNTR_x, respectively.

DQSMs of the self request and enters the standby state. Otherwise, the DQSM performs the following operations: 1) Informs other DQSMs within the node associated with Bus x of the self request. 2) Transfers the value of REQ-I-CNTR-x to CD-I-CNTR-x. This action loads the CD-I-CNTR-x with the number of QA segments queued ahead of it. 3) Indicate to the associated RQM Machine (to be explained later) that a request is to be sent at priority level I on Bus y. 4) Enters the Countdown State.

The DQSM is in Countdown State when it has a QA segment queued for transfer but is not yet permitted to access empty QA slots passing on the bus. The permission for access is controlled by the countdown counter, CD-I-CNTR-x. As in the idle state, self-requests and requests generated by the nodes downstream which have higher or equal priority increment the request counter of the DQSM, REQ-I-CNTR-x. The countdown counter, CD-I-CNTR-x, is incremented each time the DQSM receives a self-request or a request from a downstream node which has higher or equal priority. Also, each time an empty QA slot passes on the bus, if the value of CD-I-CNTR-x is not zero, it is decremented. As explained before, this slot will be used by a higher or equal priority DQSM downstream or by a higher priority DQSM within the node. On the other hand, if the value of the countdown counter is equal to zero, the QA segment queued at the DQSM is written into the empty slot and the DQSM returns to the idle state.

The Standby State is entered when a new packet moves into the first position of the DQSM and the request counter of the lowest level priority, REQ-0-CNTR-x, equals zero. This implies there are no DQSMs with higher or equal priorities downstream or within the node that have packets queued, and so the network utilization is likely to be low. Under this condition empty QA slots would probably be available on the bus which could be used without sending a request. Therefore, the DQSM attempts access in the next slot. If the next slot is an NA slot (defined in the next subsection), the DQSM stays in the Countdown State. If the next slot is an empty QA slot, the QA segment is written to it and the DQSM returns to the idle state. However, if the next slot is a busy QA slot, the DQSM informs the Request Queue Machine (the operation of which will be explained later) that a request needs to be sent on the opposite bus and moves to the Countdown State.

While in the standby state, if the DQSM receives a self-request or a request from a downstream note, it sends a request through the associated RQM and moves to the countdown state. If the request has higher priority, the associated countdown counter, CD-I-CNTR-x, is incremented. However, if the priority level of the request is equal to that of the DQSM, the associated request counter, REQ-I-CNTR-x is incremented.

It is important to notice that when there is more than one QA segment queued at a particular priority level, the DQSM can send requests for them one at a time and it is only after transmission of a

QA segment that a request can be sent for the next one. This strategy is accomplished by returning to idle state after each transmission and is intended to prevent a node with a long message from reserving the bus for a long time and delaying all other nodes. Also, by using this strategy, multi-segment messages from several nodes are serviced one-segment-per node in a round-robin discipline.

Request Queue Machine (RQM)

Each instance of a DQSM at priority level "I" and bus "x" is associated with a Request Queue Machine which is responsible for queueing and sending requests to the upstream nodes at priority level "I" and bus "y" (the opposite bus). Each RQM is equipped with a request counter, REQ-I-y, which is incremented each time the associated DQSM needs to send a request. The REQ-I-y is decremented each time the RQM manages to send the request on a QA or NA slot passing on bus y. The request can be sent only if the request bit associated with priority I in the Access Control Field (ACF) of the slot is zero. In this case, the request is sent by setting this bit to 1.

2.2.1.2 An Example

An example is provided in Figure 2.8 to help clarify the operation of the distributed queue. The example is concerned with the operation of one bus, say bus A. That is, all transmissions are

intended for Bus A. The operation of bus B is simply a mirror image of the operation of bus A. Also, it is assumed that all slots passing on bus A are busy. Therefore, almost immediately after a DQSM in a station receives its first packet to transmit, it goes from the standby state to countdown state. An ordered pair of the form <req,Cd,Pr,St> is used to show the basic state variables of a DQSM. "req," "Cd," "Pr," and "St" are the values of request counter, countdown counter, priority level, and state of the DQSM, respectively. Also note that variable St can be either equal to I (for idle state) or C (for countdown state).

Figure 2.8a shows the initial condition of the network where all the DQSMs are in the idle state. In Figure 2.8b, station 5 receives a packet at priority level 2. A self request is generated by the DQSM at priority level 2 and all the DQSMs that are within the node and have smaller priorities increment their request counters. Then the station sends a request at priority level 2 on bus B. As this request passes by the upstream stations, the DQSMs within these stations that have lower priorities increment their request counters.

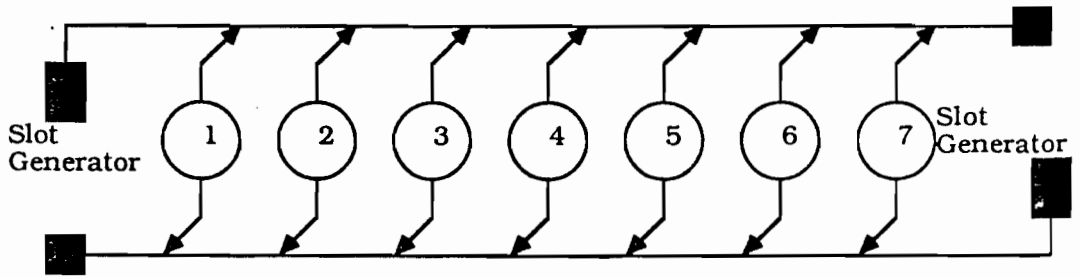
In Figure 2.8c, Station 3 receives a packet at priority 0. Because the value of the request counter is not zero, the DQSM at priority 0 loads its countdown counter with the value of the request counter and sends a request on bus B. Again the DQSMs of upstream nodes which are at priority zero increment their request counters.

In Figure 2.8d, Station 5 receives a packet at priority 3. A self request is generated and all DQSMs within the node that are in idle state (priorities 0 and 1) increment their request counters. However, the DQSM at priority 2 which is in countdown state increments its countdown counter to ensure that the packet queued at priority 3 will get access first. As the request generated by the station passes by the upstream nodes, the DQSMs that are in idle state increment their request counters and the ones in countdown state increment their countdown counter.

In Figure 2.8e, Station 4 receives a packet at priority 0. The figure also shows the counter values after the request bit generated by the station has passed by the slot generator.

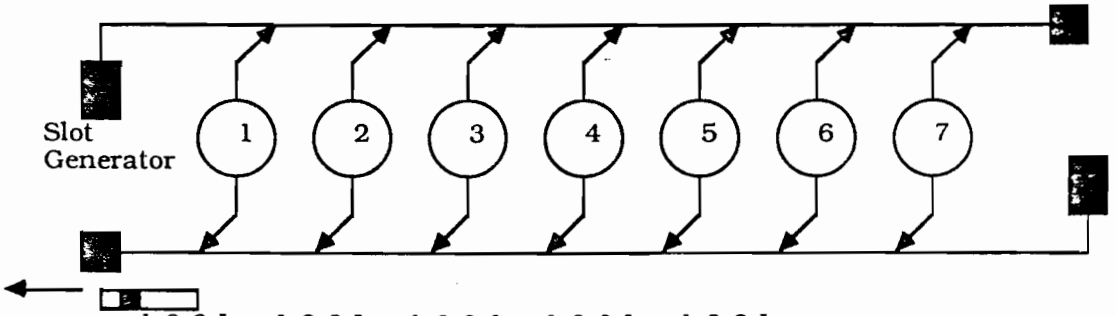
Now, assume that no further requests are received on bus B and that the empty slots pass on bus A. As the first empty slots pass by the stations on bus A, the DQSMs which are in the idle states decrement their request counters and the ones in countdown state decrement their countdown counter. Finally, Station 5 gains access at priority 3 (Figure 2.8f). The next three empty QA slots are taken in order by Station 5 at priority 2, Station 3 at priority 0, and Station 4 at priority 0.

As it can be observed in this simple example, higher priority packets gain access before all other QA segments queued at lower priorities. QA segments that are queued at the same priority level get access in the queueing order. Other characteristics of this



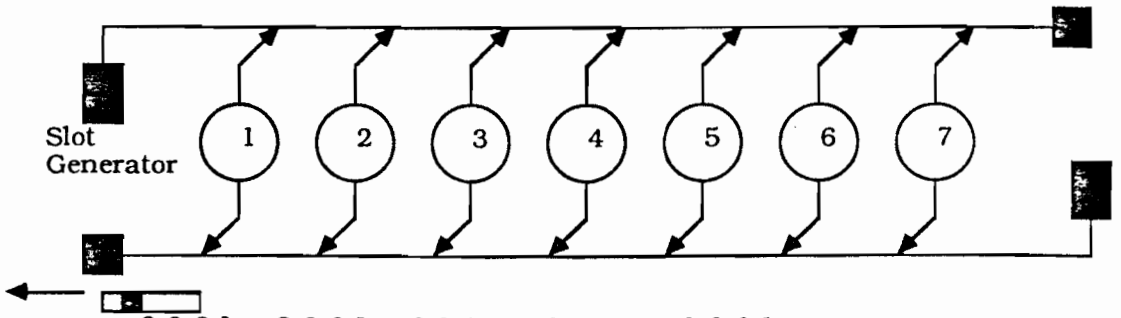
<0,0,0,I> <0,0,0,I> <0,2,0,I> <0,0,0,I> <0,0,0,I>
<0,0,1,I> <0,0,1,I> <0,0,1,I> <0,0,1,I> <0,0,1,I>
<0,0,2,I> <0,0,2,I> <0,0,2,I> <0,0,2,I> <0,0,2,I>
<0,0,3,I> <0,0,3,I> <0,0,3,I> <0,0,3,I> <0,0,3,I>

Figure 2.8a: The Network is in the Initial State



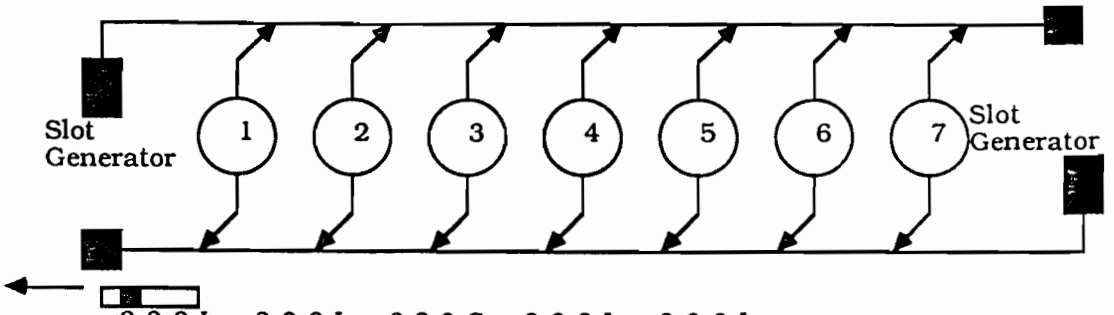
<1,0,0,I> <1,0,0,I> <1,0,0,0> <1,0,0,I> <1,0,0,I>
<1,0,1,I> <1,0,1,I> <1,0,1,I> <1,0,1,I> <1,0,1,I>
<1,0,2,I> <1,0,2,I> <1,0,2,I> <1,0,2,I> <1,0,0,I>
<1,0,3,I> <1,0,3,I> <1,0,3,I> <1,0,3,I> <0,0,3,C>

Figure 2.8b: Node # 5 Recives a Packet at Priority 2



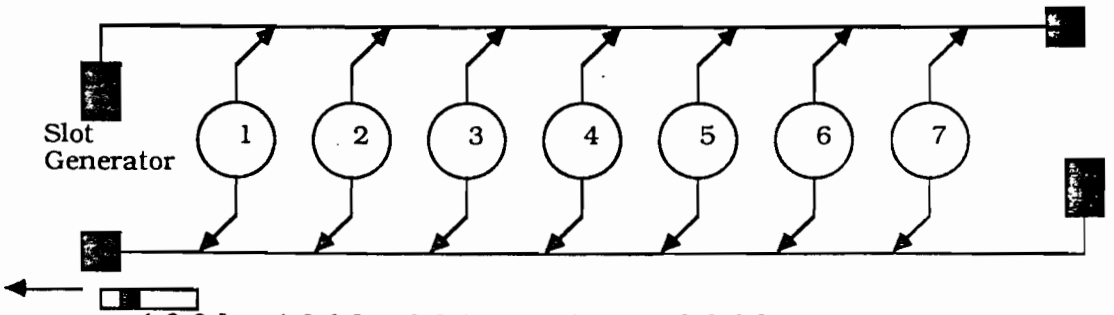
<2,0,0,I> <2,0,0,I> <0,1,0,C> <1,0,0,I> <1,0,0,I>
<1,0,1,I> <1,0,1,I> <1,0,1,I> <1,0,1,I> <1,0,1,I>
<1,0,2,I> <1,0,2,I> <1,0,2,I> <1,0,2,I> <1,0,2,C>
<1,0,3,I> <1,0,3,I> <1,0,3,I> <1,0,3,I> <0,0,3,I>

Figure 2.8c: Node # 3 Recives a Packet at Priority 0



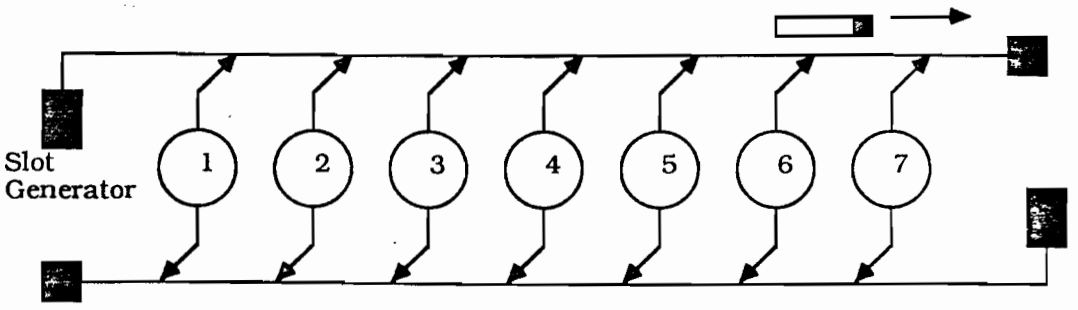
<3,0,0,I> <3,0,0,I> <0,2,0,C> <2,0,0,I> <2,0,0,I>
<2,0,1,I> <2,0,1,I> <2,0,1,I> <2,0,1,I> <2,0,1,I>
<2,0,2,I> <2,0,2,I> <2,0,2,I> <2,0,2,I> <0,1,2,C>
<1,0,3,I> <1,0,3,I> <1,0,3,I> <1,0,3,I> <0,0,3,C>

Figure 2.8d: Node # 5 Recives a Packet at Priority 3



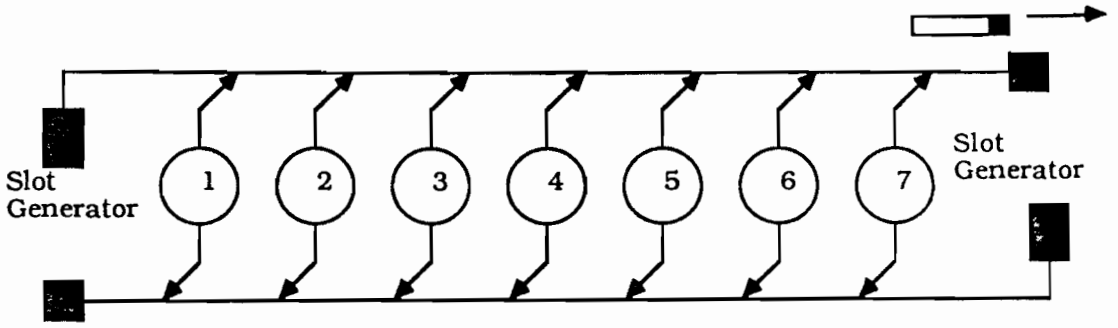
<4,0,0,I> <4,0,0,I> <1,2,0,C> <0,2,0,C> <2,0,0,I>
<2,0,1,I> <2,0,1,I> <2,0,1,I> <2,0,1,I> <2,0,1,I>
<2,0,2,I> <2,0,2,I> <2,0,2,I> <2,0,2,I> <0,1,2,C>
<1,0,3,I> <1,0,3,I> <1,0,3,I> <1,0,3,I> <0,0,3,C>

Figure 2.8e: Node # 4 Recived a Packet at Priority 0



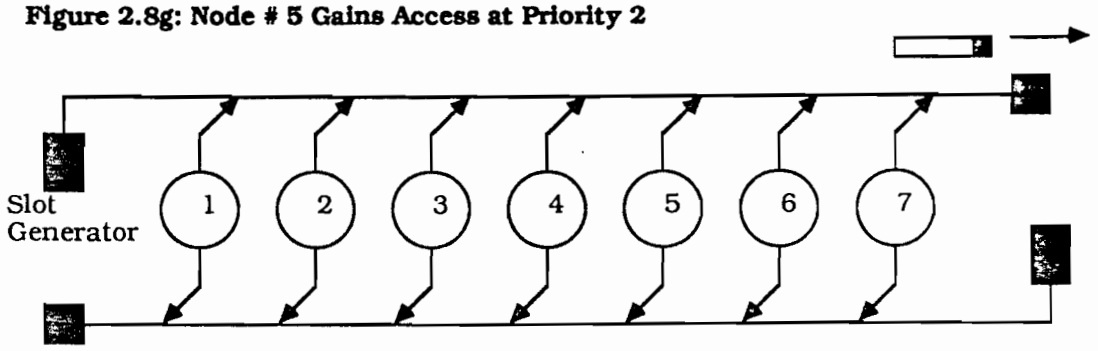
<3,0,0,I> <3,0,0,I> <1,1,0,C> <0,1,0,C> <1,0,0,I>
<1,0,1,I> <1,0,1,I> <1,0,1,I> <1,0,1,I> <1,0,1,I>
<1,0,2,I> <1,0,2,I> <1,0,2,I> <1,0,2,I> <0,0,2,C>
<0,0,3,I> <0,0,3,I> <0,0,3,I> <0,0,3,I> <0,0,3,I>

Figure 2.8f: Node # 5 Gains Access



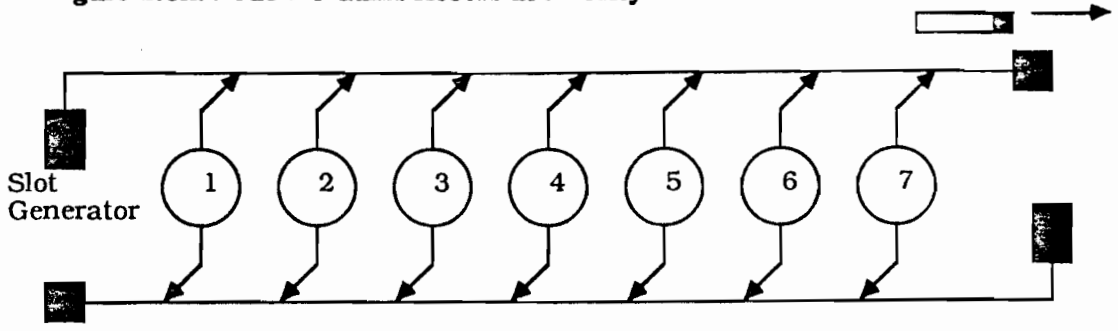
<2,0,0,I> <2,0,0,I> <1,0,0,C> <0,0,0,C> <0,0,0,I>
<0,0,1,I> <0,0,1,I> <0,0,1,I> <0,0,1,I> <0,0,1,I>
<0,0,2,I> <0,0,2,I> <0,0,2,I> <0,0,2,I> <0,0,2,I>
<0,0,3,I> <0,0,3,I> <0,0,3,I> <0,0,3,I> <0,0,3,I>

Figure 2.8g: Node # 5 Gains Access at Priority 2



<1,0,0,I> <1,0,0,I> <1,0,0,I> <0,0,0,C> <0,0,0,I>
<0,0,1,I> <0,0,1,I> <0,0,1,I> <0,0,1,I> <0,0,1,I>
<0,0,2,I> <0,0,2,I> <0,0,2,I> <0,0,2,I> <0,0,2,I>
<0,0,3,I> <0,0,3,I> <0,0,3,I> <0,0,3,I> <0,0,3,I>

Figure 2.8h: Node # 3 Gains Access at Priority 0



<0,0,0,I> <0,0,0,I> <0,0,0,I> <0,0,0,I> <0,0,0,I>
<0,0,1,I> <0,0,1,I> <0,0,1,I> <0,0,1,I> <0,0,1,I>
<0,0,2,I> <0,0,2,I> <0,0,2,I> <0,0,2,I> <0,0,2,I>
<0,0,3,I> <0,0,3,I> <0,0,3,I> <0,0,3,I> <0,0,3,I>

Figure 2.8i: Node # 4 Gains Access at Priority 0

Figure 2.8: An Example Showing The DQDB Access Scheme

access method will be studied in later chapters.

2.2.2 Non-Arbitrated Access Scheme

The Non-Arbitrated access scheme is designed for the transfer of isochronous service octets. The unit of transmission which is called an "NA Slot" has the basic format of a QA slot; however, access to NA slots is very different than access to QA slots. Although NA slots have a fixed length (like QA slots), they can be shared between a number of users.

The slot generator at the head of the bus takes responsibility for sending a sufficient number of NA slots to ensure that all isochronous service users have adequate bandwidth available. When the head of the bus generates an NA slot, it places a Virtual Channel Identifier (VCI) into the slot header. For each VCI value that the node must access, the node will maintain a table indicating which octet position(s) within the slot that it should use for reading and writing. Thus, the node will read from a particular location within the slot and write to another position within the slot. It ignores all other octets. This read and write operation must be done while the NA slot is passing the node. If the NA slot contains a VCI value which is not used by the node, the entire slot is ignored.

Chapter 3

The Simulation Model of DQDB MAN

In order to study the performance characteristics of the DQDB Access method a simulation model of the protocol was developed. In this chapter, the simulation model as well as its capabilities and limitations are described.

3.1 Simulation Model Overview

The main purpose of the simulation model is to produce statistics that can be used in studying the steady-state performance characteristics of the DQDB Access Scheme. Therefore, there is only one module in the protocol architecture of the MAC layer shown in Figure 2.3 that needs to be included in the simulation model: the Access Control Component. The simulation model is concerned with only the Medium Access Control (MAC) layer operation and assumes that the physical layer can provide an error-free bit pipe. As far as the network topology is concerned, the DQDB model assumes the open dual bus topology shown in Figure 2.2. A network model is created by specifying overall simulation controls, media access control parameters, and the parameters of each station. These parameters are listed in Table 3.1. After complete specification of a model, a simulation is performed and certain performance characteristics of the model such as throughput and delay are studied both "globally" and on a per-node

basis. A more detailed explanation of these parameters are included in section 3.2.5.

3.2 Modeling Perspectives of the DQDB MAN

The implementation method of the DQDB access scheme is described for both the non-arbitrated and queued-arbitrated access schemes.

3.2.1 Modeling the Non-Arbitrated Access Component (NA-AC)

As explained in Chapter 2, the isochronous service users gain access using the NA slots that are generated by the slot generator at the rate required by the service user. Therefore, each isochronous user is guaranteed to get the bandwidth it requires and there is no need to measure the performance characteristics of the non-arbitrated access method. However, the number of isochronous service users and their bandwidth requirements can affect the performance of queued-arbitrated service users. It is this aspect of the non-arbitrated access service that the DQDB model includes. Also, it is assumed that all isochronous service connections remain active throughout the simulation time. This assumption is very well justified considering that the duration of an isochronous service such as a phone call or a video conference is in the order of a couple of minutes whereas the steady state performance characteristics of queued-arbitrated access scheme can usually be obtained in a few seconds.

The NA-AC model requires that the number of isochronous service users, denoted by 'm', as well as their required bandwidth, denoted by IR, be specified. Based on this information, the slot interarrival time for the slowest isochronous user is calculated. This time which is referred to as Frame Interval (FI) is equal to:

$$FI = \frac{PZ}{\min(IR_i)}$$

PZ is the payload size of a slot (Figure 2.5) which is 512 bits [3] and $\min(IR_i)$ is the smallest isochronous rate required.

A frame of slots with duration of FI is formed. The number of slots in a frame is equal to:

$$N = \frac{FI}{\text{Slot-Time}}$$

where

$$\text{Slot-Time} = \frac{\text{Slot-Size}}{LR_i}$$

Slot-Size is the size of a Slot (Figure 2.5) which is 552 bits and LR_i is the line rate of one bus which is one half of the total line rate as explained in Chapter 2.

The number of NA slots in a frame, N-NA, can be calculated by dividing the sum of all the isochronous rates by the required rate of the slowest user. That is,

$$N\text{–}NA = \frac{\sum_{i=1}^{m} IR_i}{\min(IR_i)}$$

A random number generator with uniform probability density function between 1 and N is used to assign N-NA slot numbers to the NA slots in a frame. The rest of the slots in the frame are marked as QA slots. The frame generated in this manner is used by the Slot-Generation Event (to be explained later) to generate QA and NA slots. Figure 3.1 shows three consecutive frames and a typical slot assignment based on the criteria explained in this section. Note that the slot assignment is the same for all the frames on the bus.

3.2.2 Discrete Event Modeling

The approach taken for the simulation of the DQDB model was that of discrete event scheduling [7,8]. This simulation approach involves defining certain events to model the system behavior. For the proper operation of the system, each event needs to occur at the end of regular time intervals throughout the simulation. An event scheduler is implemented to create and maintain a list of events and their required execution times in the increasing order of the event execution time. This list which is referred to as the event calendar reflects the order in which the

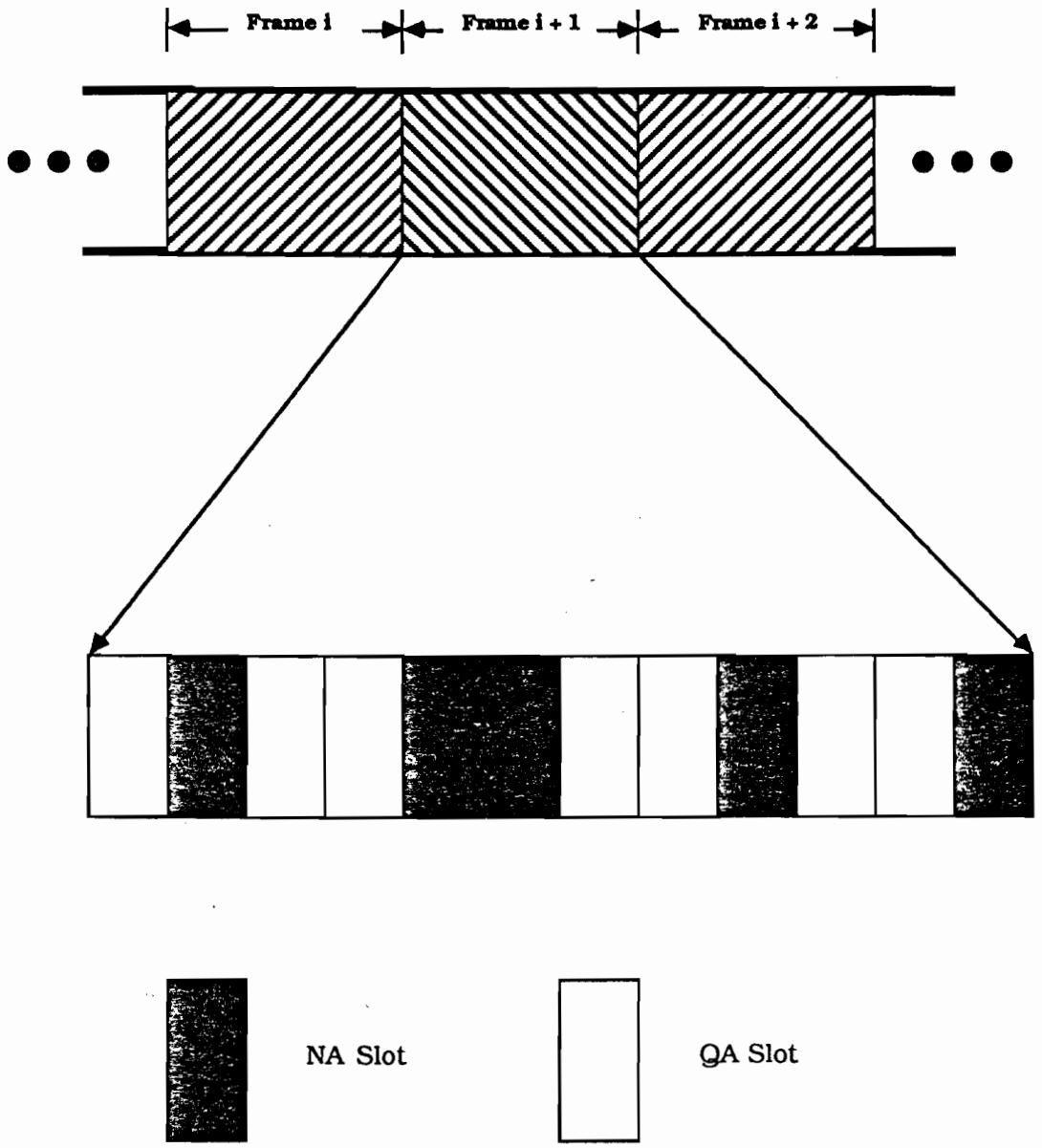


Fig. 3.1: Frames and their internal structure

events need to be executed. After the execution of each event, the executed event is removed from the event calender and the time (tnow) is advanced to the execution time of the event being executed. Then, based on the timing requirements of the event, a new execution time is entered into the event calendar for that event. This process is repeated for as long as the time (tnow) is less than the total simulation time specified by the user. Note that in each of the discrete events modeled, the time (tnow) does not advance when an event is being executed, it changes only between events. Three events were identified for the model of the DQDB. A brief description of these events is included in the next section.

3.2.3 Description of the Events Used in the DQDB Model

For the model for the DQDB MAN, three events were identified. These events included the frame-generation event, the arrival event, and the slot-arrival event. Each event is associated with a number of attributes. These attributes include the event code, the event execution time, the bus (either bus A or bus B) and the node at which the event is to be executed. The frame-generation event is always scheduled for the first station on each bus; however, the arrival event and the slot-arrival event pertain to the node under operation.

The frame-generation event models the operation of the head of the bus slot generator. Each time this event is executed, it schedules the arrival of a slot (Slot-Arrival event) at the first station

on the bus. The event code of each slot-arrival event includes the type of the slot (i.e., QA or NA Slot), the values of the Busy bit and the Request bits for each priority (Figure 2.5). The type of the slot is determined by the frame format described in 3.2.1. The values of the Busy and the Request bits are set to zero to indicate an empty slot.

The arrival of a slot at a station is modeled by the slot-arrival event. During the execution of this event, the values of the Busy and the Request bits as well as the slot type are extracted from the event code. This information is forwarded to the Distributed Queue State Machines (DQSMs) operating at that node. The DQSMs use this information to function in the way described in Figure 2.7. After the completion of each slot-arrival event at a particular node, a slot-arrival event which includes the updated values of Busy bit and Request bits is scheduled for the next station on the bus.

Finally, the arrival of a packet at a particular DQSM is modeled by the arrival event. The operation of the DQSMs after receiving the packet is done in the way shown in Figure 2.7. The time between scheduling two successive arrival events is a random variable with a probability density function specified by the user (Mean-interarrival time).

3.2.4 Other Modeling Aspects

Each DQSM is limited to a certain buffer size which indicates the number of QA slots that can be queued for access to a certain bus. Upon the arrival of a packet, the model determines the number of QA slots required to transmit the packet. If the space left in the buffer is not enough to hold the packet, the whole packet is discarded.

The distribution of the packet length and the arrival time can be exponential or constant. However, the arrival event can be easily modified to include other distributions.

The load can be distributed between the 4 priority access classes either uniformly or as the user wishes. The user can also specify the fraction of the load assigned to a certain priority that goes on a certain bus. This option is added to help model the traffic generated by LANs and other data multiplexers that are located on the DQDB MAN and generate packets with different destination addresses. The fraction of the load that goes on each bus can be figured from the distribution of the packet destination addresses.

A list of parameters required by the model as well as the type and definition of each parameter is included in Table 3.1

Table 3.1

Simulation Control Parameters

Simulation Time	: Total simulation time in seconds. Type: Real
% Transient Response Time	: Fraction of the simulation time that a DQSM requires to reach steady-state operation. Type: 0 < Real < 1

Media Access Control Parameters

Velocity of Media	: Propagation velocity of the media in meters per second. Type: Real
Line Rate	: Total line rate for both buses in Kbps. Type: Real
Number of Isoch. Users	: Total number of isochronous users. Type: Integer
Slowest Isoch. Rate	: The smallest rate required by isochronous users (Kbps). Type: Real
Total Isoch. Rate Excluding the Slowest Rate	: Sum of all but the slowest isochronous rate (Kbps) Type: Real

Node Specification

Message Length Distribution	: Type of the probability density function used for modeling the message lengths.
Average Message Length	: Average message length in bits before overhead. Type: Real
Interarrival Time Distribution	: Type of probability density function used for modeling the interarrival time of the packets.

Mean Interarrival Time on Each Bus	: Mean interarrival time of packets for each bus in seconds. Type: Real
Message Priority	: Fraction of load assigned to each priority. Type: 0 < Real < 1 for each priority
Distance to Next Node	: Distance in kilometers between the node and the next node on the bus. Type: Real

3.2.5 Performance Measures

The simulation model starts collecting the performance measure statistics after the system transient responses have disappeared and the DQSMs have started operating in steady state. The transient response time is specified by the user as a fraction of the total simulation time.

Statistics are calculated as 'Total' or 'Average' measures during the simulation. 'Total' measures report the sum of measurements during the simulation. 'Average' measures are averaged over the number of measurements taken. For example, 'Total Packets serviced by MAC' reports the total number of packets which were successfully transmitted. The MAC performance measure 'Average MAC Queueing Delay' is averaged over the number of packets transmitted during the simulation.

Statistics are also divided into 'global' and 'node' measurements. 'Global' measures are taken over all system nodes; these results are averaged for the final output as described above. 'Node' measures are taken for a specific node during the simulation. Per-node statistics are important for non-homogeneous networks.

MAC performance measures both on global and per-priority class are included in Table 3.2.

Table 3.2

MAC Performance Measures for the DQDB MAN

(All Performance statistics are defined both on a global and per-priority class basis unless specified.)

Average Total MAC Delay	: Average access delay in the MAC layer. Includes MAC queueing delay and the writing time to the bus.
Average MAC Queueing Delay	: Amount of time each packet spends in the queue.
Total Media Throughput	: Fraction of time sending information bits that got through the MAC layer.
Total Packets Offered to MAC	: Total number of packets offered to the MAC layer from the Queued-Arbitrated Transfer Users (QATUs).
Total Packets Discarded by MAC	: Total number of packets thrown away as the buffers were full at times when packets were offered to MAC.
Total Packets Services by MAC	: Total number of packets which were successfully transmitted by MAC.
Total Info Bits Offered to MAC (Per Priority Class Only)	: Total number of bits in packets offered to MAC from the QATUs.
Total Info Bits Serviced (Per Priority Class Only)	: Total number of information bits (i.e., bits offered to MAC from the QATUs) serviced by MAC.

Total Average Offered Load (Global Only)	: Total average load including overhead offered to the MAC layer of a priority class.
Total Network Utilization (Global Only)	: Total throughput including both Queued-Arbitrated and Non-Arbitrated throughput.

3.3 Limitations of the DQDB Model

The DQDB simulation model assumes the open dual bus topology shown in Figure 2.2. In this topology, the slot generator of a particular bus can send its packets on only that bus. However, in the looped dual bus topology, the slot generators for both buses are located on one station which means the slot generator can send packets on both buses.

The DQDB simulation model pertains to the steady state operation of the Medium Access Layer (MAC) and does not include network management functions, network initialization, and network reconfiguration caused by a "jabbering node" or a bus failure.

Chapter 4

Validation

This chapter deals with the procedures that were followed in the validation of the DQDB model. The validation for the model was done for both the single priority case and the mixed priority case. The validation procedures for the two cases mentioned above are included after a short explanation of the assumptions on the traffic model.

4.1 The Traffic Model

As mentioned in Section 3.2.4, in most applications of the DQDB MAN, most data stations on the network are LANs or data multiplexers that generate packets with different destination addresses. Therefore, the fraction of the total load of a station that gets transmitted on a particular bus is determined by the distribution of destination addresses of packets of that station. The traffic assignment model considered in this chapter assumes that the destination addresses of all stations have uniform distributions. That is, each station on the network sends the same fraction of its load to every destination.

As an example, consider a network which consists of five nodes connected to each other using the open dual bus topology shown in Figure 2.2. The packets generated by each node on the network can have 5 - 1 = 4 destination addresses. For instance, packets generated by station 2 can be sent to one of the nodes 1, 3,

4, or 5. The fact that the destination addresses have uniform distributions implies that 1/4 of the total load of the station goes to each destination. Again, considering station 2, this assumption requires that 1/4 of the total load of the station should be sent on bus B to node 1 which is located upstream from node 2 and 3/4 of the total load should be sent to nodes 3, 4, and 5 on bus B which are located downstream. Note that upstream and downstream in this section are defined with respect to bus A.

The above example can be generalized for a network consisting of n nodes. Figure 4.1 shows the load assignment for such a network. For station i, the total load, λ_i, is decomposed into $\lambda_{i,a}$ and $\lambda_{i,b}$ which are fractions of λ_i that are transmitted on bus A and bus B, respectively. Thus,

$$\lambda_i = \lambda_{i,a} + \lambda_{i,b}$$

Now, $\lambda_{i,a}$ = (fraction of the number of destination nodes that are located downstream) (λ_i)

or $$\lambda_{i,a} = \frac{\text{No. of destinations downstream}}{\text{No. of destinations}} \lambda_i = \frac{n-i}{n-1} \lambda_i$$

Similarly, $\lambda_{i,b}$ = (fraction of the number of destination nodes that are located upstream) (λ_i)

or $$\lambda_{i,b} = \frac{i-1}{n-1} \lambda_i$$

Other traffic model assumptions include the assumptions on the message length distributions and the mean-interarrival time

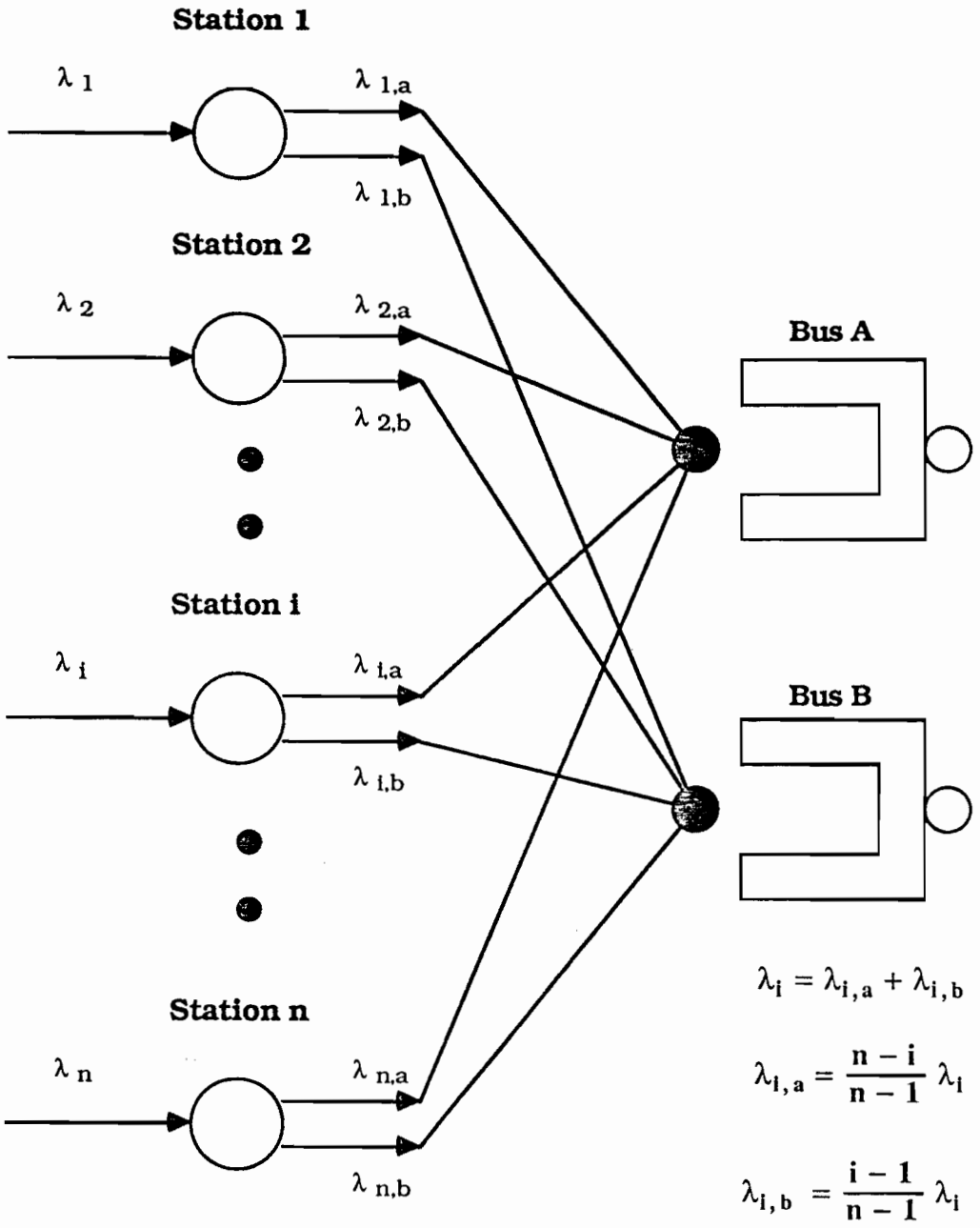


Figure 4.1: The Traffic Assignment Model

distributions. It was assumed that messages have constant lengths equal to the payload size (i.e., 512 bits) and the mean-interarrival times have exponential distributions. This assumption implies that the messages are not segmented. Also note that the simulation model developed in this study can be assigned any arbitrary traffic model. As will be explained in the next section, the traffic model explained above was chosen for comparison purposes.

4.2 Validation of the Single Priority Model

The first case considered was a DQDB network with fixed priority. A DQDB network model with 50 equally-distanced nodes was created. It was assumed that the line rate and the velocity of the media were 100 Mbps and 200 km/s, respectively. Using the traffic model explained in Section 4.1.

The performance of this model was compared to the performance characteristics obtained by Newman [9]. The results of these comparisons are given in figures 4.2-4.9. As shown in these figures, the results of the DQDB simulation model match very closely with those obtained by Newman. Note that the 95% confidence intervals for all the graphs obtained for the DQDB simulation model are within 5% of the averages shown on the graphs. Also, the parameter "a" shown on these graphs is the normalized propagation delay which is equal to end-to-end propagation time across the network divided by the slot time

(defined in Section 3.2.1). Based on the definition of "a", the length of the network can be related to "a" in the following way:

Network Length = a (Slot Time) (Velocity of Media)

The interpretation of the results shown in figures 4.2-4.9 is done in the next two subsections.

4.2.1 Global Network Characteristics

Figure 4.2 shows the average total MAC delay (defined in Table 3.2) versus load characteristic of the DQDB scheme. According to this graph, at small loads (loads smaller than 0.1), the average total MAC delay is approximately 1.5 slot times. This can be explained in the following way. At small loads, most slots on the bus are empty. Therefore, after receiving a packet to transmit, it takes an average of 0.5 slot times to get access to an empty slot and 1 slot time to write the packet into the empty slot; hence, 1.5 average total MAC delay. By increasing the load, however, the number of empty slots on the bus decreases and the stations need more often to send requests to the slot generation asking for empty slots. Therefore, the delay for accessing an empty slot on the bus increases by increasing the load.

Figure 4.3 shows the average total MAC delay versus load characteristic of the DQDB scheme for three different network sizes. The perfect scheduler in this graph is the M/D/I queueing system whose delay characteristic is given by [4]:

$$T = \frac{3-2\rho}{2(1-\rho)}$$

where T is the average total MAC delay in slot times and ρ is the load. According to Figure 4.3, the average total MAC delay for all network sizes is approximately 1.5 slot times at small loads. Note that the explanation of the 1.5 slot times delay at small loads given earlier is valid for all network sizes. However, as the load increases, longer networks have larger delays. As mentioned before, at large loads the stations need to send request bits to the slot generator to ask for empty slots. There are two sources of delay in the transferring of a request bit: propagation delay and the queueing delay of the request bit at the RQM machine. The delay caused by the latter source is a function of the load on the request channel (i.e., number of empty request bits on the bus). However, the propagation delays of the request bit to the slot generator and the empty slot to the station increases by increasing the network size. That is, in longer networks, the stations need to wait longer to receive empty slots. The propagation delay of the request bit can also result in skewing in access order of the packets. The skewing effect will be discussed in the next section.

Figure 4.3 also shows that for small network sizes (a < = 1), changing "a" does not have a significant effect on the network performance. This is because for small network sizes, the propagation delays mentioned above are small and the distributed

Fig. 4.2: Validation of Avg. Total MAC Delay Versus Load; a = 100.0

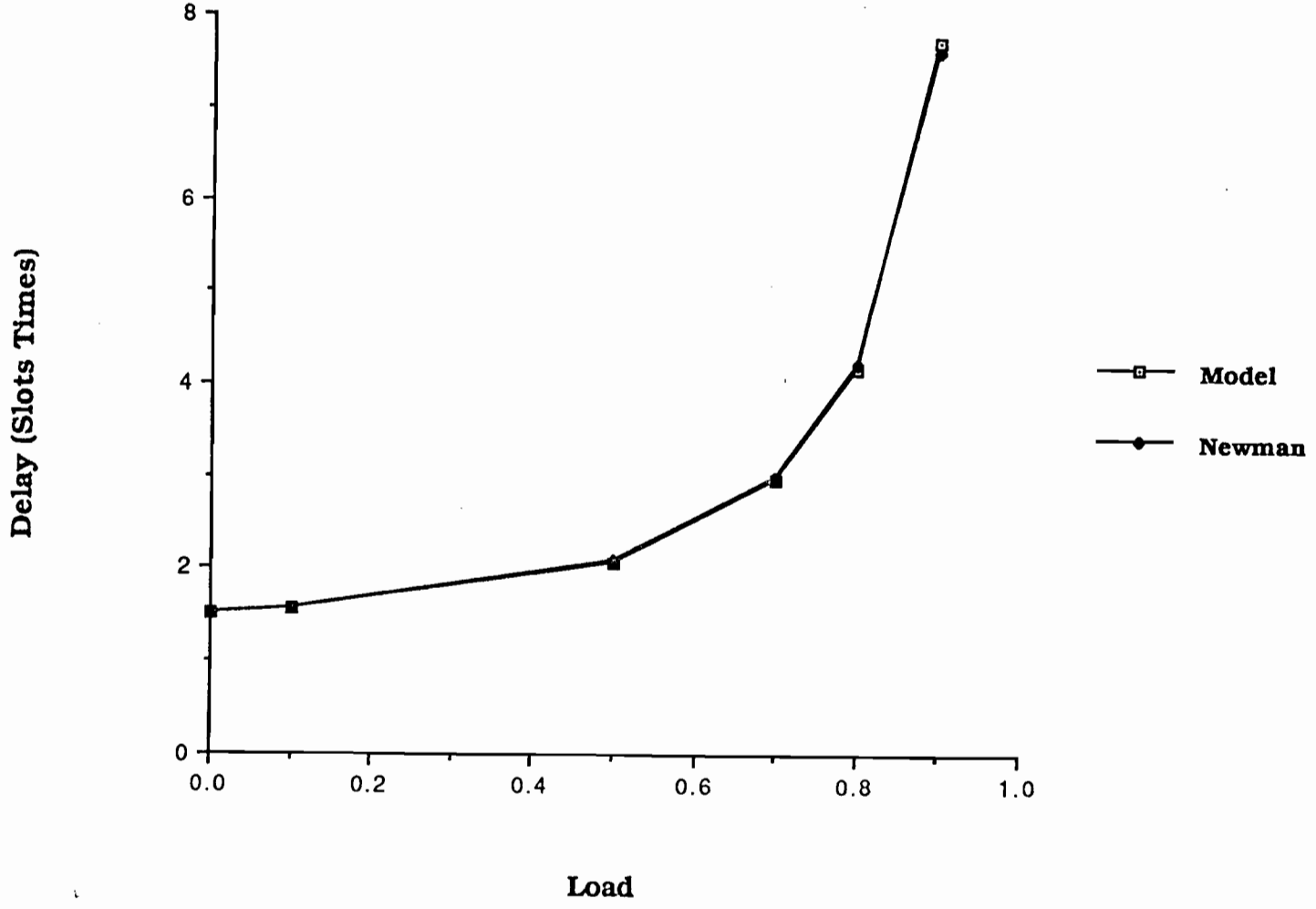
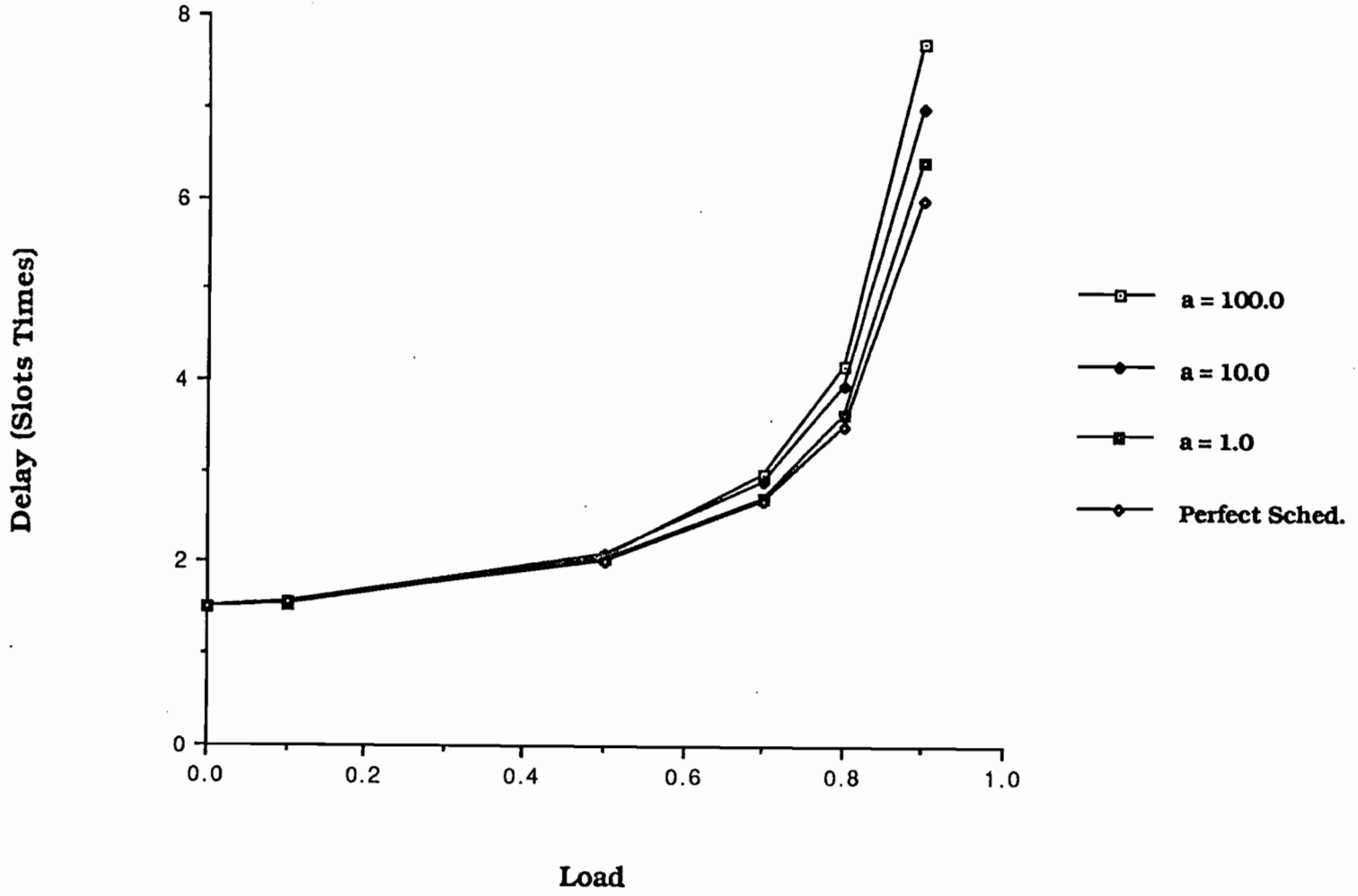


Fig. 4.3: Avg. Total MAC Delay Versus Load



queue performs almost like a perfect scheduler (i.e. the M/D/1 queueing model).

4.2.2 Fairness in the DQDB Access Scheme

Figures 4.4-4.9 show the average total queueing delay versus the position of the station on the network. The general conclusion that can be drawn from these figures is that except when operating at small loads, the DQDB scheme does not treat all the stations equally; the farther the stations from the slot generators, the larger the total queueing delay. In the following, the effects of network loading and network size on the fairness characteristics of the DQDB scheme are discussed.

Figures 4.4 and 4.5 show the effect of load on the queueing delay when the network size is held constant. As shown in Figure 4.4, for small loads, all nodes on the network can access an empty slot within 0.5 slot times. However, as the network load increases (Figure 4.5), the stations that are farther from the slot generators experience larger queueing delays. As mentioned in the previous section, this happens because at larger loads the stations need to send request bits to get empty slots. The propagation times of the request bit to the slot generator and the empty slot to the station are shorter for the stations that are closer to the slot generators.

Figures 4.6-4.9 show the effect of increasing the network size on the queueing delay characteristics when the load is held constant at 0.8. It can be concluded from figure 4.9 that for a fixed

load, increasing the network size increases the average total queueing delay. The increase in the queueing delay is even more pronounced at the stations that are farther from the slot generators. Again, this characteristic can be related to the propagation delay of the request bits. In longer networks, the propagation times of the request bit to the slot generator and the empty slot to the station are longer, especially for the stations that are farther from the slot generators.

The propagation delay of the request bit can also cause skewing in the queueing order of the distributed queue. That is, while the request bit from a station is moving toward the slot generator, the slot generator and other upstream stations that have not yet received the request from the downstream node can queue their packets in the distributed queue. This out-of-order queueing is referred to as the skewing effect in [4]. The skewing effect, which is directly proportional to the propagation delay, is larger for longer networks and for the stations that are farther from the slot generators.

The above discussion suggests that the unfairness problem of the DQDB scheme would have be if the propagation delay of the request bits could become zero (note that this requirement can be achieved only if the media has infinite velocity which is practically impossible). Hahne, Choudhry, and Maxemchuck developed a modification of the DQDB protocol that reduces the unfairness problem discussed above [19]. However, this scheme was

Fig. 4.4; Validation of Avg. Total Queueing Delay Versus Station Number
a = 100.0; Load = 0.1

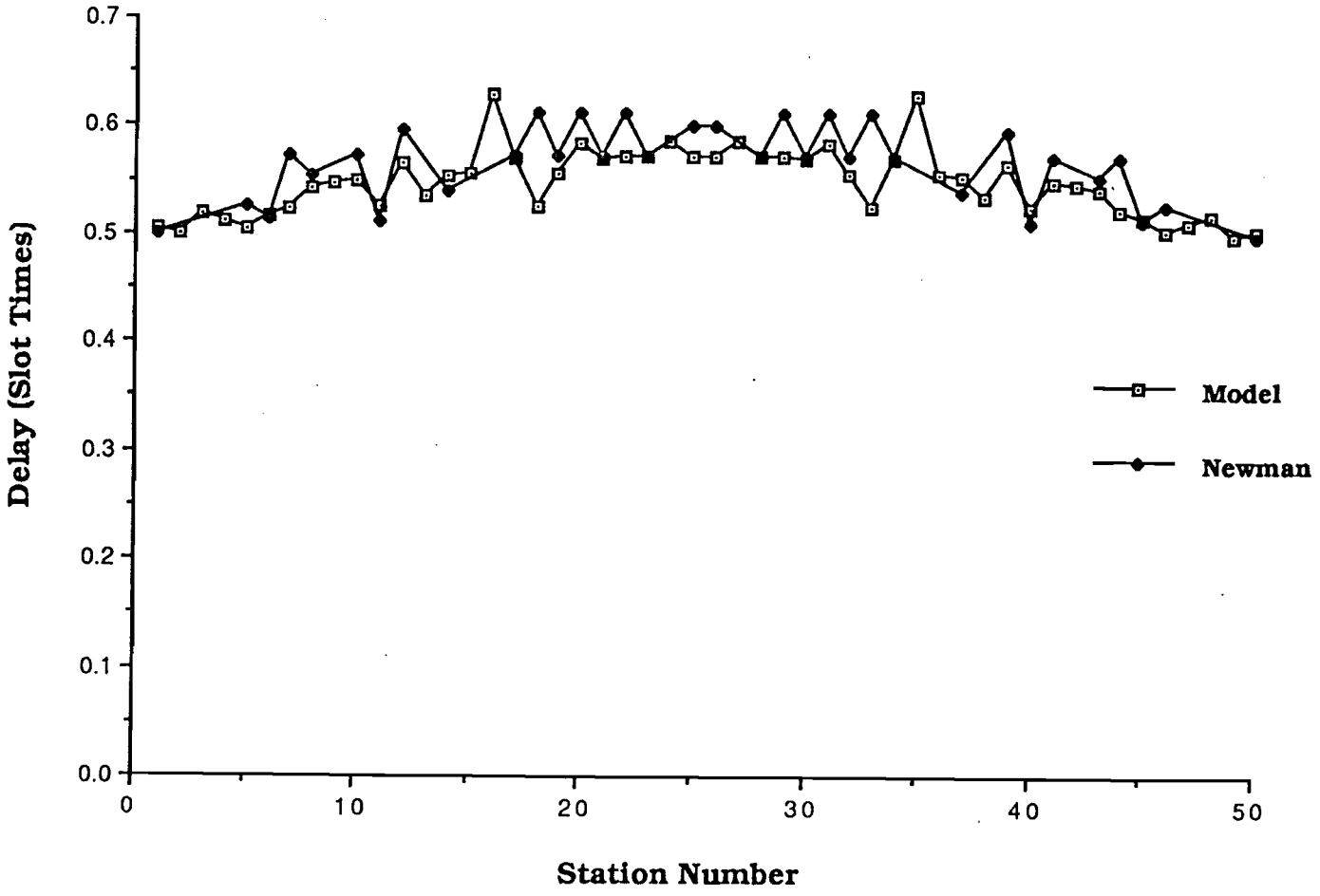


Fig. 4.5: Validation of Avg. Total Queueing Delay Versus Station Number
a = 100.0; Load = 0.5

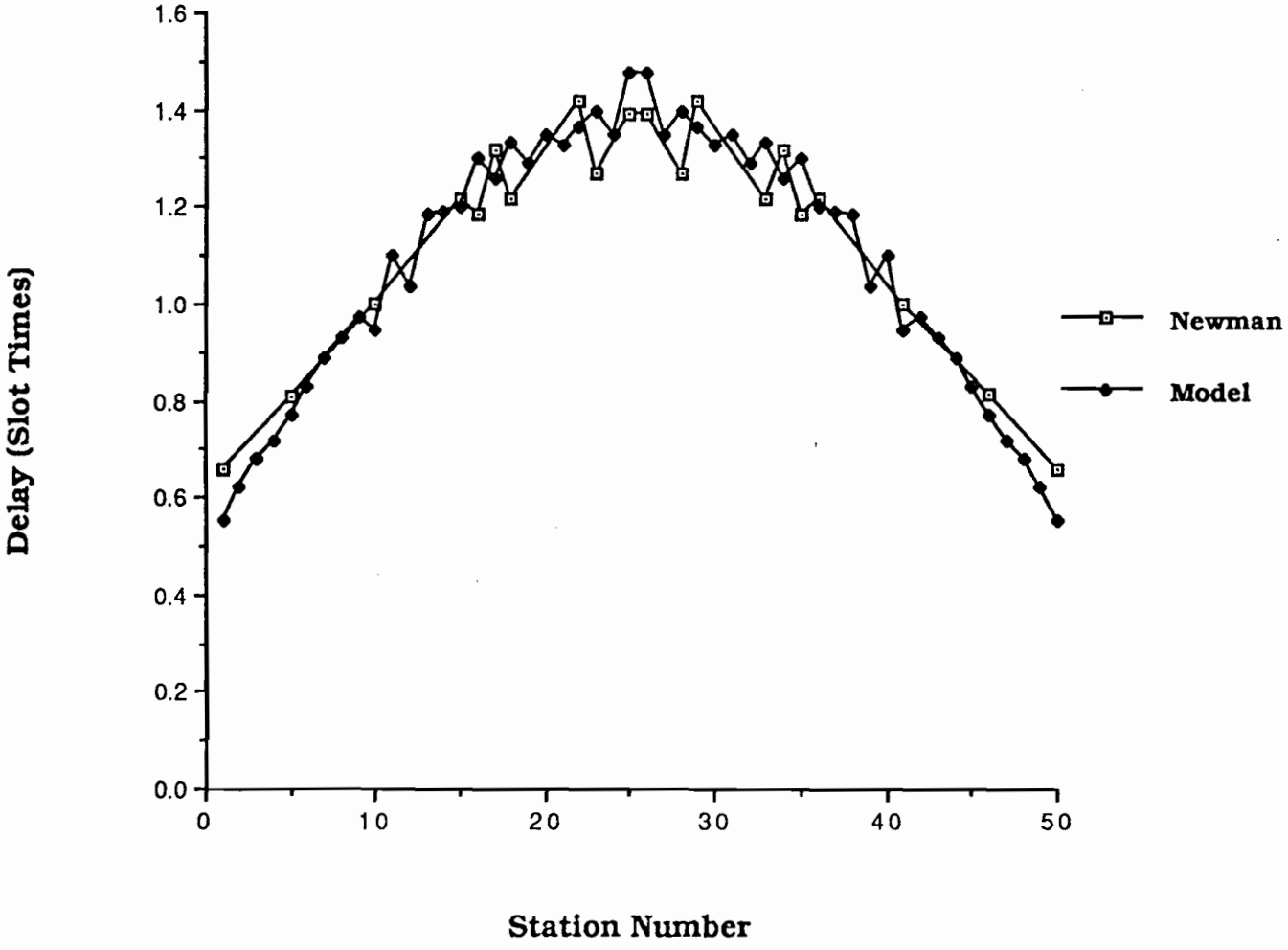


Fig. 4.6: Validation of Avg. Queueing Delay Versus Station Number
a = 1.0; Load = 0.8

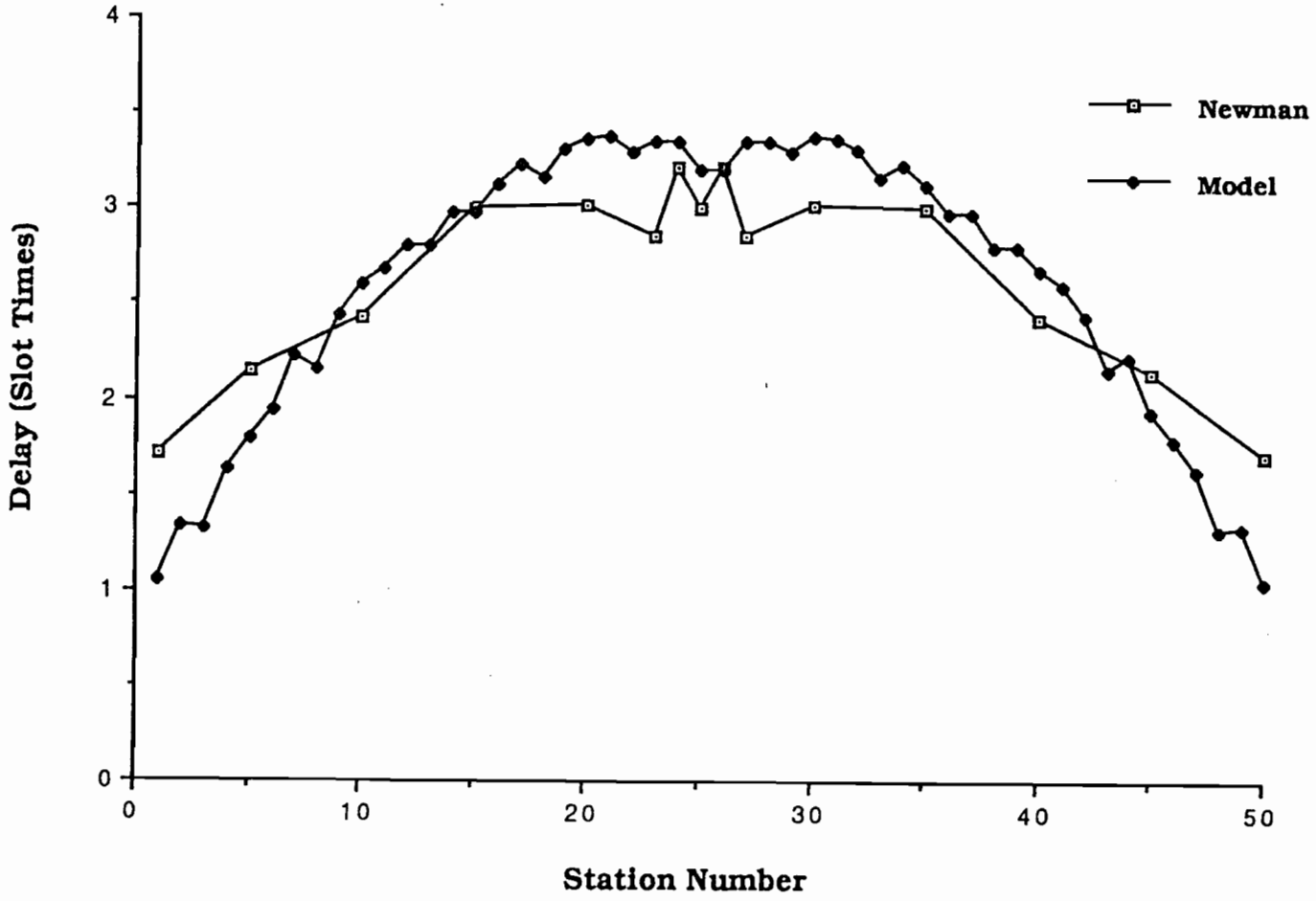


Fig. 4.7: Validation of Avg. Total Queueing Delay Versus Station Number
a = 10.0; Load = 0.8

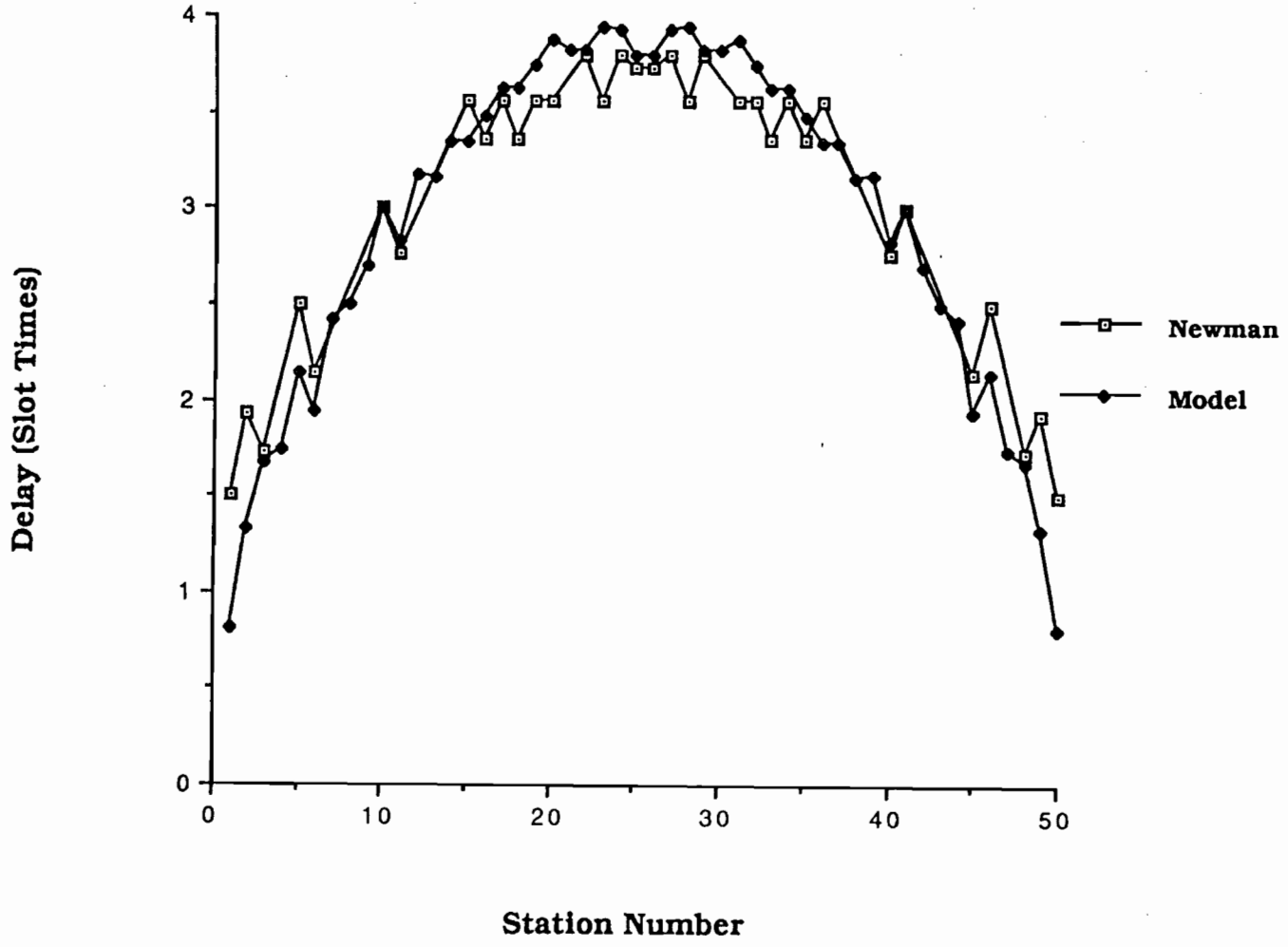


Fig. 4.8: Validation of Avg. Queueing Delay Versus Station Number
a = 100.0; Load = 0.8

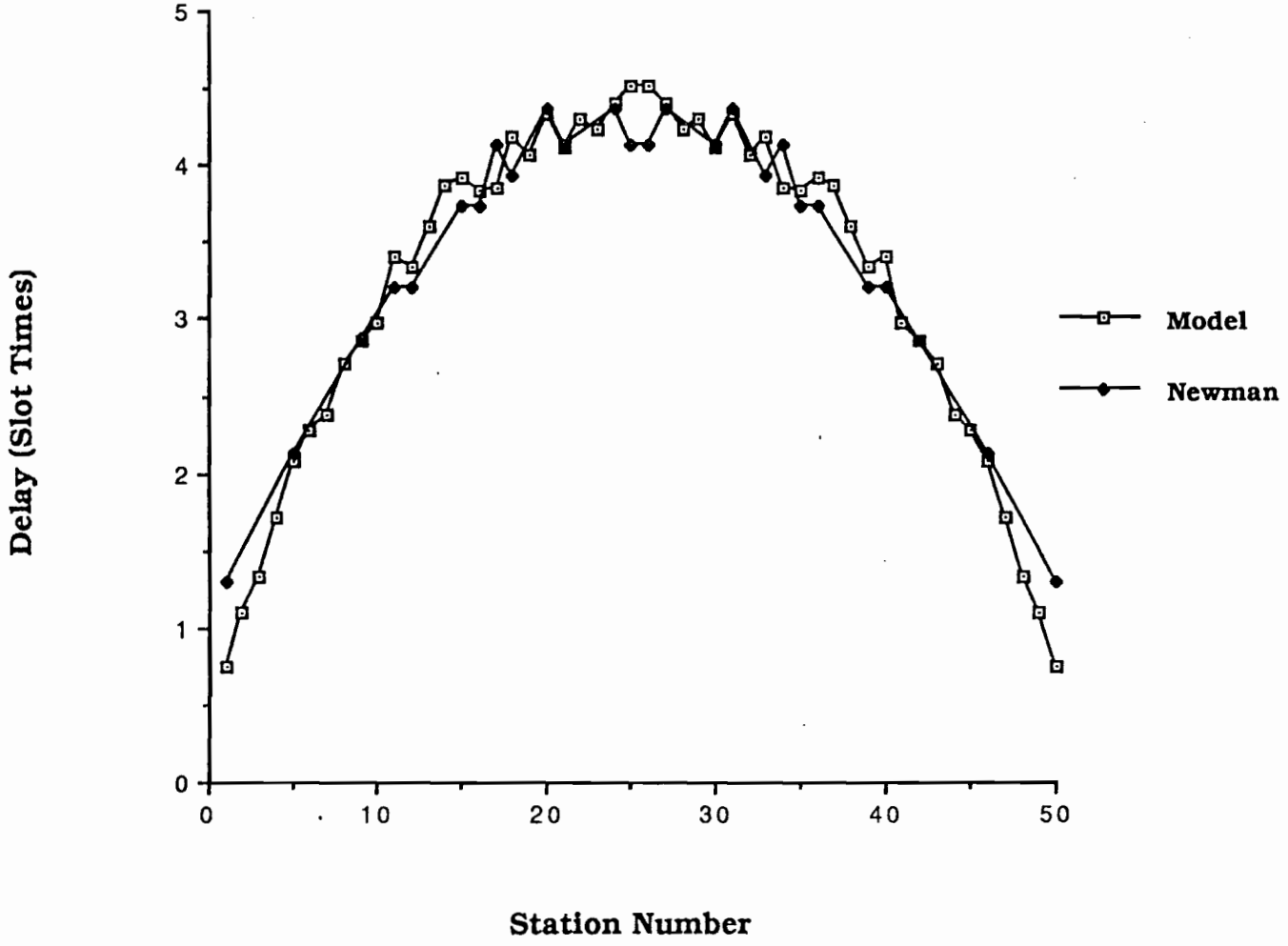
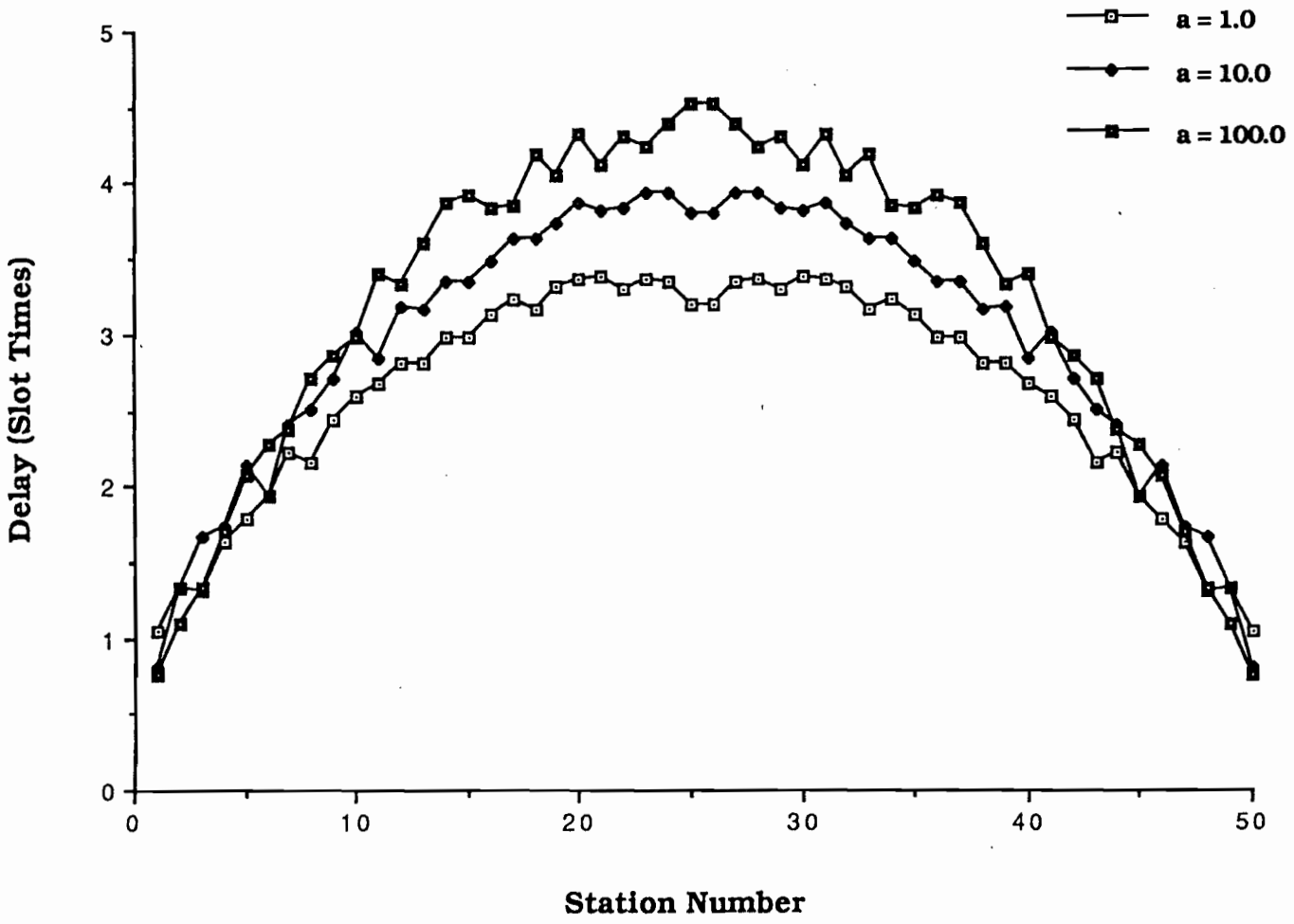


Fig. 4.9: Avg. Total Queueing Delay Versus Station Number; Load = 0.8



not implemented in the current model because the scope of this study is limited to the IEEE 802.6 DQDB MAN protocol.

4.3 Validation of the Multiple Priority Model

To obtain the performance characteristics of the DQDB model with mixed priority packets, a simulation model similar to the one described in previous section was set up. However, it was assumed that the load is distributed equally among four priorities (priority 0-3). Also, it was assumed that the buffer size at each priority level is limited to 100 messages (i.e., 100 slots). Packets that arrive when the buffers are full get discarded. Figures 4.10 and 4.11 illustrate two performance measures of this model for each priority: throughput versus load and average total MAC delay versus load.

Figure 4.10 shows the throughput versus load characteristic of the model. Throughput is the fraction of the offered load that gets transmitted properly. It is calculated, at each priority, by multiplying the offered load by the ratio of the number of packets serviced to the number of packets arrived. Also, the network utilization at a particular load is the sum of the throughputs for all priorities at that load. This figure shows that as long as the network utilization is less than one, all four packet priorities get serviced (i.e., throughput is equal to the load for all priorities). However, if the load per priority increases after the network has reached the network utilization of about one, the throughput for lower priorities decreases in such a way that the network utilization

stays constant at one. The decrease in the throughput of lower priorities is observed in the increasing order of the priorities. That is, as the load per priority increases, the first priority that starts losing throughput is priority zero. Then, the throughputs of priorities 2 and 3 are reduced. Packets of priority level 3, however, get serviced at all loads. This happens because the operation of the DQDB access scheme (Section 2.2.1.1) requires that immediate access be gained by a high priority packet, independent of the length of the low priority queue. A high priority request serves to hold off, for one packet, low priority access.

The average total MAC delay characteristics of the DQDB network, shown in Figure 4.11, is consistent with the throughput characteristics. The delay for all priorities increases by increasing the load per priority. However, as soon as a particular priority starts losing packets, the average total MAC delay rises sharply and starts approaching infinity along a vertical line. Again, this asymptotic increase in delay is observed in the increasing order of priorities.

Fig. 4.10: Throughput Versus Load
a = 100.0

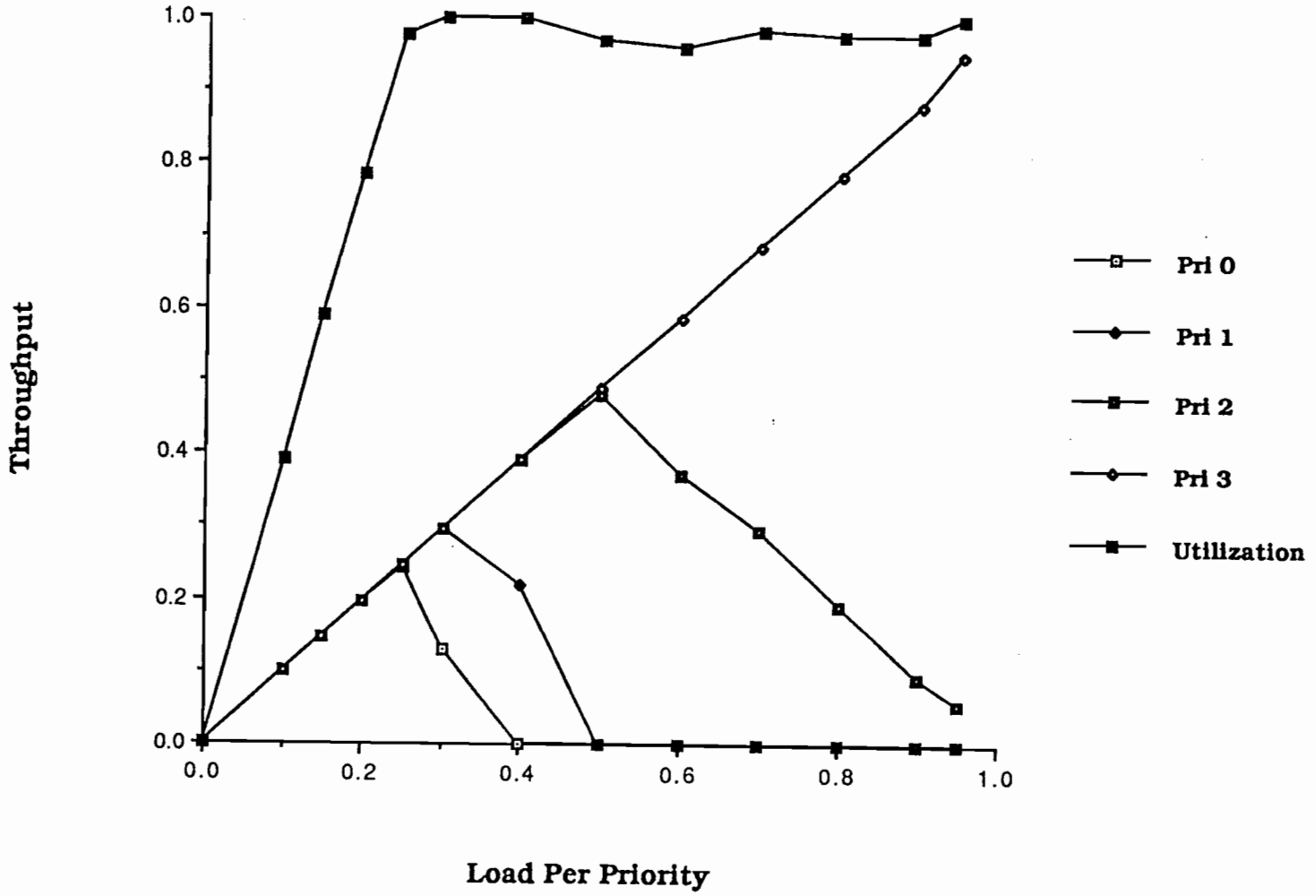
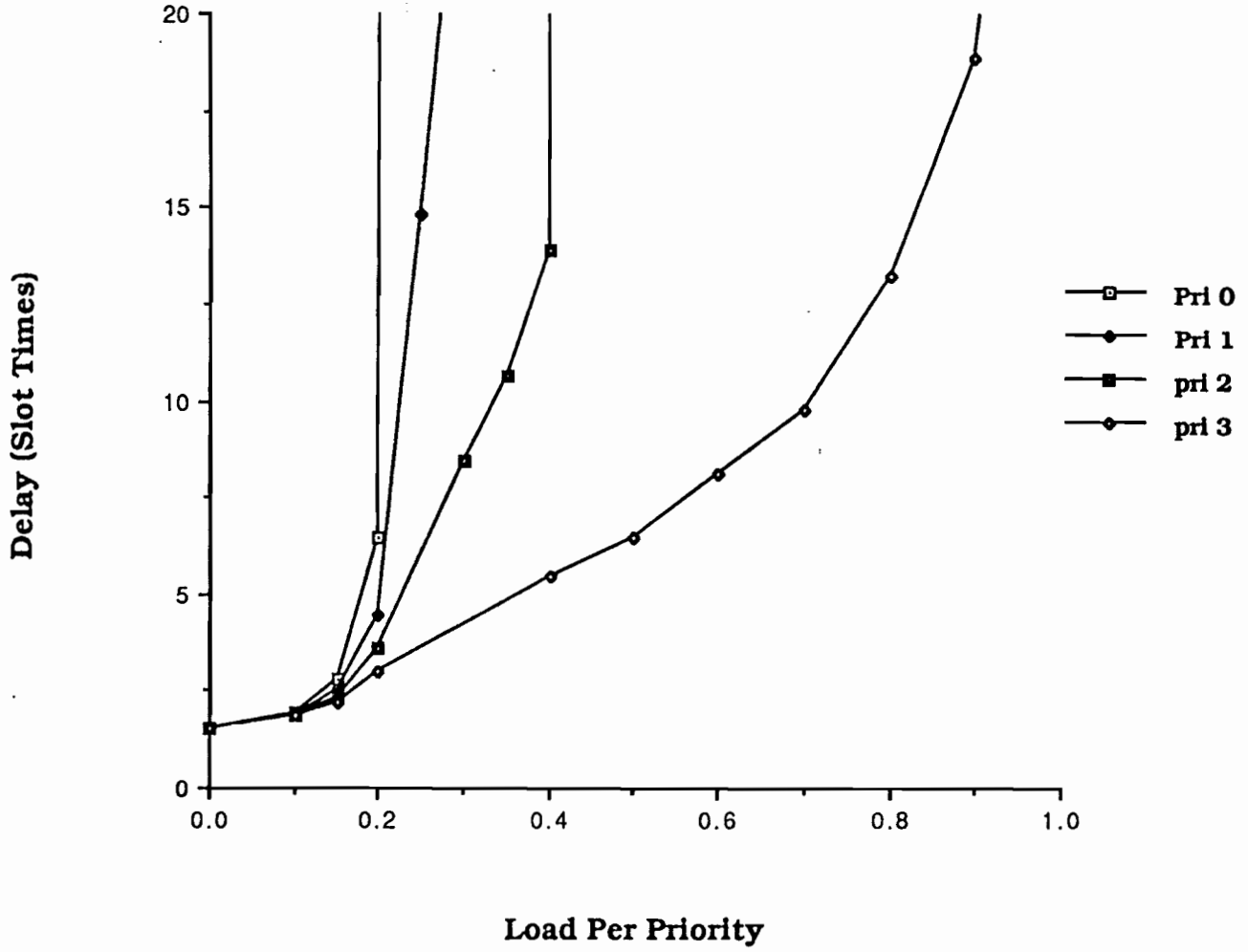


Fig. 4.11: Avg. Total MAC Delay Versus Load; a = 100.0



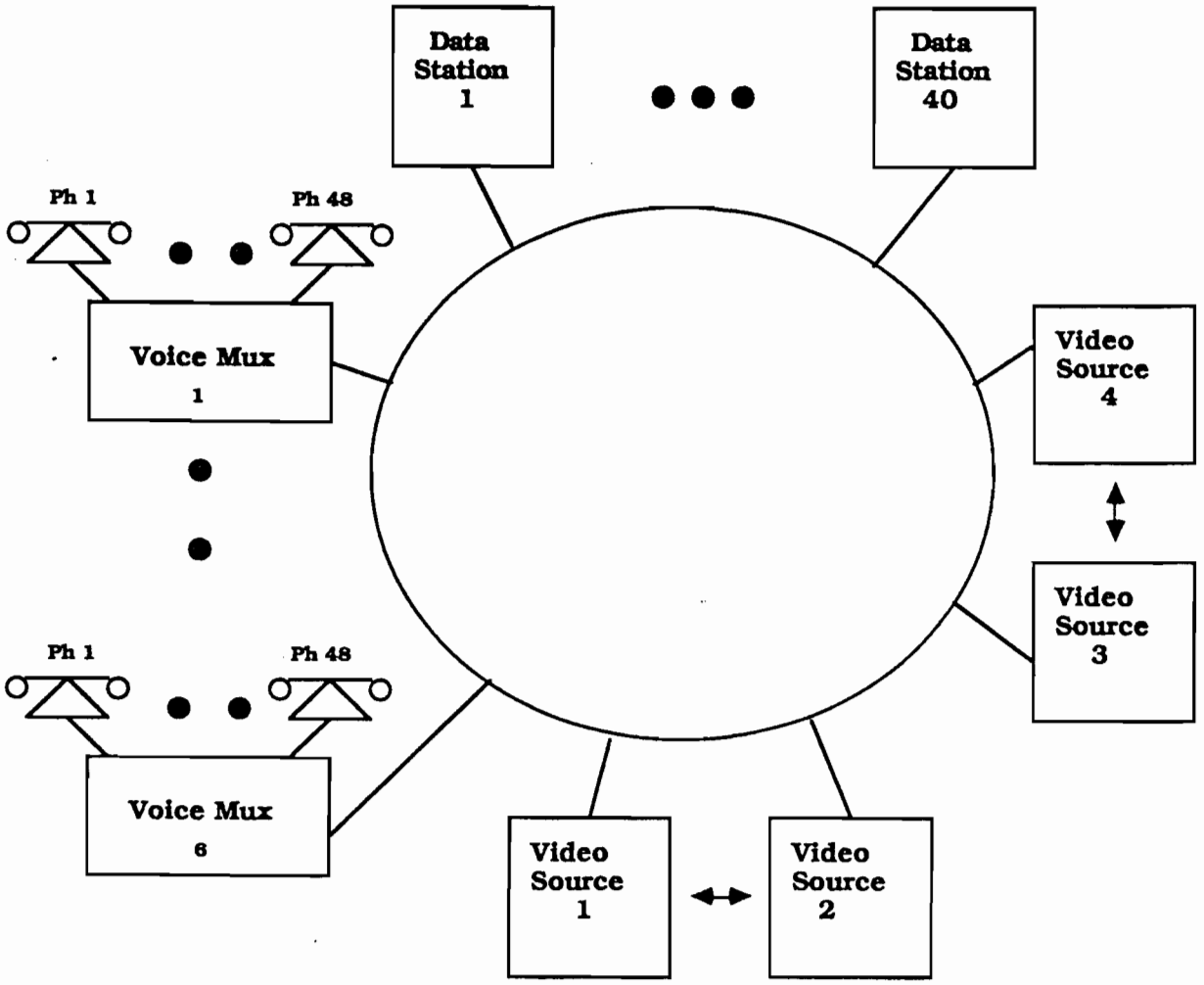
Chapter 5

Performance Comparison of the 802.6 and FDDI

In this chapter the performance characteristics of the DQDB MAN (IEEE 802.6) are compared to those of the Fiber Distributed Data Interface (FDDI) protocol [10-13]. Both of these protocols are designed to support time sensitive services, such as packet voice, and asynchronous services such as the traffic generated by computers. The study done in this chapter, however, shows that the two protocols have different performance characteristics. Using computer simulations, the FDDI and the 802.6 protocols were compared on the basis of four performance criteria: throughput, network utilization, delay, and fairness. The analysis of the performance measures is followed after the description of the network model.

5.1 The Network Model

To compare the performance characteristics of the 802.6 and the FDDI protocols, an integrated network consisting of the components shown in Figure 5.1 was considered. As shown in this figure, the network consisted of three types of components: voice multiplexers, video sources and data stations. The line rate and the network length were assumed to be 100 Mbps and 200 km, respectively. Note that these figures are the maximum allowable settings for FDDI. It was also assumed that the stations are equally



Data Stations:

- Packet Size: 1536 bits
- Exponential Arrival Distribution

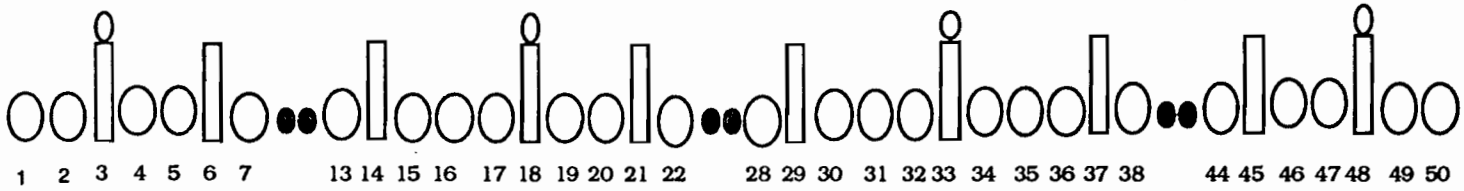
Voice Multiplexers:

- Multiplex 2 T1 lines
- Packet Size: 512 bits

Video Sources:

- Packet Size: 512 bits
- Arrival Rate: 5 Mbps/Source
- Exponential Arrival Distribution
- Two Active Video Conferencings

Fig. 5.1: The Integrated Network Components



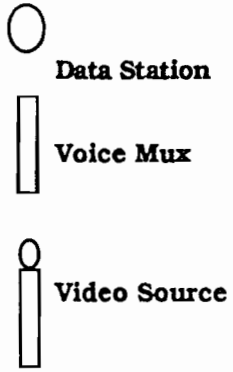


Fig. 5.2: The network arrangement used for the simulation model

distanced and that the velocity of media is 2/3 of the speed of light. The three types of stations shown in Figure 5.1 were placed on the network in such a way that the network would be homogeneous. Figure 5.2 shows the location of each station on the network.

In this section, the significance of each type of network component as well as the modeling perspectives of the network for both protocols are explained. The 802.6 protocol was modeled using the DQDB model described in the previous chapters. The FDDI protocol was modeled using the General LAN Analysis System (GLAS) [14].

5.1.1 Voice Multiplexers

A voice station in this study referres to as a voice multiplexer that multiplexes the traffic generated by 48 PCM channels used in digital telephony. That is, it multiplexes two T1 lines [15]. To provide each PCM channel with 64 Kbps of bandwidth, each channel is serviced by generating a packet voice consisting of 512 bits every 8 milliseconds. Thus, the traffic generated by 6 voice multiplexers of this kind consumes 18.432 Mbps of the total network bandwidth.

For the DQDB model, it was assumed that the 288 telephones on the network form 144 active connections such that each connection is supported by two NA slots; one on each bus. To assign 18.432 Mbps of bandwidth to the voice stations, the slowest isochronous rate and the sum of all but the slowest isochronous rate

(Table 3.1) were set to 64 kbps per bus and 9152 kbps per bus, respectively. Based on the description of the non-arbitrated service model in Chapter 3, the frame generated by this model has 724 slots out of which 144 slots on each bus are NA slots. Also, note that the voice stations in the DQDB simulation model neither receive nor transmit any packets. They are included in order to provide the same network topology for both protocols.

On the FDDI model, the Target Token Rotation Timer (TTRT) was set to 8 msec to meet the timing requirements of the voice packets. The arrival of the packet voices was modeled by constant mean-interarrival distribution with the mean value of 8 msec/48 which is 1.667×10^{-4} seconds.

5.1.2 Video Sources

The Asynchronous Transfer Mode (ATM) has been approved to be the mode of transportation for the Broadband ISDN networks of the future [16]. The concept of ATM is based on the idea of dynamic allocation of bandwidth to a particular service based on the burstiness of the traffic generated by that service. Verbiest and Pinnoo developed a CODEC system which uses variable bit rate coding to transmit video over ATM networks [17]. In this scheme, the required bandwidth depends on the type of the video service. For instance, cable TV requires an average of 16.8 Mbps whereas a video conference can be supported with an average bit rate of 5 Mbps. Kishimoto and Ogata [18] showed that, in ATM networks, the

distribution of mean interarrival time for packet video depends on not only the type of the video service but also on the length of the video packets [18]. According to this research, slow moving pictures, like video conferences, with small packet lengths (512 bits) have exponential mean interarrival distributions.

The four video sources considered in this study are modeled based on the assumptions explained above. It is assumed that each video packet contains 512 bits and that the video packets have exponential interarrival time distributions. The mean interarrival time is set to generate an average bit rate of 5 Mbps per video source. To minimize the packet loss rate caused by the network overload, the video service was assigned to the highest asynchronous priority in both protocols (i.e., priority 3 in 802.6 and class 7 in FDDI). Note that the four video sources support two active video conferences between nodes 3 and 18 and nodes 33 and 48 (Figure 5.1). The consequence of this assumption for the DQDB model is that nodes 3 and 33 and nodes 18 and 48 send all of their packets on bus A and bus B, respectively.

5.1.3 Data Stations

There are forty data stations on the network model. The load for these stations is equally distributed among three priorities (Priorities 0-3 for 802.6 and 4-6 for FDDI). It is assumed that the data packets have exponential message length distributions with the mean of 1536 bits. The mean interarrival time of the packets is

assumed to be exponential. Also, the traffic assignment model described in section 5.1 was used for the data stations in the DQDB model. Note that the parameter to be varied here was the total data rate of each station which was specified by the mean interarrival time on each bus (Table 3.1).

5.2 Comparison of Performance Measures

The network model described in the previous section was used to compare the performance measures of the two protocols. The primary consideration in adjusting the Token Rotation Timers (TRTs) for FDDI was to provide service for the voice and video users irrespective of the load generated by the data stations. By trial and error, it was found that a reasonable setting for the TRT values of class 6, 5, and 4 could be 5.4 msec, 5.3 msec and 5.3 msec, respectively. The TRT for class 7 was set to 7 msec. As mentioned before, the TTRT was set to 8 msec to meet the timing requirements of the voice packets.

The two protocols were compared on the basis of their throughput, network utilization, delay, and fairness characteristics. To make the comparisons simpler, we refer to priority 2 (or class 6), priority 1 (or class 5), and priority 0 (or class 4) as third, second, and first priorities, respectively.

5.2.1 Throughput Characteristics

The throughput characteristics of the two protocols for voice and video stations are shown in Figure 5.3. The horizontal axis in this figure is the load per priority for the data stations which is offered to the MAC layer from the higher OSI layer. This load, which does not include any MAC layer protocol overhead, is refered to as the information load. The total media throughput was calculated according to the definition in Table 3.2. The figure shows that for all values of information load, the voice traffic has total media throughput of 0.184 which is the same as the offered voice traffic. The video traffic, however, which requires 20 Mbps (i.e., total media throughput of 0.2) gets better service from the 802.6 protocol. The 802.6 supports video at almost 20 Mbps up to the information load of 0.371. Then, by increasing the information load, it gradually decreases to about 19 Mbps. This drop in throughput occurs at the information load of almost 0.0895 when FDDI is used.

Figures 5.4 and 5.5 show the total media throughput versus information load per priority for the three asynchronous priorities used in data stations. In both protocols, all three priority classes are supported at small loads. As the load increases, the throughput of the first, the second, and the third order priorities, in that order, decreases. This is consistent with the throughput characteristics of 802.6 shown in Figure 5.10 and that of FDDI shown in [11]. Also, after the load increases beyond a certain point (almost 0.15 for

FDDI and 0.3 for 802.6), the total media throughput for a particular priority becomes less than the offered load at that priority. This is because in both protocols the highest asynchronous priority is assigned to the video sources. The data stations can use only the portion of the bandwidth which is not used by the voice and video stations.

Fig. 5.3: Total Media Throughput Versus Info. Load Per Priority

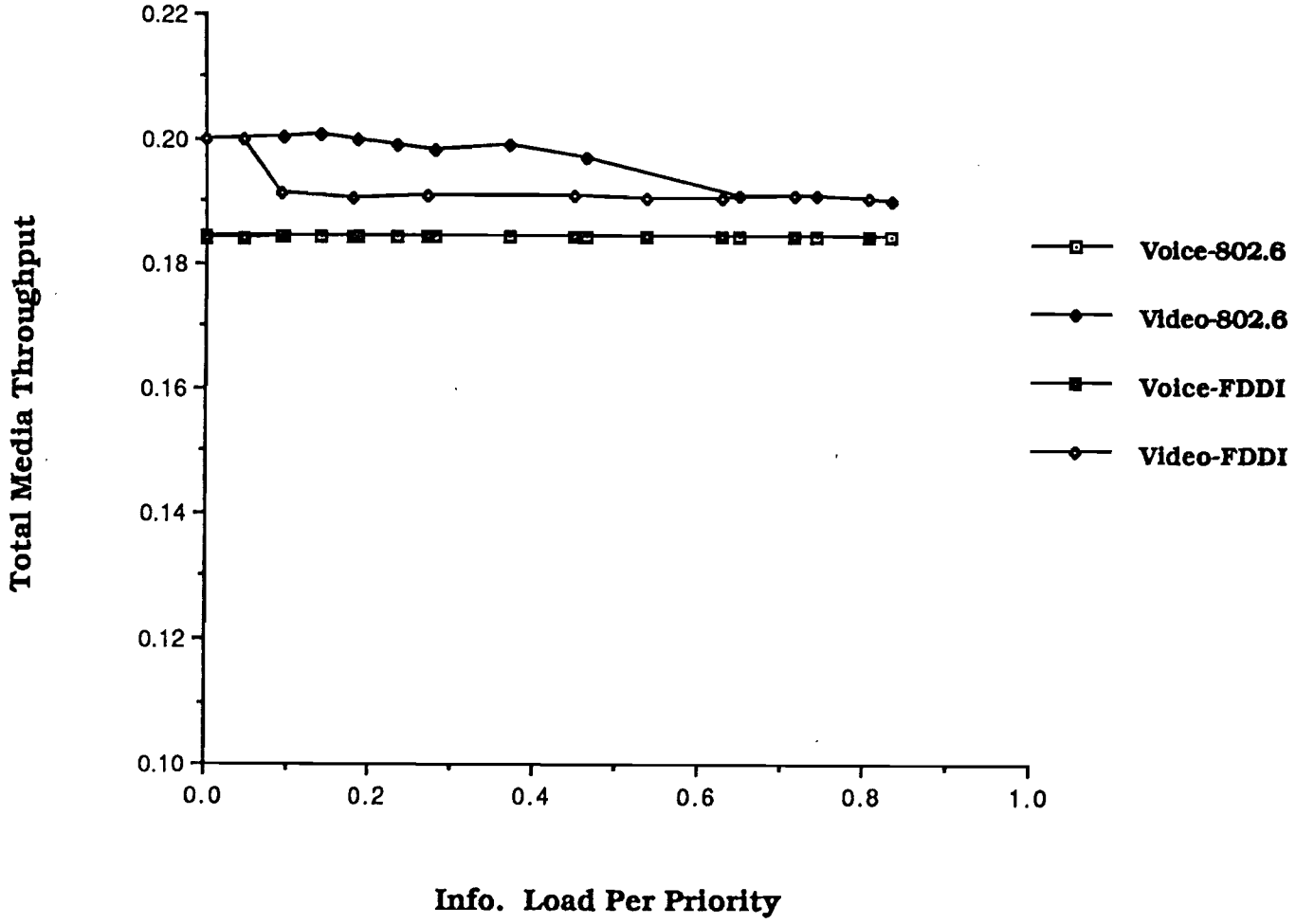


Fig. 5.4: Total Media Throughput Versus Info. Load Per Priority; 802.6

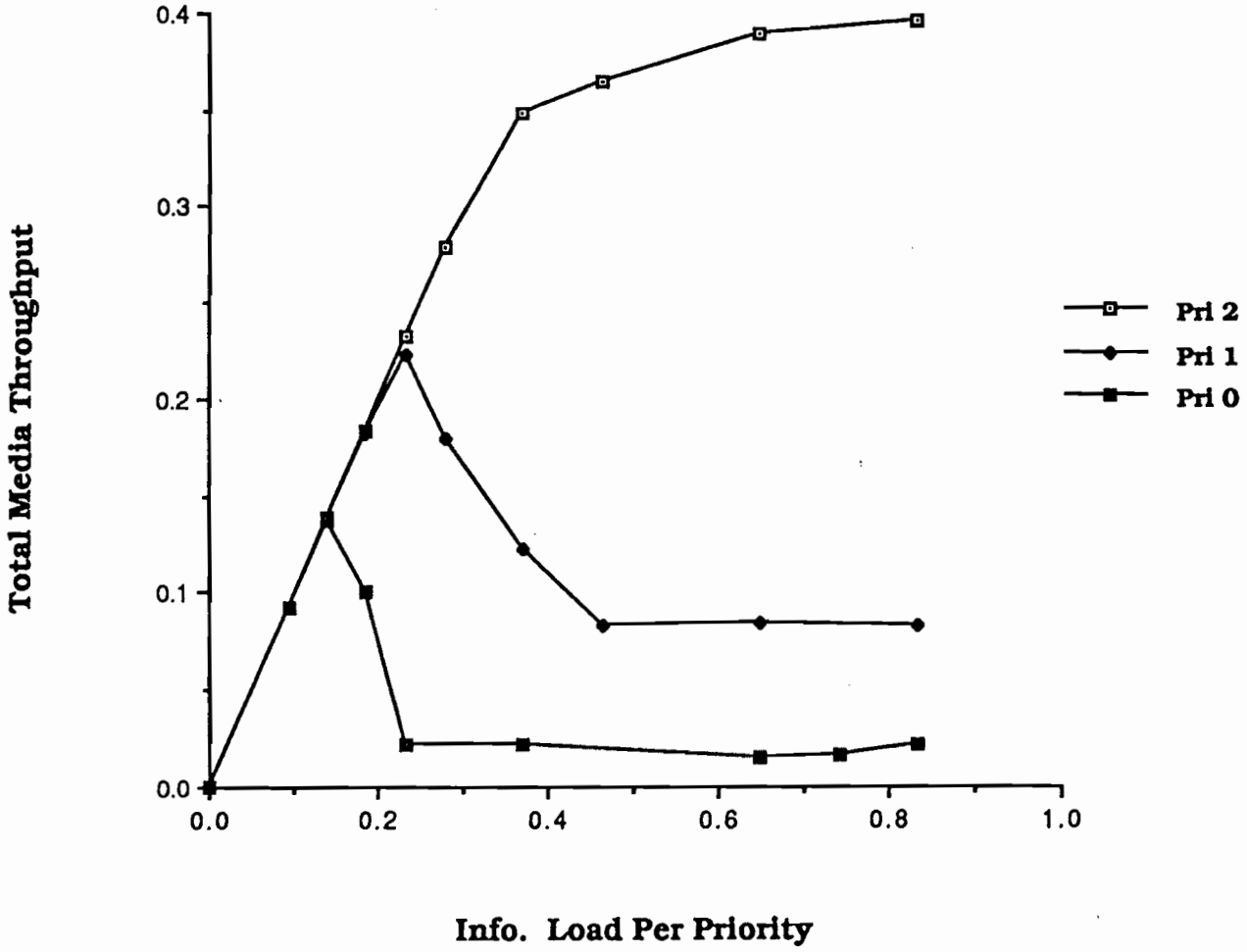


Fig. 5.5: Total Media Throughput Versus Info. Load Per Class; FDDI

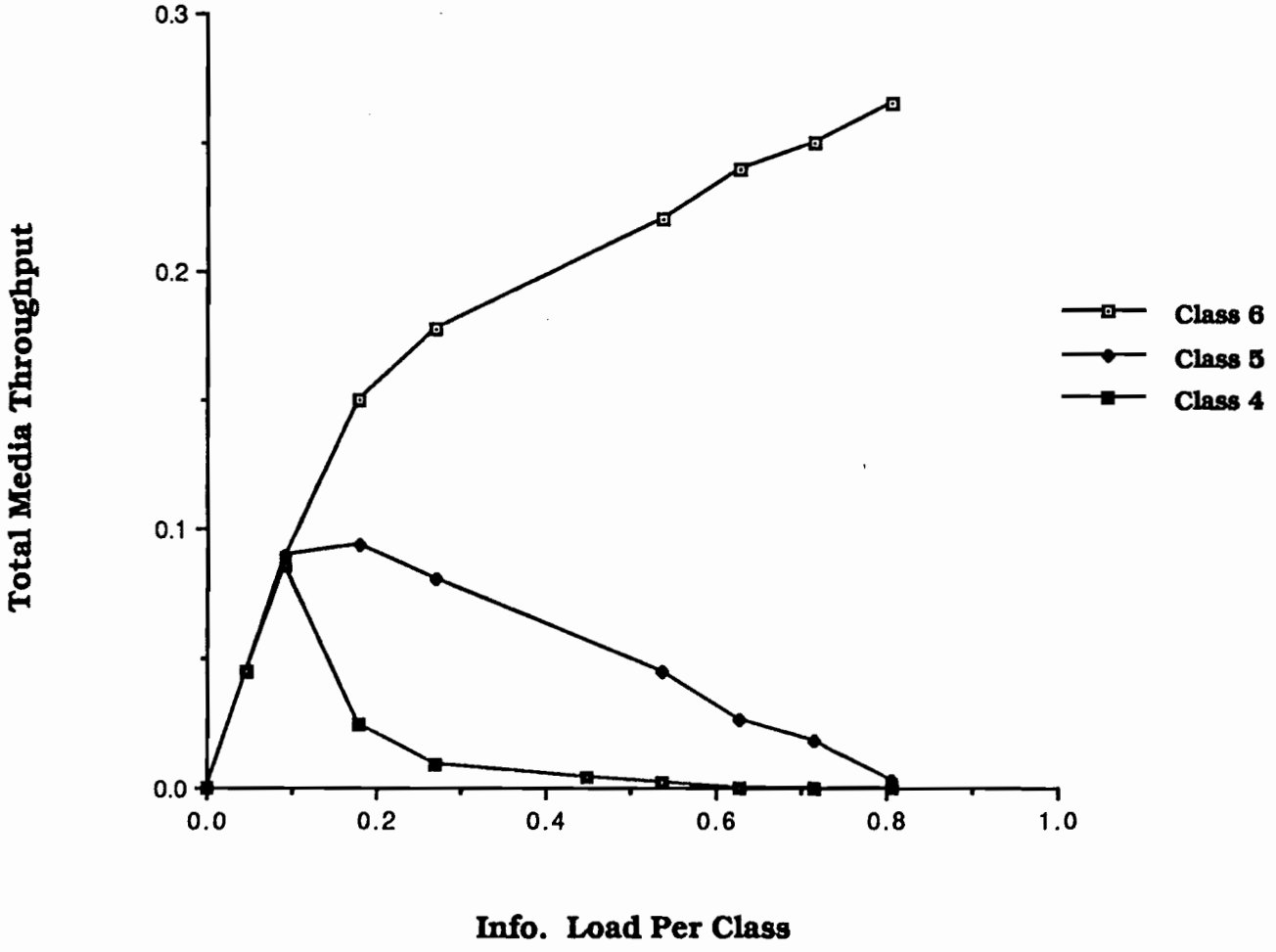


Fig. 5.6: Total Media Throughput Versus Info. Load Per Priority; Priority 0

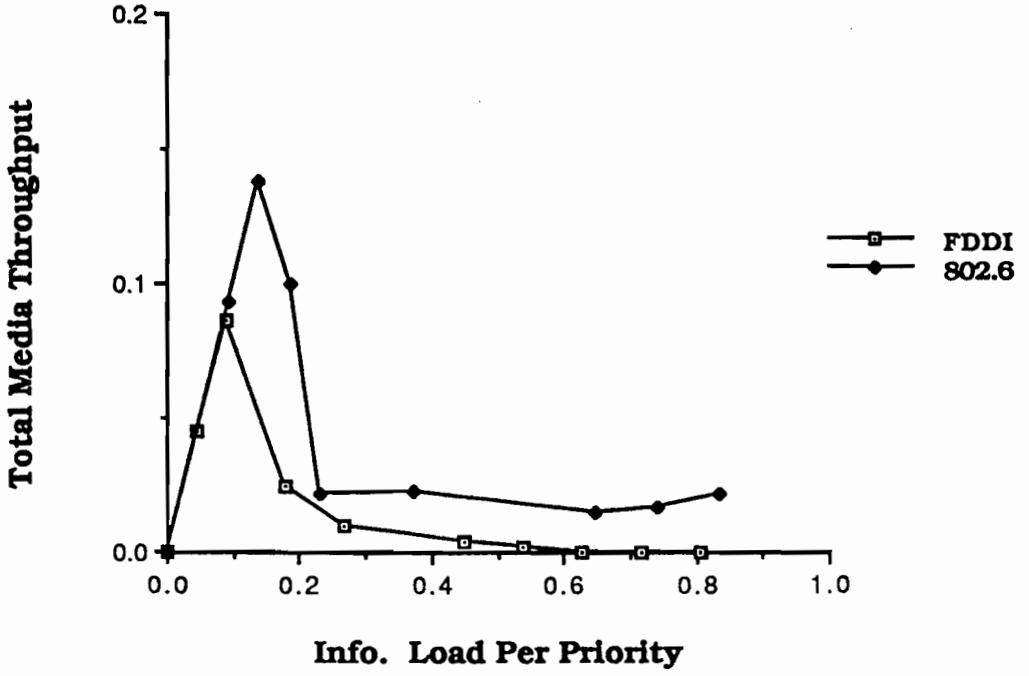


Fig. 5.7: Total Media Throughput Versus Info. Load Per Priority; Priority 1

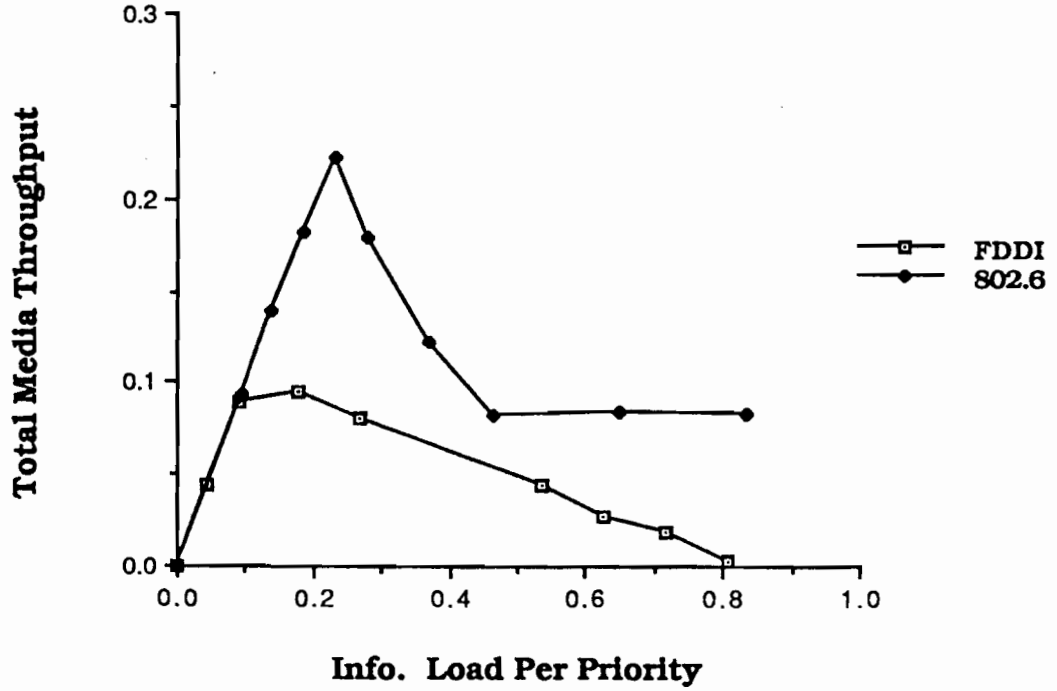
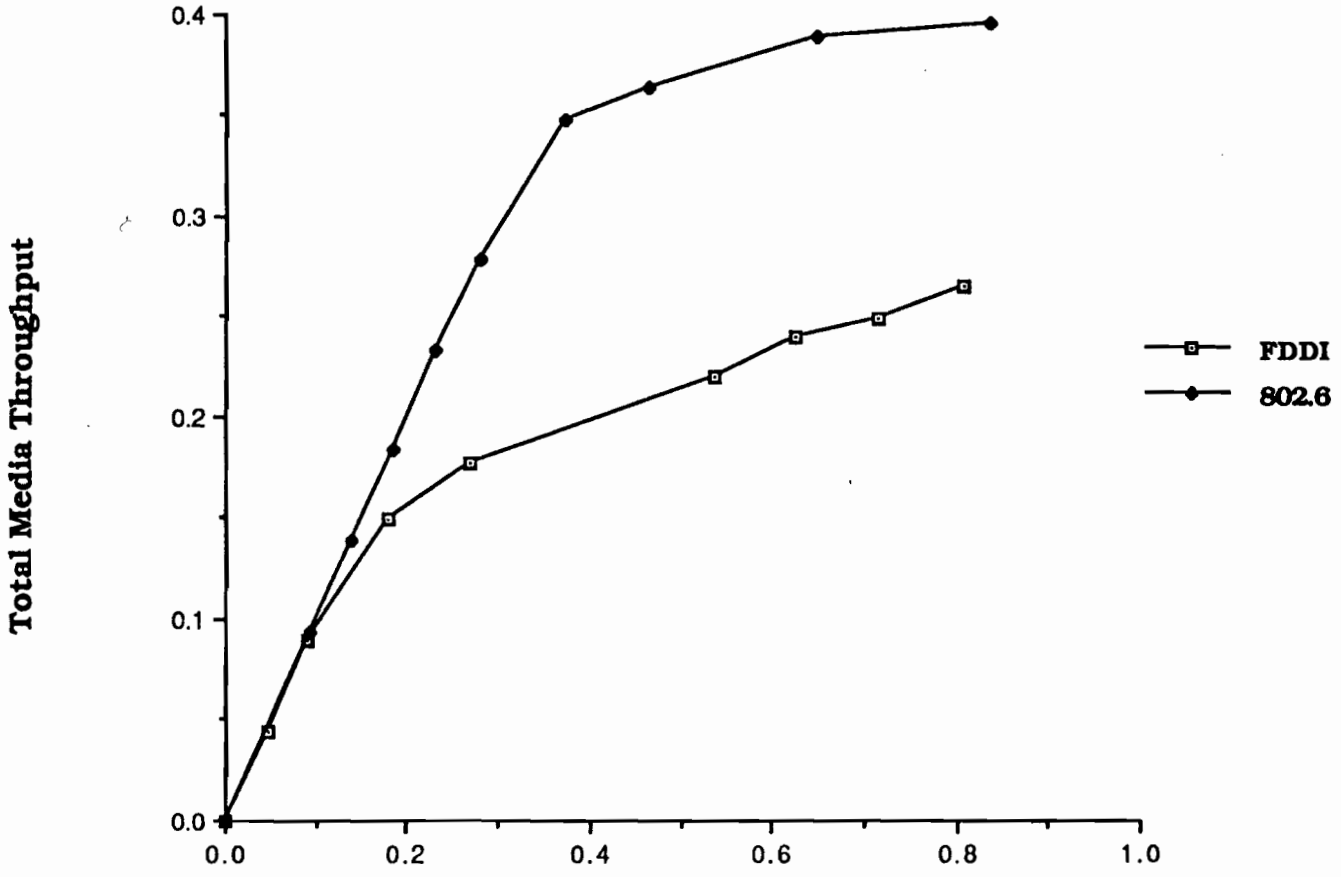


Fig. 5.8: Total Media Throughput Versus Info. Load Per Priority; Priority 2



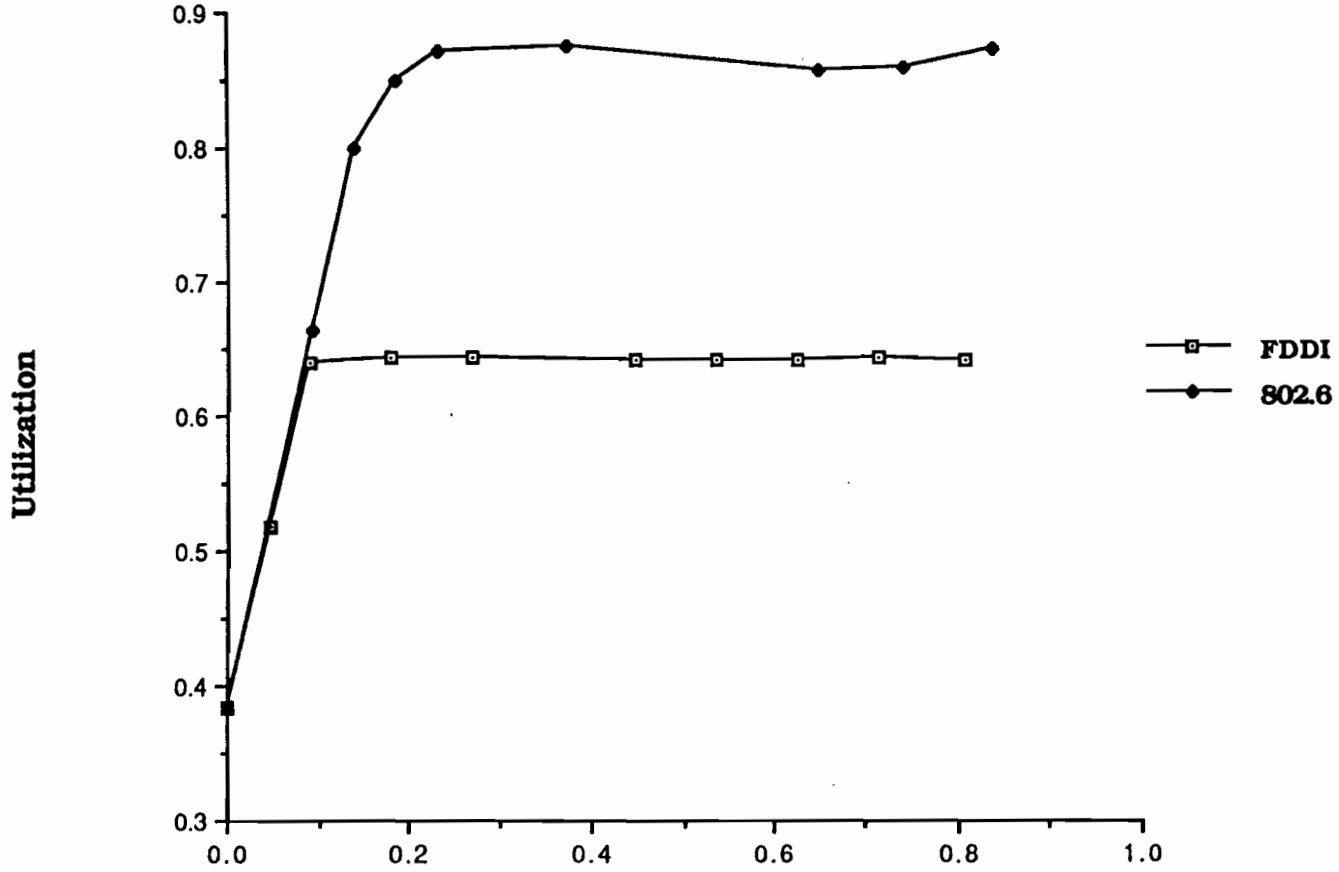
Figures 5.6–5.8 provide comparisons between throughput characteristics of the two protocols. According to these figures, all priority classes of the 802.6 protocol have higher throughputs than their corresponding priority classes of FDDI. Two possible reasons for lower throughput figures in FDDI are the following: TRT values of FDDI are small in comparison with the network size and the FDDI protocol adds more overhead to the packets than the 802.6 protocol does. To an average packet size of 1536 bits, FDDI and 802.6 add 180 and 120 bits of overhead, respectively. The former reason will be elaborated on in the next subsection.

5.2.2 Network Utilization

We define utilization at a particular load as the sum of the total media throughputs (Table 3.2) for all synchronous and asynchronous classes at that load. A comparison between network utilization of the two protocols is provided in Figure 5.9. Note that the horizontal axis is the information load per priority class of data stations.

According to the this Figure FDDI and 802.6 have almost the same utilization for small loads (loads of less than 0.1). Also, note that at the data load of zero, the utilization is the sum of the throughputs of voice and video stations. As the load increases, however, the 802.6 provides better network utilization than FDDI. This happens for the same two reasons mentioned in the last part of the previous subsection. First, FDDI adds more overhead to

Fig. 5.9: Utilization Versus Info. Load Per Priority



packets. To the 512-bit voice and video packets, FDDI adds 180 bits of overhead whereas the overhead for the 802.6 is only 40 bits. Also, the overhead for a data packet with average length of 1536 bits is 180 and 120 for FDDI and 802.6, respectively. The extra overhead of FDDI causes lower utilization because it wastes the given bandwidth by sending overhead instead of information.

The second reason is that the TTRT and the TRT values of FDDI are small in comparison with the network size. As mentioned before, the reason for selecting small TRT values was to make sure that the token rotates around the network fast enough so that the time sensitive services like packet voice and packet video could get serviced on time. The effect of this fast rotation of the token on the data stations, at higher loads, is that they will not be able to hold the token long enough to transmit all their packets. The utilization of an FDDI network can be increased by choosing larger values of TTRT. In [13], it was shown that the utilization plus overhead of 99.5 percent can be achieved using TTRT of 50 msec and a ring latency of 0.25 msec.

5.2.3 Delay Characteristics

The delay characteristics of both protocols are shown in Figures 5.10 and 5.11 . These figures indicate that, for both protocols, the total MAC delay (defined in Table 3.2) increases by increasing the information load. Also, the delay graphs increase very sharply as the load per priority approaches the peak

throughput at that priority. Because voice and video priorities have small packet loss rates, their delay graphs never increases sharply. Also, as expected, in both protocols the higher priorities experience less delay than lower priorities.

Figures 5.12–5.15 provide comparisons between each priority class of the two protocols. Figures 5.13–5.15 indicate that all asynchronous classes of data stations experience less delay when the 802.6 protocol is used. At small loads, the larger delay of FDDI is caused by the ring latency. That is, if the station which has a packet to transmit is not holding the token, it has to wait until the token arrives at the station. The 802.6, however, provides almost immediate access at small loads. The inferiority of delay characteristics of FDDI at larger loads is due to the small values of TRTs used in the FDDI network (section 5.2.2).

Figure 5.12 shows the comparison between the delay characteristics of the video sources. The 802.6 provides the video sources with less delay for loads of up to 0.55. Also, note that if the video receivers require a delay of less than 1.5 msec (minimum delay provided by FDDI), FDDI cannot be used to support the video service.

Fig. 5.10: Avg. Total MAC Delay Versus Info. Load Per Priority; 802.6

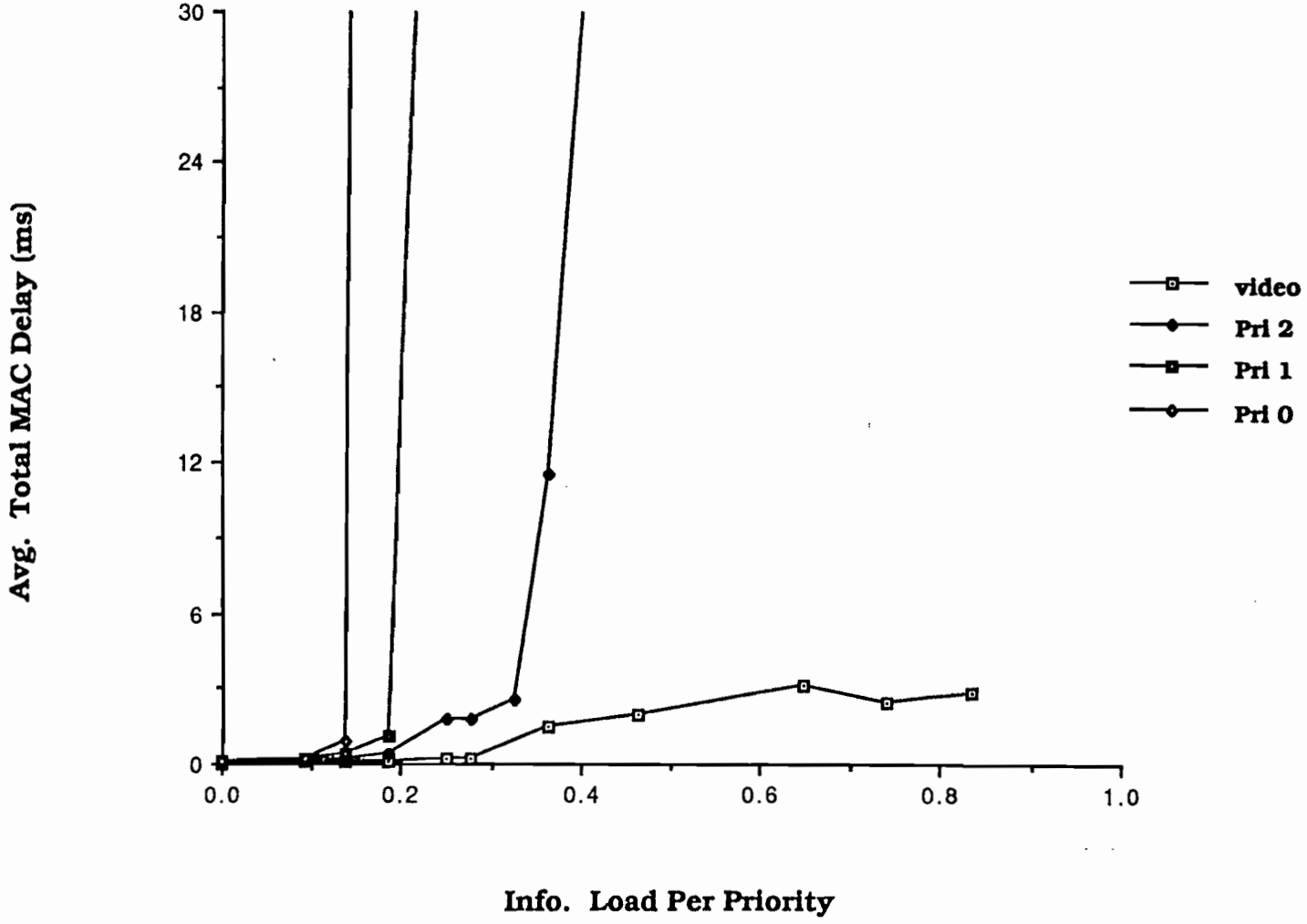


Fig. 5.11: Avg. Total MAC Delay Versus Info. Load Per Class; FDDI

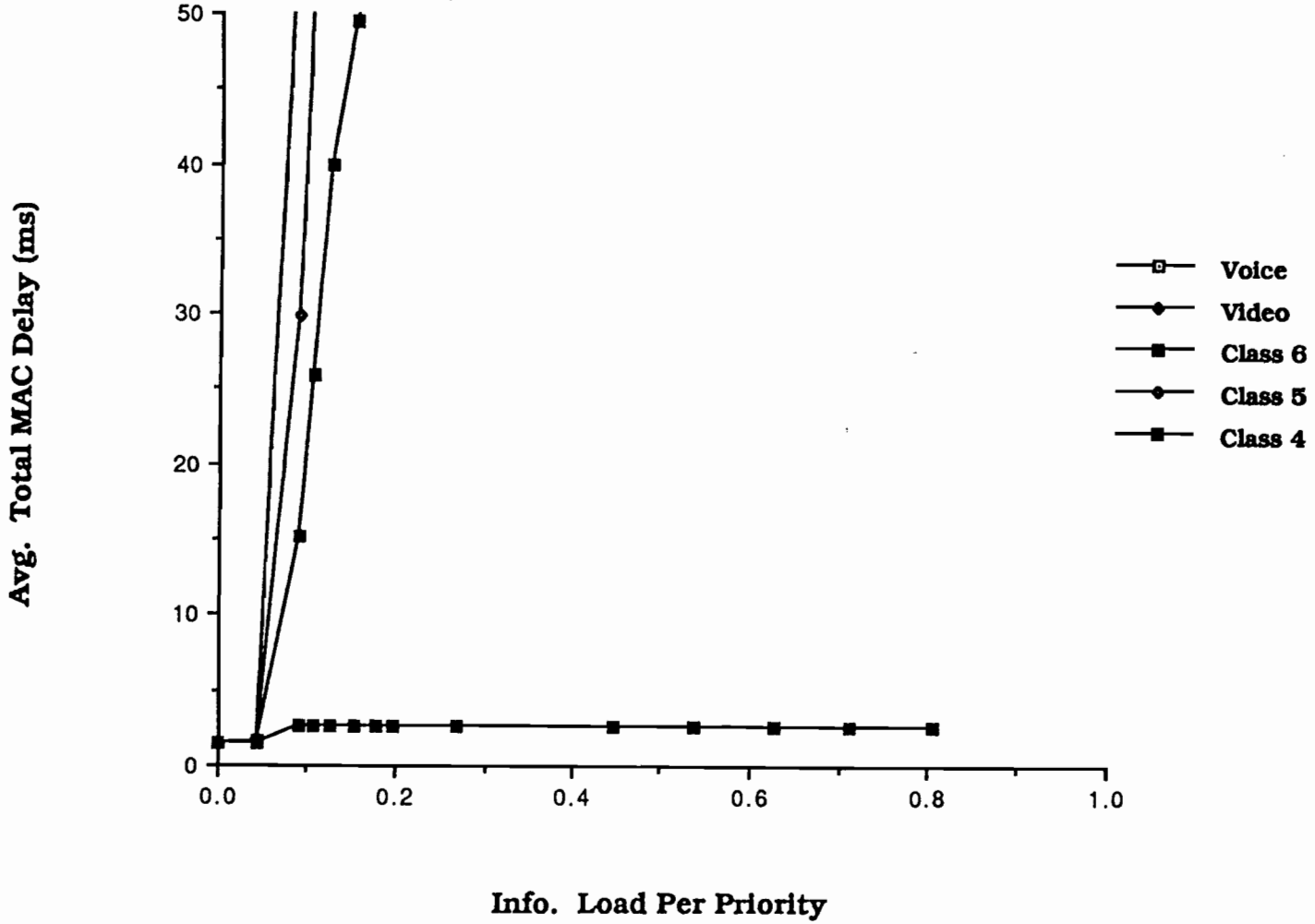


Fig. 5.12: Avg. Total MAC Delay Versus Info. Load Per Priority; Video Sources

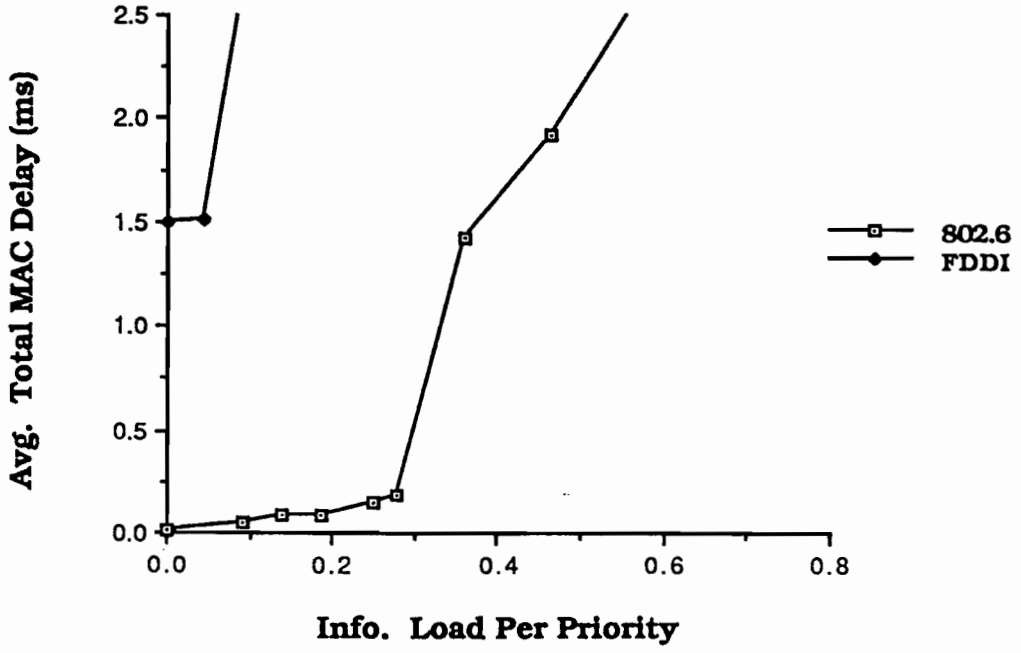


Fig. 5.13: Avg. Total MAC Delay Versus Info. Load Per Priority; Priority 2

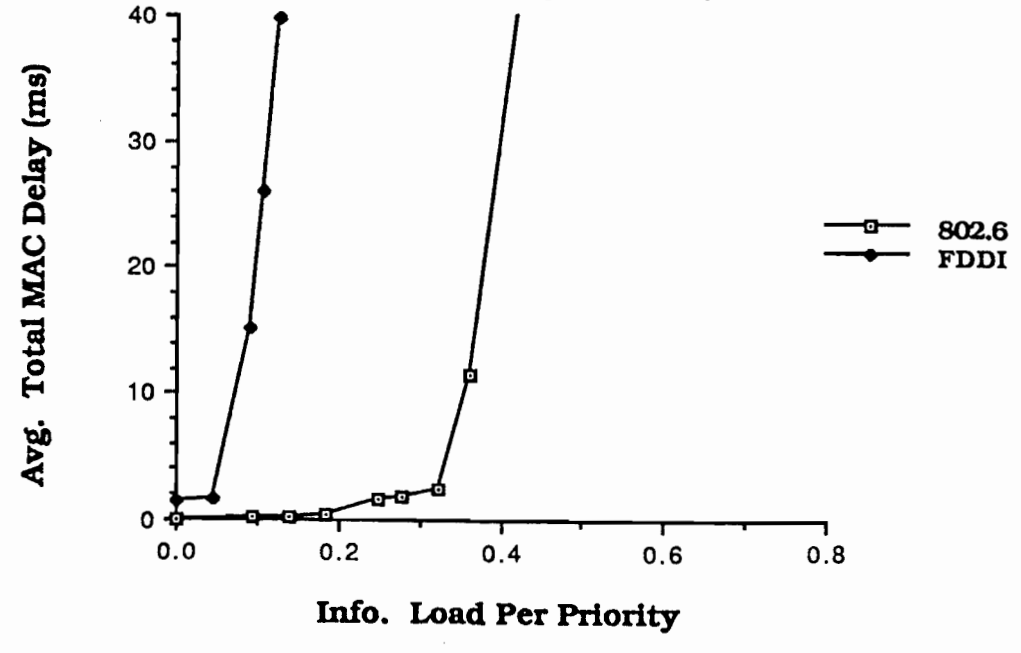


Fig. 5.14: Avg. Total MAC Delay Versus Info. Load Per Priority; Priority 1

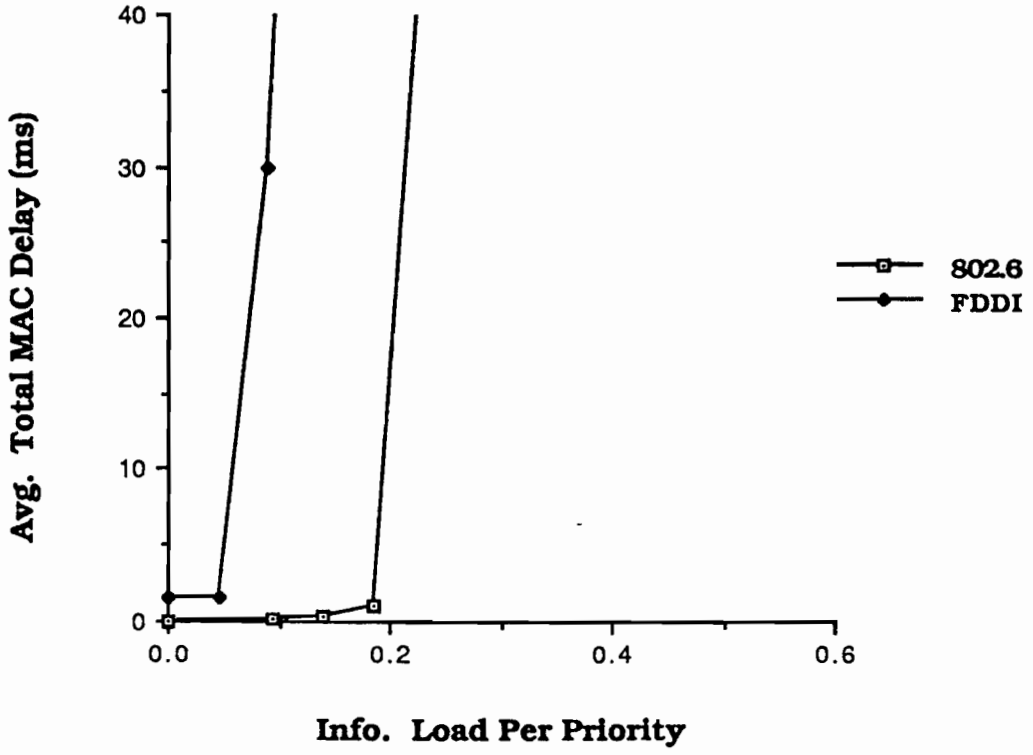
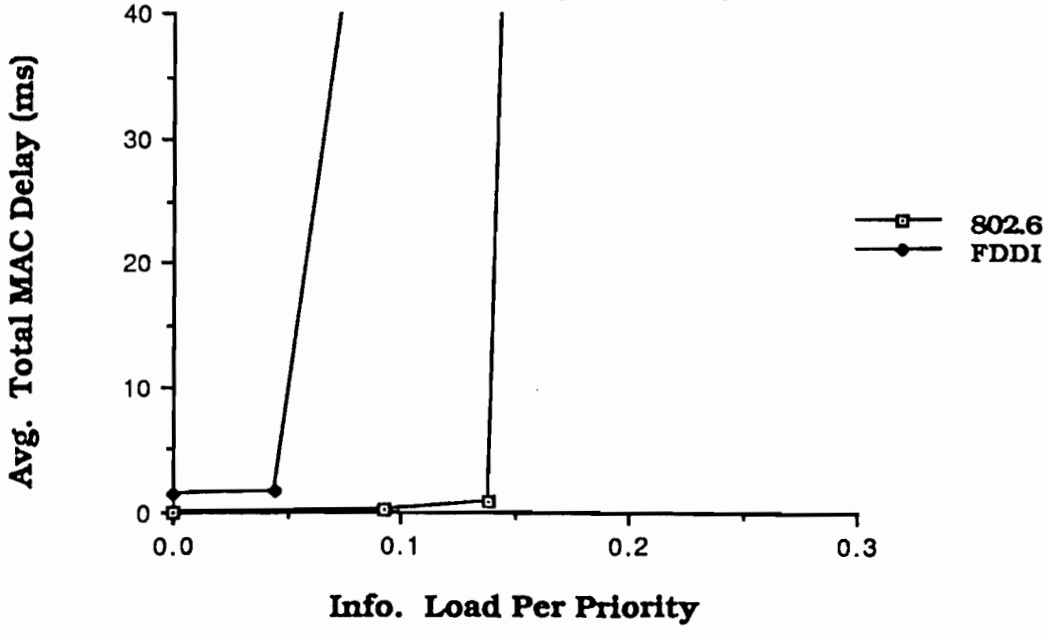


Fig. 5.15: Avg. Total MAC Delay Versus Info. Load Per Priority; Priority 0



5.2.4 Fairness

The comparisons given in the previous sections were based on the global performance measures of the network. In this section, the fairness of the two protocols are compared by considering the per-node throughput of the highest priorities of data stations. Figures 5.16 and 5.17 show throughput versus order of data stations for the two protocols. Throughput is the ratio of the number of packets serviced at a station to the number of packets arrived at a station. The horizontal axis in these figures is the order in which the data stations appear on the network. For example, the data station with the order of three is the third data station on the network which, according to Figure 5.2, is station 4.

Figure 5.16 shows the fairness characteristics of the 802.6 protocol for three different loads. At small loads (load of 0.1) all the data stations get the throughput of 1. However, as the load increases, the stations that are farther from the slot generators, particularly the ones in the middle of the bus, get smaller throughputs. As mentioned earlier (Chapter 4), this behavior is caused by the fact that the propagation delay of the request bits is larger for the stations that are located farther from the slot generators. This longer propagation delay also causes skewing in the queueing order of the distributed queue.

Figure 5.17 shows that FDDI treats the data stations more equally than the 802.6 protocol. At small loads (load of 0.1) all stations get throughputs of almost one. As the load increases,

however, the throughputs of all stations decrease. This is because as the load increases, the stations have more packets to send. However, due to the increased load of each station, the token does not rotate fast enough so that the stations can transmit all their packets. Consequently, the rate of packet loss increases which causes the per-node throughput defined in this section to decrease. This figure also illustrates some very interesting behavior of the network. The four data stations that are located right before the video sources have higher throughputs than the rest of the data stations. This effect, which is noticeable at higher loads, can be explained in the following way. The FDDI network has been tuned to allow the video sources to transmit all their packets even when the offered load from the data stations is high. That is, the amount of time that each data station gets to hold the token depends on the portion of the TTRT that is left to serve video and voice packets. Figure 5.17 suggests that,on average, by the time the token reaches the data station located right before a video source, there is enough time left to not only serve the video source but also to transmit more packets from the data station.

Fig. 5.16: Throughput Per Node Versus Order of Data Stattion 802.6; Priority 2

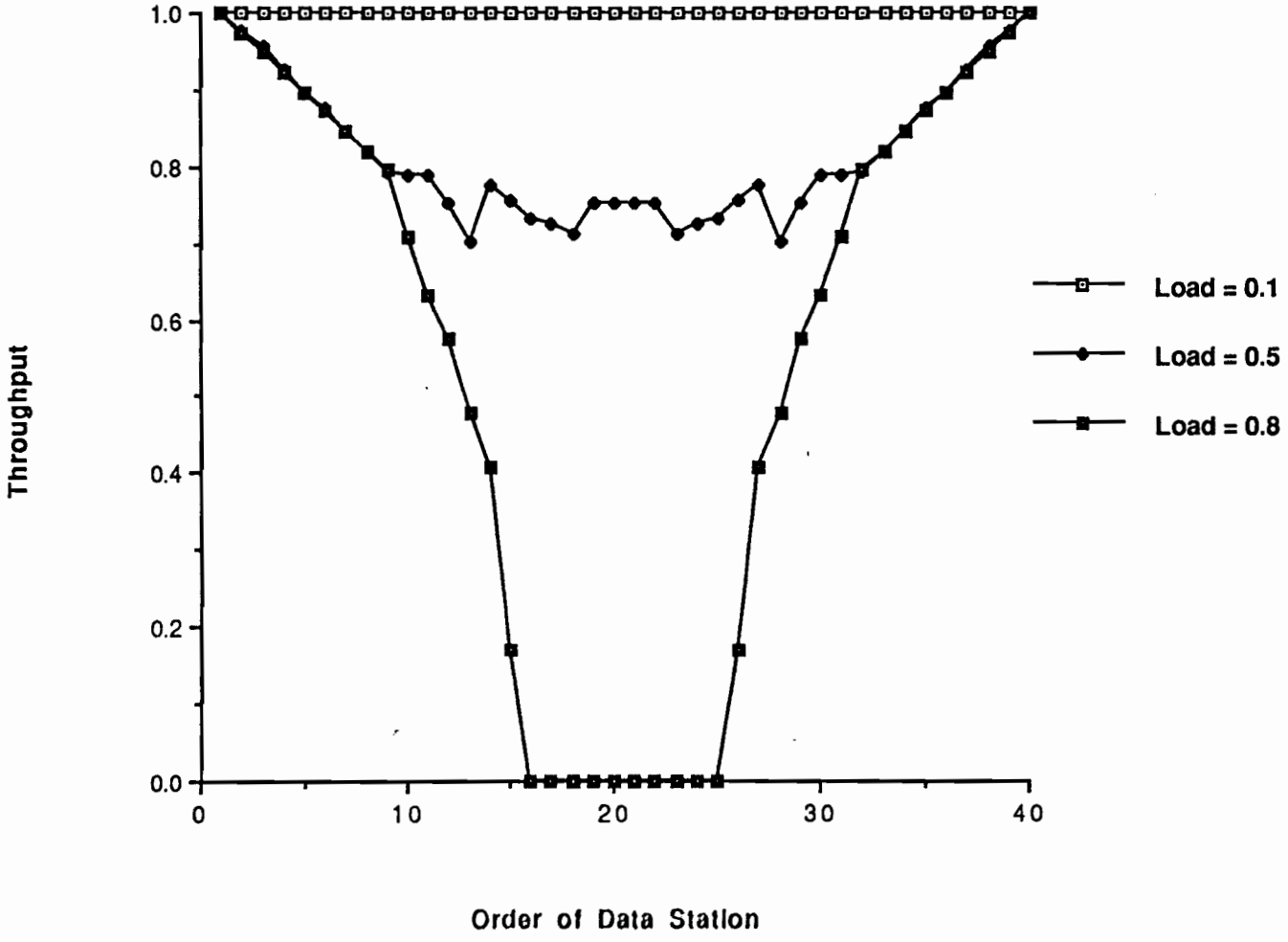
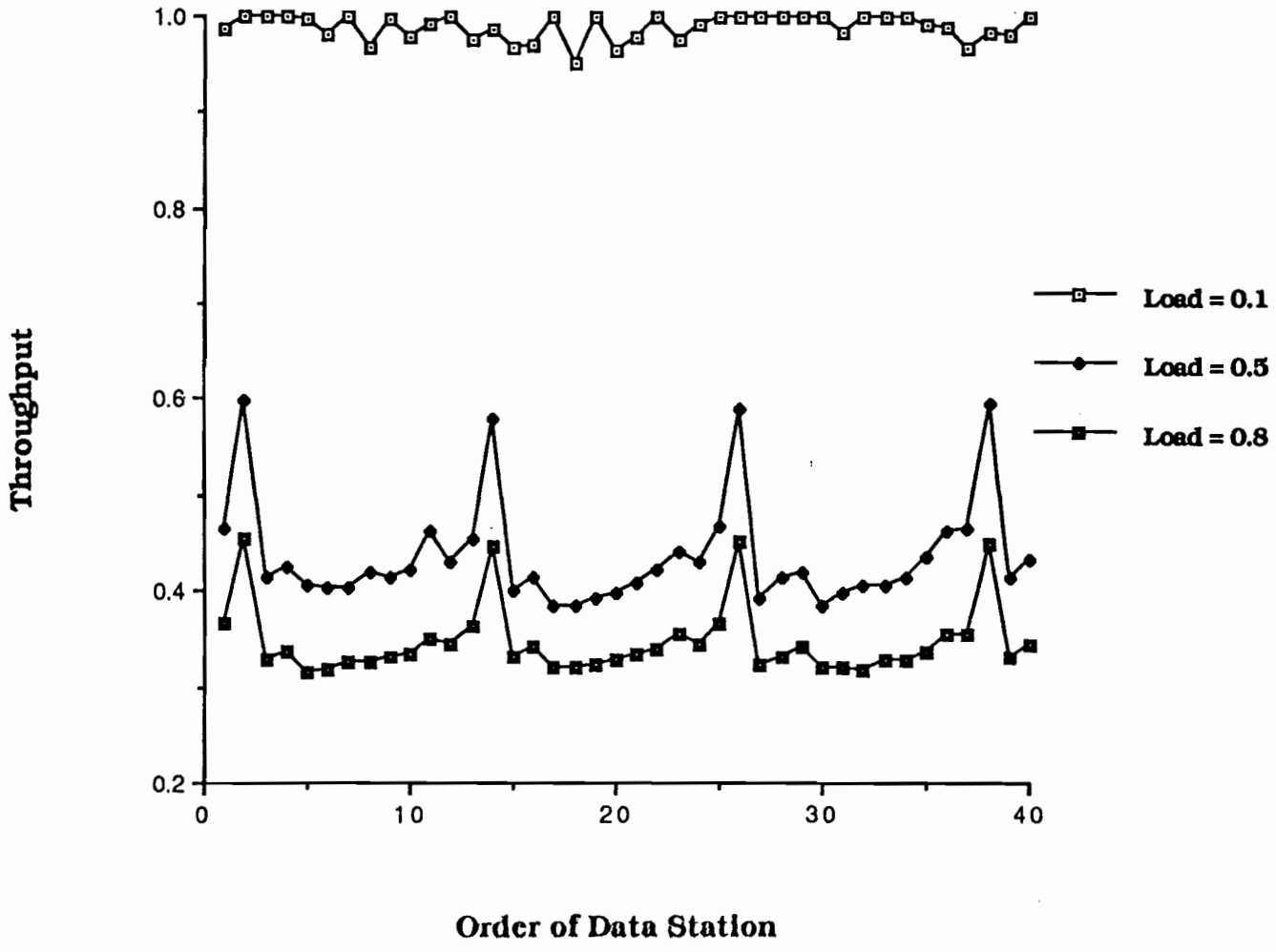


Fig. 5.17: Throughput Per Node Versus Order of Data Station
FDDI; Class 6



Chapter 6

Conclusions

Overall, all the objectives of this study were attained. For the DQDB MAN, a simulation model was developed and validated. The performance measures generated by the simulation model matched very closely with the results obtained by Newman. The single priority performance measures were validated under the assumptions of uniformly distributed destination addresses and exponential distribution of mean interarrival times of packets. These results showed that the total MAC delay of the network is close to that of a perfect scheduler for different network sizes. However, increasing the network size causes the stations that are farther from the slot generators to experience more delays than the ones that are closer to the ends of the buses. The same effect was observed when the network load was increased for a constant network size. The increase in the delay of the stations in the middle of the bus is due to the longer propagation delays of request bits to the slot generator and the empty slot to the station. Also, the longer propagation delays of request bits for these stations causes skewing in the queueing order. The multiple priority model was validated under the same assumptions. The results indicated that stations with higher priorities experienced lower delay and had higher throughput. The delay curves increased sharply as the

offered load at a priority reached the peak throughput for that priority.

The performance of the DQDB MAN was compared to that of the FDDI protocol for an integrated network consisting of voice, video, and data stations. It was assumed that the voice stations multiplex 2 T1 lines and that the video service is assigned to the highest asynchronous service class and has exponential mean interarrival distribution. While the offered load by the voice and video stations was kept constant, the performance measures of the two protocols were obtained for various loads generated by data stations. The results indicated that the DQDB protocol had higher network utilization and throughput for all priorities of data stations. Also, all priorities of data stations experienced lower delays when the DQDB protocol was used. As far as the video stations are concerned, it turned out that for loads of up to 0.55, the DQDB protocol had superior delay and throughput performance. Although both protocols support the video service without significant loss of throughput, if the maximum delay requirements of the video service is less than minimum delay provided by FDDI, this protocol can not be used to support video. As far as the fairness characteristics are concerned, it was shown that almost all the stations get the same throughput when FDDI is used. The data stations that are located right before the video stations, according to the simulation results, get slightly higher throughputs than the rest of the data stations. In the DQDB model, however, the stations

that are farther from the slot generator get lower throughput. This effect is even more pronounced at higher network loads.

Open issues for further study include studying the effects of replacing the data stations considered in this study by LANs and replacing the video stations by video multiplexers. Also, the consequences of using different network topologies on the fairness characteristics of the two protocols need to be further studied.

References

[1] J. O. Limb, C. Flores, "Description of Fasnet – A unidirectional local area communication network," BSTJ, Vol. 61, No. 7, Sep. 1982, pp. 1413-1440.

[2] F. Tobogi, F., Borgonovo, and L. Fratta, "Expressnet: A high performance integrated services local area network," IEEE J. Sel. Areas Conm., Vol. SAC-1, No. 5, Nov. 1983.

[3] Draft of proposed IEEE Standard 802.6, Distributed Queue Dual Bus (DQDB). Metropolitan Area Network (MAN), P 802.6/D6, November 1988.

[4] R. M. Newman, J. L. Hullet, "Distributed Queueing: A fast and efficient packet access protocol for QPSX," ICCC '86, Munich, pp. 294-299.

[5] R. M. Newman, Z. L. Budrikis, J. L. Hullet, "The QPSX MAN," IEEE Communications Magazine, Vol. 26, No. 4, April 1988, pp. 20-28.

[6] Gary C. Kessler, "IEEE 802.6 MAN," LAN, Vol. 5, No. 4, April 1990, pp. 102-116.

[7] Averll M. Law, W. David Kelton, *Simulation Modeling and Analysis*, McGraw Hill, 1982.

[8] Jerry Banks, John S. Carson, *Discrete-Event System Simulation*, Prentice Hall, 1984.

[9] R. M. Newman, "Distributed Queueing: Performance Characterisation," Contribution No. 802.6-88/11 to the IEEE 802.6 Working Group, 1988.

[10] American National Standard, "FDDI Token Rin Media Access Conrol (MAC)," ANSI X3.139-1987.

[11] Doug Dykeman, Werner Bux, "Analysis and Turning of the FDDI Media Access Control Protocol," IEEE Jour. on Sel. Areas in Comm., Vol. 6, no. 6, July 1988, pp. 997-1010.

[12] Floyd E. Ross, "FDDI-A Tutorial," IEEE Comm. Mag., Vol. 24, no. 5, July 1986, pp. 10-17.

[13] Floyd E. Ross, "An Overview of FDDI: The Fiber Distributed Data Interface," IEEE Jour. on Sel. Areas in Comm., Vol. 7, no. 7, Sept. 1989, pp. 1043-1051.

[14] Victor S. Frost, W. W. LaRue, et al., " A Tool for Local Area Network Modeling and Analysis", to be published in Simulation.

[15] Bernard E. Keiser, Eugene Strange, *Digital Telephony and Network Integration*, Von Nostrand Reinhold Company, 1985.

[16] Jan P. Vorstermans, Andre P. Vleeschouwer, "Layered ATM Systems and Architectural Concepts for Subscribers' Premises Networks," IEEE Jour. on Sel. Areas in Comm., Vol. 6, no. 9, Dec. 1988, pp. 1545-1555.

[17] Willem Verbiest, LUC Pinnoo, "A Variable Bit Rate Video Codec for Asynchronous Transfer Mode Networks," IEEE Jour. on Sel. Areas in Comm., Vol. 7, no. 5, June 1989, pp. 761-770.

[18] Ryozo Kishimoto, Yoshifumi Ogata, et al., "Generation Interval Distriubtion Characteristics of Packetized Variable Rate Video Coding Data Streams in an ATM Network," IEEE Jour. on Sel. Areas in Comm., Vol. 7, no. 5, June 1989, pp. 833-841.

[19] Ellen L. Hahne, Abhijit K. Choudhury, et al., "Improving the Fairness of Distributed- Queue-Dual-Bus Networks", Infocom 90.

Appendix A

Some Architectural Aspects of the Software

This appendix describes some of the architectural perspectives of the software designed for the DQDB simulation model. The software overview is presented and the initialization process of the simulation software is described. At the end, several suggestions for an implementation of a general purpose user interface for the simulation software are provided.

A.1 Software Overview for the DQDB Model

The simulation software for the DQDB model is written in the C language and is executable on all the 3B2 UNIX systems. The overall architecture of the simulation model can be divided in three distinct parts: initialization, discrete event simulation, and report generation. Initialization, which is the main focus of this appendix, involves setting the model parameters to their initial values. These parameters are defined in Chapter 3. The discrete event scheduling concept as well as the three events identified for the simulation model are also described in Chapter 3. The report generation routine, which is invoked at the end of the discrete event simulation, produces the performance measures defined in Chapter 3 by operating on the statistics that are collected in the process of discrete event simulation.

To perform a simulation, the user needs to type, at the UNIX prompt, the name of the executable file (currently called qpsx) followed by the name of an input file. This input file which will be described in the next section contains certain parameters required by the simulation model.

A.2 Initializing Routines

There are three initializing routines defined in the DQDB simulation model: Initialize_N_read, makedata, and readd. The C code for these routines is included in Appendix B. The relationship between these routines is demonstrated in Figure A.1.

The first routine that is invoked by the simulation model is the Initialize_N_read routine. This procedure starts by setting the protocol counters and the statistics collector variables to their default values. Then, the model parameters defined in Chapter 3 (Table 3.1) are initialized to their user-specified values by calling the makedata and readd subroutines. Finally, the event calender is initialized to include the first one of each type of event.

The makedata subroutine is designed to create a file including all the initial values for the parameters used by the simulation model. The current implementation of this subroutine assumes that all the station on the network have the same parameters. Also, it is assumed that the load is assigned according to the traffic model of Chapter 4. The interarrival time of the packets and the message length distributions are assumed to be

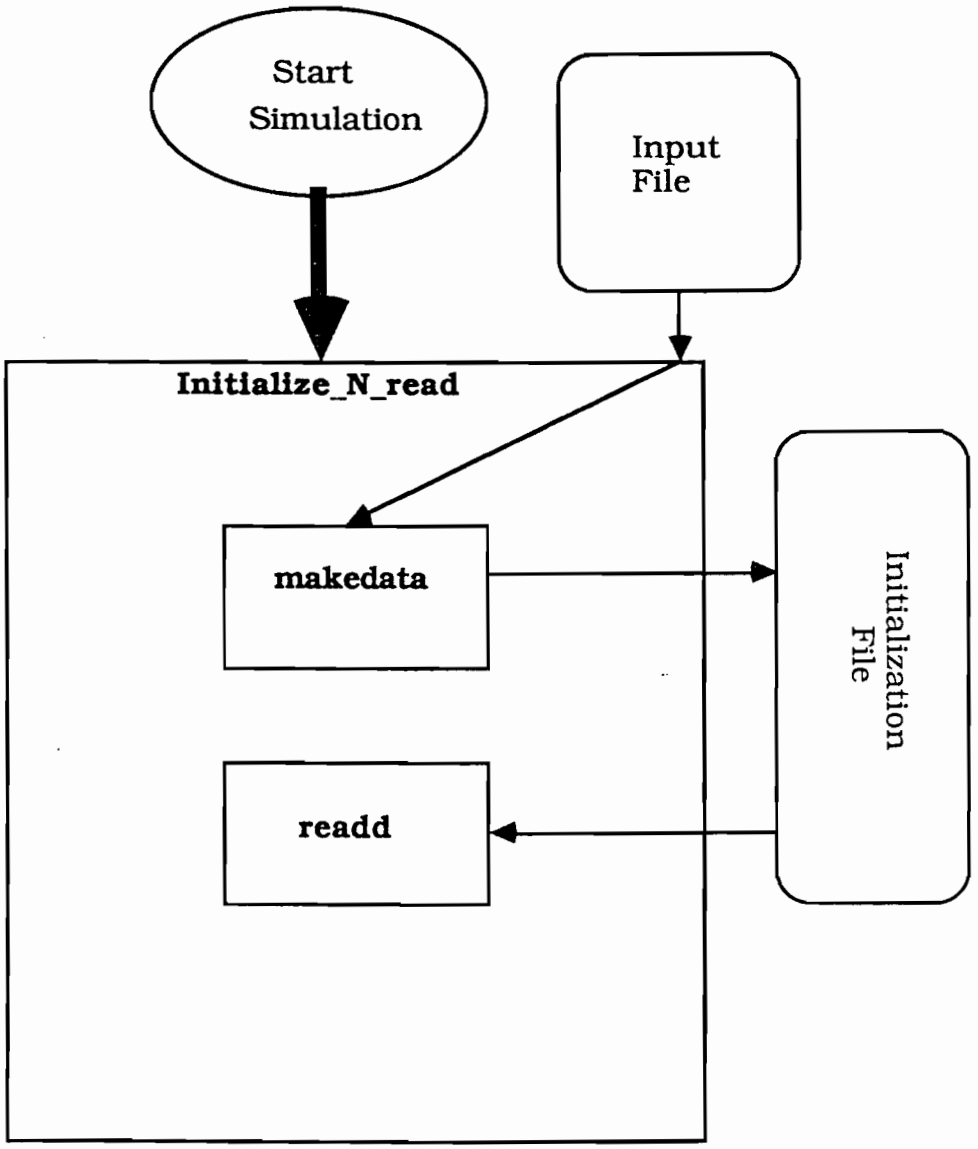


Figure A.1: The Initialization Files and Their Relashionships

exponential and constant, respectively. This subroutine operates on two data files: an Input File which is provided by the user and an Initialization File which is generated by this routine and contains the overall simulation parameters and the initial parameter values of each station. The required formats of these files are described in Tables A.1 and A.2. Upstream and downstream in Table A.2 are defined with respect to the flow direction of slots on bus A. Also note that the stations are numbered in increasing order with the smallest number, one, assigned to the slot generator of bus A. For the slot generator of bus B (i.e. the first station on bus A) the station distance from the upstream node is set to zero. For the slot generator of bus B (i.e. the last station on bus A), however, the station distance to the downstream station is set to zero.

After calling the makedata subroutine, the Initialize_N_read subroutine calls the readd subroutine which is responsible for setting the simulation parameters to the values specified in the Initialization file.

Table A.1

Format Requirements of the Input File

Line No.	File Contents	Comments
1	Name of Initialization file	
2	Name of the Output file	
3	Simulation Time (sec.)	Defined in Table 3.1
4	% Transient Response Time	Defined in Table 3.1
5	Velocity of Media (m/sec)	Defined in Table 3.1
6	Line rate of one bus (kbps)	One half of the Line Rate defined in Table 3.1
7	Isochronous users exist on bus A	There are two possible values: 2 and 0. Two is used to indicate that there are isochronous users on bus A whereas a zero on this line indicates that there are no isochronous users on bus A.
8	Isochronous users exist on bus B	Same as line 7 but for bus B.
9	Slowest isochronous rate on bus A , if any (kbps).	Defined in Table 3.1
10	Sum of all but the slowest isochronous rate on bus A, if any (kbps).	Defined in Table 3.1

11	Slowest isochronous rate on bus B, if any (kbps).	Defined in Table 3.1
12	Sum of all but the slowest isochronous rate on bus B, if any (kbps).	Defined in Table 3.1
13	Number of Stations on the network.	
14	Distance between the stations on the network (km).	
15	Mean interarrival time of packets (sec).	Defined in Table 3.1
16	Portion of the station load which is to be assigned to priority 1.	A REAL number between zero and one.
17	Portion of the station load which is to be assigned to priority 2.	A REAL number between zero and one.
18	Portion of the station load which is to be assigned to priority 3.	A REAL number between zero and one.
19	Portion of the station load which is to be assigned to priority 4.	A REAL number between zero and one.
20	Average Message Length	Defined in Table 3.1

Table A.2

Format Requirements of the Initialization File

Line No.	File Contents	Comments
1	Simulation Time	Defined in Table 3.1
2	% Transient Response Time	Defined in Table 3.1
3	Velocity of Media (m/sec)	Defined in Table 3.1
4	Line rate of one bus (kbps)	One half of the Line Rate defined in Table 3.1
5	Isochronous users exist on bus A	Defined in Table A.1
6	Isochronous users exist on bus B	Defined in Table A.1
7	Slowest isochronous rate on bus A , if any. (kbps).	Defined in Table 3.1
8	Sum of all but the slowest isochronous rate on bus A, if any (kbps).	Defined in Table 3.1
9	Slowest isochronous rate on bus B, if any (kbps).	Defined in Table 3.1
10	Sum of all but the slowest isochronous rate on bus B, if any (kbps).	Defined in Table 3.1
11	Number of Stations on the network.	

12	Upstream station	The first parameter of station 1; the station number of the upstream node which is zero for the slot generator.
13	Downstream station	The station number of the downstream node.
14	Mean interarrival time of priority 1 on bus A.	
15	Mean interarrival time of priority 2 on bus A.	
16	Mean interarrival time of priority 3 on bus A.	
17	Mean interarrival time of priority 4 on bus A.	
18	Mean interarrival time of priority 1 on bus B.	
19	Mean interarrival time of priority 2 on bus B.	
20	Mean interarrival time of priority 3 on bus B.	
21	Mean interarrival time of priority 4 on bus B.	
22	Interarrival Time Distribution	Defined in Table 3.1; zero and one are used to identify exponential and constant

		interarrival time distributions, respectively.
23	Average Message Length	Defined in Table 3.1
24	Message Length Distribution	Defined in Table 3.1; zero and one are used to identify exponential and constant interarrival time distributions, respectively. This is the last parameter of Station 1.

Note that the parameters of lines 1 through 11 are global simulation parameters whereas the parameters appearing on the next lines are specifications of each station on the network. For the second to the last station on the network, the node specifications are included after the specifications of station one. All the station parameters are specified in the same order as those of station one. The last station, however, would have zero for the downstream station.

A.2.1 An Example

An example is provided to help clarify the simulation process. Table A.3 shows an Input File specifying an integrated network consisting of five nodes. The assumption on this example is that the stations are capable of supporting both isochronous and asynchronous services. Assuming that the name of the Input File shown in Table A.3 is "input.example", the simulation of the model can be scheduled by typing at the UNIX prompt:

```
UNIX Prompt > batch
qpsx input.example
<CNTRL> D
```

After the simulation starts, the makedata routine generates the Initialization File shown in Table A.4. Also, the simulation results are sent to the output file specified by the user in the Input File (i.e. Out.example).

Table A.3

An example Input File for an integrated network

Init.example

Out.example

0.8

0.1

2e+5

50000

2

2

64

9152

64

9152

5

4.0

3.45e-4

1.0

0.0

0.0

0.0

512.0

Table A.4

The Initialization File for the Input File of Table A.3

```
0.800000
0.100000
2.000000e+05
50000
2
2
64
9152
64
9152
5
0
2
0
4.0
3.450000e-04
0.000000e-04
0.000000e-04
0.000000e-04
0.000000e-04
0.000000e-04
0.000000e-04
0.000000e-04
0
512.000000
1
1
3
4.0
4.0
4.600000e-04
0.000000e-04
0.000000e-04
0.000000e-04
1.380000e-03
0.000000e-04
0.000000e-04
0.000000e-04
0
512.000000
1
2
4
4.0
4.0
6.900000e-04
0.000000e-04
0.000000e-04
0.000000e-04
```

6.900000e-04
0.000000e-04
0.000000e-04
0.000000e-04
0
512.000000
1
3
5
4.0
4.0
1.380000e-03
0.000000e-04
0.000000e-04
0.000000e-04
4.600000e-04
0.000000e-04
0.000000e-04
0.000000e-04
0
512.000000
1
4
6
4.0
0.0
0.000000e-04
0.000000e-04
0.000000e-04
0.000000e-04
3.450000e-04
0.000000e-04
0.000000e-04
0.000000e-04
0
512.000000
1

A.3 A General Purpose User Interface

To be able to model various network configurations considered in this study, the makedata routine was modified incorporate the necessary changes to the Initialization file. Because making frequent changes to the makedata routine can become tedious, it is recommended that a general purpose user interface be implemented for the simulation model. The user interface should provide the user with an environment in which the user can specify the simulation parameters, network configuration, and the station parameters. Once these parameters are specified, the user interface generates an Initialization File in the format given in Table A.2. In essence, the user interface replaces the makedata routine. The interaction between the simulation software and the user interface can be implemented by replacing the call to the makedata routine (this call is done inside the Initialize_N_read) by a call to start the user interface.

Appendix B

The Code for the DQDB Simulation Model

This appendix includes the C code for the DQDB simulation model which is currently kept at /usr3/tari/qpsx4.c.

```
/* Declaration of Variables */


#include <stdio.h>
#include <math.h>

#define  MNSTA     100                  /* Maximum number of staions */
#define  MaxQSize  100                  /* Maximum queue size (buffer size) */
#define  CCSG_a    0
#define  bus_a     1
#define  bus_b     2
#define  idle      0
#define  cntdn     1
#define  stdby     2
#define  fmgen     20
#define  slot_size 552
#define  payload   512
#define  MxFrSize  1000                 /* Maximum frame size          */
#define  Expon     0
#define  Constant  1


void     readd(),                /* subroutine to read the input data file */
         initialize_N_read(),    /* subroutine to initialize the nodes */
         fram_gen(),             /* frame generation event */
         slot_arrival(),         /* slot arrival event */
         departure(),            /* departure suroutine */
         pr(),
         aval(),                 /* packet arrival event */
         report(),               /* subroutine to generate statistics of
                                    the performance measures */
         makedata();

int      rate[MNSTA+1][3],       /* array to store the isochronous rate
                                    requested for each station on each bus;
                                    first dimension is the station number,
                                    the second dimension is the bus number */
         ls,                     /* line rate in kbps */
         niso_a,                 /* number of isochronous users on bus A */
         niso_b,                 /* number of isochronous users on bus B */
         nsta,                   /* total number of stations on the network */
         num_qa[3];              /* number of QA slots on each bus */

float    prop_sp,                /* velocity of the media in m/s */
         tnow,                   /* current simulation time */
         fi[3],                  /* frame interval in seconds for each bus */
         tns[3],                 /* total number of slots in the frame for
                                    each bus */
         tsim,                   /* total simulation time */
         expo(),                 /* function to generate random numbers with
                                    exponential distribution */
         pty[5],                 /* array to store the portion of each
                                 /* priority class from the total load */
         trans;

char     datafile[30],
         outfile[30],
         slotout[30];
```

```
struct  station {                   /* data structure to keep track of parameters
                                       of each node; note that the last two
                                       dimensions of the arrays correspond to
                                       the bus number and the priority number */
  float  queue[MaxQSize+1][3][5];
                                    /* array to construct a queue for each bus
                                       and each prioirty */
  float  info_bits[MaxQSize+1][3][5];
                                    /* array to maintain the number of information
                                       bits in each slot */
  int    redn[MaxQSize+1][3][5];
                                    /* array to maintain the number of slots
                                       corresponding to each packet in the queue
                                       for each bus and each priority */

  float  avg_mess;                  /* average message length of the packet */
  int    req_cntr[3][5];            /* request counter of all DQSMs in the node */
  int    cd_cntr[3][5];             /* countdown counter of all DQSMs within the
                                       node */
  int    req_q[3][5];               /* array to be used by the RQM Machine in each
                                       node to queue the request bits */
  float  tdelay[3][5];              /* array to accumulate total queueing delay
                                       of each station for each DQSM within the
                                    /* node */
  float  tadelay[3][5];             /* array to accumulate total MAC delay
                                       for each DQSM within the node */
  int    ncus[3][5];                /* array to accumulate total number of packets
                                       served by each DQSM */
  int    ndiscard[3][5];            /* array to accumulate total number of
                                       packets discarded at each DQSM */
  float  narrive[3][5];             /* array to store number of packet arrivals
                                       at each DQSM */
  int    state[3][5];               /* array to record the state of each DQSM
                                       within the node */
  int    origin;                    /* number of the upstream station on bus A */
  int    destination;               /* number of the downstream staion on bus A */
  float  dis_origin;                /* distance in km to the origin */
  float  dis_dest;                  /* distance in km to the destination */
  float  miat[3][5];                /* array to store the mean interarrival time
                                       of each DQSM  */
  int    niq[3][5];                 /* array to store number of packets in the
                                       queue for each DQSM */
  int    nrdn[3][5];                /* array to store the length of redn array for
                                       each DQSM */
  float  arcount[3][5];             /* array to accumulate the number of
                                       information bits in the arriving packets
                                       at each DQSM */
  float  depcount[3][5];            /* array to accumulate the number of
                                       information bits in the departing packets
                                       at each DQSM */
  float  bitsdep[3][5];             /* array to accumulate total number of bits
                                       departing from each station */
  int    arr_dist;                  /* Integer to specify the arrival distribution
                                       of the station.  Exponential and constant
                                       arrival distributions are represented by
                                       zero and one, respectively.          */
  int    mess_len_dist;             /* Integer to specify the message length
                                       distribution of the station.  Exponential
                                       and constant distributions are represented
```

```
                                   by zero and one, respectively.      */
} sta[MNSTA+1];                  /* array to store parameters of each node */


struct  time{                    /* structure to record and manipulate
                                    the frame on each bus */
  int     frame[MxFrSize+1];     /* array to store the frame; QA slots are
                                    represented by a one wheras NA slots are
                                    represented by a zero */
  int     index;                 /* number of the current slot in the frame;
                                    is used by the frame generation event
                                    to determine whether the next slot is a QA
                                    slot or an NA slot */
} slot[3];                       /* array to store the frame for each bus */


typedef struct  calender{        /* structure of each item of the event
                                    calender */
  int     evt_code;              /* code used in the event calender to represent
                                    the event and its attributes.  The event
                                    codes are the followings:

                                         frame_gen -> 20;

                                         aval -> 1p; where p ( 1 <= p <= 4)
                                         is the packet priority.

                                         slot_aval -> 3Babcd; where B is
                                         the Busy bit which can be either
                                         zero or one, and a,b,c,d correspond
                                         to request bits of priorities 1,2,3,4,
                                         respectively.  Zero for these values
                                         would indicate no request and the
                                         priority number would indicate a
                                         request. */

  float   evt_time;              /* execution time of the event */
  int     node;                  /* node at which the event will be executed */
  int     bus;                   /* bus associated with the event */
  struct calender *next;         /* pointer to the next item in the linked
                                    list */
} *list;                         /* pointer of the above type */


list    evs,                     /* pointer to the event set */
        push(),                  /* function to schedule an event in the
                                    event calender */
        pop(),                   /* function to remove the first element in the
                                    event set */
        makelist();              /* function to initialize the event set */
```

```
/*****************************************************************/
/*                                                               */
/*      Module Name:    Main                                     */
/*      Return Values:  None                                     */
/*      Author:         Farzad Tari                              */
/*                                                               */
/*      Description:                                             */
/*                                                               */
/*      The global simulation seed is randomly set and the       */
/*      parameters of stations are initialized to their          */
/*      specified values.  Then, as long as there is time left   */
/*      to perform the simulation, the simulation time is        */
/*      to the time of the next event in the event calender and  */
/*      the control is passed to the subroutine performing the   */
/*      functions of that event.  At the end of the simulation   */
/*      the performance measures of the model are obtained using*/
/*      the simulation results.                                  */
/*                                                               */
/*****************************************************************/


main(argc,argv)

int argc;
char *argv[];

{
long time(),ranseed;
void srand48();

/* Generate a random seed and initialize drand48() using srand48(seed) */

ranseed = time((long *) 0);
srand48(ranseed);

/* Initialize each station with the values specified by the user */

initialize_N_read(pty,argv,&tsim,&trans,&tnow,rate,&ls,&niso_a,&niso_b,&nsta,
                  &prop_sp,sta,slot,fi,tns,num_qa);


/*  While there current simulation time is less that the total
    simulation time, advance the simulation time to the event time of the
    first item in the event calender and execute that event */

while(tnow <= tsim){
        tnow = evs->evt_time;

        if(evs->evt_code == fmgen){
                fram_gen(slot,tns,tnow,nsta,prop_sp,ls);
         }
        else if(evs->evt_code < fmgen){
                aval(tnow,tsim,trans);
        }
                else {
         slot_arrival(sta,tnow,tsim,trans,prop_sp,ls);
        }
}
```

```
/* Call the report subroutine to generate the performance measures of the
   model based on the simulation results */

report(sta,nsta,ls,tsim,trans,tns,num_qa);
}
```

```
/*****************************************************************/
/*                                                               */
/*      Module Name:    Initialize_N_read                        */
/*      Return Value:   None                                     */
/*      Author:         Farzad Tari                              */
/*                                                               */
/*      Description:                                             */
/*                                                               */
/*      This modules initilizes the station parameters           */
/*      and the simulation parameters to their user-specified    */
/*      values.  The format of the frame on both buses is        */
/*      figured and saved.  At the end, an arrival event is      */
/*      scheduled for all stations that have non-zero mean       */
/*      interarrival times.                                      */
/*                                                               */
/*****************************************************************/

void initialize_N_read(pty,argv,tsim,trans,tnow,rate,ls,niso_a,niso_b,
                       nsta,prop_sp,
                       sta,slot,fi,tns,num_qa)

int   rate[MNSTA+1][3],*ls,*niso_a,*niso_b,*nsta,num_qa[];
float *prop_sp,*tnow,fi[],tns[],*tsim,*trans,pty[5];
struct  station sta[];
struct  time  slot[];
char  *argv[];

{  int   slowest[3],qa_left[3],pos,last,count,uni(),
         current,num_na[3],i,l,k,j,diff,code,st,bs;
   float miat,tnev,dif;
   list temp, push(),pop();

   /* Initialize the event set */

evs = makelist();

  /* Initialize the Stations to their defult values.  This process
  includes setting the values of counters, queues, and arrival rates
  to zero.  */

for(i=1; i<= MNSTA ; i++) {

  sta[i].dis_origin = 0;
  sta[i].dis_dest = 0;
  sta[i].origin = 2*MNSTA;
  sta[i].destination = 2*MNSTA;
  sta[i].avg_mess = 0;

  for(j=1; j <= 2; j++) {

     for(l=1; l <=4; l++){
        for (k=1; k<= MaxQSize; k++){
           sta[i].queue[k][j][l] = 0;
           sta[i].redn[k][j][l] = 0;}
        sta[i].req_cntr[j][l] = 0;
        sta[i].cd_cntr[j][l] = 0;
        sta[i].req_q[j][l] = 0;
        sta[i].niq[j][l] = 0;
        sta[i].nrdn[j][l] = 0;
        sta[i].tdelay[j][l] = 0;
```

```
        sta[i].tadelay[j][1] = 0;
        sta[i].ncus[j][1] = 0;
        sta[i].ndiscard[j][1] = 0;
        sta[i].narrive[j][1] = 0.0;
        sta[i].state[j][1] = 0;
        sta[i].miat[j][1] = 0;
        sta[i].arcount[j][1] = 0.0;
        sta[i].depcount[j][1] = 0.0;
        sta[i].bitsdep[j][1] = 0.0;
      }
    }

}

  /* Initialize the frame on each bus to zero which is the code
     used for NA slots.  Also set the index to point at the next
     slot to be generated to zero.  */

for(j=1; j <= 2; j++) {
   for(i=1; i<= MxFrSize; i++)
     slot[j].frame[i]=0;


    slot[j].index = 0;
}
slowest[1] = slowest[2] = 1000000000;

makedata(argv,pty);

/* call subroutine readd to initialize the staions to their user_specified
/* values. */

readd(tsim,trans,sta,ls,niso_a,niso_b,nsta,prop_sp,rate);

/* if there is isochronous service on only one bus (case of having a     */
/* station that broadcasts) the frame size is set to its maximum and     */
/* the frame on the opposite bus is dedicated to the QA slots.  However,*/
/* if neither buses  carry isochronous traffic, set their frame size     */
/* to its maximum and assign all the slots to QA slots by assigning one */
/* to all frame elements.                                                */


if(*niso_a != 0){
  j=1;
  if(*niso_b == 0){
    tns[2] = MxFrSize;
    for(i=0; i<=MxFrSize; i++)
      slot[2].frame[i] = 1;
  }
}
else if(*niso_b != 0){
  j = 2;
  tns[1] = MxFrSize;
  for(i=1; i<=MxFrSize; i++)
   slot[1].frame[i] = 1;
}
else {
 tns[1] = tns[2] = MxFrSize;
 for(i=0; i<=MxFrSize; i++){
   for(j=1; j <=2; j++)
```

```
    slot[j].frame[i] = 1;
 }
 goto step2;
}

/* for the bus that carries isochronous traffic, calculate the frame
   interval by deviding the payload size (552 bits) by the slowest
   isochronous rate requested.  Also, calculate the total number of
   slots.                                                          */


while(j < 3) {
 if(j == 1)
  count = *niso_a;
 else
  count = *niso_b;
/*   Find the slowest ISO user  */

  for(i=1; i<= count; i++) {
    if(rate[i][j] < slowest[j])
       slowest[j] = rate[i][j];
  }

/* Calculate Frame Interval and Total Number of Slots */

fi[j] = (float) payload /(float) (slowest[j]*1000);
tns[j] = (((float)1000* *ls)*fi[j])/(float) slot_size;

j++;
if(*niso_b == 0)
  j++;

}

/* for the bus that carries isochronous traffic Calculate
   the number of NA slots in the frame.  Note that the number
   of NA slots is rounded to the nearest integer less than the number.
   This number is calculated by first deviding the rate of each iso. user
   by the slowest iso.  rate (this operation yields the number of NA slots
   required by that user) and then summing these values for all users.  */

num_na[1] = num_na[2] =0;

if(*niso_a != 0)
  j = 1;
else
  j = 2;
while( j < 3) {

if(j==1)
 count = *niso_a;
else
 count = *niso_b;

 for(i=1; i<= count; i++) {

  dif = ((float) rate[i][j]/(float)slowest[j]) -
        (int) (rate[i][j]/slowest[j]);
  if(dif != 0)
    num_na[j] += (int) (rate[i][j]/slowest[j] + 1);
```

```
  else
    num_na[j] += rate[i][j]/slowest[j];
 }
j++;
if(*niso_b == 0)
  j++;
}

/* check to see if the number of NA slots on each bus exceeds or is
   equal to the total number of slots in the frame.  If either condition
   is true print an error message and exit the simulation.  Otherwise,
   calculate the number of QA slots in the frame by subtracting the number
   of the NA slots from the total number of slots in the frame          */


if(*niso_a != 0)
  j = 1;
else
  j =2;
while(j < 3) {

  last  = 1;
  current = 2;
  if(num_na[j] > (int) tns[j]){
    printf("Too Many ISO users on bus %d\n",j);
    exit();}
  else if(num_na[j] == (int) tns[j]){
    printf("No Room For ATSUs on bus %d\n",j);
    exit();}
  else
    num_qa[j] = (int) tns[j] - num_na[j];

/* Generate a frame in the following way.  Assign the first element
   to an NA slot.  For each NA slot that is assigned decrement the
   value of qa_left by one.  Use a uniform randon number generator
   (module uni() ) to generate random numbers between 1 and the total
   number of slots.  If the generated number has already been assigned
   to an NA slot generate another randon number and repeat the process  */

  qa_left[j] = num_qa[j];
  slot[j].frame[1] = 1;
  qa_left[j] --;
  while(qa_left[j] != 0) {

    l = (int) tns[j];
    pos = uni(1,l);
    if(slot[j].frame[pos] == 0){
      slot[j].frame[pos] = 1;
      qa_left[j]--;
    }
  }
j++;
if(*niso_b == 0)
  j++;
}

step2:
;

/* Scedule the first frame generation event and initialize the simulation  */
```

```
/* time to zero.  Scheduling event is accomplished by calling the function
   push() (to be explained later).                                          */

code =20;
tnev = 0.0;
*tnow = 0.0;
st =0;
bs = bus_a;
slot[1].index = slot[2].index = 1;

temp = push(evs,tnev,code,st,bs);
evs = temp;

st = MNSTA;
bs = bus_b;

temp = push(evs,tnev,code,st,bs);
evs = temp;

/* if the mean interarrival time of a station at a particular priority     */
/* and on particular bus is not equal to zero, schedule an arrival event. */
/* Note that no arrival event is scheduled for the last */
/* station on each bus.                                                    */

for(l=1; l<=4; l++) {
   for(st=1; st<=*nsta; st++){
      for (bs =1; bs<=2; bs++){
        if(!((bs==bus_a && st==*nsta) || (bs ==bus_b && st==1))){
           if(sta[st].miat[bs][l] != 0.0){
                tnev = expo( sta[st].miat[bs][l]);
                code = 10 + l;
                temp = push(evs,tnev,code,st,bs);
                evs = temp;}
         }
      }
   }
}
return;
}
```

```
/*********************************************************/
/*                                                       */
/*      Function Name:  uni                              */
/*      Return Values:  Integer from a uniform distribu-*/
/*      tion between integer parameters "a" and "b".     */
/*                                                       */
/*      Author:         Farzad Tari                      */
/*                                                       */
/*      Description:                                     */
/*                                                       */
/*      Function to generate an integer value from       */
/*      a uniform distribution between parameters "a"    */
/*      and "b".  The system function drand48() is used */
/*      to generate randon numbers between zero and one.*/
/*      The u(a,b) pdf is built using this function.     */
/*                                                       */
/*********************************************************/

int uni(a,b)

int a,b;

{ double  drand48();
  int     num;

  num = (int) (a + ((double) (b-a) * drand48()));
  return num;
}
```

```
/*********************************************************/
/*                                                       */
/*      Function Name:  expo                             */
/*      Return Value:   Real random number from an       */
/*      exponential distribution with mean of "meam"     */
/*                                                       */
/*      Author:         Farzad Tari                      */
/*                                                       */
/*      Description:                                     */
/*                                                       */
/*      Function to generate exponentially distributed   */
/*      randon numbers using the system provided         */
/*      function drand48().  drand48() generates         */
/*      uniformly distributed randon numbers which are   */
/*      converted to real numbers with exponential pdfs.*/
/*                                                       */
/*********************************************************/

float expo(mean)

float mean;
{
double drand48();
float m;


  m = - mean * log( (float) drand48() );
  return m;
}
```

```
/*********************************************************/
/*                                                       */
/*      Function Name:  makelist                         */
/*      Return Value:   Pointer to the end f of a linked*/
/*      list.                                            */
/*                                                       */
/*      Author:         Farzad Tari                      */
/*                                                       */
/*      Description:                                     */
/*                                                       */
/*      Fuction to initialize the fields of the event    */
/*      calender.                                        */
/*                                                       */
/*********************************************************/

list makelist()
{
list temp = (list) malloc(sizeof(struct calender));
temp->evt_time = -1.0;
temp->evt_code = -1;
temp->node = -1;
temp->bus = -1;
temp->next = NULL;
return temp;
}
```

```
/*********************************************************/
/*                                                       */
/*      Function Name:  push                             */
/*      Return Value:   Pointer to the end of the linked*/
/*      list after pushing the new item into the list.   */
/*                                                       */
/*      Author:         Farzad Tari                      */
/*                                                       */
/*      Description:                                     */
/*                                                       */
/*      Function to insert a new event into the event    */
/*      calender.  The new element is inserted into the  */
/*      in the increasing order of the event time.       */
/*                                                       */
/*********************************************************/


list push(set,t,cd,s,b)

int  cd,s,b;
float t;
list set;

{list current, back, temp;

/* If this is the first element in the list, insert it  */
/* at the head of the list.                             */

 if((set->next == NULL) && (set->evt_time == -1.0)){
     set->evt_time = t;
     set->evt_code = cd;
     set->node = s;
     set->bus = b;
     return set;
  }

/* if the new element needs to be inserted at the head of the list
   do the followings.                                              */

  else if(t < set->evt_time){
     temp = (list) malloc(sizeof(struct calender));
     temp->evt_time = t;
     temp->evt_code = cd;
     temp->node = s;
     temp->bus = b;
     temp->next = set;
     return temp;}
/* search through the event set and find the place where the new
   element needs to be inserted.  Then, insert the new element.  */

  else{
     back = set;
     current = back->next;
     while(  (t >= current->evt_time) && (current != NULL)){
           back = current;
           current = back->next;
     }
     temp = (list) malloc(sizeof(struct calender));
     temp->evt_time = t;
     temp->evt_code = cd;
```

```
        temp->node = s;
        temp->bus = b;
        temp->next = current;
        back->next = temp;
        return set;
    }
}
```

```
/*********************************************************/
/*                                                       */
/*      Function Name:  pop                              */
/*      Return Value:   Pointer to the head of a linked */
/*      list after the first element in the list is      */
/*      removed.                                         */
/*                                                       */
/*      Author: Farzad Tari                              */
/*                                                       */
/*      Description:                                     */
/*                                                       */
/*      Function to remove the fist event in the event  */
/*      calender and return the new pointer to the first*/
/*      element in the event set.                        */
/*                                                       */
/*********************************************************/

list  pop(set)
list set;

{
list temp;
temp = set->next;

/* if there is more than one element in the list, free the
   memory location of the first element in the list     */

if(temp != NULL){
    free(set);
    return temp;
}

/* if the element to be removed is the only link left in the list,
   initialize the link the same way as in function makelist     */

else{
    set->evt_time = -1.0;
    set->evt_code = -1;
    set->node = -1;
    set->bus = -1;
    return set;
}
}
```

```
/*********************************************************/
/*                                                       */
/*      Module Name:    makedata                         */
/*      Return Values:  none                             */
/*      Author:         Farzad Tari                      */
/*                                                       */
/*      Description:                                     */
/*                                                       */
/*      This module creates a data file which is used by*/
/*      module readd to initialize the stations          */
/*      parameters and the simulation parameters to      */
/*      their user-specified values. The load assignment*/
/*      is done under the assumption of uniform          */
/*      distribution of destination addresses.           */
/*                                                       */
/*      To be able to use this program as a general      */
/*      purpose simulator of the DQDB protocol, this     */
/*      module can be replaced by a general purpose      */
/*      user interface.                                  */
/*                                                       */
/*********************************************************/
void makedata(argv,pty)
char  *argv[];
float pty[5];
{
  FILE  *dt;
  FILE  *in;
  float  miat_a,miat_b,ps,trans,avmess,dist,tsim,miati[5],const,miat2[5];
  int    p,nsta,j,ls,nisoa, nisob, rate,typ,ndat,ctr,k1,k2;

  in = fopen(argv[1],"r");
  fscanf(in,"%s",datafile);
  fscanf(in,"%s",outfile);

  dt = fopen(datafile,"w");

  fscanf(in,"%f",&tsim);
  fprintf(dt,"%f\n",tsim);
  fscanf(in,"%f",&trans);
  fprintf(dt,"%f\n",trans);
  fscanf(in,"%f",&ps);
  fprintf(dt,"%e\n",ps);
  fscanf(in,"%d",&ls);
  fprintf(dt,"%d\n",ls);
  fscanf(in,"%d",&nisoa);
  fprintf(dt,"%d\n",nisoa);
  fscanf(in,"%d",&nisob);
  fprintf(dt,"%d\n",nisob);
  if(nisoa != 0){
     for(j=1; j <= nisoa; j++){
         fscanf(in,"%d",&rate);
         fprintf(dt,"%d\n",rate);}
  }
  if(nisob != 0){
     for(j=1; j <= nisob; j++){
          fscanf(in,"%d",&rate);
          fprintf(dt,"d\n", rate);}
  }
  fscanf(in,"%d",&nsta);
  fprintf(dt,"%d\n",nsta);
```

```
   fscanf(in,"%f",&dist);
   fscanf(in,"%f",&miat_a);
   fscanf(in,"%f",&miat_b);

   ctr = 0;

   for(j=1; j <= 4; j++)
      fscanf(in,"%f",&pty[j]);
   fscanf(in,"%f",&avmess);
   for(j=1; j<=nsta; j++){
    fprintf(dt,"%d\n",j-1);
    fprintf(dt,"%d\n",j+1);
    if(j==1)
     fprintf(dt,"0\n");
    else
     fprintf(dt,"%f\n",dist);

    if(j==nsta)
      fprintf(dt,"0\n");
    else
      fprintf(dt,"%f\n",dist);

          ctr++;
          for(p=1; p<=4; p++){
           if(pty[p] != 0.0){
             const = 1.0 / pty[p];
          kl = ctr -1;
            if(kl != 0)
             miat2[p] = const * (float)(nsta-1)/(float)(ctr-1) * miat_a;
            else
             miat2[p] = 0.0;

          k2 = nsta-ctr;
           if(k2 != 0)
             miati[p] = const * (float)(nsta-1)/(float)(nsta-ctr) * miat_a;
           else
             miati[p] = 0.0;}

           else{
             miat2[p] = 0.0;
             miati[p] = 0.0;}
           }


   for(p=1; p<=4; p++)
       fprintf(dt,"%e\n",miati[p]);
   for(p=1;p<=4; p++)
       fprintf(dt,"%f\n",miat2[p]);
   fprintf(dt,"0\n");
   fprintf(dt,"%f\n",avmess);
   fprintf(dt,"1\n");
 }
   fclose(in);
   fclose(dt);
}
```

```
/*********************************************************/
/*                                                       */
/*      Module Name:    readd                            */
/*      Return Value:   none                             */
/*      Author:         Farzad Tari                      */
/*                                                       */
/*      Description:                                     */
/*                                                       */
/*      This module initializes the node parameters      */
/*      and the simulation parameters to the user-       */
/*      specified values.  The parameters are read from  */
/*      the simulation datafile whose name is provided   */
/*      to the module.                                   */
/*                                                       */
/*********************************************************/
void readd(tsim,trans,sta,ls,niso_a,niso_b,nsta,prop_sp,rate)

struct station sta[];
float  *prop_sp,*tsim,*trans;
int    *ls,*niso_a,*niso_b,*nsta,rate[MNSTA][3];
{
int     i,j,count;

FILE *dat;
dat = fopen(datafile,"r");
fscanf(dat,"%10f",tsim);
printf("Simulation Time = %f seconds\n",*tsim);
fscanf(dat,"%10f",trans);
printf(" % transient time = %f\n",*trans);
fscanf(dat,"%e",prop_sp);
printf("prop. speed =%e\n",*prop_sp);
fscanf(dat,"%10d",ls);
printf("link rate in kbits/sec =%d\n",*ls);
fscanf(dat,"%10d",niso_a);
printf("Num of ISO users on bus A =%d\n",*niso_a);
fscanf(dat,"%10d",niso_b);
printf("Num of ISO users on bus B =%d\n",*niso_b);

count = *niso_a;
for(i=1; i<=count; i++){
  fscanf(dat,"%10d",&rate[i][1]);
  printf("bus A,rate for user%d =%d\n",i,rate[i][1]);
}


count = *niso_b;
for(i=1; i<=count; i++){
  fscanf(dat,"%10d",&rate[i][2]);
  printf("bus B,rate for user %d = %d\n",i,rate[i][2]);
}

fscanf(dat,"%10d",nsta);
printf("Num of Sta's =%d\n",*nsta);
for(i=1; i<=*nsta; i++){
  fscanf(dat,"%10d",&sta[i].origin);
  printf("sta %d origin %d\n",i,sta[i].origin);
  fscanf(dat,"%10d",&sta[i].destination);
  printf("sta %d destination %d\n",i,sta[i].destination);
  fscanf(dat,"%10f",&sta[i].dis_origin);
  printf("sta %d dis to orig %f\n",i,sta[i].dis_origin);
```

```
   fscanf(dat,"%10f",&sta[i].dis_dest);
   printf("sta %d dis to dest %f\n",i,sta[i].dis_dest);

   for(j=1; j <=4; j++){
      fscanf(dat,"%e",&sta[i].miat[1][j]);
      printf("sta %d miat @ pri %d = %e\n",i,j,sta[i].miat[1][j]);}
   for(j=1; j<=4; j++){
      fscanf(dat,"%10f",&sta[i].miat[2][j]);
      printf("sta %d miat @ pri %d = %f\n",i,j,sta[i].miat[2][j]);}

   fscanf(dat,"%d",&sta[i].arr_dist);
   printf("sta %d arrival dist is ",i);
   if(sta[i].arr_dist == Expon)
      printf("Exponential\n");
   else if (sta[i].arr_dist == Constant)
      printf("Constant\n");
   else
      printf("Undefined\n");
   fscanf(dat,"%f",&sta[i].avg_mess);
   printf("sta %d avg mess length = %f\n",i,sta[i].avg_mess);
   fscanf(dat,"%d",&sta[i].mess_len_dist);
   printf("sta %d message length distribution is ",i);
   if(sta[i].mess_len_dist == Expon)
      printf("Exponential\n");
   else if (sta[i].mess_len_dist == Constant)
      printf("Constant\n");
   else
      printf("Undefined\n");


}
fclose(dat);
return ;
}
```

```
/*****************************************************************/
/*                                                               */
/*      Module Name:    fram_gen                                 */
/*      Return Value:   None                                     */
/*      Author:         Farzad Tari                              */
/*                                                               */
/*      Description:                                             */
/*                                                               */
/*      This modules models the operation of the head of the     */
/*      bus slot generator.  Slots are generated with constant   */
/*      interarrival time of one slot time which is the time     */
/*      required to transmit one  slot.  This modules refers to  */
/*      the frame format on the bus to determine the type of the */
/*      next slot.  The codes for  QA and NA slots are one and   */
/*      zero, respectively.                                      */
/*                                                               */
/*****************************************************************/

void fram_gen(slot,tns,tnow,nsta,prop_sp,ls)

int nsta,ls;
float tns[],tnow,prop_sp;
struct time slot[];

{ int b,nd,l,cd,ind;
  float  tnev,slot_time;
  list temp;

/* Extract the bus number and node number fron the first event
   in the event calender.  Then, remove the first event from the
   calender by calling the function pop().                      */

  b = evs->bus;
  nd = evs->node;
  temp = pop(evs);
  evs = temp;

/* Calculate the slot time by deviding the QA slot size (512 bits)
   by the line rate of one bus which os half of the total capacity. */

  slot_time = (float) slot_size/(float) (1000*ls);

/* Schedule the next fram_gen event by calling the function push */

  ind = slot[b].index;
  tnev = tnow +  slot_time;

  cd = 20;
  temp = push(evs,tnev,cd,nd,b);
  evs = temp;

  /* Schedule the next slot arrival at the first station on the bus */

  if(b == bus_a){
   nd = 1;}
  else if(b==bus_b){
   nd = nsta;}

/* if the frame index is pointing to zero generate an NA slot by
   setting the code to 400000; otherwise, set the code to 300000 to
```

```
   generate an empty  QA slot.                                    */

  if(slot[b].frame[ind] == 0)
   cd = 400000;
  else
   cd = 300000;

  tnev = tnow;

  temp = push(evs,tnev,cd,nd,b);
  evs = temp;

/* if the frame is pointing to the last item in the frame, set the
   index to one.  This initializes the index to the beginning of the
   frame and causes the same pattern of QA and NA slots to be generated
   again. If the index is not pointing to the end of the frame, increment
   the index to the next slot in the frmae.                       */

  if(slot[b].index == (int)tns[b])
    slot[b].index = 1;
  else
    slot[b].index ++;

  return;
}
```

```
/*************************************************************************/
/*                                                                       */
/*      Module Name:    slot_arrival                                     */
/*      Return Value:   None                                             */
/*      Author:         Farzad Tari                                      */
/*                                                                       */
/*      Description:                                                     */
/*                                                                       */
/*      This modules models the operation associated with arrival of     */
/*      a slot on either of the buses.  These operations where extracted*/
/*      from the state diagram of the DQDB protocol.                     */
/*                                                                       */
/*************************************************************************/

void slot_arrival (sta,tnow,tsim,trans,prop_sp,ls)

int  ls;
float tnow,tsim,trans,prop_sp;
struct station sta[];

{ int b, nd, cd,l,busy,req[5],opp,code,nod,na,by,i;
  float  dist,tnev,slot_time;
  list   temp;

/* Extract the bus number and the node number from the event code
   Then, remove the first event (i.e. the current event) from the
   event calender.                                                  */

b = evs->bus;
nd = evs->node;
cd = evs->evt_code;
temp = pop(evs);
evs = temp;

/* set the by variable to zero.  If anywhere inside this module the
   conditions call for the tramsmission of a QA slot, this bit is
   set to one.  At the end, when the slot is assembled the value of this
   bit and the busy bit in the slot are both checked.  If either value
   is equal to one, the Busy bit in the outgoing slot is set to one.    */


by = 0;


cd -= 300000;
na = 0;

for(i=1; i<=4; i++)
  req[i] = 0;

/* Determine the type of the slot.  If this is an NA slot set the
   variable na to one; otherwise, it maintains the value of zero.  */

if(cd > 14321){
  na = 1;
  cd -= 100000;}

/* Extract the value of the Busy bit from the slot and assign it
   to the variable busy.                                            */
```

```
if(cd > 4321){
  cd -= 10000;
  busy = 1;}
else
  busy = 0;

/* Extract the values of the request bits for each priority */

if (cd > 321){
   req[4] = 4;
   cd -= 4000;}

if(cd > 21){
   req[3] = 3;
   cd -= 300;}

if(cd > 1){
   req[2] = 2;
   cd -= 20;}

if(cd == 1){
   req[1] = 1;
   cd -= 1;}

/* Determine the opposite bus */

if(b==1)
  opp = 2;
else
  opp = 1;

/* Perform the protocols opertation specified by the systems state
   diagram for:

   A: DQSM's on the same bus. (The bus on which the QA slot arrived) */

/* Note that the "transitions" refered to in the documentation are
   the state diagram transitions described in the documentation of
   the DQDB protocol.                                              */

for(l=1; l<=4; l++){    /* for each prioiryt do the followings */

/* In the idle state, if an empty QA slot is observed on the same bus
   and the request counter at that priority is not equal to zero,
   decrement the request counter at that prioirty.                 */

   if(sta[nd].state[b][l] == idle){ /* IDLE state */

      if (na == 0){ /* Transition 11c */
         if((busy == 0) && (sta[nd].req_cntr[b][l] != 0))
              sta[nd].req_cntr[b][l]--;}
    } /* End IDLE State */


   else if(sta[nd].state[b][l] == cntdn) { /* Countdown state */

/* in the countdown state, if an empty QA slot is observed on the
   same bus and the countdown counter at the priority is not equal
   to zero, decrement its value by one.                            */
```

```
      if(na == 0){ /* Transition 22d */
         if((busy == 0) && (sta[nd].cd_cntr[b][l] != 0))
              sta[nd].cd_cntr[b][l] --;

/* In the countdown state, if an empty QA slot is observed on the same
   bus and the countdown counter at that prioiry is equal to zero
   arrange for the departure of a QA slot in the queue by calling
   the subroutine departure.                                          */

         else if((busy == 0) && (sta[nd].cd_cntr[b][l] == 0)) { /* Trans 21 */
              by = l;

              departure(sta,ls,nd,b,l,tnow,tsim,trans);

         }
      }
    }/* End Countdown State */

   else if(sta[nd].state[b][l] == stdby) { /*Standby State */
      if(na == 0){

/* In the standby state, if an empty QA slot is observed on the same
   bus, arrange for the departure of a QA slot by calling the
   subroutine departure.                                          */

         if (busy == 0) { /* Transition 31 */
            by = l;

            departure(sta,ls,nd,b,l,tnow,tsim,trans);

         }

/* In the standby state, if a busy QA slot arrives, indicate to the RQM
   of the opposite bus that a request needs to be sent.  This is implemented
   by incrementing the RQM counter of the opposite bus. Then, move
   to the countdown state.                                            */
         else{ /* Transition 32a */
           sta[nd].req_q[opp][l]++;
           sta[nd].state[b][l] = cntdn;}
      }
   } /* End Standby state */
} /* End for */

/* B: DQSM's on the opposite bus */

for(i=1; i<=4; i++){
   for(l=1; l<=4; l++){

      if(sta[nd].state[opp][l] == idle) { /* IDLE */

/* in the idle state, if a request arrives at a priority
   greater than or equal to the prioirty of the DQSM, increment
   the request counter at that priority by one.                   */

         if(req[i] >= l) /* Transition 11a */
               sta[nd].req_cntr[opp][l]++;
      } /* End IDLE */

      else if(sta[nd].state[opp][l] == cntdn){ /* Countdown */
```

```
/* in the countdown state, if a request is received which is greater than
   the pririty of the DQSM, increment the countdown counter of the DQSM
   by one.                                                          */
        if(req[i] > l)    /* Transition 22a */
             sta[nd].cd_cntr[opp][l]++;

/* in the countdown state, if a request is received at the same
   prioiry level, increment the request counter of that priority
   by one.                                                          */

        else if(req[i] == l)  /* Transition 22b */
             sta[nd].req_cntr[opp][l]++;
     } /* End Countdown */

     else if(sta[nd].state[opp][l] == stdby) { /* Standby */

/* in the standby state, if a request is received at a priority
   less than the priority of the DQSM, indicate to the RQM associated
   with the bus that a request is to be sent at that priority and
   move to the countdown state.                                     */

         if((req[i] != 0) && (req[i] < l)) {  /* Transition 32b */
             sta[nd].req_q[b][l]++;
             sta[nd].state[opp][l] = cntdn;}

/* in the standby state, if a request is received at a priority greater
   than the priority of the DQSM, increment the countdown counter of the
   DQSM by one, indicate to the RQM associated with the bus that a request
   at that priority is to be sent, and move to the countdown state.     */

         else if(req[i] > l) { /* Transition 32c */
             sta[nd].cd_cntr[opp][l]++;
             sta[nd].req_q[b][l]++;
             sta[nd].state[opp][l] = cntdn;}

/* in the standby state, if a request is received at the same priority
   increment the request counter of the DQSM by one, indicate to the RQM
   associated with the bus that a request is to be sent (i.e. increment
   the RQM counter, req-q, of the associated bus), and move to the
   countdown state.                                                 */

         else if(req[i] == l){ /* Transition 32d */
             sta[nd].req_cntr[opp][l]++;
             sta[nd].req_q[b][l]++;
             sta[nd].state[opp][l] = cntdn;}
     } /* End Standby */
   }
}

/* The RQM Operations */

for(l=1; l<=4; l++){

/* for each priority, if the request bit in the QA slot associated with
   that priority is equal to zero and the RQM counter of that priority
   is not equal to zero, send a request at that priority and decrement
   the RQM counter of that priority (i.e. that DQSM).               */

    if(sta[nd].req_q[b][l] != 0){
        if(req[l] == 0){
```

```
                sta[nd].req_q[b][l]--;
                req[l] = l;}
    }
}


/* Unless the station is the last one on the bus, schedule a "slot arrival"
   at the next station on the bus */

if(!((b == bus_a && nd == nsta) || (b == bus_b && nd == 1))){


/* Assemble the outgoing slot.                              */

  code = 300000;
  if(na == 1)
    code = 400000;

/* if either the Busy bit in the Qa slot or the by variable are
   equal to one set  the Busy bit in the outgoining QA slot to one. */

  if((by == 1) || (busy==1))
    code += 10000;

/* insert the values of the request bits into the outgoing slot.        */

  l=1;
  for(i=1; i<=4; i++){
    code = code + (l * req[i]);
    l *= 10;
  }

/* Schedule the arrival of this slot at the next station on the bus
   by calling function push.  The time of this event would be
   the current simulation time, tnow, plus the propagation time betwen
   the two stations which is equal to the distance between the two
   stations devided by the velocity of the media.                */


  if(b == 1){
    dist = sta[b].dis_dest;
    nod = sta[nd].destination;}
  else if(b == 2) {
    dist = sta[b].dis_origin;
    nod = sta[nd].origin;}

tnev = tnow + (dist/prop_sp);
temp = push(evs,tnev,code,nod,b);
evs = temp;
}

return;

}
```

```
/*****************************************************************/
/*                                                               */
/*      Module Name:    departure                                */
/*      Return Values:  None                                     */
/*      Author:         Farzad Tari                              */
/*                                                               */
/*      Description:                                             */
/*                                                               */
/*      This module is responsible for the operations associated*/
/*      with the departure of a QA slot such as extracting the   */
/*      departing slot from the queue, updating the counters     */
/*      and the state of the DQSMs, and collecting statistics.   */
/*                                                               */
/*****************************************************************/

void departure(sta,ls,nd,b,pp,tnow,tsim,trans)

int nd,b,pp,ls;
float tnow,tsim,trans;
struct station sta[];

{ int i,opp;
  float  delay,slot_time;

slot_time = (float) slot_size / (float)(1000 * ls);

/* if the transient time period is over, start collecting
   statistics.                                                  */

if(tnow >= (tsim * trans)){
  if(sta[nd].queue[1][b][pp] >= (tsim * trans)){

/* Update the count of total MAC delay, number of information
   bits serviced and total number of bits serviced at the station. */

   sta[nd].tadelay[b][pp] += (float) slot_time;
   sta[nd].depcount[b][pp] +=  sta[nd].info_bits[1][b][pp];
   sta[nd].bitsdep[b][pp] += (float) slot_size;}
}

/* The redn array is used to keep track of the number of slots
   associated with each packet.  Each time a slot is transmitted
   the redn counter is decremented by one. The delay associated with
   each packet is calculated when the last slot associated with the
   packet leaves the system.                                    */

 if(sta[nd].redn[1][b][pp] == 1){

  for(i=0; i<= sta[nd].nrdn[b][pp]; i++)
     sta[nd].redn[i][b][pp] = sta[nd].redn[i+1][b][pp];

  sta[nd].nrdn[b][pp] --;
  delay = tnow - sta[nd].queue[1][b][pp];

/* If the transiant time period is over, update the counts of
   total queueing delay, total MAC delay, and number of packets
   serviced at the station                                      */

  if(tnow >= (tsim * trans)){
   if(sta[nd].queue[1][b][pp] >= (tsim * trans)){
```

```
    sta[nd].tdelay[b][pp] += delay;
    sta[nd].tadelay[b][pp] += delay;
    sta[nd].ncus[b][pp] ++;}
   }
 }
 else
  sta[nd].redn[1][b][pp] --;

/* Determine the opposite bus   */

  if(b==1)
    opp = 2;
  else
    opp = 1;

/* Remove the first element from the queue      */

  for(i=0; i<= sta[nd].niq[b][pp]; i++){
    sta[nd].queue[i][b][pp] = sta[nd].queue[i+1][b][pp];
    sta[nd].info_bits[i][b][pp] = sta[nd].info_bits[i+1][b][pp];}

  sta[nd].niq[b][pp] --;
  sta[nd].cd_cntr[b][pp] = 0;

/* Update the counters and the state of the DQSM        */

/* if there is no more packets to be transmitted return to
   the idle state.                                      */

  if(sta[nd].niq[b][pp] == 0)
    sta[nd].state[b][pp] = idle;

/* if queue at this priority is not emty but the request counter
   and the number of packets in the queue at the lowest prioirty
   are both equal to zero, move to the standby state.          */

  else if(sta[nd].req_cntr[b][1] == 0 &&
          sta[nd].niq[b][1] == 0)

    sta[nd].state[b][pp] = stdby;

/* if the queue is not empty and moving to standby state is not
   possible, move to the countdown state, put the value of
   the request counter into the countdown counter, set the value
   of the countdown counter to zero, and indicate to the associated
   RQM that a request is to be sent on the opposite bus (i.e. increment
   the req_q counter of the opposite bus.                      */

  else {
    sta[nd].cd_cntr[b][pp] = sta[nd].req_cntr[b][pp];
    sta[nd].req_cntr[b][pp] = 0;
    sta[nd].req_q[opp][pp] ++;
    sta[nd].state[b][pp] = cntdn;}

  return;

}
```

```
/*******************************************************/
/*                                                     */
/*      Module Name:    aval                           */
/*      Return Value:   none                           */
/*      Author:         Farzad Tari                    */
/*                                                     */
/*      Description:                                   */
/*                                                     */
/*      This module handles the state opertations      */
/*      associated with the arrival of a packet at a   */
/*      DQSM. Counters as well as the states of all    */
/*      DQSMs are updated.                             */
/*                                                     */
/*******************************************************/
void aval(tnow,tsim,trans)

float tnow,tsim,trans;


{int  b,nd,l,opp,nq,i,num_pac,num,cd,pr,dum;
 float dif,m,tnev;
 list temp;

/* Extract from the event calender information about the current event.
   That is, extract the bus number, the node number and the event code.
                                                                    */
b = evs->bus;
nd = evs->node;
pr = evs->evt_code;
pr -= 10;

/* Remove the current event from the event calender.           */

temp = pop(evs);
evs = temp;

/* Determint the opposite bus                    */

if(b==1)
  opp = 2;
else
  opp =1;

/* Generate a message length from the specified message length
   distribution.                                             */

if(sta[nd].mess_len_dist == Expon){
  while((m = expo( sta[nd].avg_mess) ) < 1.0);
  dum = (int) m;
  m = dum;
}
else
  m = sta[nd].avg_mess;

/* if the transient response period is over, collect the nuber of
   packets offered to the MAC layer (narrive) and the number
   of information bits arrived offered to the node.
                                                          */
if(tnow >= (tsim * trans)){
  sta[nd].arcount[b][pr] += m ;
```

```
  sta[nd].narrive[b][pr]++;
}

/* Determint the number of slots required to transmit the packet.  */

dif = m/(float) payload - (int) (m/(float)payload);
if(dif != 0)
  num_pac = (int) (m/(float) payload + 1);
else
  num_pac = m/payload;


/* Schedule the next arrival for the station on the same bus */
/* First determine the arrival distribution and then call the
   push module to schedule the arrival event in the event calender */

cd = 10 + pr;

if(sta[nd].arr_dist == Expon)
  tnev = tnow + expo( sta[nd].miat[b][pr]);
else
  tnev = tnow + sta[nd].miat[b][pr];

temp = push(evs,tnev,cd,nd,b);
evs = temp;

/* if there is not enough buffer space to hold the packet, discard
   the entire packet                                              */

if((sta[nd].niq[b][pr] + num_pac) >= MaxQSize){

   if(tnow >= (tsim*trans))
    sta[nd].ndiscard[b][pr]++;
    goto discard;}

for(l=4; l>=1; l--) {

   if(sta[nd].state[b][l] == idle) {

/* In the idle state, if a packet arrives with a priority higher
   than the priority of the DQSM, increment the request counter of
   the DQSM by one.                                               */

        if(l < pr) /* Transition 11b */
           sta[nd].req_cntr[b][l]++;

/* In idle state, if a packet arrives at the DQSM, examine the
   request counter and the length of the queue at the lowest priority
   DQSM (i.e. priority 0).  If the request counter or the queue length
   at the lowest prioirty is greater than zero, do the followings:

   - Load the value of the request counter to the countdown counter.
   - Set the value of the request counter to zero.
   - Increment the RQM counter (req_q) of the opposite bus by one
     to indicate that a request is to be sent on the opposite bus.
   - Move to the countdown state.

   if the request counter and the queue length of the lowest priority
   are both zero, move to the standby state.
```

```
                                                                    */
        if(l == pr) { /* Transition 12 */

           if((sta[nd].req_cntr[b][l] > 0) || (sta[nd].niq[b][l] > 0)) {
                sta[nd].cd_cntr[b][pr] = sta[nd].req_cntr[b][pr];
                sta[nd].req_cntr[b][pr] = 0;
                sta[nd].req_q[opp][pr]++;
                sta[nd].state[b][pr] = cntdn;}

            else if((sta[nd].req_cntr[b][l] == 0) && (sta[nd].niq[b][l]==0))
                sta[nd].state[b][pr] = stdby;
        }
    } /* End IDLE */

    else if(sta[nd].state[b][l] == cntdn){

/* In the countdown state, if a packet arrives which has larger prioiry
   than the priority of the DQSM, increment the countdown counter.    */

         if(l < pr)     /* Transition 22c */
            sta[nd].cd_cntr[b][l]++;
    }  /* End Countdown */

    else if(sta[nd].state[b][l] == stdby) {

/* In the standby state, if a packet arrives which has priority larger
   than the prioirty of the DQSM, do the followings:

   - Increment the countdown counter.
   - Increment the RQM counter of the opposite bus to indicate that
     a request is to be sent on  the opposite bus.
   - Move to the countdown state.
                                                                    */
          if(l < pr) { /* Transition 32f */
                sta[nd].cd_cntr[b][l]++;
                sta[nd].req_q[opp][l]++;
                sta[nd].state[b][l] = cntdn;}

/* In the standby state, if a packet arrives which has lower priority
   than the priority of the DQSM, increment the RQM counter of the
   opposite bus to indicate that a request is to be sent on that bus and
   move to the countdown state.                                     */

           else if(l > pr) {  /* Transition 32e */
                sta[nd].req_q[opp][l]++;
                sta[nd].state[b][l] = cntdn;}
    } /* End Standby */

}

/* Insert the new packet in the queue                  */

for(i=1; i<=num_pac; i++){
  sta[nd].niq[b][pr] ++;
  nq = sta[nd].niq[b][pr];
  sta[nd].queue[nq][b][pr] = tnow;

/* Update the count of the total information bits offered to the
   node.                                                            */
```

```
  if(dif == 0)
     sta[nd].info_bits[nq][b][pr] = payload;
  else if((dif != 0.0) && (i<= (num_pac - 1)))
     sta[nd].info_bits[nq][b][pr] = payload;
  else if((dif != 0.0) && (i == num_pac))
     sta[nd].info_bits[nq][b][pr] = dif * (float) payload;

}

/* Record the number of slots in the queue that are associated with
   the new packet.                                                  */

sta[nd].nrdn[b][pr] ++;
num = sta[nd].nrdn[b][pr];
sta[nd].redn[num][b][pr] = num_pac;

discard:
;
return;

}
```

```
/*****************************************************************/
/*                                                               */
/*      Module Name:    report                                   */
/*      Return value:   none                                     */
/*      Author:         Farzad Tari                              */
/*                                                               */
/*      Description:                                             */
/*                                                               */
/*      This module generates the simulation statistics based    */
/*      on the statistics collocted throughout the simulation.   */
/*      The statistics are devided into 'per node' and 'global'  */
/*      The per-node statistics pertain to the characteristics   */
/*      of the particular node and are important especially in   */
/*      non-homogeneous networks.  The global statistics reflect*/
/*      the overall characteristic of the network.  The          */
/*      statistics are futher devided into average and total.    */
/*      Total measures report the sum oof measurements during    */
/*      the simulation.  Average measurements are averaged over  */
/*      the number of measurements taken.                        */
/*                                                               */
/*****************************************************************/
void report(sta,nsta,ls,tsim,trans,tns,num_qa)

int  nsta,ls,num_qa[3];
struct station sta[];
float tsim,trans,tns[3];

{ int    i,j,p,num_pac,ndat,cp[51],found,used,ndiscard,npack;
  float  avg,avg2,slot_time,th,cus,delay,da,db,da2,db2,la,lb,count[5],
         count2[5],laststa,miat,tia,tib,thi,tiap,tibp,thp,num[5]
         ,dif,rt,tput[5],tputp[5],load[5],time,ninfo,noff,ninoff
         ,tia2,tib2,tput2[5],thi2,th2,total,tdep,tarr,mt,ld,tth;
  FILE *fp;

  fp = fopen(outfile,"w");

/* Determine the slot time by deviding the slot size (552 bits) by
   the line rate of one bus.                                     */

  slot_time = (float) slot_size/ (float) (ls*1000);
  time = (1.0 - trans) * tsim;

  total = 0.0;

/* In the following block, calculate and report the per-node statistics */

  fprintf(fp,"   Per Node Statistics \n");
  fprintf(fp,"-------------------------\n");
  fprintf(fp,"\n");

/* Initialize the counters */

    for(p=1; p<=4;p++){
      num[p] = 0.0;
      tput[p] = 0.0;
      tputp[p] = 0.0;
      count2[p] = 0.0;
      count[p] = 0;}

  for (i=1; i<=nsta; i++) {
```

```
   fprintf(fp,"Station %d\n",i);


   for(p=1; p<=4; p++){

    avg = th = th2 = avg2 = thp = 0.0;

    fprintf(fp,"\n");
    fprintf(fp,"priority %d\n",p);
    fprintf(fp,"\n");

     da = db = 0.0;
     tia = tib = 0.0;
     tia2 = tib2 = 0.0;

/* For the DQSMs on Bus A, calculate the average queueing delay
   and average total MAC delay.  Average delays are calculated
   by deviding the total delay by the total number of packets serviced.
                                                                    */
        if(sta[i].ncus[1][p] != 0){
          da = sta[i].tdelay[1][p] / (float) sta[i].ncus[1][p];
          da2 = sta[i].tadelay[1][p] / (float) sta[i].ncus[1][p];}
        else
          da = da2 = 0;

          tia = sta[i].depcount[1][p];
          tia2 = sta[i].bitsdep[1][p];

        if(sta[i].arcount[1][p] != 0)
          tiap = sta[i].depcount[1][p] / sta[i].arcount[1][p];

/* For the DQSMs on Bus B, calculate the average queueing delay
   and average total MAC delay.  Average delays are calculated
   by deviding the total delay by the total number of packets serviced.
                                                                    */

       if(sta[i].ncus[2][p] != 0){
          db = sta[i].tdelay[2][p] /
               (float) sta[i].ncus[2][p];
          db2 = sta[i].tadelay[2][p] /
               (float) sta[i].ncus[2][p];}
        else
          db = db2 =  0;

          tib = sta[i].depcount[2][p];
          tib2 = sta[i].bitsdep[2][p];

          if(sta[i].arcount[2][p] != 0)
             tibp = sta[i].depcount[2][p] /
                    sta[i].arcount[2][p];

/* Calculate the total mean interarrival time of the station    */

       ld = 0.0;
       if (sta[i].miat[1][p] != 0)
          ld += 1.0/sta[i].miat[1][p];
       if (sta[i].miat[2][p] != 0)
           ld += 1.0/sta[i].miat[2][p];
       if(ld != 0)
```

```
         mt = 1.0/ld;
      else
         mt = 0.0;

/* Determine the ratio of the arrival rate on bus A to the
   total arrival rate at the station and store it in 'la'       */

      if(sta[i].miat[1][p] != 0)
        la = mt / sta[i].miat[1][p];
      else la = 0;


/* Determine the ratio of the arrival rate on bus B to the
   total arrival rate at the station and store it in 'lb'       */

      if(sta[i].miat[2][p] != 0)
        lb = mt / sta[i].miat[2][p];
      else lb = 0;

/* Calculate the average delays by multiplying the average delay of
   each bus (say bus A) by the ratio of arrival rate at the bus to
   the total arrival rate at the station (la).  Then, for each bus
   add the two products.  This is the weighted sum of the delay figures
   of each bus where the weighting coefficients are 'la' and 'lb'  */

      avg = (la * da) + (lb * db);
      avg2 = (la * da2) + (lb * db2);
      thp = (la * tiap) + (lb * tibp);

/* Add the troughput coefficients of both buses */
      th2 = tia2 + tib2;
      th =  tia + tib;

/* Update the average delay and total throughput counts which
   will be used in calculation of Global Statistics     */

     count[p] += avg;
     count2[p] += avg2;
     tput[p] += thp;
     tput2[p] += th2;
     tput[p] += th;


/* Calculate the following statistics by adding the statistics of
   both buses.  ncus, ndiscard, ninfo, noff, and ninoff are
   total number of packets serviced at the station, total number
   of packets discarded at the station, total number of information
   bits serviced at the station, number of packets offered to the
   station and number of information bits offered to the station,
   respectively.                                                 */

     npack = sta[i].ncus[1][p] + sta[i].ncus[2][p];
     ndiscard = sta[i].ndiscard[1][p] + sta[i].ndiscard[2][p];
     ninfo = sta[i].depcount[1][p] + sta[i].depcount[2][p];
     noff = sta[i].narrive[1][p] + sta[i].narrive[2][p];
     ninoff = sta[i].arcount[1][p] + sta[i].arcount[2][p];

/* Calculate the total MAC throughput by deviding the ratio of
   the time spent by the station to service information bits to
   the effective simulation time.  Effective simulation time is
```

```
   the total simulation time minus the transient period.  The time
   spent on serving information bits is calculated by deviding the
   total number of the information bits by the line rate.       */

     thi = th /(time * (float)(2 * 1000 * ls));
     thi2 = th2/(time * (float)(2 * 1000 * ls));

/* Print the per-node statistics to the simulation output file  */

     fprintf(fp,"Avg  Total MAC Queueing  Delay\t\t%e seconds\n",avg);
     fprintf(fp,"Avg  Total MAC Delay\t\t\t%e seconds\n",avg2);
     fprintf(fp,"Total Packets Serviced by MAC\t\t%d\n"
                ,npack);
     fprintf(fp,"Total Info bits Serviced by MAC\t\t%f\n"
           ,ninfo);
     fprintf(fp,"Total Packets Discarded by MAC\t\t%d\n",ndiscard);
     fprintf(fp,"Total Packets Offered to MAC\t\t%e\n",noff);
     fprintf(fp,"Total info bits Offered to MAC\t\t%f\n",ninoff);
     fprintf(fp,"Total Media Throughput\t\t\t%f\n",thi);
     fprintf(fp,"Throughput Including Overhead\t\t%f\n",thi2);
     fprintf(fp,"% Correct Transmissions\t\t\t%f\n",thp);

  }


    fprintf(fp,"\n");
    fprintf(fp,"--------------------\n");
  }

/* Count the number of stations with a certain priority and store
   the count in the array num[].                                */

for(p=1; p<=4; p++){
   for(j=1; j<=nsta; j++){
      if((sta[j].miat[1][p] != 0) || (sta[j].miat[2][p] != 0))
        num[p]++;
   }
}

/* In the following block, calculate and report the global statistics */

  fprintf(fp,"\n");
  fprintf(fp,"      Global Statistics\n");
  fprintf(fp,"-------------------------------\n");

  fprintf(fp,"\n");

  fprintf(fp,"\n");
  fprintf(fp,"\n");
  fprintf(fp,"Average  Total Queueing Delay\n");
  fprintf(fp,"\n");

 /* For each priotity, calculate the average queueing delay by
    deviding the total queueing delay of all stations at that priority
    by the number of stations with that prioirty.               */

  for(p=1; p<=4; p++){

     if(num[p] != 0)
         avg =  (count[p] / num[p]);
```

```
    else
        avg = 0.0;

  fprintf(fp,"queueing delay @ pri %d = %e seconds\n",p,avg);
  }


  fprintf(fp,"\n");
  fprintf(fp,"Average Total MAC Delay\n");
  fprintf(fp,"\n");

/* For each priotity, calculate the average total MAC delay
   by deviding the total MAC delay of all stations at that priority
    by the number of stations with that prioirty.                */

  for(p=1; p<=4; p++) {

        if(num[p] != 0)
          avg2 = (count2[p] / num[p]);
        else
          avg2 = 0.0;

     fprintf(fp,"total MAC delay @ pri %d = %e\n",p,avg2);
   }



  fprintf(fp,"\n");
  fprintf(fp,"\n");
  fprintf(fp,"Total Media  Throughput\n");
  fprintf(fp,"\n");

/* For each priority, calculate the total media throughput by deviding
   the ratio of the time spent to servive information bits of that
   priority to the effective simulation time.  Effective simulation
   time is the total simulation time minus the transient period.  The
   time spent on transmitting informatiom bits is calculated by deviding
   the total number of information bits serviced at that proirity by the
   line rate.                                                    */

/* Note that the variable 'total' is used to accumulate the total
   media throughput of all priorities and isochronous services.
   This sum which is reffered to as the network utilization does not
   include any protocol overhead.                                */

  for(p=1; p<=4; p++){

      thi = tput[p] / (time * (float) ( 2 * 1000 * ls));

    fprintf(fp,"Throughput @ pri %d = %f\n",p,thi);

    total += thi;
  }


/* Calculate Global Throughput Including Overhead.  The procedure
   is the same as in total media throughput except that instead of
   the number of information bits, the total number of bits (i.e.
   information bits + overhead) is used.
                                                                 */
```

```
fprintf(fp,"\n");
fprintf(fp,"\n");
fprintf(fp,"Total MAC Throughput including Overhead\n");
fprintf(fp,"\n");

  for(p=1; p<=4; p++){

        thi2 = tput2[p] / (time * (float) ( 2 * 1000 * ls));

     fprintf(fp,"Throughput @ pri %d = %f\n",p,thi2);

  }


/* Calculate the average number of packets per arrrival */

  dif = sta[1].avg_mess/(float) payload -
        (int) ((int)sta[1].avg_mess/(float)payload);
  if(dif !=  0)
       num_pac = (int) (sta[1].avg_mess/(float)payload + 1);
  else
       num_pac = sta[1].avg_mess/payload;



  fprintf(fp,"\n");
  fprintf(fp,"\n");
  fprintf(fp,"Avg Total Offered Load including Overhead\n");
  fprintf(fp,"\n");

/*  Calculate the total offered load by first finding the total
    number of bits per seconds offered to the MAC and then deviding
    that by the line rate (i.e. load = arrival rate / departure rate)
                                                                    */
     for(p=1; p<=4; p++){
        th = ld = tth  =  0.0;
        for(j=1; j<= nsta; j++){
               if(sta[j].miat[1][p] != 0)
                  ld += 1.0/sta[j].miat[1][p];
               if(sta[j].miat[2][p] != 0)
                  ld += 1.0/sta[j].miat[2][p];
         }
               th = ld * ((float) (num_pac * slot_size));
               th /= (float)(2 * 1000 * ls);


        load[p] = th;
        fprintf(fp,"Offred Load @ pri %d = %f \n",p,th);

     fprintf(fp,"\n");
  }


/* Calculate the offered isochronous load by deviding the number
   of the NA slots in the frame by the total number of slots in the
   frame.  To eliminate the protocol overhead, this result is then
   multiplied by the ratio of the payload size (512 bits) to the
   slot size (552 bits).
```

```
   Note that the parameter 'total( i.e. network utilization) is updated
   to include the isochronous load serviced by the network.
                                                                    */

  if(num_qa[1] != 0) {
       rt = (float) ((int) tns[1] - num_qa[1])/(float)((int) tns[1]);
       rt *= (float) payload / (float) slot_size;
       }
  else
       rt = 0.0;

  total += rt;

  fprintf(fp,"Offered Isochrounous Load =  %f \n",rt);
  fprintf(fp,"\n");


/* Print the total network utilization accumulated earlier to the
   simulation output file.  Network utilization is defined as the sum
   of total media throughput for all priorities and isochronous services
                                                                    */
  fprintf(fp,"\n");
  fprintf(fp,"Total Network Utilization = \t %f\n",total);
  fprintf(fp,"\n");

  fprintf(fp,"\n");
  fprintf(fp," Total Packets Offered to MAC\n");
  fprintf(fp,"\n");

/* For each priority, sum the number of packets offered to the MAC
   and report the result to the simulation output file          */

  for(p=1; p<=4; p++){

   tarr = 0.0;

   for(j=1; j<=nsta; j++){
    for(i=1; i<=2; i++)
      tarr += sta[j].narrive[i][p];

   }
   fprintf(fp,"Number of Packets Offered to MAC @ pri %d = %e\n"
          ,p,tarr);
  }
  fprintf(fp,"\n");



  fprintf(fp," Total Packets Discarded by MAC\n");
  fprintf(fp,"\n");

  for(p=1; p<=4; p++){

    tdep = 0.0;

    for(j=1; j<=nsta; j++) {
     for(i=1; i<=2; i++)
      tdep += sta[j].ndiscard[i][p];
```

```
    }
    fprintf(fp,"Number of Packets Discarded @ pri %d = %e\n",
          p,tdep);
  }

  fclose(fp);

  return;
}
```

```
/*****************************************************************/

void pr(evs,sta,nsta)

/*      This subroutine is not used in the simulator code; however,
        it can be used as a debugging tool which prints the event
        calender and the station parameters on the screen as the simulation
        is performed. */


list evs;
int nsta;
struct station sta[];

{ list curr,temp;
  int j,k,p;

  printf("time    event code  bus   node\n");
  curr = evs;
  temp = curr->next;
  while(curr != NULL){
  printf("%f   %d          %d   %d\n",curr->evt_time,curr->evt_code,curr->bus,
  curr->node);
  curr = temp;
  temp = curr->next;}

  for (j=1; j<=nsta; j++){
      printf("sta %d\n",j);
      for(p=1; p<=4; p++){
         printf("priority %d\n",p);
         printf("req_cntr = %d\n",sta[j].req_cntr[1][p]);
         printf("cd_cntr = %d\n",sta[j].cd_cntr[1][p]);
         printf("state @ pri %d = %d\n",p,sta[j].state[1][p]);
         if(sta[j].niq[1][p] !=0){
            printf("queue\n");
            for(k=1; k<=sta[j].niq[1][p]; k++)
               printf("%f\n",sta[j].queue[k][1][p]);
            printf("info_bits\n");
            for(k=1; k<=sta[j].niq[1][p]; k++)
               printf("%f\n",sta[j].info_bits[k][1][p]);
         }
         printf("num of cus = %d\n",sta[j].ncus[1][p]);
         printf("depcount = %f\n",sta[j].depcount[1][p]);
         printf("arcount = %f\n",sta[j].arcount[1][p]);
         printf("\n");
      }
      printf("--------------------\n");
  }
  return;
}
```