Modeling and Simulation of a Fiber Distributed Data Interface Local Area Network (FDDI LAN) Using OPNET for Interfacing Through the Common Data Link (CDL)

by

Ernest E. Nix, Jr.

June 1994

Thesis Advisor: Shridhar Shukla

Approved for public release; distribution is unlimited.
**Modeling and Simulation of a Fiber Distributed Data Interface Local Area Network (FDDI LAN) Using OPNET**

**Author:** Ernest E. Nix, Jr.

**Supplementary Notes:**

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

**Abstract:**

The Optimized Network Engineering Tool (OPNET®) is a commercially available communications network simulation package. This thesis involves the modification of OPNET®'s Fiber Distributed Data Interface Local Area Network (FDDI LAN) model in order to enhance its usefulness as an aid in the development of recommendations for the characteristics and metrics to be eventually included in the Defense Service Project Office's (DSPO) Common Data Link (CDL) project. This work includes a step-by-step guide for FDDI simulation in OPNET®, and a discussion of the changes made to the original model to enhance its performance and data display characteristics. Simple tests are provided to verify the completed model's performance and usefulness as a working tool for further development.

**Subject Terms:**

FDDI, Synchronous, LAN, Simulation, Multicast

**Number of Pages:** 213

**Price Code:** UL
Modeling and Simulation of a Fiber Distributed Data Interface Local Area Network (FDDI LAN) Using OPNET® for Interfacing Through the Common Data Link (CDL)

by

Ernest E. Nix, Jr.
Lieutenant, United States Navy
B.S. Ed. University of South Carolina, 1985

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
Monterey, California
June 1994

Author: 

Ernest E. Nix, Jr.

Approved by: 

Shridhar Shukla, Thesis Advisor

Michael A. Morgan, Chairman
Department of Electrical and Computer Engineering
ACKNOWLEDGEMENTS

The author gratefully acknowledges the efforts of those whose assistance made this thesis achievable: Dr. Shridhar Shukla, who provided guidance, support and motivation; MIL 3, Inc. Technical Support Staff who were available days, nights, weekends and holidays, and treated every question seriously, regardless of content; Mr. James "The Wizard" Scott, thesis preparer extraordinaire, who performed miracles on request, and Mrs. Hernestina Nix, who never once complained.

OPNET® is a registered trademark of MIL 3 Inc.

BONeS® is a registered trademark of Comdisco Systems, Inc.
ABSTRACT

The Optimized Network Engineering Tool (OPNET®) is a commercially available communications network simulation package. This thesis involves the modification of OPNET®'s Fiber Distributed Data Interface Local Area Network (FDDI LAN) model in order to enhance its usefulness as an aid in the development of recommendations for the characteristics and metrics to be eventually included in the Defense Service Project Office's (DSPO) Common Data Link (CDL) project. This work includes a step-by-step guide for FDDI simulation in OPNET®, and a discussion of the changes made to the original model to enhance its performance and data display characteristics. Simple tests are provided to verify the completed model's performance and usefulness as a working tool for further development.
# TABLE OF CONTENTS

## I. INTRODUCTION

A. PROBLEM STATEMENT ........................................ 1  
B. SCOPE ........................................................... 2  
C. BENEFITS ....................................................... 2  
D. ORGANIZATION .................................................. 3  

## II. MODELING AN FDDI LAN IN OPNET®

A. OVERVIEW ....................................................... 4  
B. PRELIMINARY THEORY
   1. Theoretical vs OPNET® Model of FDDI ..................... 5  
   2. FDDI LAN Equation Parameters
      a. "F_Max" .................................................. 8  
      b. "D_Max" ................................................. 9  
      c. "SA" .................................................. 10  
      d. "TTRT" ................................................ 11  
C. MODEL STRUCTURE .............................................. 11  
   1. FDDI LAN .................................................. 15  
   2. FDDI Station .............................................. 17  
   3. Processes
      a. Source Node .......................................... 24  
      b. Sink Node ............................................ 26  
      c. Medium Access Control (MAC) ......................... 27  
   4. Packets
      a. ICI Formats ........................................... 30  
      b. Packet Formats ...................................... 33  
   5. Environment File
      a. "station_address" ..................................... 34  
      b. "ring_id" .............................................. 35  
      c. "low dest address" .................................. 35  
      d. "high dest address" .................................. 35  
      e. "arrival rate" ....................................... 35  
      f. "mean_pk_len" ........................................ 36  
      g. "async_mix" .......................................... 36  
      h. "sync bandwidth" .................................... 36  
      i. "T_Req" ............................................... 38  
      j. "spawn station" ...................................... 38  
      k. "station_latency" .................................... 38  

v
I. "prop_delay" .................................................. 38
m. "acceleration_token" ........................................ 39
n. "duration" ..................................................... 39
o. "verbose_sim" .................................................. 39
p. "upd_int" ....................................................... 39
q. "os_file" ......................................................... 39
r. "ov_file" ......................................................... 40
s. "seed" .......................................................... 40
t. "debug" .......................................................... 40

D. SIMULATION .................................................. 41
1. Build the LAN .................................................. 41
2. Correct "OPBUG 2070" ......................................... 41
3. Implement "OPBUG 2081" Patch ............................ 41
4. Update Environment File ..................................... 41
5. Generate Probe File ......................................... 42
6. Simulation Editor ............................................. 42
7. Start the Simulation ......................................... 45
8. Analysis ......................................................... 45
9. Debug Tool ...................................................... 47

III. MODEL MODIFICATIONS ...................................... 50
A. OVERVIEW ..................................................... 50
B. PRIORITIZATION .............................................. 51
1. Activating Prioritization in OPNET®'s FDDI Model ......... 51
2. Changing the Scheme and the Code ........................... 53
3. Subqueues ....................................................... 53
   a. "RCV_TK" ....................................................... 54
   b. "TX_DATA" .................................................... 55
4. Modifications to Prioritization ............................... 57
   a. Station Model Changes .................................... 58
   b. Environment File Changes ................................ 60
   c. Source Modifications ...................................... 61
   d. MAC Modifications ........................................ 63
   e. Sink Node Modifications ................................. 64
   f. Packet Format .............................................. 64
C. PERFORMANCE MEASURES ..................................... 65
1. Overview ....................................................... 65
2. Variables ...................................................... 67
3. Initialization State ......................................... 68
4. "DISCARD" State .............................................. 70
5. "STATS" State ................................................ 72
D. BRIDGE LINK .................................................. 73
1. Station Model Modifications .................................. 74
2. Process Model Modifications .................................................. 75
E. MULTICAST ................................................................. 77
1. Overview ................................................................. 77
2. Environment File .......................................................... 79
3. Station Model ............................................................ 80
4. Source Process Model .................................................... 81
   a. Variables .............................................................. 81
   b. Initialization State .................................................. 83
   c. "ARRIVAL" State ..................................................... 84
5. MAC Process Model ..................................................... 86
   a. Variables .............................................................. 86
   b. Encapsulation State ................................................ 86
   c. Frame Repeat State ................................................ 87
6. Limitation ..................................................................... 90

IV. MODEL TESTING ............................................................... 91
A. SYNCHRONOUS THROUGHPUT ........................................... 92
   1. Overview ................................................................. 92
   2. Setup ..................................................................... 92
   3. Results .................................................................. 95
B. PRELIMINARY LINKING MODEL ........................................ 95
   1. Overview ................................................................. 95
   2. Setup ..................................................................... 99
   3. RESULTS ................................................................ 100
C. SYNCHRONOUS TIMING ................................................... 102
   1. Overview ................................................................. 102
   2. Setup ..................................................................... 102
   3. Results .................................................................. 104
D. ASYNCHRONOUS EFFICIENCY ........................................... 104
   1. Overview ................................................................. 104
   2. Results .................................................................. 107
E. MULTICASTING ............................................................... 107
   1. Overview ................................................................. 107
   2. First Test .................................................................. 109
      a. Setup .................................................................. 109
      b. Results .............................................................. 110
   3. Second test ................................................................ 112
      a. Setup .................................................................. 112
      b. Results .............................................................. 112
   4. Third Test .................................................................. 114
      a. Setup .................................................................. 114
      b. Results .............................................................. 114
   5. Fourth Test ............................................................... 115
V. CONCLUSIONS AND RECOMMENDATIONS .................................. 118
   A. CONCLUSIONS ........................................ 118
   B. RECOMMENDATIONS .................................. 119

APPENDIX A. FILE RETRIEVAL VIA FTP ................................... 121

APPENDIX B. PACKET AND ICI FRAME STRUCTURES ...................... 123
   A. PACKET FORMATS ...................................... 123
      1. "fd_di_lfc_fr" ..................................... 123
      2. "fd_di_mac_fr" ..................................... 123
      3. "fd_di_mac_tk" ..................................... 123
   B. ICI FORMATS .......................................... 124
      1. "fd_di_mac_ind" ..................................... 124
      2. "fd_di_mac_req" ..................................... 124

APPENDIX C. EXAMPLE ENVIRONMENT FILE FOR 32-STATION FDDI LAN .... 125

APPENDIX D. DEBUG TOOL EXCERPT ...................................... 128

APPENDIX E. MAC "C" CODE: "fd_di_mac_mult.pr.c" .................... 133

APPENDIX F. SOURCE "C" CODE: "fd_di_gen_mult.pr.c" ............... 175

APPENDIX G. SINK "C" CODE: "fd_di_sink_mult.pr.c" ................ 184

APPENDIX H. ENVIRONMENT FILE FOR 50-STATION MULTICAST CAPABLE FDDI LAN ........................................ 196

APPENDIX I. CONVENTIONS ............................................ 200

APPENDIX J. GLOSSARY ............................................. 201

LIST OF REFERENCES .............................................. 202

INITIAL DISTRIBUTION LIST ......................................... 203
I. INTRODUCTION

A. PROBLEM STATEMENT

The simulation model described in this thesis was developed in support of the Defense Support Project Office's (DSPO) Common Data Link (CDL) project. The Common Data Link is a full duplex, jam resistant, point-to-point microwave communication system for use in imaging and signals intelligence collection systems (DSPO, 1993, p. 1). Essentially, CDL is to provide a protocol for communication between two or more Fiber Distributed Data Interface Local Area Networks (FDDI LAN). These include an airborne LAN providing sensor information with high data transfer rates, and a ground based LAN providing command and control information.

This work is concerned primarily with the modification and testing of a commercially available communications network simulation program, MIL 3, Inc.'s Optimized Network Engineering Tool (OPNET®). This thesis represents the first portion of three relatively independent research tasks being performed as MS theses to provide evaluations of several Network Interfaces (NI) to the CDL and a multilink point-to-point protocol, in support of the CDL project.
B. SCOPE

The scope of this thesis includes the following:

- Introduce the CDL concept as the context in which the FDDI simulation model is to be modified and tested.
- Provide a tutorial style introduction to the OPNET® FDDI model, designed to expand upon the tutorial provided by the manufacturer. This is directed to those who will conduct further studies in the CDL project, and also to students whose class laboratory work will include simulations in OPNET®.
- Discuss in detail the modifications made to the given model. Provide analysis of the model's actual simulation performance as a validation of the model's usefulness to the CDL work at NPS through comparisons against trials published in the research literature using other simulation tools.

C. BENEFITS

The primary contribution of this thesis is the development of a functioning simulation model that will support the features typically required in a CDL deployment scenario. Typical data communication requirements include the following:

- a wide range of data rates,
- a wide range of error rates and types of error correction required,
- real-time requirements such as user-specified delivery delays and its variation (jitter),
• connection requirements (whether connection-oriented or connectionless, multicasting, broadcasting, etc.),
• retransmission requirements,
• coupling and synchronization with other data sources, and
• adjustable prioritization relative to other sources.

The second benefit is to document in detail the MIL 3, Inc.'s FDDI LAN simulation model in its operation and in its modification. The third benefit lies in the use of the developed model as an instructional tool for classroom laboratory exercises supporting the study of FDDI LAN operation.

D. ORGANIZATION

This thesis is organized as follows. Chapter II provides a tutorial on the use of the FDDI LAN model provided with OPNET®. Chapter III addresses the details of the modifications made to the given model to simulate multicasting and priority-based traffic. Where applicable, clarifications regarding the OPNET® manuals are highlighted. Chapter IV presents the results of simulation tests intended to verify the validity of the modified model. The thesis ends with conclusions and recommendations for future work in Chapter V.
II. MODELING AN FDDI LAN IN OPNET®

A. OVERVIEW

This chapter is intended to provide a tutorial on the use of OPNET® to model an FDDI LAN by providing a brief set of steps to build and execute a simulation. The current version as of this writing is Release 2.4.A, dated 02/27/93, which is the third revision. Release 2.4.A, Errata 1, dated 08/01/93 is a manual update. Some prerequisite knowledge is required of the user, including "C" programming language syntax, ability to use a UNIX workstation, and an understanding of the FDDI protocol. MIL 3, Inc. provide thorough documentation in the form of an eleven volume set of manuals, the first of which is Vol. 1.0, entitled: Tutorial Manual. It includes a general introduction to OPNET®, a trouble-shooting guide, and five chapters presenting different communications network models. While none of these discusses FDDI in particular, all are designed to familiarize the novice user with the mechanics of the user interface, and should be studied prior to working with OPNET®. Volumes 4.0 and 4.1, the Tool Operations Manual, describes the editors of the user interface, and should likewise be studied. The chapter entitled "FDDI" in Vol. 8.1.0, Example Models Manual, Protocol Models, discusses the FDDI simulation in detail, and provides the essential information to build, develop and execute a simulation. Most of the information presented here is available in the manuals, but a number of idiosyncrasies exist which are not readily documented. These required trial and error experimentation to discover, and in many cases required explanation.
from MIL 3, Inc.'s excellent technical support organization. The new user is advised to heed every sentence regarding mechanical details; much of the advice given is hard-earned.

This thesis will not present an explanation of the FDDI protocol in detail, except as necessary to emphasize or clarify the operation of the model. Many discussions exist in the research literature and textbooks, for example, Stallings, (1991, 1993). Those interested in the physical characteristics of optical fiber systems are referred to Powers, (1993). A useful introduction to modeling FDDI in OPNET® is Modeling and Simulation of a Fiber Distributed Data Interface Local Area Network, a Naval Postgraduate School MSEE thesis by Aldo Schenone, which summarizes OPNET®'s FDDI model, includes a detailed description of the FDDI protocol, presents the results of several simulations, and ends with a challenge to other researchers to further develop the model. (Schenone, 1993)

The remainder of this chapter will briefly introduce the structure of OPNET®'s FDDI LAN model and its components, introduce some preliminary modifications, then lead the reader through a simple simulation.

B. PRELIMINARY THEORY

1. Theoretical vs OPNET® Model of FDDI

The setting of parameters in OPNET® simulations is based on the following equation and discussion, which is found in most literature treating FDDI LANs, including

\[
D_{Max} + F_{Max} + TokenTime + ESA, \leq TTRT
\]  

where:

\( SA, = \) synchronous allocation for station \( i, \)

\( D_{Max} = \) propagation time for one complete circuit of the ring,

\( F_{Max} = \) time required to transmit a maximum-length frame (4500 octets), and

\( TokenTime = \) time required to transmit a token.

In an actual LAN, a station management protocol handles the assignments of \( SA, \), which may be changed in real time. OPNET® simulations represent steady-state performance, and do not contemplate changing network conditions.

All stations negotiate a common value of TTRT. Also, these timers and variables are maintained at each station

- Token Rotation Timer (TRT)
- Token Holding Timer (THT)
- Late counter (LC)

Each station is initialized to the same TRT, which is set to TTRT. Note that LC, TRT and THT are not global; each station maintains its own copies, which will differ from those of other stations. If a given station receives the token before its TRT has expired, then that TRT is reset to TTRT. On the other hand, should the token arrive after the expiration
of TRT, then its lateness is recorded by setting LC to "1" (at that station). Two consecutive late tokens will increment LC to "2", in which case the token is considered to be lost, and a re-initialization process will commence. OPNET® has no provision for re-initialization. On the other hand, as a computerized simulation, it never permits LC=2 to occur.

When the token arrives early, (before TRT expires), THT is set to the current value of TRT. The transmission rules are as follows:

1. A station may transmit synchronous traffic for a time \( S_{AI} \), as specified for that station.
2. THT is enabled after synchronous traffic is sent, or if there was no synchronous traffic to send. The station may transmit asynchronous traffic while \( THT > 0 \).

In the TX_DATA state of the MAC process model, THT is incremented from zero to THT. This is an important point in regard to the prioritization scheme by which a value \( T_{Pri[i]} \) is assigned to each priority setting, and the eligibility of a given priority to transmit depends on \( T_{Pri[i]} \) in comparison to THT. That is, for THT decrementing, priority \( i \) traffic may be transmitted as long as \( T_{Pri[i]} \) is less than THT. This implies lower \( T_{Pri[i]} \) is assigned to higher priority. In OPNET®, THT increments up from zero. Priority \( i \) traffic may be transmitted as long as \( T_{Pri[i]} \) is greater than THT, implying that higher \( T_{Pri[i]} \) is assigned to higher priority. This subtle point is important to know when setting values to \( T_{Pri[i]} \) in the INIT state of the MAC process model.

There is an important distinction in the timing of transmission eligibility for synchronous and asynchronous traffic:
The time spent sending synchronous traffic may not exceed $S_A$, for station $i$. That is, the protocol will not allow a synchronous packet transmission to commence if it can not be completed without exceeding $S_A$. OPNET® supports this criterion.

Asynchronous transmissions may commence as long as THT has not expired. Any packet transmission in progress when THT expires is allowed to complete, but no more will commence.

The protocol allows the actual token rotation time to have a maximum value (2)TTRT, with an average value of TTRT over time.

2. FDDI LAN Equation Parameters

Each of the terms in Equation 1 given above is addressed in this section, with reference to its representation in OPNET®'s Environment file attributes.

a. "F_Max"

The time required to transmit a maximum length packet is based on the assumption that any station is capable of transmitting at the rate of 100Mbps. Since the maximum packet length is 36,000 bits (4500 octets or bytes), the algebra yields 0.360 ms for $F_{Max}$. Powers agrees (1993, p. 340), but Tari et. al. use 0.361 ms (1988, p. 55). In OPNET®, the fddi_mmc process model defines the transmission rate as 100 Mbps, in the Header Block. $F_{Max}$ is not directly assigned as an attribute, but simply exists as a physical characteristic which must be considered in determining TTRT and $S_A$ assignments.
"D\_Max"

The Maximum Ring Latency is the time required for a frame to travel around the ring. The maximum value is often assumed in textbook discussions, but it should be calculated for individual cases. The total delay may be defined as follows:

\[D\_\text{Max} = (\text{total fiber length} \times \text{delay rate}) + (\text{number of stations} \times \text{station latency})\]

Powers uses 1.73 ms, Tari et al. uses 1.62 ms, and Dykeman and Bux use 1.62 ms. \(D\_\text{Max}\) includes the time required for a frame (which is basically a number of light flashes) to travel the length of the fiber on the ring, plus the time required to cross each station interface. \(5.085 \times 10^{-8}\) sec/km is the value used in the literature for the delay rate of a signal in optical fiber. The reciprocal results in 1.9665e+08 m/s, which agrees with the generally accepted value of 2/3 \(c\) for the speed of light in glass. OPNET® and Dykeman & Bux use a station delay of 60.0e-08 sec. Powers assumes 1 \(\mu\)s as a representative value. Ultimately the value is a physical characteristic that could be measured on a real device, and may be declared in a computer simulation. The value 1.617 ms derives from using the maximum possible dimensions: 500 dual attachment stations or 1000 single attachment stations on a 200 km ring yield the following:

\[(1000 \times 60.0 \times 10^{-8}) + (200 \times 5.085 \times 10^{-8}) = 1.617 \text{ ms} .\]

The environment file attribute prop\_delay represents the actual time the packet is on the fiber between two stations, rather than the delay rate, and therefore defines the size of the ring. OPNET® has no safety feature to prevent the user from entering
attributes that would define a ring larger than 200 km. Note that the stations are assumed to be equally spaced. The user should realize that the value of 5.085e-06 given with the original example environment file implies a one kilometer length of fiber, rather than a delay rate.

Powers (1993, p. 328) notes that the early proposal for FDDI limits fiber length between stations to 2 kilometers. In OPNET®, the attributes prop_delay and station_latency are used in the "C" code to postpone the reception of a packet until sufficient time has passed to allow for physical delays.

c. "SA_t"

Synchronous allotment, or synchronous bandwidth, is the time a station is granted to transmit synchronous traffic, regardless of the lateness of the token. It is a form of prioritization, providing a means by which certain types of traffic are not delayed. For example, voice traffic would be synchronous. Textbook discussions represent SA, in units of time for each station.

Asynchronous traffic is transmitted whenever there is THT remaining after the transmission of all synchronous traffic. It is the responsibility of the Station Management Protocol (in OPNET®, the user) to ensure that the synchronous bandwidth is sufficient to handle the synchronous offered load. One nuance involves the inviolate nature of SA, for each station. A given station's synchronous offered load may amount to relatively little in terms of bits per second, while the packet size is assigned a value too large to be transmitted in the time SA,. In this case, synchronous traffic would never be transmitted, and outbound packets would simply accumulate in the buffers of the MAC. The Environment file attribute
"sync bandwidth" corresponds to $SA_i$, but is expressed as a unitless fraction of TTRT, rather than as a time.

**d. "TTRT"**

Equation 1 suggests that physical requirements of the fiber and the stations are used to determine a workable TTRT value. The FDDI specification allows a range of settings from 4 ms to 165 ms (Powers, 1993, p. 339). Powers also notes that synchronous voice transmission requires $\sum SA_i = 10$ ms. In OPNET®, the Environment file attribute $T_{Req}$ corresponds to TTRT.

### C. MODEL STRUCTURE

OPNET®'s FDDI LAN model structure is hierarchical. The LAN is a ring made of stations and the connections between them. Figure 1 shows a 50-station FDDI LAN as shown in the user interface window. Figure 2 is a ten-station ring provided for greater clarity of detail. The stations are modeled as connected nodes, each of which is in turn defined by a process model. The processes are represented by state transition diagrams, which are the ultimate source of the "C" language code that describes the model's behavior. Figure 3 illustrates the FDDI station model. Figures 4-6 are the process models for the source, sink, and MAC processes, respectively. These correspond one-to-one to the nodes "llc_sink," "llc_sink," and "mac" shown in Figure 3. The packets of information that travel between stations on the ring, and between nodes within the station, are also modeled and may be
Figure 1. 50-Station FDDI LAN, "fddi_net_50"

Figure 2. Ten-Station FDDI LAN, "fddi_net_10"
Figure 3. FDDI Station Model, "fddi_station"

Figure 4. Source Process Model, "fddi_gen"
Figure 5. Sink Process Model, "fddi_sink"

Figure 6. MAC Model, "fddi_mac"
modified. Model parameters may be entered by several methods, with the Environment file being by far the most convenient.

1. **FDDI LAN**

Figure 7 shows a 32-station FDDI ring in the user interface window as it would appear on a computer screen. This image displays the Network Editor, whose icon appears toward the upper left corner of the figure. (A note on the mechanics of activating the various editors: as indicated in the tutorial manual, the center mouse button activates the desired editor. If instead either the left or right button is pressed, then the opposite button must also be pressed to "cancel" the first; only then will the center button work as expected). To the right of the ring are on-screen menus of attributes for one station (actually, three menus are shown to display simultaneously all the fields). This menu is invoked by placing the cursor over the desired station, then pressing the right mouse button. The FDDI protocol supports up to 500 dual-attachment stations on a ring, and OPNET® permits from two to 500 stations in a ring.

Actual generation of the ring is best done outside OPNET®, through a UNIX command window set to the "-\op_models\fddi" directory path. The command "fddi_build_em.x <number_of_stations>" will automatically generate an FDDI LAN with the number of stations specified. The user should verify that this function is present in the desired subdirectory. This operation is described in manual Vol. 8.1.0, "FDDI," and refers the user to Vol. 6.0, *External Interfaces Manual*, which has a more complete description of the ring building protocol. Were OPNET® active during this ring-building process, then the "Rehash" icon toward the lower right of the user interface window must be activated to update the
Figure 7. Network and Attributes Menus in User Interface Window
program's access to models in the subdirectory. In general, the "rehash" command should be used frequently, particularly when new files are generated through simulation runs or through model editing.

The LAN as shown is not actually a true ring architecture, as no dedicated physical layer object exists in OPNET® for modeling ring architectures. The model is in fact a circle of point-to-point links; the ring is an abstraction whose characteristics and behavior are represented in the "C" programs that comprise the process models. (OPNET®, Vol. 8.1.0, "FDDI," p.23)

2. FDDI Station

Figure 8 illustrates the FDDI station in the user interface window, summoned and printed from the Node Editor. Also shown are the menus listing the attributes associated with each part of the station model. Message traffic in the form of packets is generated at the source, llc_src, at a rate specified by the user. The source model does not function as a true Logical Link Control (LLC) beyond correctly interfacing with the Medium Access Control (MAC) model. (OPNET®, Vol. 1.0, "FDDI," p. 21) The MAC entity is represented by mac in the model, and is responsible for encapsulating packets generated by the source, holding these packets until they can be transmitted, receiving packets from other stations, destroying packets as needed, and maintaining the locally held Token Holding Timer (THT) and Token Rotation Timer (TRT). Packets are counted and statistics gathered at llc_sink. These three nodes are modeled in detail by respective process models, which may be assigned with the attributes menus shown in Figure 8. The field "process model" may be changed
by cursor action, with selection made from the resulting submenu presenting a list of available process models. The nodes "phy_rx" and "phy_tx" represent the receiver and transmitter interfaces to the ring, and are not further defined by process models.

The user may modify the station model within the Node Editor by setting the attributes fields as desired, then saving the model by activating the "write node model" icon toward the lower left corner of the user interface window. By then exiting the Node Editor, entering the Network Editor, and calling the desired network model (e.g., "fdtnet_32"), each station acquires the new setting when the network model is archived and bound ("A+B" icon). The same modification may be effected from within the Network Editor by calling the attributes menus for each station and setting them individually. This method would be preferred only if the user desires to set differing attributes in various stations. Note that setting the attribute fields is not the same as modifying the process model itself, which is accomplished with changes to the "C" programming code accessed through the Process Editor.

3. Processes

Process models are specified by State Transition Diagrams (STD) representing the actions of the nodes within the station model. Figure 9 illustrates the process model fdd1_mac as it appears in the Process Editor in the user interface window. Figure 10 shows the on-screen menu that appears when the cursor is placed over one of the states (ENCAP in this case), and the right mouse button pressed. Invoking the "enter execs" attribute calls the text editor shown in Figure 11. Here the user may inspect the programming code behind
Figure 9. Process Model "fiddi_mac" in User Interface Window
Figure 10. Process Model "fiddi_mac" with On-Screen Attributes Menu
Figure 11. Process Model "fiddi_mac" with Text Editor
the model's behavior, and modify it if desired. Each state has its own section of code, and the
icons to the left of the window include additional editors, all described in Vol. 4.0, Tool
Operations Manual. These are primarily variable and function declaration sections. The
entire code for the process may be called with the icon "C," but this editor is for viewing
only. Any changes made in the "C" editor will remain when the editor is dismissed, but will
disappear if the process is compiled. If calling the "C" editor returns a "C file
unavailable" message, then recompiling will generate the file again (sixth icon above the
lower "EXIT" icon in the Process Editor). Many UNIX stations include a cleanup command
that deletes certain temporary files upon logging out of the system, and the "C" language
codes ending in "c" are not necessary once simulations are generated. They may always be
recovered by recompiling the models. (In the file directory containing OPNET®, the model
source codes have the suffix "pr.m".) If changes are made (in a proper editor), the model
must be recompiled. If several changes are being made within different sections, each may
be saved with the keystrokes <CTRL+S>, deferring compiling to the end. (The set of manuals
includes a summary page of OPNET®'s text editor keyboard commands.) If desired, the
model may be saved without compiling by using the "Write Process Model" icon. When
a process is changed, the station model in the Node Editor must also be called and written
afresh. Then the corresponding network model must be called into the Network Editor, and
be archived and bound again. If the modified process was not compiled earlier, then the
"C+A+B" icon will compile all the process models in addition to archiving and binding them.
a. **Source Node**

The source node of the FDDI station model generates packets at a rate and size specified by the user. It also determines the destination address for each packet, the priority if applicable, and records the packet’s creation time so that delay statistics can later be gathered. These data are passed to the MAC for encapsulation. In the Node Editor, the source is labelled "l1c_src," and the process model is "fdd1_gen."

In the source process' original form, as released by MIL 3, Inc. in version 2.4.A, the packet arrival (generation) rate is stochastically assigned on an exponential distribution approximating that specified by the user. If a precise, invariant arrival rate is desired, it may be assigned with the following change to the INIIT state in the Process Editor, where "constant" is substituted for "exponential" in the line:

```
inter_dist_ptr = op_dist_load("exponential", 1.0 / arrival_rate, 0.0);
```

A voice traffic transmitter station, for example, would require a constant packet arrival rate from the source. Similarly, packet length is originally assigned a constant value in the given model, but may be set to a stochastic approximation of the requested value by replacing <"constant"> with <"exponential"> in the line:

```
len_dist_ptr = op_dist_load("constant," mean_pk_len, 0.0);
```

If all stations on the ring are to have the same assigned attributes and the same source code (i.e., all <"constant"> or all <"exponential">), then the remaining steps
are to save and compile the process in the Process Editor, then call and save the station model in the Node Editor, and finally archive and bind the relevant LAN model in the Network Editor. If the stations on the LAN do not all have identical source code (i.e., some "constant" and others "exponential"), then the modified process models and their corresponding node models must be renamed. The following steps illustrate the creation of a station modified to allow a ring to simulate a number of voice stations amid other transmitters:

1. In the Process Editor, substitute "constant" for "exponential" in the "fdd1-gen" model's INIT state editor. Save the change by keying \(<CTRL+S>\) while the cursor is inside the INIT state's editor.

2. Use the "Write Process" icon to save the modified process under a new name, for example fdd1_gen_const.

3. Compile the new process model, then exit the Process Editor.

4. The Node Editor is used to create and save a new station model, by calling the original model and changing the "process model" field in the on-screen menu for the relevant node (in this example the "fdd1_gen" process in the llcsrc node is changed to "fdd1_gen_const").

5. The new model is saved by invoking the "Write Node Model" icon. Exit the Node Editor.

6. In the Network Editor, the desired stations on the relevant LAN are reassigned using the on-screen menus: when the "model" attribute field is invoked at a
particular station, a list of available station models appears, and the desired one is chosen. If the expected model does not appear in the list, activate the User Interface Window "Rehash" icon to refresh OPNET’s access to recently created files.

7. Desired stations are reassigned as required, and the LAN is saved, then re-archived and bound.

8. Differently named models using the same functions may cause naming conflicts at simulation run time. Should this occur, then the word "static" must be inserted in the Function Blocks ("FB" icon) of both the original and the new source model processes in the Process Editor, just prior to \texttt{fddi\_gen\_schedule()}.

The above steps illustrate a change made to the source code, and do not represent the same situation in which identical stations are assigned different values in the given on-screen attributes menus.

\textbf{b. Sink Node}

The \texttt{llc\_sink} node of the station model is the final destination of all message traffic. The INIT state establishes counters to hold statistical information regarding network performance (throughput and delay). The STATS state updates these counters as packets are received. The DISCARD state reports the statistical information at specified intervals, and finally destroys the packet. Because new packets are created for each transmission, they must eventually be destroyed when received, or the host computer conducting the simulation will soon fill its memory.
The "fdd1_sink" model in the current version of OPNET® (Release 2.4.A, dated 02/27/93) is defective. It will cause the simulation to abort upon completion, with the error message "Program Abort: packet pointer is NIL." in the event any station did not receive traffic. Figure 12 illustrates the State Transmission Diagram as originally given, and the corrected version is shown in Figure 13. The user should correct the defective version, referring to Vol. 1.0, Tutorial Manual, "Bpt," pp. 6-10. Saving this modification requires the same steps described for the source model, with the exception that no text has been changed. This defect and its correction are documented by Mil 3, Inc. as OPBUG 2070.

MIL 3, Inc. maintains an electronic bulletin board containing information on model corrections and upgrades between OPNET® revisions. Users may acquire these upgrades using file transfer protocol (ftp) procedures to download desired files. Appendix A includes a sample of dialogue used to acquire an upgraded file from MIL 3, Inc.

c. Medium Access Control (MAC)

The MAC process model encapsulates frames received from the source node for transmission to other stations, maintains token holding and token rotation timers, inspects received packets, decapsulates received frames, and determines token usability. Vol. 8.1.0, Example Models Manual, "FDDI," provides a detailed description of the MAC process and of the functions of its component states.

The MAC model provided with OPNET® Version 2.4.A, (filename: "fdd1_mac.pr.m") has been upgraded by MIL 3, Inc. and the newer model and
Figure 12. Original (Defective) Sink Process Model

Figure 13. Corrected Sink Process Model
documentation are available via ftp on Internet, under the subdirectory 
"-/patches/2.4.8/opbug_2081". When retrieving a file via ftp, the user should verify the 
entire file is received by checking the file size listed on the bulletin board against the size of 
the file received. Entering "type image" at the ftp prompt should ensure a full and intact 
file retrieval. The original "fdd1_mac_pr.m" file is then removed from the user's directory 
(~/op_models/fddi), stored in a safe place, and replaced with the newer version. The new file 
must then be compiled from the command window with the command, "op_mkpro -m 
fddi_mac", the procedure for which is described in Vol. 6.0, External Interfaces Manual, 
"Env". It has been observed that the "drag and drop" method of transferring files in using the 
File Manager in the SunOS Windows environment sometimes causes the subject file to gain 
or lose a byte or two, leading to "bitsum error" messages when said file is compiled in 
OPNET®. Standard UNIX commands are the most reliable method for moving OPNET® 
source files.

The patch is not necessary to operate the simulation; it is a refinement of 
the model, and will be included in the next revision of OPNET®. The patch OPBUG 2081 
actually includes three repairs, documented as OPBUGs 2081, 2095, and 2097. OPBUG 
2081 corrects existing timing and efficiency inaccuracies connected with the token 
acceleration feature, by which the token is destroyed and the simulation enters a "fast 
forward" mode in order to reduce the number of events while no station has a need to 
transmit. In a real FDDI LAN, TRT is reset each time the token passes, whether or not the 
token is used. The model in its original form allows the TRT timer to continue running when
the simulation enters "token acceleration," resulting in unexpected Late_Count occurrences. OPBUG 2095 is also related to the token acceleration feature, correcting the existing incorrect initialization of several variables when the simulation enters token acceleration. In particular, the variable Fdd1_Num_Station, the number of stations on the ring, is always reset to one, upsetting calculations predicting the proper location of the token at the end of an idle period. Finally, OPBUG 2097 addresses the fact that the original model neglects to properly account for the transmission delay associated with the token itself.

4. Packets

All communications between stations in a LAN and between the internal nodes of a station are conducted using data framed into packets. The Parameter Editor, described in Vol. 4.0, Tool Operations Manual, "Pm," is illustrated in Figure 14, which shows the packet structure fdd1_macifr, which is used to encapsulate the frames generated in the Source node and sent to the MAC. Appendix B lists the five packet structures used in the FDDI LAN simulation, giving their fields and assignments. OPNET® simulation does not enforce limits on packet size required by the standard IEEE 802.5.

a. ICI Formats

Interface control information packets (ICI) are used for internal communication within a station, reporting for example service options, error conditions, and packet arrivals. Figure 15 shows the ICI Editor within the Parameter Editor, with the ICI fdd1_mac_req. This ICI specifies the control information passed from the source to the MAC when transmission requests are generated. The ICI fdd1_mac_ind specifies control information passed from the MAC to the LLC when a packet has been received by a station.
Figure 14. Packet Format "fddi_mac_fr" in the Parameter Editor
Figure 15. ICI Format "fddi_mac_rec" in Parameter Editor
For OPNET simulation purposes, both structures are created once per station in the simulation, and reused as needed.

b. Packet Formats

Three types of packet frame formats exist in OPNET to simulate communications between the stations. For the simulation these are created as needed and destroyed when no longer needed. Packets of format "fddi_llc_fr" are created in the LLC source as arrivals are generated. The format has only one field, containing the creation time, which is used to generate delay and throughput statistics when the packet is finally received at its destination address. The packet format "fddi_mac_fr" is used in the MAC state ENCAP to encapsulate the generated packets for transmission on the LAN. The "info" field contains the "fddi_llc_fr" structure providing the data of interest. Because OPNET simulates only the characteristics of transmission and not the actions of stations in response to information received, the packets used are not precise replicas of real FDDI frames. The token is represented by the frame format "fddi_mac_tk". The field "fc" is inspected by each MAC process receiving a packet to determine whether it is a token or a message packet.

5. Environment File

Appendix C is an example of an environment (or configuration) file used to assign station attributes to a 32-station FDDI LAN. Inspection shows that the fields specified in the file correspond to the "promoted" fields in the on-screen attributes menus that appear in the Network and Node Editors, and to the fields in the Simulation Editor. Promoted attributes may be assigned directly within these editors, a tedious and error-prone process at best. If
a simulation is begun with none of these attributes specified, OPNET® will prompt the user for inputs at the command screen, another error-prone and tedious process. The environment file is the most efficient way to assign parameters of interest, and may be quickly modified between simulations, using the UNIX text editor. Vol. 6.0, External Interfaces Manual, “Env,” discusses the environment file, and points out that attributes assigned in the environment file supersede those assigned in the Node and Network Editors. The file is recognized to OPNET® by its “.ef” suffix. The assigned attributes are described in chapter “FDDI” of manual Vol 8.1.0, Example Models Manual, and are presented here in their order of appearance for summary and in some cases for required elaboration. The attributes are not declared as variables in the “C” programs, but rather are generated by adding them to the on-screen menus in the Node and Network Editors. The user can create new attributes by adding them to the “extended attributes” field in the Node Editor, then including them in the environment file. This procedure is discussed in Chapter III. “Env,” pp.33-34, in the External Interfaces Manual discusses the use of name wildcards in the attributes given below. Sequence of entries is not significant in the environment file.

a. “station_address”

This attribute is required for station identification; numbering of stations is from zero to \( N-1 \), where \( N \) is the number of stations on the ring. The INIT state in the MAC process calls this variable. Note that for quick changes to the file, lines may be commented with the pound key “#”.

34
b. "ring_id"

This attribute identifies the ring, in the event more than one may be modeled simultaneously. It is set to zero if there is only one ring.

c. "low dest address"

This attribute assigns the lowest identification address that may receive traffic from this station. The use of the wild-card asterisk, shown in Appendix C, assigns the same value to all stations. Quotation marks are used here because the attribute assigned has spaces vice underscore marks between words.

d. "high dest address"

This attribute assigns the high end of the range of addresses to which a station may send traffic. It is used in conjunction with the previous attribute by the "llc_src" process in the INIT state. It is permissible to limit the range of target addresses, down to one, but all target addresses must lie in a contiguous sector. For example, the code as given has no provision for allowing a particular station to send packets to two different stations without also possibly sending to the stations between them.

e. "arrival rate"

This attribute assigns the rate at which the source process will generate packets, and it is called by the INIT state. It may be set to zero for any station intended to be idle. As originally used in the process code, arrival rate is a stochastic approximation exponentially distributed about the assigned value. To make this a precise unchanging value, as in the case of a synchronous voice transmitter, the code would have to be modified as described in the previous discussion of the source process.
f. "mean_pk_len"

Mean packet length is expressed in bits. Despite the name, this value is actually held constant by the INIT state of the source process model. The user may modify the code using the procedure described earlier to substitute "exponential" for "constant" in the line:

```
len_dist_ptr = op_dist_load("constant," mean_pk_len, 0.0),
```

which appears toward the bottom of the INIT state enter executives in the Process Editor. Then the station will generate non-identical packets, which may be more realistic behavior. OPNET® will permit any number of bits for the packet length; the user should know that FDDI packets have a maximum size of 36,000 bits.

g. "async_mix"

FDDI stations may generate synchronous and asynchronous traffic. This attribute sets the proportion, with 1.0 indicating all asynchronous traffic generated by the given station, 0.5 indicating half synchronous and half asynchronous. Any value between zero and one inclusive may be chosen. The INIT state in the 11c_src model calls this attribute with the statement:

```
op_ima_obj_attr_get(my_id, "async_mix", &async_mix);
```
h. "sync bandwidth"

This attribute is used in the MAC process, INIT state, where it is expressed as a percentage of TTRT. It is analogous to $SA_i$ in Equation 1, but is numerically a fraction of $T_{-req}$ (TTRT), while $SA_i$ is an amount of time. Synchronous bandwidth should not be confused with synchronous offered load. Bandwidth is expressed in time, while synchronous offered load is a bit transmission rate. Therefore, synchronous bandwidth is the time allotted for the transmission of synchronous offered load. It is entirely possible to set parameters so that these two attributes do not match well.

In describing "sync bandwidth," Vol. 8.1.0, Example Models Manual, "Fddi", warns the user not to allow the sum of all assigned attributes "sync bandwidth" to exceed one, since OPNET® does not enforce FDDI protocol standards. However, this warning neglects to consider the physical delay parameters in Equation 1, which indicate that total synchronous bandwidth must be somewhat less than TTRT. The correct assignment of "sync bandwidth" involves some algebra, and is perhaps best explained with an example.

Given:

- $D_{Max}$: 1.617 ms.
- $F_{Max}$: 0.360 ms.
- Token_Time: 0.00088 ms.
- TTRT: 8.0 ms.

Using Equation 1 yields:

$$8.0 \text{ ms.} - (1.617 \text{ ms.} + 0.360 \text{ ms.} + 0.00088 \text{ ms.}) = 6.02112 \text{ ms.} = \sum SA_i$$

This is 6.02112 ms. of total bandwidth to be divided among as many stations assigned. Assume there are five such stations:
6,02112 ms. + 5 stations = 1.204224 ms = SA_i

SA_i is converted to "sync bandwidth" with a division by TTRT:

1.204224 ms. + 8.0 ms. = 0.150528

This is the value entered into the Environment file.

i. "T_Req"

This attribute is called in the MAC process, INIT state, and represents the specified station's requested value of TTRT. A real FDDI LAN has a TTRT negotiation phase; OPNET® simply chooses the smallest \( T_{\text{req}} \) value available. The user may, but need not set different values to each. This value is in units of seconds, which is not apparent from the manuals nor from the default value that appears at the command prompt if no value is assigned.

j. "spawn station"

The spawn station is simply the starting point for the token, and may be assigned to any station on the LAN.

k. "station_latency"

This is the delay incurred by packets as they pass a station's ring interface. Powers gives 1 \( \mu \text{sec.} \) (1993, p.336); 60.0e-08 sec. agrees with Dykeman and Bux (1988).

Station latency is a component of \( D_{\text{Max}} \) in Equation 1.

l. "prop_delay"

Propagation delay is the time separating stations on a ring, based on the amount of fiber between them. It is given here in seconds, and may be used to define the ring size. The INIT state of process fddi_mac calls this value, which is used as one of the delay
parameters applied to transmission commands. FDDI standards limit the ring size to a maximum of 200 miles of fiber (Dykeman, 1988, p. 997), and OPNET<sup>®</sup> assumes that the fiber length is divided evenly among the stations. That is, all stations are evenly spaced on the LAN in OPNET<sup>®</sup> simulations, whatever the number of stations and length of fiber. Dykeman and Bux (1988, p. 1000) define propagation delay in units of time per distance, and give a value of 5.085 μs/km. The value given in the original example environment file, 5.085 × 10<sup>4</sup> seconds, corresponds to one kilometer of fiber between stations.

m. "acceleration_token"

This attribute speeds the simulation by removing the token during idle periods when no station has packets to transmit, significantly reducing the number of events.

n. "duration"

This is the simulated run time in seconds. Most systems should reach steady state in less than a second.

o. "verbose_sim"

This feature enables on-screen reports regarding event numbers, time remaining until completion, etc.

p. "upd_int"

This specifies in seconds the intervals at which to make on-screen simulation status updates. It must be less than duration to be useful.
q. "os_file"

The output scalar file receives scalar data accumulated over several simulations. It is useful in observing the effect of varying one or more attributes, for example TTRT, over a series of experiments.

r. "ov_file"

The output vector file receives throughput and delay information relevant to one simulation run. Output vector files can be quite large, on the order of several megabits, and for this reason are often automatically deleted by a "cleanup" command included in a UNIX station's logoff sequence. The user should alter the filename or save desired plots as "*.ac" files using the Analysis Tool, rather than log off planning to study the vector data at some future time.

s. "seed"

This is a constant used by the simulation's random number generator. It may be any positive integer, but should be left constant once chosen.

t. "debug"

This enables the Debug Tool, allowing the user to step through a simulation one event at a time. Once enabled, the command "help" provides a listing of the debugger's features.
D. SIMULATION

This section presents the steps involved in running a simulation and observing the resulting output data. The user must keep in mind that OPNET® is unaware of IEEE 802.5. That is, it is the user's responsibility to ensure reasonable input parameters are assigned in keeping with the established standards. The steps given will use the original model provided.

1. Build the LAN

If a 32-station LAN is not already available in the Network Editor, then one should be created using the command "fddi_build.em.x 32", as described earlier.

2. Correct "OPBUG 2070"

The simulation will abort if the original process model fddi_sink is used for the SINK node llc_sink, and some station happens to have not received any packets. The correction described earlier should be applied, and the model recompiled and saved.

3. Implement "OPBUG 2081" Patch

As described earlier, this repair corrects minor timing inaccuracies in the model, related to the token and to the token acceleration feature. The simulation will work without aborting if the patch is not applied, but the user planning on implementing code changes within the model over the long term should patch the model before doing so.

4. Update Environment File

Refer to the configuration file in Appendix C for input parameters. As mentioned before, use of this file will save the user the effort involved in setting parameters by hand through the Node, Network, and Simulation Editors. Note that attributes assigned in the
Environment file will supersede any that are assigned through these editors. The file should be given some distinguishing name, for example "fddi32.ef".

5. Generate Probe File

Use of a Probe file is optional. The process code as written will generate vector file outputs only for overall throughput, delay, and mean delay. Additional outputs may be monitored through the use of a Probe file, illustrated in Figure 16. Vol. 4.1, Tool Operations Manual, "Pb," describes the Probe Editor. With it the user may monitor, for example, each station's arrivals (packet generation) and throughput at any physical interface point on the LAN, measured in packets and/or in bits per second. For simplicity, only packet arrivals at station f11 will be assigned a probe in this simulation.

6. Simulation Editor

Figure 17 illustrates the Simulation Tool in the user interface window, with the settings necessary to run this simulation. Use of this tool is discussed in Vol. 4.1, Tool Operations Manual, "Sm". Fields are assigned by use of the cursor, and are filled by choosing from on-screen menus or keyboard entry. The "Simulation" field should be assigned the LAN filename, <fddi_32_net> (note that filename suffixes are not visible to the user in the various Editors). The fields "Probe File", "Vector File", "Scalar File", "Seed", "Duration", and "Upd Intvl" may all be left blank if they are assigned in the configuration file; "Probe File" is optional in any case. The "Arg Name" field should be assigned <environment file>, and the "Arg Value" field should be assigned the <filename> given to the environment file. The user may then save the work area using the
Figure 16. Probe Assignment
Figure 17. Simulation Tool
"Write Simulation" icon, to spare the effort of filling in these fields again on future simulations.

7. Start the Simulation

Once the fields are set, the simulation is started with the "Execute Simulation Sequence" icon. Had the user neglected to assign some parameter, the simulation will wait until the command line prompt has been answered; the user should keep the command screen in view. Upon completion of the simulation, a vector file (suffix ".ov") will be generated, along with a scalar file (suffix ".os"). The "Rehash" icon must be invoked to refresh OPNET's access to the files. Then the user may exit the Simulation Tool and enter the Analysis Tool.

8. Analysis

Figure 18 shows the Analysis tool in the user interface window, whose operation is explained in Vol. 4.1, Tool Operations Manual. The first action upon entering this tool is to call the available vector outputs, using the "Open Output Vector File" icon, then selecting from the choices presented. If more than one are present, choose the one that was assigned in the environment file. The on-screen menu will then disappear, leaving the user to select the "Create Single Vector Panel" icon, which presents the on-screen menu shown in Figure 18. The entries "end-to-end delay (sec.)", "throughput (bits/sec)", and "mean delay (sec.)" are generated directly from the SINK process model. The remaining field, "ring0.f11.mac[0].pksize," comes from the Probe Editor. Each may be plotted by selection with the cursor, then dragging the box corners to the desired
Figure 15. Vector Trace Selection in the Analysis Tool
Having placed the panel, the plotted points are fired by clicking the left mouse button, or by placing the cursor over the "Fire All Panels" icon and clicking the same button. Figure 19 shows all four plots generated, placed together on the screen. The "Create Multi-Vector Panel" icon is used to place several plots in the same panel, an operation that is meaningful when the Probe Editor is used to generate comparable outputs. Once the desired plots are on the screen, they may be saved with the "Write Analysis Configuration" icon, which will store the plots in a ".ac" file for later recovery. This is important because the vector file will be written over the next time a simulation is run using the same output vector filename. In addition, the UNIX station's logout sequence may include a "<remove *.ov>" command to prevent the accumulation of large vector files in memory.

The output scalar file, on the other hand, accumulates steady-state data over several simulations, allowing the generation of plots showing, for example, the effect of various TTRT values on total throughput. The user wishing to create such a plot should ensure the file is empty of previous data before commencing a series of simulations.

9. Debug Tool

The debug tool may be activated from the environment file. The command "help" will list the available commands. The user may step through a simulation one event at a time, or specify stopping points. The "fulltrace" command causes every variable to be reported at each event, allowing the user to follow the sequence of events in a
Figure 19. Analysis Tool Display, Four Panels
simulation, and to search for logic errors should failures occur. Appendix D is a short section of the debugger's output when the <"fulltrace"> command is active.
III. MODEL MODIFICATIONS

A. OVERVIEW

The FDDI station model provided with OPNET® is shown by Schenone (1993) to perform as expected by performance equations provided by research literature, for example, Dykeman and Bux (1988). However, the model as given lacks the flexibility to adequately demonstrate the characteristics and metrics required to formulate recommendations for the development of a CDL network interface. To begin, no way exists in the original model to monitor the throughput and delay statistics of synchronous traffic separately from asynchronous traffic. In addition, the code must be modified to allow the implementation of different asynchronous priority levels, and further altered to allow the generation of statistics of these subcategories. Also desirable is a method to hold traffic in the sink process of one station on the LAN, rather than destroying all packets, so that a bridging protocol may eventually be developed for communication between LANs, which is the ultimate goal of the Common Data Link Project. Finally, a multicast/broadcast facility, by which a packet may be addressed to multiple stations, is needed. Modifications implementing these features were generated for this thesis, and are described in this chapter. Appendices E, F, and G contain the final form of the "C" programs representing the process models "fddi_mac_mult", "fddi_gen_mult", and "fddi_sink_mult", respectively. All contain inserted comments to indicate where changes have been made. In some cases, modifications are extensive.
enough that the original structure is not apparent. For these cases the reader who is interested in comparisons is referred to the original code available in the Process Editor.

The modifications described here were suggested in large part from readings in the research literature. In particular, Tari, Schaffer, Poon and Mick (1991) published results generated from another commercially produced network simulation tool, the Block Oriented Network Simulator (BOnNeS®), demonstrating that increased asynchronous offered traffic load has minimal effect on the throughput of synchronous voice data traffic. At the same time, the throughput of the various asynchronous levels was shown to degrade in order of priority with increasing asynchronous offered load. These findings emphasize the fact that OPNET® has no facility in place providing for the monitoring and plotting of throughput or delay data in respect to class or priority levels.

Closely interrelated are the setting of asynchronous priority levels, a system of subqueues to segregate traffic by priority, and the gathering and display of performance statistics according to priority. In the following discussion, reference is sometimes made from one to the others before all are complete.

B. PRIORITIZATION

1. Activating Prioritization in OPNET®'s FDDI Model

As given in the original released model, code exists to support a prioritization scheme, but it is not implemented. The station model includes a priority field that may be set in the Node Editor, but no setting will take effect until the INIT state in the MAC process is
modified. As given, \( T_{\text{Pri}}[i] \) is simply assigned the value \( Fdd1\_T\_Opr \) (which is the negotiated value of TTRT, with the negotiation consisting of selection of the lowest requested \( T_{\text{Req}} \) from all stations) for all priority settings \( i \), resulting in no distinction made between priorities. The state \( TX\_DATA \) in the MAC process model contains the code that determines transmission eligibility, then transmits packets if timing conditions are satisfied. The user should notice here that unlike the real FDDI protocol, the Token Holding Timer is incremented from zero to \( \text{tht-value} \) (THT), not decremented from THT to zero. This results in a reorientation of \( T_{\text{Pri}}[i] \) settings, wherein higher settings allow a larger transmission window, and therefore, higher priority. To re-emphasize: in the OPNET\textsuperscript{®} model, \( T_{\text{Pri}}[i] \) is larger for higher priority stations. Actual settings of \( T_{\text{Pri}}[i] \) are a matter of user's choice and real-world physical characteristics. One quick approach is to alter the code in INIT from the original:

```c
for (i = 0; i < 8; i++)
{
    T_Pri [i] = Fdd1_T_Opr;
}
```

by substituting the text:

```c
T_Pri [i] = (double)Fdd1_T_Opr/(8-i);
```

to impart some weight to the priority settings (Appendix E, line 250). Note that priority settings in OPNET\textsuperscript{®} are counted from zero to seven, in keeping with the "C" programming
language convention of numbering elements of an N-element array from zero to N-1. As mentioned earlier, actual FDDI convention numbers the priority settings from one to eight.

2. Changing the Scheme and the Code

For the purposes of CDL, an ability in a station to generate traffic of differing priorities is a desirable characteristic. The modifications discussed here allow this behavior, though with a certain amount of abstraction included. Essentially, each packet generated in the source process is assigned a priority setting, in a manner that is functionally identical to the determination of the destination address. However, the priority of one packet has no influence on the priority of the next one generated at the same station. Of course in real-world transmissions, packets are grouped into messages, meaning that thousands of consecutive packet arrivals should have the same priority settings to reflect real behavior. In fact, the subqueue structure imparted to the MAC causes outgoing packets to be sent in decreasing order of priority, thereby modeling expected behavior to some small degree. More significantly, the user should keep in mind that the model's purpose is to model a LAN's handling of the traffic it does receive. The fact that packets are transmitted with random priorities in a scattershot fashion is not significant to the LAN's overall performance.

3. Subqueues

Because subqueues are essential to the development of the prioritization scheme, their construction is addressed first. As is seen in the FDDI station model in the Node Editor, the MAC node is represented as a queue. Therefore, only a change to "subqueue count" field in the attributes menu is necessary to change the MAC's structure into a bank of subqueues. Code is already in place that treats the MAC as a set of subqueues, although by
default, only one is available at first (this is labelled <subqueue (0)>, as is seen when the "subqueues" field is selected from the on-screen attributes menu for the MAC node. Nine subqueues are desired here: one for each asynchronous priority setting, plus another to handle synchronous traffic. Subqueue indexing corresponds to priority settings, so that subqueue (0) receives and releases the lowest priority asynchronous traffic, while subqueue (7) is assigned the highest priority traffic. Subqueue (8) is designated for synchronous traffic. This segregation of traffic into subqueues is necessary to support the recording of performance statistics and plots of traffic generation through the use of the Probe Editor. Also, while the Kernel Procedures (KP) available to the user include one that allows packets to be removed from the transmission queue in order of priority, rather than in the usual first-in-first-out (FIFO) order, subqueues allow simpler logic (Vols. 5.0 and 5.1 are directories of the commands and functions used by OPNET®). Vol. 2.0, Modeling Manual, "Nddef," pp. 27-29, describes the procedure to adjust code so that references to queues may be replaced with references to subqueues, particularly in relation to prioritization schemes. In summary, the subqueues represent a way point for packets. They are created and assigned a priority setting in the source, encapsulated for transmission in the MAC (ENCAP state), and placed in the appropriate subqueue while the station awaits the next token arrival to transmit them.

a. "RCV_TK"

In the RCV_TK (receive token) state, the first test of token usability is a determination of the presence of outbound traffic. The statement:

```c
if (op_q_stat(OPC_QSTAT_PKT_ID) > 0.0)  
   ... etc ...
```
calls for an inspection of the queue. This statement is replaced with a loop structure that searches all subqueues:

```c
for (i = NUM_PRIOS - 1; i > 1; i--)
{
    if (op_subq_stat (i, OPC_QSTAT_PKSIZE) > 0.0)
        ... etc ...
}
```

(Appendix E, line 453)

Of course the above requires a declaration of the variable NUM_PRIOS and the loop counter i.

b. "**TX_DATA**"

The TX_DATA (transmit data) state contains the code that transmits packets while the token remains available, and monitors THT. The THT (tht_value) is checked inside a loop whose condition is, "while packets remain in the queue, transmit." This loop contains most of the code in TX_DATA. This condition must be set inside another loop which counts through each desired subqueue, and which must include an additional number of "break loop" points. Accordingly, the code:

```c
while (op_q_stat (OPC_QSTAT_PKSIZE) > 0.0)
{
    /* Remove the next frame for transmission.*/
    pkptr = op_subq_pk_remove (0,OPC_QPOS_HEAD);
    ... etc ...
}
```

is rendered into the following:
for (i = NUM_PRIOS - 1; i > -1; i--)
{
    while (op_subq_stat (i, OPC_QSTAT_PKSIZE) > 0.0)
    {
        /* Remove the next frame for transmission.*/
        pkptr = op_subq_pk_remove (0, OPC_QPOS_HEAD);
        ... etc ...
    }
}

(Appendix E, line 920)

The transmission loop is broken when any of the following occur:

- No more packets are enqueued.
- In the case of synchronous transmission, insufficient bandwidth remains to complete a transmission. (Note that synchronous traffic is allocated by the user an inviolate amount of time in which to transmit, regardless of the lateness of the token. However, the model checks the bandwidth remaining to ensure a transmission can be completed within the allotted time, and will not commence a transmission that would delay the token. This is in agreement with the actual protocol, and in contrast to the asynchronous case, in which packet transmission may commence while the THT is active, even if the transmission will keep the token past THT expiration.)
- The remaining packets are of too low a priority to be transmitted in the time remaining to THT ($T_{Pr} + i < THT$).

After closure of the outer loop, the station deregisters its interest in the token by indicating it has no more traffic to send. This information is used by the token
acceleration mechanism, which will destroy the token for the time period no station has traffic
to transmit, then recreate it when needed, thereby significantly reducing the number of events
and the amount of time spent in a simulation. The original code appears at the bottom of the
TX_DATA state:

```c
if (tk_registered && op_q_stat (OPC_QSTAT_PKSIZE) == 0.0)
{
    tk_registered = 0;
    fddi_tk_deregister ();
}
```

As before, this must be altered to search through a set of subqueues first
before deciding no traffic remains to be sent:

```c
q_check = 1;
for (i = NUM_PRIOS - 1; i < -1; i--)
{
    if (op_subq_stat (i, OPC_QSTAT_PKSIZE) == 0.0)
    {
        q_check = 0;
    }
    else
    {
        q_check = 1;
        break;
    }
}
if (tk_registered && q_check == 0)
{
    tk_registered = 0;
    fddi_tk_deregister ();
}
```

(Appendix E, line 1084)
4. Modifications to Prioritization

In order to enact the priority scheme described above, changes are needed in the station model (Node Editor), all three process models (Process Editor), packet format (Parameter Editor), and to the Environment file (UNIX text editor).

a. Station Model Changes

For the modified priority scheme, new attributes are needed in the on-screen menus that appear for the mac node of the fddi_station model in the Node Editor. The original field "priority" is left in place, but not used. Note that the "super priority" field (described in Vol. 6.0, External Interfaces Manual, "Rel," p. 15) is not related to the FDDI protocol, but is a tool for scheduling of events in the simulation; it is not used, and should be left disabled.

In order to support the priority setting protocol that occurs in the source model (described later), two new attributes are created: "high pkt priority" and "low pkt priority." The new fields are created as listed in the following steps:

1. Call the on-screen attributes menu for the "mac" node.

2. Place the cursor over the "extended attrs" field, and press the left mouse button to call the submenu (the right mouse button dismisses the attributes menu; try again).

3. Assign the fields as shown in Figure 20: names will be "low pkt priority" and "high pkt priority", units are <none>, type is <integer> (selected from another on-screen menu that will appear.

58
(llc_src) Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin intrpt</td>
<td>enabled</td>
</tr>
<tr>
<td>endsim intrpt</td>
<td>disabled</td>
</tr>
<tr>
<td>failure intrpts</td>
<td>disabled</td>
</tr>
<tr>
<td>recovery intrpts</td>
<td>disabled</td>
</tr>
<tr>
<td>priority</td>
<td>0</td>
</tr>
<tr>
<td>super priority</td>
<td>disabled</td>
</tr>
<tr>
<td>icon name</td>
<td>processor</td>
</tr>
<tr>
<td>extended attrs.</td>
<td>--&gt;</td>
</tr>
</tbody>
</table>

Figure 20. Adding Extended Attributes to the Station Model
rather than typed in), and defaults are at the user's discretion. Zero for both are reasonable. These assignments are preserved with the keyboard combination <CTRL+S>. Further, the model must be saved using the "Write Node Model" icon, as described in Vol. 4.0, Tool Operations Manual, "Nd," pp.13-15.

4. Once the model is saved, then called again to the Node Editor, unexpected values will probably appear in the fields corresponding to the new attributes (which will now appear in the primary on-screen menu as well as in the "extended attrs" submenu). These values should be set to <promoted> so that they may be assigned in the Environment file. To set the newly created field to <promoted>, place the cursor over the field, then type <CNTL+O> at the keyboard, invoking the literal "promoted."

This last item is not described in the manuals; it was obtained from MIL 3, Inc.'s technical support via electronic mail.

b. Environment File Changes

Corresponding to the attributes created and then assigned <promoted> fields above, the following code is added to the Environment file:

"*.*.llc_src.high pkt priority" : 7
"*.*.llc_src.low pkt priority" : 0

The quotation marks are required here because of the spaces in the attribute names. Had low_pkt_priority been used instead, then the quotation marks would have been omitted. Examples of both styles appear in Appendix H, an example Environment file that includes all
attributes added to the model (some of which are still to be described). The Environment file may be modified in the UNIX text editor. Pound signs (#) indicate comments. Order of attributes is not significant, and any attributes not used by a model are simply ignored.

c. Source Modifications

The approach to assigning a random priority to each packet as its arrival is generated is functionally identical to the procedure by which the destination address is generated. The OPNET® kernel function op_ima_obj_attr_get() is used to call attributes from the node model or from the Environment file. The KP op_dist_load() is used to load a distribution to be used in generating a stream of stochastic values. These are used in the source process model "fdd1_gen" INIT state in the following manner:

\[
\text{op_ima_attr_get(my_id,"high pkt priority", } \\
\text{ &high_pkt_priority);} \\
\text{op_ima_attr_get(my_id,"low pkt priority" } \\
\text{ &low_pkt_priority);} \\
\text{pkt_priority_ptr = op_dist_load("uniform_int", } \\
\text{ low_pkt_priority, &high_pkt_priority);} \\
\]

(Appendix F, line 108)

In the preceding, the first two lines call the desired attributes from the Environment file to the calling station ("my_id"). The second field in the procedure call is taken verbatim from the Environment file, while the third field is the address of the attribute. The address may have any name; <&high_pkt_priority> is purely a memory aid for the user, and is not required by syntax to resemble the field name to which it is assigned. The value returned by op_dist_load() is used in the ARRIVAL state to finally generate the priority setting with the statement:

61
which finally returns an integer between the values set in the Environment file. This integer is assigned to the packet with the commands:

\[
\text{pkt\_prio} = \text{op\_dist\_outcome (pkt\_priority\_ptr)};
\]

(Appendix F, line 195)

In the SY (State Variable) edit window, high_pkt_priority and low_pkt_priority are declared as integers, and pkt_priority_ptr is declared as a pointer of type "Distribution." Because OPNET® uses a form of proto-C, the declarations made in the editor actually have the following form:
When the code is compiled and the ".C" icon is invoked to present the entire process model code, the above will have the following appearance:

```
Distribution* sv_pkt_priority_ptr;
int sv_high_pkt_priority;
int sv_low_pkt_priority;
```

OPNET® will also produce the following in the ".C" code:

```c
#define pkt_priority_ptr
   pr_state_ptr->sv_pkt_priority_ptr
#define high_pkt_priority
   pr_state_ptr->sv_high_pkt_priority
#define low_pkt_priority
   pr_state_ptr->sv_low_pkt_priority
```

The above commands have the effect of choosing, on a uniform distribution, a priority setting from a given range whose endpoints are retrieved from the Environment file.

\textit{d. MAC Modifications}

Because a priority scheme is already supported in the original model, little change is required in the \texttt{mac} node once a priority value is assigned in the source node. The user should note that communication between nodes is conducted with locally held variables; globals are avoided. This may result in different declared variable names for the same data,
which is acceptable. Thus, the pri of the source becomes the req_pri of the mac node. In the MAC node, priority settings are used as the indexes for the subqueues. NUM_PRIOS is declared for use as a loop counter.

e. Sink Node Modifications

The changes to the "fddl_slnk" process are nearly all related to the generation of performance parameters, which are discussed in the next section. In the original model, the received packet's priority setting is not even relayed to the sink, since the only information necessary to the calculation of overall throughput and delay statistics are the packet's creation time and its time of receipt. A fundamental addition to the code in the DISCARD state is the line:

\[
\text{op_pk_nfd_get (pkptr, "pri", &pri_set);} \\
\text{(Appendix F, line 78)}
\]

which recovers the priority from the field "pri" in the frame structure fddi_llc_fr. With this information, additional modifications will bring about the ability to create throughput and delay information for each asynchronous priority setting, and also to separate synchronous traffic statistics from asynchronous. But before any of this will work, the fddi_llc_fr packet structure must be modified.

f. Packet Format

The frame that is created in the source and passed to the MAC, then encapsulated into a more extensive frame, then ultimately passed from the destination station's MAC to the sink process for accounting and final destruction, is of the format fddi_llc_fr.
To support the prioritization scheme, the format needs to include more information than only the frame's creation time. To enhance its characteristics, the Parameter Editor's "Packet Format" icon is invoked, and the format fddi_11c_fr is called. Another line is added, as shown in Figure 21, making "priority" an attribute of the packet. Type is set to "integer," size can be "0," default value is "0," and default set is "unset." The changes are saved with the "Write Model" icon.

C. PERFORMANCE MEASURES

1. Overview

OPNET's original FDDI LAN model provides no ready way to monitor synchronous traffic separately from asynchronous, and no way to monitor the throughput and delay statistics for individual asynchronous levels. The inability of OPNET's original FDDI model to provide anything besides overall performance is a serious limitation to its usefulness in the CDL project. A major goal of this work is to augment the code in the sink process so that additional output vectors are generated, allowing the effects on individual class and priority levels to be seen. For example, in the original model, the user may assign any desired proportion of the generated traffic to be synchronous, but the original model has no facility to measure the synchronous traffic alone. The following paragraphs describe the modifications made to allow the display of statistics segregated by class and priority.

As an incidental note, the user should realize there is no particular significance to the sequence in which the states appear in the ".C" file. That is, the order DISCARD,
Figure 21. Adding a New Field to the "fddi_llc_fr" Frame Format
STATS, then INIT that appears in the "fddi_sink" process code is no indication of the sequence in which the simulation "visits" these states for each station. In fact, the more logical order, INIT, DISCARD, then STATS will be followed in this discussion, though preceded by a discussion of the variables needed. Appendix G is the file "fddi_sink_mult.pr.c", containing the modifications described here.

2. Variables

There are essentially four primary variables of interest in the sink process model: fddi_sink_accum_delay, fddi_sink_total_pkts, fddi_sink_total_bits, and fddi_sink_peak_delay. These exist as single integers or as floating point numbers, and are incremented or recomputed as packets are received by the station. The overall idea is to expand these into vector arrays, in which each element represents a running total for one priority setting, with the last element representing synchronous traffic totals. As mentioned earlier, this approach requires the "fddi_mac_fr" format in the Parameter Editor to be modified to include the priority as a field, since the original model needs only the packets' creation time and time of arrival in order to compute the overall throughput, mean delay, and end-to-end delay. However, as was discovered through trial and error, while the given variables can be changed to vector arrays, and the model can be modified to accommodate the new structure, and the code will compile (if the syntax is correct), any attempted simulation will abort with a segmentation violation error. This is because the "C" programs, which may be modified by the user, must interface with OPNET®'s kernel procedures, which are beyond the user's access. Ultimately, the given variables must be kept and new ones
created. Since the overall performance remains a useful statistic, the variables mentioned are left in place, while new ones are declared with the desired vector structure. These are fddi_sink_accum_delay_a, fddi_sink_total_pkts_a, fddi_sink_total_bits_a, and fddi_sink_peak_delay_a, which are declared and initialized in the Header Block.

The following declarations are added to the State Variables section:

```
Ghandle thru_gshandle_a[10];
Ghandle m_delay_gshandle_a[10];
Ghandle ete_delay_gshandle_a[9];
```

Once compiled, the following appear in the "C" file:

```
Ghandle sv_thru_gshandle_a[10];
Ghandle sv_m_delay_gshandle_a[10];
Ghandle sv_ete_delay_gshandle_a[9];
#define thru_gshandle_a pr_state_ptr->sv_thru_gshandle_a
#define m_delay_gshandle_a
    pr_state_ptr->sv_m_delay_gshandle_a
#define ete_delay_gshandle_a pr_state_ptr->sv_ete_delay_gshandle_a
```

(Appendix G, line 43)

With these declarations in place, the modifications to the rest of the code are straightforward, and generally follow the examples set by the original code. In addition, the variables Offered_Load and Asynch_Offered_Load are declared for use in generating scalar plots over a series of simulations. These are assigned values called from the Environment file by the state STATS.
3. Initialization State

The primary purpose of the Initialization State is to assign handles to the global statistics that are generated at the end of the simulation. The "KP" statement \texttt{op\_stat\_global\_reg (<gstat\_name>)} returns a handle used to reference a globally accessible statistic. This handle is needed to furnish new values (as they arrive with new packets) for the "KP" \texttt{op\_stat\_global\_write()}, which appears in the DISCARD state. The field entry \texttt{<gstat\_name>} is the text in the on-screen menu that appears in the Analysis Tool when the Create Single Vector Panel or "Create Multi Vector Panel" icons are invoked. The code added is very similar to what is already in place. For example:

\begin{verbatim}
thru_gshandle_a[0] = op_stat_global_reg("pri 1 thruput (bps)");
\end{verbatim}

(Appendix G, line 325)

creates a handle for the collection of data for priority 1 level throughput, and creates a field which will appear in the on-screen menu in the Analysis Tool. In another example:

\begin{verbatim}
m_delay_gshandle_a[9] = op_stat_global_reg("async mean delay (sec)");
\end{verbatim}

(Appendix G, line 364)

creates a handle for total asynchronous mean delay, and generates another field which will appear in the on-screen menu in the Analysis Tool. Each priority level has its corresponding handle assignment line. The actual statistics to accompany these handles are generated in the DISCARD state. As seen in the code itself (Appendix G), each element has a handle assignment line, for throughput, mean delay, and end-to-end delay. Figure 22 shows off the
resulting on-screen menu from the "Create Single Vector Panel" icon, reflecting the new data that may be plotted. Note that the code here reconciles "C" language vector numbering conventions with real-world priority level settings.

4. "DISCARD" State

The DISCARD state is where the received packet is "opened" and statistics generated from the packet contents. As mentioned before, "KP op_stat_global_write (<gstat_handle>, <value>)" is the statement that accumulates data. The <value> field may be a previously computed figure, or may be calculated within the "KP". The <gstat-handle> field is the same used in the INIT state. DISCARD uses the priority value found with the arriving packet as the index for the vector structure. For example:

```
  op_stat_global_write (thru_gshandle_a[5],
                     fddi_sink_total_bits_a[5] / op_sim_time());

  (Appendix G, line 128)
```

generates a current throughput figure for asynchronous priority level six (recall the necessary offset for vector element numbering convention) Also necessary is the recording of delay values for each priority, which is done with the following:

```
  op_stat_global_write (ete_delay_gshandle_a[pri_set],
                        delay);

  (Appendix G, line 182)
```

This state also destroys the packet once its contents are recorded. This is necessary to prevent the simulation from filling the host computer's memory with dead packets.
Figure 22. Newly Created Vector Traces Available
5. "STATS" State

The STATS state produces the steady-state scalar data that may be plotted using the "Create Scalar Panel" icon. These are saved, rather than written over, so that the user may observe changes to output as the input is varied. For example, the throughput of synchronous traffic over several simulations as different TTRT values are used. From these, a plot of throughput vs. TTRT may be generated. The statement:

```
op_stat_scalar_write (<scstat_name>, <value>);
```

is similar to the "write" command described before, and writes a scalar steady-state statistic in this case. The field "scstat" appears in the on-screen menu called with the "Create Scalar Panel" icon. Examples of the use of this statement appear in Appendix G, lines 210-226.

Another on-screen menu line item is drawn from the Environment file. The values <total_offered_load> and <asynch_offered_load> are placed and assigned in the Environment file (see Appendix H), as described earlier. These correspond to the
variables Asynch_Offered_Load and Offered_Load declared in the header block. These are joined by the commands:

```c
op_ima_sim_attr_get (OPCIMA_DOUBLE, "total_offered_load", &Offered_Load);
op_ima_sim_attr_get (OPCIMA_DOUBLE, "asynch_offered_load", &Asynch_Offered_Load);
```

(Appendix G, line 299)

and added to the on_screen menu with the commands:

```c
op_stat_scalar_write ("Total Offered Load (Mbps)", Offered_Load);
op_stat_scalar_write ("Asynchronous Offered Load (Mbps)", Asynch_Offered_Load);
```

(Appendix G, line 305)

This code in Appendix G contains a warning to the user that the offered load settings are not automatically updated in any way. If the user desires to plot throughout or delay as a function of offered load over a series of simulations, then the user must remember to keep the offered load assignments current in the Environment file for each simulation.

**D. BRIDGE LINK**

The alteration described here represents a simple first step toward a network interface. Instead of destroying frames after they are received, one station on the LAN is modified to hold its packets in subqueues. Further development will bring about a protocol for removing these buffered packets from the original LAN and transferring them to another.
1. **Station Model Modifications**

The received frames are stored in subqueues according to their priority in a manner analogous to that already described for the mac node. This requires that the sink node, llc_sink, be changed from its original processor form into a queue node. To affect this modification to the station model, the following steps are followed:

1. The sink node is selected by placing the cursor over it and clicking the left mouse button.

2. The node is removed by invoking the "Cut" icon. The "Packet Stream" between the sink and the mac also disappears.

3. The "Create Queue" icon is selected, and the resulting box is dragged to the location just vacated. Clicking the left mouse button places the new node. The station now has a queue node rather than a processor node.

4. The on-screen attributes menus is called with the right mouse button, and the fields are all set to the same values that were in effect before, including the node name.

5. The "process model" field will be set to the newly modified sink process model. If the process has not yet been modified, then the original assignment may be used, then changed when the process has been modified and saved under a new filename.

6. The "subqueues" field is set to <9>, accommodating eight levels of asynchronous traffic and also synchronous traffic.
7. The packet stream line between the sink and the mac nodes must be replaced.

8. The new station model is saved under a new filename, for example, "fddi_sink_link".

Figure 23 shows the resulting station model, with the appropriate on-screen attributes menu.

2. Process Model Modifications

The changes to the process model code are few, and are included in Appendix G as inactive code ("commented out"). The code that destroys received packets:

```c
op_pk_destroy (pkptr);
```

(Appendix G, line 98)

is replaced by code that enqueues the packets according to their priority settings:

```c
op_subq_pk_insert (pri_set, pkptr, OPC_OPOS_TAIL);
```

(Appendix G, line 105)

The user should realize that for the moment, no more code exists for the disposition of these enqueued packets; a long simulation simply accumulates packets and fills computer memory. The subqueues are infinite by default, but may be limited (using another on-screen menu) to demonstrate overflows. In that case, there is no code for the disposition of packets that are lost through buffer overflow, and these will simply accumulate in the host computer's memory as well. In sum, the user must be aware of the memory demands of OPNET® simulations.
Figure 23. Modified Station Model, "fddi_sink_link"
E. MULTICAST

1. Overview

Multicast is the addressing of a packet to more than one station. Broadcast is a special case with the transmission of the packet to all stations. In the original model, a station desiring to send the same message to all stations would repeat the packet transmission for each destination station. Actually, this last is an abstraction; OPNET® simply generates packets from each station addressed to randomly generated destinations, with no indication that any particular transmission represents a copy of any previous transmission. However, the fact remains that each packet is addressed to only one station. In the actual FDDI protocol, each packet is passed from station to station until it reaches its destination, but then continues past its destination until it is finally removed from the ring by its originating station. The OPNET® simulation economizes on the number of simulation events (and therefore the simulation time) by having the destination stations remove the packets they receive. This occurs in the mac state FR_REPEAT, which also contains comments suggesting that the user may wish to overrule this economizing feature in the event that group addressing is desired. Figure 24 is the State Transition Diagram for the mac process, repeated from Figure 6 for the reader's reference. The state FR_STRIP includes the code by which an originating station removes packets that have completed a circuit of the LAN. It is not used in the original code, nor will it be used in the modifications described here.
Figure 24. MAC Process Model State Transition Diagram
The basic idea is as follows: rather than carry a destination address, each packet carries an array with a number of elements equal to the number of stations on the ring. These elements are simply ones and zeros, with a one indicating by its location that the packet is addressed to a station corresponding to that location. For example, in a five station LAN, the address field $[0 \ 1 \ 0 \ 1 \ 1]$ would indicate the packet is addressed to stations one, three, and four (as in the case of indexing vector array elements, the stations on an N-station LAN in OPNET® are numbered from zero to N-1). As the packet is passed around the LAN, each station inspects this array, passing the packet on if the station is not designated a destination address. Destination addresses keep a copy of the packet's information, set their place in the address field to zero, then pass the packet on to the next station. The last destination address will destroy the packet after verification that only zeros remain in the destination address array, thereby preserving some of the economy gained in minimizing the number of events in the simulation. While the destination address array would need to be transported with each packet, OPNET®'s Kernel Procedures can only accommodate a pointer to the array. The following sections describe the changes necessary to effect multiple addressing. The reader is again referred to Appendices E, F and G, containing the ".C" files for the MAC, source, and sink process models, respectively. However, the implementation of multicasting involves no changes to the sink process model.
2. Environment File

Each station is assigned a maximum and a minimum number of possible destination addresses to use in addressing each packet. The following are added to the Environment file:

```
"*.llc_src.min num addees": <user assigned integer>
"*.llc_src.max num addees": <user assigned integer>
```

These will be called by the source process model's INIT state. Appendix H contains an example Environment file including these new attributes. The minimum number must be at least one, and the maximum should be no greater than N-1, where N is the number of stations (the logic written in the source model's code does not allow stations to address packets to themselves). If it is desired that some station generate no traffic, then the "arrival rate" field should be set to zero. Setting "min num addees" and "max num addees" to zero will only result in packets transmitted with no destination addresses assigned. Setting "max num addees" to a number greater than N-1 will cause an endless loop in the code that generates destination address assignments in the source process (the logic in the loop is, "assign x different destinations, but do not repeat any.").

3. Station Model

The additions to the environment file must be added to the station model in the Node Editor, in a manner analogous to the method described in the discussion of prioritization (III.B.4.a. Station Model Changes). The steps required to add another attribute to the on-screen menu are not recounted here, but the desired final result is shown in
Figure 25. It is a good practice to save the changed station model under a new name, for example "fddi_sta_mult", until the user is certain that the modifications do more good than harm to the original model.

4. Source Process Model

A real packet on a real LAN would necessarily be self-contained, carrying with it all its destination addresses. However, the functions used to assign the packet address field in OPNET® will not support a vector structure, and so pointers to memory locations must be used, with these memory locations containing the destination addresses. The modifications to the source process to effect multiple addressing are summarized in the following sections. As is the case with the station node model, each of the process models should be saved under new names, for example fddi_gen_mult, as a matter of practice.

a. Variables

To begin, a global variable, NUM_STATIONS, is defined in the Header Block (it is also defined in the mac process model), to be used as a loop counter. This variable must be kept updated to accurately reflect the correct number of stations on the FDDI LAN. This is easily forgotten when different LANs are created, using the same station model with different numbers of stations.

As was mentioned, the destination address field, originally a single integer value, must now be made a pointer to an array of integers. Although "C" programming language treats the name of a vector array as the array's pointer, the established Kernel Procedures do not support the simple change of syntax. In short, a new variable, *da_ptr (destination address pointer), is declared as an integer pointer in the Temporary Variables
Figure 25. Multicast-Capable FDDI Station with Attributes Menus
(TV) editor. The original variable, dest_addr, is removed from the TV section, and declared with the State Variables (SV) as an array of integers, with dimensions \([\text{NUM\_STATIONS} + 1]\).

In addition, variables to accompany the Environment file attributes are declared with the State Variables: min_num_addees and max_num_addees. The resemblance between these variables and the attributes they accompany is meaningless in respect to "C" language syntax, but is of course a useful memory aid to the user. The pointer num_addees_dist_ptr represents the value used in determining stochastic values. Also, the integer num_addees is declared in the State Variables, to represent the number of stations to which a given packet will be assigned. It is used as a loop counter, and will be different for each packet.

b. Initialization State

In the source model's Initialization state, the range bounding the number of destination addresses is determined with a call to the Environment file:

```c
opimaobjattr_get(my_id, "min num addees", &min_num_addees);
opimaobjattr_get(my_id, "max num addees", &max_num_addees);
```

(Appendix F, line 86)
These result in the assignment of the values from the Environment file to the addresses of the corresponding variables. A distribution is established with the following:

\[
\text{num\_addees\_dist\_ptr} = \text{op\_dist\_load ("uniform\_int", min\_num\_addees, max\_num\_addees)};
\]

(Appendix F, line 114)

This value is used to generate streams of stochastic values, and is used in the ARRIVAL state.

c. "ARRIVAL" State

In the ARRIVAL state, the actual number of stations to receive the new packet is determined, and then a loop is used to choose these stations one at a time. Each loop iteration is very similar to the original procedure that was in place when only one station was assigned to each packet. The loop contains a provision to prevent the repeated assignment of the same station, and also to prevent the assignment of the originating station as a destination address. The following statement determines the number of destination addresses for a given packet

\[
\text{num\_addees} = \text{op\_dist\_outcome (num\_addees\_dist\_ptr)};
\]

(Appendix F, line 172)
The following loop is used to find and assign the chosen stations:

```c
for (i = num_addees; i > 0; i--)
{
    gen_packet;
    nix = op_dist_outcome (dest_dist_ptr);
    if (dest_addr[nix] == 1 || nix == station_addr)
    {
        goto gen_packet;
    }
    dest_addr[nix] = 1;
}
```

(Appendix F, line 174)

This loop continues to iterate until the specified number of stations, without repetition, is assigned.

Recall that the destination address array is declared with one element more than the number of stations in the LAN (dest_addr [NUM_STATIONS + 1]). Here, all the elements in the destination array must be shifted one space to the right, and a simple loop is used to set dest_addr[i+1] equal to dest_addr[i], for i iterations. This step is necessary because the first array element will be overwritten with the array's memory address in the course of the packet's travels, as it is transmitted from one station, received by the next, and its destination address field is opened, inspected, then closed by each station in turn. This behavior is verified by use of the debug tool set to "fulltrace." accompanied by strategically placed printf statements. The author does not pretend to know why this happens. The array element shift is a deft enough way to sidestep the problem. However, all references in other states to the destination address array must be offset to accommodate this shift.
5. MAC Process Model

The MAC process model receives each packet, inspects it, the decides whether or not the packet is addressed to the station. In the original model, the packet is removed from the ring by the destination station, and relayed by other stations. With multicasting of packets, a third case arises, in which a station receives a packet addressed to it, but must also pass the packet on to other destination stations. A number of print commands are placed in the code, but left inactive. They are of much use in the verification of the model’s operation.

a. Variables

The same variable NUM_STATIONS used in the source model is also defined in the Header Block of the MAC. The user must remember to keep this value updated in both places when the same station model is used in a different size LAN. The destination address is changed from an integer into an open-ended array of integers, dest_addr[], in the Temporary Variables editor. An integer pointer, *da_ptr, is declared as well.

b. Encapsulation State

The Encapsulation state receives frames from the source process, and place them inside the format fddi_mac_fr for transmission on the LAN. An intermediate step is to inspect the frame received from the source for its destination address, which must be written into the encapsulating frame’s destination address field as well. The original statement:

```c
op_ici_attr_get (ici_ptr, "dest_addr", &dest_addr);
```
is unworkable with `dest_addr` in array form, which is why the pointer `*da_ptr` is declared. Instead, the following is used:

```c
op_ici_attr_get (ici_ptr, "dest_addr", &da_ptr);
```

(Appendix E, line 806)

followed by a loop assigning each element in the array a value from the corresponding element in the array found at address `da_ptr`. Vol. 5.0, *Simulation Kernel Manual*, discusses this command statement. This loop uses as a counter the value `NUM_STATIONS+1`, for the reason mentioned earlier: use of `printf` statement would reveal that the first element in the array has been written over and replaced with a number representing the memory address of the array. Fortunately, the entire original array of zeros and ones has been shifted, so the first element is intact. Correspondingly, all references to the array from within the MAC process must be made with respect to this shift.

c. **Frame Repeat State**

The Frame Repeat state inspects each received packet and acts on one of three cases: the packet is addressed only to this station, or the packet is not addressed to this station at all, or the packet is addressed to this station and to other stations as well. The first two cases are already present in the original model, and require some simple modifications. The third case represents a significant change, requiring the addition of an entire block of code to the state, in which the packet information is copied first, then passed on to the next station.
The first statement in the FR_REPEAT state opens the arriving packet's address field for inspection:

\[
\text{op_pk_nfd_get (pkptr, "dest addr", &da_ptr);
}\]

(Appendix E, line 604)

Here, \&da_ptr has been substituted for the original \&dest_addr. As is the case in the ENCAP state, the arriving pointer is used to initialize the locally held destination address array:

\[
\text{for (i = 0; i < NUM_STATIONS+1; i++)}
\text{dest_addr[i] = da_ptr[i];}
\]

(Appendix E, line 610)

This destination address array is then passed through a loop to determine if it has more than one destination address, and to see if the element corresponding to this station is set to one, indicating the packet is addressed to this station. If the packet proves to be addressed to this station only, then the actions of the original code are carried out: relevant fields are copied to an ICI packet format for transmission to the sink process, then the packet is destroyed. If inspection of the destination address array shows the packet is not addressed to this station at all, the original code is again sufficient to place the packet back on the ring.

The third case represents a new situation. When the packet is addressed to this station and to others as well, the information must be saved here, and be transmitted onto the ring again. These actions are carried out with commands borrowed from the first two cases, and include some new considerations as well. The function "op_pk_nfd_get()"
is used to retrieve data from specified fields in the packet. That is, it is a decapsulation function. When the data happens to be in a structure format, the function has the effect of destroying the information. This is an important point because the information must be preserved for retransmission. Therefore the function:

\[
\text{op_pk_nfd_get (pkptr, "info", &data_pkptr);}
\]

(Appendix E, line 719)

must be followed at some point with the statement:

\[
\text{info_ptr = op_pk_copy (data_pkptr);}
\]

(Appendix E, line 727)

in which \text{info_ptr} has been declared in the \text{Temporary Variables} to have the same type as the structure in the "info" field. When this information is summoned for re-encapsulation with the function:

\[
\text{op_pk_nfd_set (pkptr, "info", info_ptr);}
\]

(Appendix E, line 747)

the field information has been preserved. The other fields, "src_addr", "dest_addr", and "pri", do not require this procedure, since they are not lost with decapsulation. Two packets result: one is an ICI frame carrying the received information to the sink process, and the other is a re-encapsulated packet sent to the LAN. The station's last action before
re-encapsulating the destination address array is to zero its corresponding address element.
The last station to receive the packet will destroy it, upon determining that only zeros remain
in the destination address array (Appendix E, lines 644 and 692).

6. Limitation

The primary limitation of the model with multicasting active is in the generation
of throughput and delay statistics. As the code currently stands, each packet is counted by
every station that receives it, leading to multiplication of throughput data. On the other hand,
some use may possibly be made of this characteristic by comparing the tallied throughput
against the actual offered load, as a measure of the effectiveness of the multicasting scheme.
For example, a measured throughput of 100 Mbps versus a known offered load of 50 Mbps
could indicate that multicasting effectively generates 50 Mbps without physically taxing the
bandwidth capability of the FDDI LAN, since no additional packets are generated.
IV. MODEL TESTING

This chapter provides several test results intended to verify the enhanced capabilities added to the basic FDDI station model in OPNET®. In addition to improving the performance and display features, the modifications must preserve the basic behavior to be useful. The tests presented here include an illustration of the model's treatment of synchronous and priority-based asynchronous traffic, a simple demonstration that the modified sink process does indeed store the traffic it receives, a check against theoretical performance equations, and a demonstration on the monitoring of multicast.

Not all of the modifications described in this thesis are incorporated simultaneously in all station models. The State Transition Diagram correction to the original fddi_sink process model is of course installed in all models that use the llc_sink node. Likewise, the patch for OPBUG is installed for all MAC model versions. The generation of randomly differing priority assignments for all packets is closely interrelated to the generation of output statistics segregated by class and priority; both involve extensive modifications to the original process models for the source, sink and MAC, which are stored as fdd1_gen, fdd1_mac, and fdd1_sink. (The actual file names found in the UNIX subdirectory have the suffix ".pr.m," and when these are compiled, corresponding files with the suffix ".pr.c" are created.) The original models of those names are retired under the ending "_orig.pr.m." Likewise, the original node-level station is stored under the name fdd1_sta_orig, while fdd1_station is the upgraded
version. The multicasting capability is stored in a separate station model, fdd1_sta_mult, which includes the processes fdd1_mac_mult, fdd1_gen_mult, and fdd1_sink_mult, and contains all of the other modifications as well. The multicast capability was the last feature installed for this thesis, and has not been completely developed.

A. SYNCHRONOUS THROUGHPUT

1. Overview

This test was motivated by an earlier study, in which a software simulation tool was used to demonstrate the effect of increasing offered load on the asynchronous and synchronous throughput of an FDDI LAN. Using the Block Oriented Network Simulator (BONeS®), Tari et. al. were able to show that synchronous throughput does not degrade appreciably, even when the offered load is well in excess of 100 Mbps. They were able further to show the decay in throughput suffered in the asynchronous priority levels as offered load is increased. Figure 26 is taken from this study. In addition to providing an idea of expected performance, this figure also highlights the fact that in its original form, OPNET®'s FDA model provides no method to display any throughput data other than the overall performance parameters, total throughput, total mean delay, and total end-to-end delay.

2. Setup

The set up was intended to imitate the older experiment as closely as possible. Ten simulations were conducted using the same 50-station LAN. Ten stations (f0, f5, f10, f15, f20, f25, f30, f35, f40, f45) were designated "constant" generators.
Figure 26. Throughput Measurements Using BONeS® (Tari, et al., 1988, p. 58)
That is, they were modified inside the source process "C" code to generate packets at a constant rate, rather than at an exponentially distributed approximation of the rate specified in the Environment file. These ten stations transmitted synchronous traffic only, while the other 40 stations were allocated no synchronous bandwidth, sending asynchronous traffic only. The synchronous stations generated 512-bit packets at a constant arrival rate of 6000 packets per second. The asynchronous stations transmitted 1000-bit packets, at a rate stepped so as to increase the total offered load by 10 Mbps in each simulation. Therefore, total offered load ranged from 40.72 Mbps to 140.72 Mbps over ten tests, with steady state data recorded in a data file of scalar data. TTRT was set to 10.7 ms, in order to support the delay of 50 stations on a 50 km ring while maintaining $\Sigma SA_i \geq 10$ ms. This differed from the older study, for which a TTRT value of 8.0 ms was used, despite the authors' stated intention to model voice network traffic. As noted by Powers (1993, p.340), synchronous voice transmission requires that the token visit each voice transmitter every 20 milliseconds, resulting in $\Sigma SA_i = 10$ ms. This in turn requires TTRT to exceed 10 ms by an amount sufficient to account for physical delays inherent to the ring fiber and the stations.

Asynchronous priority threshold levels. $T_{Pri} \{1\}$, in the INIT state of the MAC process code were set in increments of 0.125 TTRT, so that Priority 1 traffic transmission was cut off when THT incremented to 1.3375 ms, and Priority 8 traffic could be sent for the entire THT period. Recall that this incrementing THT is a function of OPNET, reversed from the decrementing timer described in most literature.
3. Results

Figure 27 shows a plot of total throughput over a series of ten simulations in OPNET, demonstrating a roughly linear rise until the offered load begins to exceed 90 Mbps, which is in qualitative agreement with the older experiment, and agrees with results published by Dykeman and Bux (1988, pp. 1003-1007). This plot also serves to show off one of the model's improvements in data display, allowing "Offered Load" to be an abscissa for scalar plots.

Figure 28 illustrates the fact that synchronous bandwidth allocation is not affected by the asynchronous offered load; therefore, synchronous throughput remains nearly constant as the offered load is increased.

Figures 29-31 illustrate the effect of increasing offered load on the throughputs of asynchronous traffic at priority levels two, three, and four, respectively. No priority settings above four suffered any degradation within the range of offered load observed in this test. The decay of Priority 3 traffic at approximately 100 Mbps of offered load suggests that (\(3/a\))TTRT may be a guiding point for setting priority thresholds that will take effect before the LAN's capacity is reached.

B. PRELIMINARY LINKING MODEL

1. Overview

The modifications made to the sink process as a step toward a bridging node are the simplest of all described in this thesis, and the testing of the finished model is also simple.
Figure 27. Total Throughput vs. Total Offered Load

Figure 28. Synchronous Throughput vs. Total Offered Load
Figure 29. Priority Two Throughput vs. Total Offered Load

Figure 30. Priority Three Throughput vs. Total Offered Load
Figure 31. Priority Four Throughput vs. Total Offered Load
It is necessary only to show that all the traffic transmitted to the station is received and held.

To demonstrate this action, a ten-station LAN was created, with station f9 designated the link node. The Environment file assignments were made so that all stations directed all traffic to station f9, which itself was not transmitting. A modest offered load was used to prevent an overload of packets in the simulation host computer's memory. All transmitting stations were assigned the same full range of asynchronous priority settings, and were assigned equal portions of synchronous bandwidth. Each of the buffers at station f9 should have then been seen to receive packets pre-sorted for eventual transmission. The Probe Editor was used to monitor \texttt{"pksize,"} the number of packets in each subqueue of the modified sink process model.

2. Setup

The following calculations and Environment file settings in Table 1 were chosen as reasonable:

<table>
<thead>
<tr>
<th>Table 1. ENVIRONMENTAL FILE SETTINGS.</th>
</tr>
</thead>
<tbody>
<tr>
<td>packet size: 2000 bits</td>
</tr>
<tr>
<td>arrival rate: 10 pkt./sec.</td>
</tr>
<tr>
<td>offered load: $9 \times 10 \times 2000 = 180,000$ bps.</td>
</tr>
<tr>
<td>TTRT ($T_{Req}$): 0.004 sec.</td>
</tr>
<tr>
<td>async mix: 0.9</td>
</tr>
<tr>
<td>prop delay: $5.085 \times 10^{-6}$ sec./km. $\times$ 1 km.) = $5.085 \times 10^{-6}$ sec.</td>
</tr>
<tr>
<td>station latency: $60.0 \times 10^{-6}$ sec.</td>
</tr>
<tr>
<td>$D_{Max}$: $(5.085 \mu\text{sec.} \times 10) + (60.0 \times 10^{-6} \times 10) = 0.05685$ ms.</td>
</tr>
<tr>
<td>$F_{Max}$: 0.360 ms.</td>
</tr>
<tr>
<td>Token Time: 0.00088 ms.</td>
</tr>
<tr>
<td>synchronous BW: $4-(0.05685 + 0.360 + 0.00088) = 4 - 0.41773 = 3.58227$ ms.</td>
</tr>
</tbody>
</table>
Dividing bandwidth evenly among 9 stations gives the following:

$$\frac{3.58227 \text{ ms.}}{9 \text{ stations}} = 0.39803 \text{ ms./station.}$$  \hspace{1cm} (2)$$

This result is compared with TTRT to determine the "sync bandwidth" attribute:

$$\frac{0.39803 \text{ ms.}}{4.0 \text{ ms.}} = 0.0995075 \text{ ms.}$$  \hspace{1cm} (3)$$

which is a unitless fraction of TTRT.

3. RESULTS

The given parameters were applied, and the receiving buffers were inspected at the end of one second of simulation time. Figure 32 is the plot obtained illustrating the accumulation of packets in all nine subqueues. (Although all the plots may be placed in one panel, the data are divided into two panels for readability of the hardcopy). The plots show the following after one second:

<table>
<thead>
<tr>
<th>Subqueue</th>
<th>Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>subqueue(0)</td>
<td>13 packets</td>
</tr>
<tr>
<td>subqueue(1)</td>
<td>11 packets</td>
</tr>
<tr>
<td>subqueue(2)</td>
<td>10 packets</td>
</tr>
<tr>
<td>subqueue(3)</td>
<td>7 packets</td>
</tr>
<tr>
<td>subqueue(4)</td>
<td>7 packets</td>
</tr>
<tr>
<td>subqueue(5)</td>
<td>12 packets</td>
</tr>
<tr>
<td>subqueue(6)</td>
<td>11 packets</td>
</tr>
<tr>
<td>subqueue(7)</td>
<td>10 packets</td>
</tr>
<tr>
<td>subqueue(8)</td>
<td>9 packets</td>
</tr>
</tbody>
</table>

\[ \text{total: 90 packets} \]

This agrees with the given packet generation rate of 90 packets/sec., and verifies that the preliminary linking model holds all received packets on station.
Figure 32. Packet Accumulation at Linking Model "fddi_sink_link"
C. SYNCHRONOUS TIMING

1. Overview

In order to preserve the real-time nature of synchronous traffic, the bandwidth allocated to a given station may not be exceeded. Although transmission time (bandwidth) may remain to a station, the protocol determines a priori whether the next packet transmission would cause the allotted bandwidth to be exceeded. If the bandwidth limit would be exceeded, then the station will not transmit. The code in the TX_DATA state of the MAC process ensures OPNET's FDDI LAN models adhere to this behavior, as shown in the following simple experiment.

2. Setup

A new LAN was created, using 13 stations on 91 km. of fiber, resulting in seven kilometers of fiber between stations. The odd figures were chosen in order to avoid symmetries in the arithmetic involved, thereby enhancing the instructional value of the test. All 13 stations were assigned an equal portion of the total available bandwidth for synchronous traffic, and all stations transmitted only synchronous traffic. Physical attributes were identical for each station. The following calculations apply:
Table 3. 13-STATION LAN ENVIRONMENT SETTINGS.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>prop_delay</td>
<td>$7 \text{ km} \times 5.085 \times 10^{-6} \text{ sec./km.} = 0.0355950 \text{ ms.}$</td>
</tr>
<tr>
<td>station latency</td>
<td>$60.0 \times 10^{4} \text{ sec./station}$</td>
</tr>
<tr>
<td>F_Max</td>
<td>0.360 ms.</td>
</tr>
<tr>
<td>D_Max</td>
<td>$(13 \text{ stations} \times 60.0 \times 10^{4} \text{ sec./station}) + (13 \text{ links} \times 0.035595 \text{ ms./link}) = 0.4627428 \text{ ms.}$</td>
</tr>
<tr>
<td>Token_Time</td>
<td>0.88 $\mu$sec.</td>
</tr>
<tr>
<td>TTRT</td>
<td>4.0 ms.</td>
</tr>
</tbody>
</table>

$D_{\text{Max}} + F_{\text{Max}} + \text{Token\_Time} = 0.8236228 \text{ ms.}$

Divide the synchronous allotment evenly among 13 stations:

$3.1763772 \text{ ms./13 stations} = 0.2443367 \text{ ms./station}.$

Given the standard transmission rate of 100 Mbps, each station may transmit 24,433 bits with its synchronous allotment. This bandwidth is converted to a fraction for the Environment file attribute "sync bandwidth":

$$\frac{0.2443367 \text{ ms.}}{4.0 \text{ ms.}} = 0.06108418 \quad (4).$$

For this test, all stations were constant transmitters. That is, the code for the source process was adjusted so that packet transmission rates and packet lengths were assigned invariant values, rather than stochastic approximations of the attributes assigned. A TTRT of 4 ms indicates 250 token passes per second, which was chosen as the packet arrival rate for all stations. Packet size was 24,000 bits, resulting in a total offered load of 78.0 Mbps. A larger packet size, for example 25,000 bits, should be too large to transmit, resulting in no throughput at all.
3. Results

Figure 33 shows the resulting total throughput derived from the given setup. Invoking the "convert to text" attribute of the on-screen menu indicates a value of 77.7 Mbps after one second, with a slow rise still in progress. This is in close agreement with the offered load.

Figure 34 is an empty panel, accompanied by a text screen indicating that no throughput results when the packet size exceeds that allowed by the synchronous bandwidth assignment. In this case, packet size was increased to 25,000 bits. Arrival rate was also reduced to 100 packets/sec., resulting in a total offered load of 32.5 Mbps. Figure 35 illustrates the accompanying accumulation of packets in station f3's subqueue(8), which is reserved for synchronous traffic. The perfectly linear shape is a result of the constant arrival rate assignment.

D. ASYNCHRONOUS EFFICIENCY

1. Overview

A network's throughput efficiency is calculated from the following equations, which assume that only asynchronous traffic is being transmitted:
\[ E = \frac{N(T-D)}{(TN+D)} \] (5).

where:

- \( N \) = number of stations,
- \( T = TTRT \) and,
- \( D = \) ring latency (total time required for a token to circulate the ring in the absence of data traffic).

Ring latency \( D \) is in turn defined as

\[ D = L\left(\frac{\pi}{c}\right) + N\cdot T_s \] (6).

where:

- \( L = \) length of ring in kilometers
- \( ^c/\pi = 5.085 \times 10^{-6} \text{ sec./m.} \)
- \( T_s = \) token processing time (0.88 \( \mu \text{s.} \))

(Powers, 1993, pp. 336-337; uses \( T_s = 1.0 \mu \text{s. as a typical value} \)

The same 13 station, 91 km LAN described previously was used here, with the Environment file adjusted so that no synchronous bandwidth was assigned, and only asynchronous traffic was generated. Offered load was 78.0 Mbps. The given equations indicate the expected efficiency is 87.44%.
Figure 33. Synchronous Throughput: BW Exceeds Packet Transmission Time

Figure 34. Synchronous Throughput: Packet Transmission Time Exceeds BW
2. Results

Figure 36 illustrates the resulting throughput, 68.48 Mbps, which is 87.79% of the offered load. This agrees quite well with the predicted throughput.

E. MULTICASTING

1. Overview

The multicasting function is designed to assign to each packet a randomly chosen number of destination addresses, then to use this selected number as an index for a loop in which a different destination address is assigned on each iteration. Upon completion of the loop, a vector array of ones and zeros has been created, in which a one in position \( i \) indicates the packet is addressed to station \( f_i \). Of course, a real transmitter would send many consecutive packets to the same set of addresses, in streams that comprise messages. However, this modification is in keeping with the original model's actions, which randomly assigned destinations on a packet-by-packet basis.

Preliminary tests indicated the model is not fully developed. Because no change was made to the statistics generation mechanism in the sink process model, the new model as written should have resulted in packets being counted toward total throughput each time they were received at a destination, giving an inflated throughput computation. On the other hand, this inflated figure could possibly be of value as a measure of "virtual throughput." In any event, tests indicate that throughput statistics are not being counted correctly. However, a study of the model's behavior using the debug tool in conjunction with "\texttt{printf}" statements
Figure 35. Accumulating Synchronous Packets in MAC

Figure 36. Asynchronous Throughput
indicated that no packets were being lost, and all stations seemed to carry out correctly the operation of receiving a multicast packet, saving the information, and passing the packet on to the next station.

Three tests are described here. The first is an attempt to reproduce the results generated for one of the simulations used in Section A, then uses the same input parameters to see if any differences exist in the handling of synchronous-only and asynchronous-only traffic. The second test compares all synchronous throughput with all asynchronous throughput. The third test is a verification that no packets are actually being lost. The fourth test is an attempt to generate a plot of expected throughput when one station broadcasts all its traffic.

The tests generally indicate that the multicast capable model, when limited to single addressing, does not behave in the same manner as the single-address-only model. In particular, throughput is lower than expected. Further development will be necessary to make the multicasting model a reliable tool.

2. First Test

a. Setup

Figure 37 displays the throughput plots resulting from the third run in the set of simulations used to construct Figure 27. This test is based on the idea that if the multicast-capable model is limited to single addressing, and given the same input parameters provided in the earlier test, then the resulting throughput ought to be identical for total asynchronous and synchronous traffic. Therefore, the following inputs were used for a 50 station, 50 km. LAN using the multicasting station model.
Ten stations (f0, f5, f10, f15, f20, f25, f30, f35, f40, f45) transmit only synchronous traffic in 512-bit packets with a constant arrival rate of 6000 packets/sec, resulting in a synchronous offered load of 30.72 Mbps. The remaining 40 stations transmit asynchronous traffic only, in 1000-bit packets at an arrival rate of 750 packets/sec, for an asynchronous offered load of 30 Mbps. TTRT is set to 10.7 ms. For all stations, the Environment file attributes "min num addees" and "max num addees" are both set to 1, enforcing the limit of one destination address per packet. Appendix H shows the Environment file used here.

b. Results

Figure 38 illustrates the resulting throughput. Table 1 summarizes the throughput results for the two simulations. The unpredictable throughput of the multicast-capable model's synchronous and asynchronous modes would suggest some logic error in coding. The reduction in overall throughput suggests perhaps some difficulty with the simulation's timing mechanism.

Table 4. THROUGHPUT COMPARISON, FIRST TEST

<table>
<thead>
<tr>
<th></th>
<th>Offered Load</th>
<th>Single Address Capable</th>
<th>Multicast Capable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>60.72 Mbps</td>
<td>60.60 Mbps</td>
<td>55.00 Mbps</td>
</tr>
<tr>
<td>Synchronous</td>
<td>30.72 Mbps</td>
<td>30.80 Mbps</td>
<td>43.00 Mbps</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>30.00 Mbps</td>
<td>30.00 Mbps</td>
<td>12.00 Mbps</td>
</tr>
</tbody>
</table>

A real FDDI station would not generate destination addresses in the manner coded into the simulation model; the time required would be unreasonable. On the
Figure 37. Throughput: Single Addressing Only Stations

Figure 38. Throughput: Multicast Capable Stations, Single Addressing Mode
other hand, although the simulation's execution is noticeably slowed by the extra events generated in the destination address assignment loop, the throughput rate should not be affected because the simulated passage of time is controlled by the Kernel Procedures. That is, one microsecond does not pass until the simulation has completed one microsecond's worth of events, at all points on the LAN. This is how simultaneous events around the LAN are conducted, one at a time. For example, a new packet arrival at station f7, and a packet destruction at station f23, and a token release at station f0 may all occur simultaneously. A study of the sequence of events, revealed through use of the debug facility, shows that the simulation's clock is incremented after these events are all completed, thereby modeling simultaneous events. The eight percent reduction in total throughput may indicate the simulation timer is proceeding without waiting for the completion of the loop.

3. Second test

a. Setup

This test was intended to compare the throughput resulting from two nearly identical simulations, in which the first involved only synchronous traffic, and the second involved only asynchronous traffic. A ten-station LAN was created, with all stations having the same packet arrival rate (750 packets/sec) and packet size (1000 bits). As in the previous test, destination addresses were limited to one station per packet. TTRT was set to 4.0 ms.

b. Results

Figure 39 and Figure 40 illustrate the throughputs resulting from 7.5 Mbps offered load of all-synchronous and all-asynchronous traffic, respectively. Interestingly, both plots are identical, suggesting perhaps that the disparate results of the previous test indicate
Figure 39. Synchronous Throughput

Figure 40. Asynchronous Throughput
a coding error whenever mixed loads of synchronous and asynchronous traffic must be tracked simultaneously. Also noteworthy is the fact that the resulting throughput in both cases, 6.86 Mbps, was 91.5% of the offered load. This is comparable with the previous test: the 55 Mbps throughput in Figure 38 represents 91.7% of the 60.72 Mbps offered load. Again, this suggests a coordination problem between the addressing loop and the simulation's timekeeping function.

4. Third Test

a. Setup

This test was intended to compare the throughput from two different perspectives available. One result was generated in the familiar accounting procedure conducted in the sink process model, and the other result used the Probe Editor to place a monitoring probe on one of the station transmitter nodes, phyTx. If the transmissions are arranged so that all traffic passes this node, then the result should be two identical throughputs.

The same LAN of ten multicast-capable stations was used, with only station f9 transmitting, and with all of its traffic directed to station f8. In between, a probe was placed on the transmitter node of station f7. The packet arrival rate was 7500 packets/sec., and the packet length was 1000 bits, giving an offered load of 7.5 Mbps, all of which was asynchronous, with the full range of prioritization available. TTRT was 4.0 ms.

b. Results

Figure 41 shows a roughly constant difference of approximately 0.15 Mbps between the throughput monitored from the physical transmitter and that calculated in the
receiving station's sink process. The irregular plots are unusual, since the simulations
normally show a smooth steady state after less than half a second. Significantly, both curves
remain close to the offered load, though they are jagged.

The 40 bits per packet overhead (created in the Parameter Editor, where
the fields "fc", "src_addr," and "dest_addr" are assigned sizes of 8, 16 and 16 bits,
respectively) is a possible source of disparity, although a difference of 0.3 Mbps would be
expected in that case (7500 x 40). A study of the sink process model shows that overhead
is not included in throughput calculations, while the probes do count the bits in the
encapsulating packet structure. As before, the problem requires more study.

5. Fourth Test

a. Setup

A final test of the expected throughput of the multicast-capable model
actually observed the multicasting facility, or more exactly, the broadcasting facility of the
model. Again, a ten station LAN was used, with only one station transmitting, with TTRT
set to 4.0. This transmitter, station f7, generated 7500 packets at 1000 bits per packet, for
a total offered load of 7.5 Mbps, all asynchronous. In addition, each packet was addressed
to all nine of the other stations, which would be expected to yield a throughput of 67.5 Mb/s.
(The throughput statistics are gathered by comparing timestamps at the receipt of a packet
with the packet's creation time, which is carried as a field in the fddi_llc_fr packet format).
b. Results

Figure 42 illustrates the actual result. The probe at station f7's "phy_tx" node reflects the offered load of 7.5 Mbps with reasonable accuracy. However, the expected throughput of 67.5 Mbps was not nearly realized. The throughput plotted, 48.9 Mbps, represents 72.44% of the predicted amount. Again, this indicates a need to further develop the model, and to better define the meanings of throughput and offered load when packets are addressed to more than one station at a time.
Figure 41. Throughput from Two Vantage Points

Figure 42. Broadcast from One Station
V. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

This thesis has been directed to two main purposes: to explain the use and operation of OPNET's FDDI LAN simulation, and to describe the changes that were installed in the model in order to make it a more useful, accurate and versatile tool for the Common Data Link project. Both goals share another common objective, which is to develop and document the "corporate knowledge" of the CDL working group which will continue to work with the model studied here.

One immediate conclusion is the observation that OPNET is a powerful and flexible tool, but it requires much time and study to be used effectively. Where desired model attributes are lacking, the patient user may code his or her own. The following are the accomplishments documented with this thesis:

1. Throughput, mean delay, and end-to-end delay data are recorded separately and may be displayed separately for synchronous packet traffic and each priority level of asynchronous packet traffic. In addition, two new scalar plot axes, "Asynchronous Offered Load" and "Total Offered Load" are available for the display of LAN performance data in relation to usage.

2. The FDDI station model is capable of randomly choosing different priority threshold settings to assign to generated asynchronous packets, where before the
model was bound to one setting per simulation. This modification allows an additional measure of flexibility in assigning the transmission characteristics of an FDDI LAN.

3. A rudimentary linking node has been created, which accumulates received traffic in buffers for eventual transmission to another LAN.

4. A multicasting capability has been added to the FDDI station model, enabling packets to be addressed to more than one destination station. Preliminary tests and studies indicate the model correctly generates, transmits, receives, and disposes of the multicast packets, although unexpected throughput data suggest the model possibly has coding inaccuracies or improper interfacing of simulation timing and destination address generation.

5. A modest series of preliminary tests verifies the continuing valid operation of the modified models, for the most part.

6. A significant number of unexpected features of OPNET's FDDI LAN model are documented, for the benefit of those researchers continuing to work on the CDL project.

B. RECOMMENDATIONS

Because the eventual goal of the work begun in this thesis is the development of a large scale model simulation for a Network Interface, the development of the FDDI LAN model may be expected to continue. The following are some possible areas for further development:
1. As mentioned, the multicasting feature yields unexpectedly low throughput. The complex nature of the changes made to the code to enable multicasting suggests the likelihood of some logic error or timing interface discrepancy. This ought to be found and corrected.

2. The testing presented in this thesis is only preliminary, and could well be expanded upon. Because of time constraints, delay characteristics were not addressed at all.

3. The buffered subqueues of the preliminary bridging model are infinite by default, but may be assigned limits through the use of on-screen attributes. The code currently has no provision for the proper disposal of packets lost to buffer overflows; lost frames will fill the host terminal's memory until none remains. A related but separate issue is the fact that there is no retransmission protocol in effect for the current model.
APPENDIX A. FILE RETRIEVAL VIA FTP

The following is an excerpt from a screen dialogue demonstrating access to MIL 3 Inc.'s bulletin board, movement among subdirectories, and retrieval of files.

sun24:/home3/nix
% ftp
ftp> open
(to) mil3.com
Connected to mil3.com.
220 rmaxwell FTP server (SunOS 4.1) ready.
Name (mil3.com:nix): anonymous
331 Guest login ok, send 'ident as password.
Password: nix@ece.nps.navy.mil
230 Guest login ok, access restrictions apply.
ftp> dir
200 PORT command successful.
150 ASCII data connection for /bin/ls (131.120.20.124,2926) (0 bytes).
total 18
-rw-r--r-- 1 0 100 2110 Jul 21 00:32 FTP.instructions
-r--r--r-- 1 0 5135 Jan 21 1992 README.OPSIG
dr-xr-xr-x 2 0 512 Jan 14 1992 bin
dr-xr-xr-x 2 0 512 Jan 14 1992 dev
dr-xr-xr-x 2 0 512 Jan 14 1992 etc
drwxr-xr-x 5 0 512 Oct 27 1992 examp
drw-xr-xr-x 2 0 512 Dec 20 16:40 incoming
dr-xr-xr-x 2 0 512 Dec 23 17:34 model_depot
drwxr-xr-x 10 0 512 Dec 15 10:09 patches
drwxrwrxrwx 3 0 512 Dec 29 03:05 tmp
dr-xr-xr-x 3 0 512 Jan 14 1992 usr
226 ASCII Transfer complete.
700 bytes received in 3.1 seconds (0.22 Kbytes/s)
ftp> get README.OPSIG
200 PORT command successful.
150 ASCII data connection for README.OPSIG (131.120.20.124,2927) (5135 bytes).
226 ASCII Transfer complete.
local: README.OPSIG remote: README.OPSIG
5257 bytes received in 6.8 seconds (0.75 Kbytes/s)
ftp> cd patches
250 CWD command successful.
ftp> dir
200 PORT command successful.
150 ASCII data connection for /bin/ls (131.120.20.124,2928) (0 bytes).
total 8
drwxr-xr-x 6 101 1 512 Mar 10 1993 2.3.L
drwxr-xr-x 4 101 100 512 Apr 2 1993 2.3.Lhp
drwxr-xr-x 6 101 1 512 Sep 24 22:42 2.3.M
drwxr-xr-x 4 101 1 512 Dec 6 18:37 2.4.A
drwxr-xr-x 5 101 1 512 Dec 17 22:32 2.4.B
drwxr-xr-x 4 101 1 512 Jan 29 1993 2.4.beta2
drwxr-xr-x 13 101 1 512 Apr 26 1993 2.4.beta3
drwxr-xr-x 5 101 1 512 Jun 10 1993 2.4.beta4
226 ASCII Transfer complete.
511 bytes received in 0.63 seconds (0.8 Kbytes/s)
ftp> cd 2.4.B
250 CWD command successful.
ftp> dir
200 PORT command successful.
150 ASCII data connection for /bin/ls (131.120.20.124,2934) (0 bytes).
total 3
   drwxr-xr-x 2 101 100 512 Dec 15 10:11 opbug_2081
   drwxr-xr-x 2 0 1 512 Dec 17 22:33 tsup_3139
   drwxr-xr-x 2 0 1 512 Dec 17 20:15 tsup_3182
226 ASCII Transfer complete.
205 bytes received in 0.28 seconds (0.7 Kbytes/s)
ftp> cd opbug_2081
250 CWD command successful.
ftp> dir
200 PORT command successful.
150 ASCII data connection for /bin/ls (131.120.20.124,2935) (0 bytes).
total 57
   -rw-r--r-- 1 101 100 3988 Dec 15 07:12 README
   -rw-r--r-- 1 101 100 53379 Dec 15 07:12 fddi_mac.pr.m
226 ASCII Transfer complete.
141 bytes received in 0.22 seconds (0.62 Kbytes/s)
ftp> get README
200 PORT command successful.
150 ASCII data connection for README (131.120.20.124,2936) (3988 bytes).
226 ASCII Transfer complete.
local: README remote: README
4092 bytes received in 5 seconds (0.81 Kbytes/s)
ftp> bin
200 Type set to I.
ftp> get fddi_mac.pr.m
200 PORT command successful.
150 ASCII data connection for fddi_mac.pr.m (131.120.20.124,2937)
   "(53379 bytes).
226 ASCII Transfer complete.
local: fddi_mac.pr.m remote: fddi_mac.pr.m
53388 bytes received in 64 seconds (0.81 Kbytes/s)
ftp> quit

122
APPENDIX B. PACKET AND ICI FRAME STRUCTURES

The following packet structures are used in the FDDI LAN model.

A. PACKET FORMATS

1. "fddi_lkc_fr"

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Size (bits)</th>
<th>Default Value</th>
<th>Default Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>cr_time</td>
<td>double</td>
<td>0</td>
<td>0.0</td>
<td>set</td>
</tr>
<tr>
<td>pri</td>
<td>integer</td>
<td>0</td>
<td>0</td>
<td>unset</td>
</tr>
</tbody>
</table>

Note: Added to allow generation of statistics related to prioritized traffic

2. "fddi_mac_fr"

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Size (bits)</th>
<th>Default Value</th>
<th>Default Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>fc</td>
<td>integer</td>
<td>8</td>
<td>-</td>
<td>unset</td>
</tr>
<tr>
<td>src_addr</td>
<td>integer</td>
<td>16</td>
<td>-</td>
<td>unset</td>
</tr>
<tr>
<td>dest_addr</td>
<td>integer</td>
<td>16</td>
<td>-</td>
<td>unset</td>
</tr>
<tr>
<td>info</td>
<td>packet</td>
<td>-1</td>
<td>-</td>
<td>unset</td>
</tr>
<tr>
<td>svc_class</td>
<td>integer</td>
<td>0</td>
<td>-</td>
<td>unset</td>
</tr>
<tr>
<td>pri</td>
<td>integer</td>
<td>0</td>
<td>-</td>
<td>unset</td>
</tr>
<tr>
<td>tk_class</td>
<td>integer</td>
<td>0</td>
<td>-</td>
<td>unset</td>
</tr>
</tbody>
</table>

3. "fddi_mac_tk"

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Size (bits)</th>
<th>Default Value</th>
<th>Default Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>fc</td>
<td>integer</td>
<td>8</td>
<td>-</td>
<td>unset</td>
</tr>
<tr>
<td>class</td>
<td>integer</td>
<td>0</td>
<td>-</td>
<td>unset</td>
</tr>
<tr>
<td>res_station</td>
<td>integer</td>
<td>0</td>
<td>-</td>
<td>unset</td>
</tr>
</tbody>
</table>
# B. ICI FORMATS

1. "fddi_mac_ind"

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>src_addr</td>
<td>integer</td>
<td>0</td>
</tr>
<tr>
<td>dest_addr</td>
<td>integer</td>
<td>0</td>
</tr>
</tbody>
</table>

2. "fddi_mac_req"

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>svc_class</td>
<td>integer</td>
<td>0</td>
</tr>
<tr>
<td>dest_addr</td>
<td>integer</td>
<td>0</td>
</tr>
<tr>
<td>pri</td>
<td>integer</td>
<td>0</td>
</tr>
<tr>
<td>tk_class</td>
<td>integer</td>
<td>0</td>
</tr>
</tbody>
</table>
APPENDIX C. EXAMPLE ENVIRONMENT FILE

FOR 32-STATION FDDI LAN

# fdd132.ef
# sample simulation configuration file for fddi example model
# 32 station network

### Attributes related to loading used by "fddi_gen" ###

# station addresses
*.*.f0.mac.station_address: 0
*.*.f1.mac.station_address: 1
*.*.f2.mac.station_address: 2
*.*.f3.mac.station_address: 3
*.*.f4.mac.station_address: 4
*.*.f5.mac.station_address: 5
*.*.f6.mac.station_address: 6
*.*.f7.mac.station_address: 7
*.*.f8.mac.station_address: 8
*.*.f9.mac.station_address: 9
*.*.f10.mac.station_address: 10
*.*.f11.mac.station_address: 11
*.*.f12.mac.station_address: 12
*.*.f13.mac.station_address: 13
*.*.f14.mac.station_address: 14
*.*.f15.mac.station_address: 15
*.*.f16.mac.station_address: 16
*.*.f17.mac.station_address: 17
*.*.f18.mac.station_address: 18
*.*.f19.mac.station_address: 19
*.*.f20.mac.station_address: 20
*.*.f21.mac.station_address: 21
*.*.f22.mac.station_address: 22
*.*.f23.mac.station_address: 23
*.*.f24.mac.station_address: 24
*.*.f25.mac.station_address: 25
*.*.f26.mac.station_address: 26
*.*.f27.mac.station_address: 27
*.*.f28.mac.station_address: 28
*.*.f29.mac.station_address: 29
*.*.f30.mac.station_address: 30
*.*.f31.mac.station_address: 31
.*.*.mac.ring_id:0

# destination addresses for random message generation
"**.llc-src.low dest address":0
"**.llc-src.high dest address":31

# arrival rate(frames/sec), and message size (bits) for random message
generation at each station on the ring.
"**.arrival rate":200
"**.mean pk length":500

# set the proportion of asynchronous traffic
# a value of 1.0 indicates all asynchronous traffic
"**.async_mix":1.0

### Ring configuration attributes used by "fddi_mac" ###

# allocate percentage of synchronous bandwidth to each station
# this value should not exceed 1 for all stations combined; OPNET does not
# enforce this: 01FEB94: this must be less than 1: see equation below
"**.mac.sync bandwidth":0.0
# "f0.mac.sync bandwidth":.0935487
"**.mac.T-Req":.010

# Index of the station which initially launches the token
"spawn station":0

# Delay incurred by packets as they traverse a station's ring interface
# see Powers, p. 351 for a discussion of this (Powers gives lusec.
# but 60.0e-08 agrees with Dykeman & Bux)
station_latency:60.0e-08

# Propagation Delay separating stations on the ring.
prop_delay:5.085e-06

### Simulation related attributes

# Token Acceleration Mechanism enabling flag.
# It is recommended that this mechanism be enabled for most situations
accelerate_token:1
seed:10

# Run control attributes
duration:5
verbose_sim: TRUE
upd_int: .1
os_file: fdd132mod
ov_file: fdd132mod
# Opnet Debugger (odb) enabling attribute
# debug: TRUE
APPENDIX D. DEBUG TOOL EXCERPT

The following is an excerpt from the debugger in fulltrace, showing the arrival and reception of a message traffic packet at its destination address. In this case, station f11 has sent a packet to station f31. The packet has been passed "hand to hand" through each station in between, with the simulator enacting every step along the way. Note that when reception is complete, the simulation returns to station f11, which still has the token.

________________________________________ (ODB 2.4.A: Event)________________________________________

* Time : 0.995744186808 sec. [00d 00h 00m 00s . 995ms 744us 186ns 80Bps]
* Event : execution ID (461704), schedule ID (#502933), type (self intrpt)
* Source : execution ID (461703), top.ring0.f30.phy_tx (pt-pt transmitter)
* Data : code (0)
> Module : top.ring0.f30.phy_tx (pt-pt transmitter)
odb> next

  * Kernel Action: Pt-Pt Transmitter object
    Completing transmission of packet(s)
    channel (0)
    packet ID (4459)

________________________________________ (ODB 2.4.A: Event)________________________________________

* Time : 0.995744186808 sec. [00d 00h 00m 00s . 995ms 744us 186ns 80Bps]
* Event : execution ID (461705), schedule ID (#502934), type (remote intrpt)
* Source : execution ID (461703), top.ring0.f30.phy_tx (pt-pt transmitter)
* Data : code (0)
> Module : top.ring0.f31.phy_rx (pt-pt receiver)
odb> next

  * Kernel Action: Pt-Pt Receiver object
    Beginning reception of packet
    channel (0)
packet ID (4459)

(ODB 2.4.A: Event)

* Time : 0.995744186808 sec. [00d 00h 00m 00s . 995ms 744us 186ns 808ps]
* Event : execution ID (461706), schedule ID (#502935), type (remote intrpt)
* Source : execution ID (461703), top.ring0.f30.phy_tx (pt-pt transmitter)
* Data : code (0)
> Module : top.ring0.f31.phy_rx (pt-pt receiver)

odb> next

* Kernel Action: Pt-Pt Receiver object
  Completing reception of packet
  channel (0)
  packet ID (4459)

* Kernel Action: Pt-Pt pipeline
  Calling (error) pipeline stage
  packet ID (4459)

* op_td_get_int (pkptr, tda_index)
  packet ID (4459)
  TDA attribute (OPC_TDA_PT_LINK_OBJID)
  TDA value (82)

* op_ima_obj_attr_get (objid, attr_name, value_ptr)
  objid (82)
  attr_name (ber)
  value (0.0)

* op_pk_total_size_get (pkptr)
  packet ID (4459)
  total size (25040)

* op_td_set_int (pkptr, tda_index, value)
  packet ID (4459)
  TDA attribute (OPC_TDA_PT_NUM_ERRORS)
  TDA value (0 bit errs)

* Kernel Action: Pt-Pt pipeline
  Calling (ecc) pipeline stage
  packet ID (4459)

* op_td_is_set (pkptr, tda_index)
  packet ID (4459)
  TDA attribute (OPC_TDA_PT_ND_FAIL)
tda is set (false)

* op_td_get_int (pkptr, tda_index)
  packet ID (4459)
  TDA attribute (OPC_TDA_PT_RX_OBJID)
  TDA value (734)

* op_ima_obj_attr_get (objid, attr_name, value_ptr)
  objid (734)
  attr_name (ecc threshold)
  value (0.0)

* op_pk_total_size_get (pkptr)
  packet ID (4459)
  total size (25040)

* op_td_get_int (pkptr, tda_index)
  packet ID (4459)
  TDA attribute (OPC_TDA_PT_NUM_ERRORS)
  TDA value (0 bit errs)

* op_td_set_int (pkptr, tda_index, value)
  packet ID (4459)
  TDA attribute (OPC_TDA_PT_PK_ACCEPT)
  TDA value (1)

* Kernel Action: Pt-Pt Receiver object
  Packet successfully received
  channel (0)
  packet ID (4459)

ODB 2.4.A: Event

* Time : 0.995744186808 sec, [00d 00h 00m 00s . 995ms 744us 186ns 808ps]
* Event : execution ID (461707), schedule ID (#502936), type (stream intrpt)
* Source : execution ID (461706), top.ring0.f31.phy_rx (pt-pt receiver)
* Data : instrm (0), packet ID (4459)
> Module : top.ring0.f31.mac (queue)

odb> next

* invoking process ("fdd1_mac")

state (IDLE): exit executives

* op_intrpt_type ()
  intrpt type (stream intrpt)
* op_intrpt_strm ()
  active strm (0)

* op_pk_get (instrm_index)
  strm. index (0)
  packet ID (4459)

* op_pk_nfd_get (pkptr, fd_name, value_ptr)
  packet ID (4459)
  field name (fc)
  value (0)

* op_intrpt_type ()
  intrpt type (stream intrpt)

* op_intrpt_type ()
  intrpt type (stream intrpt)

* op_intrpt_strm ()
  active strm (0)

  state (FR_RCV): enter executives

* op_pk_nfd_get (pkptr, fd-name, value-ptr)
  packet ID (4459)
  field name (srcaddr)
  value (11)

  state (FR_RCV): exit executives

  state (FR_REPEAT): enter executives

* op_pk_nfd_get (pkptr, fd-name, value_ptr)
  packet ID (4459)
  field name (dest-addr)
  value (31)

* op_pk_total_size_get (pkptr)
  packet ID (4459)
  total size (25040)

* op_pk_nfd_get (pkptr, fd_name, value_ptr)
  packet ID (4459)
  field name (info)
  value (pk id (4458))

* op_ici_attr_set (iciptr, attr_name, attr_value)
  ICI id (81)
  attr name (src_addr)
  value (11)
* op_ici_attr_set (iciptr, attr_name, attr_value)
  ICI id   (81)
  attr name (dest_addr)
  value   (31)

* op_ici_install (iciptr)
  ICI ID   (81)

* op_pk_send_delayed (pkptr, outstrm_index, delay)
  packet ID   (4458)
  stream index (1)
  delay   (0.0002504)

* op_pk_destroy (pkptr)
  packet ID   (4459)

* Kernel Action: Destroying Packet
  packet ID   (4459)

  state (FR_REPEAT): exit executives
  state (IDLE): enter executives

* returning from process (*fddl_mac*)

________________ (ODB 2.4.A: Event) ________________________________

* Time : 0.995886571808 sec, [00d 00h 00m 00s . 995ms 886us 571ns 808ps]
* Event : execution ID (461708), schedule ID (#502837), type (stream intrpt)
* Source : execution ID (461607), top.ring0.fll.mac (queue)
* Data : instrm (0), packet ID (4457)
> Module : top.ring0.fll.phy_tx (pt-pt transmitter)

odb> next
APPENDIX E. MAC "C" CODE:

"fddi_mac_mult.pr.c"

The line numbering in this appendix is used for reference within this thesis only, and does not correspond with that seen in OPNET®'s text editors.

1 /* Process model C form file: fddi_mac_mult.pr.c */
2 /* Portions of this file Copyright (C) MIL 3, Inc. 1992 */

3 /* OPNET system definitions */
4 #include <opnet.h>
5 #include "fddi_mac_mult.pr.h"
6 FSM_EXT_DECS

7 /* Header block */
8 /* Define a timer structure used to implement */
9 /* the TRT and THT timers. The primitives defined to */
10 /* operate on these timers can be found in the */
11 /* function block of this process model. */
12 typedef struct {
13     int enabled;
14     double start_time;
15     double accum;
16     double target_accum;
17 } FddiT_Timer;

18 /* 08FEB94: define the number of stations here. -Nix */
19 #define NUM_STATIONS 50

20 /* Declare certain primitives dealing with timers */
21 double fddi_timer_remaining ();
22 double fddi_timer_create ();
23 double fddi_timer_value ();

24 /* Scratch strings for trace statements */
25 char str0 [512], str1 [512];

26 /* define constants particular to this implementation */
27 #define FDDI_MAX_STATIONS 512
/* define possible values for the frame control field */
#define FDDI_FC_FRAME 0
#define FDDI_FC_TOKEN 1

/* define possible service classes for frames */
#define FDDI_SVC_ASYNC 0
#define FDDI_SVC_SYNC 1

/* define input stream indices */
#define FDDI_LL никогда строки 1
#define FDDI_PHY Kostenlos_In

/* define output stream indices */
#define FDDI_LL никогда строки 1
#define FDDI_PHY Kostenlos_OUT

/* define token classes */
#define FDDI_TK_NONRESTRICTED 0
#define FDDI_TK_RESTRICTED 1

/* Ring Constants */
#define FDDI_TXRATE 1.0e+08
#define FDDI_SA_SCAN_TIME 28.0e-08

/* Token transmission time: based on 6 symbols plus 16 symbols of preamble */
#define FDDI_TOKEN_TX_TIME 88.0e-08

/* Codes used to differentiate remote interrupts */
#define FDDI_TRT_EXPIRE 0
#define FDDI_TK_INJECT 1

/* Define symbolic expressions used on transition */
/* conditions and in executive statements. */
#define TRT_EXPIRE (op_intrpt_type () - OPC_INTRPT_REMOTE &&
op_intrpt_code () == FDDI_TRT_EXPIRE)

#define TK_RECEIVED_phy_arrival && frame_control == FDDI_FC_TOKEN
#define RC_FRAME_phy_arrival && frame_control == FDDI_FC_FRAME
#define FRAME_ARRIVAL phy_arrival && frame_control == FDDI_FC_FRAME

#define TRT_EXPIRE (op_intrpt_type () == OPC_INTRPT_STRM &&
op_intrpt_strm () == FDDI_LL никогда строки)

#define STRIP my_address == src_addr

/* Define the maximum value for ring_id. This is the */
/* maximum number of FDDI rings that can exist in a */
/* simulation. Note that if this number is changed, */
/* the initialization for fddi_claim_start below must */
/* also be modified accordingly. */
#define FDDI_MAX_RING_ID 8

/* Declare the operative TTRT value 'T_Opr' which is the final */
/* negotiated value of TTRT. This value is shared by all stations */
/* on a ring so that all agree on its value. */
double fddi_t_opr [FDDI_MAX_RING_ID];
#define Fddi_T_Opr (fddi_t_opr [ring_id])

/* This flag indicates that the negotiation for the final TTRT */
/* has not yet begun. It is statically initialized here, and */
/* is reset by the first station which modifies T_Opr. */
/* Initialize to 1 for all rings. */
int fddi_claim_start [FDDI_MAX_RING_ID] = (1,1,1,1,1,1,1,1);
#define Fddi_Claim_Start (fddi_claim_start [ring_id])

/* Declare station latency parameters. */
/* These are true globals, so they do not need to be arrays. */
double Fddi_St_Latency;
double Fddi_PROP_Delay;

/* Declare globals for Token Acceleration Mechanism. */
/* Hop delay and token acceleration are true globals. */
double Fddi_Tk_Hop_Delay;
int Fddi_Tk_Accelerate = 1;

/* These are actually values shared by all nodes on a ring, */
/* so they must be defined as arrays. */
double fddi_tk_block_base_time [FDDI_MAX_RING_ID];
#define Fddi_Tk_Block_Base_Time (fddi_tk_block_base_time [ring_id])

int fddi_tk_base_station [FDDI_MAX_RING_ID];
#define Fddi_Tk_Base_Station (fddi_tk_base_station [ring_id])

int fddi_tk_blocked [FDDI_MAX_RING_ID];
#define Fddi_Tk Blocked (fddi_tk_blocked [ring_id])

int fddi_num_stations [FDDI_MAX_RING_ID];
#define Fddi_Num_Stations (fddi_num_stations [ring_id])

int fddi_num_registered [FDDI_MAX_RING_ID];
#define Fddi_Num_Registered (fddi_num_registered [ring_id])

Objid fddi_address_table [FDDI_MAX_RING_ID] [FDDI_MAX_STATIONS];
#define Fddi_Address_Table (fddi_address_table [ring_id])

/* Below is part of the OPBUG 2081 patch: FB ended here, before. -Nix */
/* Event handles for the TRT are maintained at a global level to */
/* allow token acceleration mechanism to adjust these as necessary */
/* when blocking and reinjecting the token. TRT_handle simply */
/* represents the TRT for the local MAC */
Evhandle fddi_trt_handle [FDDI_MAX_RING_ID][FDDI_MAX_STATIONS];
define Fddi_Trt_Handle (fddi_trt_handle [ring_id])
define TRT_handle Fddi_Trt_Handle [my_address]

/* Similarly, the TRT data structure is maintained on a global level. */
FddiT_Timer* fddi_trt [FDDI_MAX_RING_ID][FDDI_MAX_STATIONS];
define Fddi_Trt (fddi_trt [ring_id])
define TRT Fddi_Trt [my_address]

/* Registers to record the expiration time of each TRT when token is */
/* blocked. */
double fddi_trt_exp_time [FDDI_MAX_RING_ID][FDDI_MAX_STATIONS];
define Fddi_Trt_Exp_Time (fddi_trt_exp_time [ring_id])

/* the 'Late.Ct' flag is declared on a global level so that it can be */
/* set at the time when the token is injected back into the ring. */
double fddi_late_ct [FDDI_MAX_RING_ID][FDDI_MAX_STATIONS];
define Fddi_Late.Ct (fddi_late_ct [ring_id])
define Late.Ct Fddi_Late.Ct [my_address]

/* Convenient macro for setting TRT for a given station and absolute */
/* time */
define TRT_SET(station_id,abs_time) fddi_timer_set (Fddi_Trt
[station_id], abs_time - op_sim_time()); Fddi_Trt_Handle [station_id]
- op_intrpt_schedule_remote (abs_time, FDDIC_TRTEXPIRE,
Fddi_Address_Table [station_id]);

/* State variable definitions */
typedef struct
{
  FSM_SYS_STATE
  int sv_ring_id;
  FddiT_Timer* sv_THT;
  double sv_T_Req;
  double sv_T_Pri [8];
  Objid sv_my_objid;
  int sv_spawn_token;
  int sv_my_address;
  Packet* sv_tk_pkptr;
  double sv_sync_bandwidth;
  double sv_sync_pc;
  int sv_restricted;
  int sv_res_peer;
  int sv_tk_registered;
  Icl* sv_to_llc_ici_ptr;
134
} State Variable Definitions
```c
int sv_tk_trace_on;

#define pr_state_ptr ((fddi_mac_mult_state*) SimI_Mod_State_Ptr)
#define ring_id pr_state_ptr->sv_ring_id
#define T_Req pr_state_ptr->sv_T_Req
#define T_Pri pr_state_ptr->sv_T_Pri
#define my_objid pr_state_ptr->sv_my_objid
#define spawn_token pr_state_ptr->sv_spawn_token
#define my_address pr_state_ptr->sv_my_address
#define sync_pkptr pr_state_ptr->sv_sync_pkptr
#define sync_bandwidth pr_state_ptr->sv_sync_bandwidth
#define sync_pc pr_state_ptr->sv_sync_pc
#define restricted pr_state_ptr->sv_restricted
#define res_peer pr_state_ptr->sv_res_peer
#define tk_registered pr_state_ptr->sv_tk_registered
#define to_llc_ici_ptr pr_state_ptr->sv_to_llc_ici_ptr
#define tk_trace_on pr_state_ptr->sv_tk_trace_on

/* Process model interrupt handling procedure */

void fddi_mac_mult () {
  /* Packets and ICI's */
  Packet* mac_frame_ptr;
  Packet* pdu_ptr;
  Packet* pkptr;
  Packet* data_pkptr;
  Ici* ici_ptr;

  /* Packet Fields and Attributes */
  int req_pri, svc_class, req_tk_class;
  int frame_control, src_addr;
  int pk_len, pri_level;
  static int *da_ptr, dest_addr[];

  /* Token - Related */
  int tk_usable, res_station, tk_class;
  int current_tk_class;
  double accum_sync;

  /* Timer - Related */
  double tx_time, timer_remaining, accum_bandwidth;
  double tht_value;

  /* Miscellaneous */
  int i;
```
int spawn_station, phy_arrival;
char error_string [512];
int num_frames_sent, num_bits_sent;

/* 26DEC93: loop management variables, used in RCV_TK */
/* and ENCAP states. -Nix */
int NUM_PRIOS;
int punt;
int q_check;

/* 08FEB94: case management variables, used in FR_REPEAT. -Nix */
int for_me;
int count_addees;

/* 08MAR94: "field holding" variables, used in FR_REPEAT. -Nix */
Packet* info_ptr;

FSM_ENTER (fddi_mac_mult)

FSM_BLOCK_SWITCH
{
    /*---------------------------------------------------------*/
    /** state (UNIT) enter executives **/
    FSM_STATE_ENTERFORCED (0, state0_enter_exec, "INIT")
    {
        /* Obtain the station's address. This is an attribute */
        /* of this process. Addressing is simplified by */
        /* simply using integers, and only one mode. */
        /* This mode is 16 bit addressing unless the */
        /* packet format 'fddi_mac_fr' is modified. */
        /* my_objid = op_id_self(); /* 29DEC93 */
        op_im_obj_attr_get (my_objid, "station_address", &my_address);

        /* Register the station's object id in a global table. */
        /* This table is used by the mechanism which improves */
        /* simulation efficiency by 'jumping over' idle periods */
        /* rather than circulating an unusable token. */
        fddi_station_register (my_address, my_objid);

        /* Obtain the station latency for tokens and frames. */
        /* Default value is set at 100 nanoseconds. */
        Fddi_St_Latency = 100.0e-09;
        op_im_sim_attr_get (DPC_IMA_DOUBLE, "station_latency",
                            &Fddi_St_Latency);

        /* Obtain the propagation delay separating stations. */
        /* This value is given in seconds with default value 3.3 microseconds. */
        Fddi_Prop_Delay = 3.3e-06;
/* Derive the Delay for a 'hop' of a freely circulating packet. */
Fddi_Tk_Hop_Delay = Fddi.Prop_Delay + Fddi.St_Latency;

/* The T_Pri[] state variable array supports priority */
/* assignments on a station by station basis by */
/* establishing a correspondence between integer priority */
/* levels assigned to frames and the maximum values of the*/
/* Token holding timer (THT) which would allow packets to be*/
/* sent. Eight levels are supported here, but this can easily */
/* be changed by redimensioning the priority array. */
/* By default all levels are identical here, allowing */
/* any frame to make use of the token, so that in fact */
/* priority levels are not used in the default case. */
/* 01JAN94: (8-I) is a quick attempt to impart different weighting */
/* scales on each priority level, and is not necessarily realistic.-Nix */
/* for (i = 0; i < 8; i++)

for (i = 0; i < 8; i++)
{
    T_Pri[i] = (double) Fddi_T_Opr / (8.0 - i); /* 01JAN94 */
    printf("INIT: T_Pri[%d] = %f, Fddi_T_Opr = %d\n", i, T_Pri[i], Fddi_T_Opr);
}

/*Create the token holding timer (THT) used to restrict the */
/* asynchronous bandwidth consumption of the station */
THT = fddi_timer_create();

/* Create the token rotation timer (TRT) used to measure the */
/* rotations of the token, detect late tokens and initialize */
/* the THT timer before asynchronous transmissions. */
TRT = fddi_timer_create();

/* Set the TRT timer to expire in one TTRT */
TRT_SET (my_address. op_sim_time () + Fddi_T_Opr);

/* Initialize the Late_Ct variable which keeps track. */
/* of the number of TRT expirations. */
Late_Ct = 0;

/* initially the ring operates in nonrestricted mode */
restricted = 0;

/* Create an Interface Control Information structure */
/* to use when delivering received frames to the LLC. */
to_1ic_ici_ptr = op_ici_create ("fddi_mac_ind");

/* The 'tk_registered' variable indicates if the station */
/* has registered its intent to use the token. */
tk_registered = 0;

/* Determine if the model is to make use of the token */
/* 'acceleration' mechanism. If not, every passing of the */
/* token will be explicitly modeled, leading to large */
/* number of events being scheduled when the ring is idle */
/* (i.e. no stations have data to send). */
op_ima_sim_attrget (OPC_IMA_INTEGER, "accelerate_token", 
                   &Fddi_Tk_Accelerate);

/* Obtain the synchronous bandwidth assigned */
/* to this station. It is expressed as a */
/* percentage of TTRT, and then converted to seconds */
op_ima_obj_attrget (my_oid, "sync bandwidth", &sync_pc);
sync_bandwidth = sync_pc * Fddi_T_Opr;

/* Only one station in the ring is selected to */
/* introduce the first token. Test if this station is it. */
/* If so, set the 'spawn_token' flag. */
op_ima_sim_attrget (OPC_IMA_INTEGER, "spawn station", 
                    &spawn_station);
spawn_token = (spawn_station == my_address);

/* If the station is to spawn the token, create */
/* the packet which represents the token. */
if (spawn_token)
{
    tk_pkptr = op_pk_create_fmt ("fddi_mac_tk");

    /* assign its frame control field */
    op_pk_nfd_set (tk_pkptr, "fc", FDDI_FC_TOKEN);

    /* the first token issued is non-restricted */
    op_pk_nfd_set (tk_pkptr, "class", FDDI_TK_NONRESTRICTED);

    /* The transition will be made into the ISSU_TK */
    /* state where the tk_usable variable is used. */
    /* In case any data has been generated, prset */
    /* this variable to one. */
    tk_usable = 1;

    /* When sending packets the variable accum_bandwidth is */
    /* used as a scheduling base. Init this value to zero. */
/* This statement is required in case this is the spawning */
/* station, and the next state entered is ISSUE_TK */

    accum_bandwidth = 0.0;

/** state (INIT) exit executives **/
FSM_STATE_EXIT_FORCED (0, state0_exit_exec, "INIT")

/** state (INIT) transition processing **/
FSM_INIT_COND (spawn_token)
FSM_DFLT_COND
FSM_TEST_LOGIC ("INIT")

    FSM_TRANSIT_SWITCH
    
    FSM_CASE_TRANSIT (0, 2, state2_enter_exec, ;)
    FSM_CASE_TRANSIT (1, 1, state1_enter_exec, ;)

    /*******************************-*/

/** state (IDLE) enter executives **/
FSM_STATE_ENTER_UNFORCED (1, state1_enter_exec, "IDLE")

/** blocking after enter executives of unforced state. **/
FSM_EXIT (3, fddi_mac_mult)

/** state (IDLE) exit executives **/
FSM_STATE_EXIT_UNFORCED (1, state1_exit_exec, "IDLE")

    /* Determine if a trace is activated for the FDDI model */
    tk_trace_on = op_prgrd_b1_trace_active ("fddi_tk");

    /* Trap packets arriving from physical layer so that their */
    /* FC field can be extracted before evaluating conditions */
    if (op_intpr_type () == OPC_INTRPT_STRM && op_intpr_strm ()
        != FDDI_LLCC_STRM_IN)
    
    /* Acquire the arriving packet. */
    pkptr = op_pk_get (FDDI_PHY_STRM_IN);

    /* Determine the type of packet by extracting */
    /* the frame control field. */
    op_pk_nfd_get (pkptr, "fc", &frame_control);

    /* Physical layer arrival flag is set. */
    phy_arrival = 1;
else{
    /* The interrupt is not due to a physical layer arrival. */
    phy_arrival = 0;

    /* If the interrupt is a remote interrupt with specified code, it
     * signifies */
    /* the reinsertion of the token into the ring after an idle period. This
     * only */
    /* occurs if the token acceleration mechanism is active. */
    if (op_intrpt_type () == OPC_INTRPT_REMOTE && op_intrpt_code
        () == FDDIC_TK_INJECT)
    {
        /* create a new token */
        tk_pkptr = op_pk_create_fmt ("fddi_mac_tk");

        /* assign its frame control field */
        op_pk_nfd_set (tk_pkptr, "fco", FDDI_FC_TOKEN);

        /* the token is non-restricted */
        op_pk_nfd_set (tk_pkptr, "class", FDDI_TK_NONRESTRICTED);

        /* insert it into the ring */
        op_pk_send (tk_pkptr, FDDI_PHY_STRM_OUT);
    }
}

/** state (IDLE) transition processing **/
FSM_INIT_COND (TK_RECEIVED)
FSM_TEST_COND (RC_FRAME)
FSM_TEST_COND (TRT_EXPIRE)
FSM_TEST_COND (FRAME_ARRIVAL)
FSM_DFLT_COND
FSM_TEST_LOGIC ("IDLE")

FSM_TRANSIT_SWITCH
{
    FSM_CASE_TRANSIT (0, 3, state3_enter_exec, );
    FSM_CASE_TRANSIT (1, 4, state4_enter_exec, );
    FSM_CASE_TRANSIT (2, 7, state7_enter_exec, );
    FSM_CASE_TRANSIT (3, 8, state8_enter_exec, );
    FSM_CASE_TRANSIT (4, 1, state1_enter_exec, );
}

FSM_STATE_ENTER_FORCED (2, state2_enter_exec, "ISSUE_TK")

/* If the token is sent without having been used, and the station */
/ has no data to send, then indicate this fact to the */
/* token acceleration mechanism which may have an */
/* oppurtunity to block the token. */
if (!tk_usable && op_q_stat (OPC_QSTAT_PKSIZE) == 0.0)
{
/* Note that if the token cannot be blocked. */
/* this procedure will forward the token physically. */
    fddi_tk_indicate_no_data (tk_pkptr, my_address, 
                        accum_bandwidth);
}
else(
    if (tk_trace_on == OPC_TRUE)
    {
        sprintf (strO, "Issuing token. accum_bw (%.9f), prop_del
                  (%.9f)", accum_bandwidth, Fddi_Prop_Delay);
        op_prd_odb_print_major (strO, OPC_NIL);
    }
/* Send out the token packet using the accumulated */
/* consumed bandwidth as a scheduling base. */
/* In the case of the initial spawning of the token */
/* this will be zero; otherwise this variable will */
/* reflect the bandwidth consumed since the last capture */
/* of the usable token. Propagation delay is also accounted for. */
    op_pk_send_delayed (tk_pkptr, FDDI_PHY_STRM_OUT, 
                 accum_bandwidth + Fddi_Prop_Delay);
}
/** state (ISSUETK) exit executives **/
FSM_STATE_EXIT_FORCED (2, state2_exit_exec, "ISSUETK")
/** state (ISSUETK) transition processing **/
FSM_TRANSIT_FORCE (1, statelenterexec. ;)
/*-----------------------------------------------*/
/** state (RCV_TK) enter executives **/
FSM_STATE_ENTER_FORCED (3, state3_enter_exec, "RCV_TK")
/* The arriving packet, when received in the IDLE state */
/* is placed in the variable 'pkptr'. Since it is now */
/* known that it is a token, it can be placed in 'tk_pkptr'. */
    tk_pkptr = pkptr;
/* Load the token's class into the temporary variable 'tk_class.' */
    op_pk_nfd_get (pkptr, "class", &tk_class);
/* If the token is restricted, determine for which station. */
if (tk_class == FDDI_TK_RESTRICTED)
{
	/* Place the station address in the variable 'res_station' */
	/* which may factor in to the determination of token usability. */
	op_pk_nfd_get (tk_pkptr, "res_station", &res_station);
}

/* Determine if the token is usable: */
/* assume by default that it is not */
/* The token can only be usable if there are frames enqueued */
/* 27DEC93: the entire bank of subqueues must be checked. */
/* starting at the highest priority (corresponding to synchronous traffic), and stopping when a packet is */
/* found. Then the loop is broken. -Nix */

tkusable = 0;

/* The token can only be usable if there are frames enqueued */
/* 27DEC93: the entire bank of subqueues must be checked, */
/* starting at the highest priority (corresponding to */
/* synchronous traffic), and stopping when a packet is */
/* found. Then the loop is broken. -Nix */

NUM_PRIOS = 9;
for (i = NUM_PRIOS - 1; i > -1; i--)
{
	if (op_subq.stat[i, OPC_QSTAT_PKSIZE] > 0.0)
	{
	/* examine the attributes of the packet at the */
	/* head of the queue. */
	/* fddi_load_frameattrs (&dest_addr, &svc_class, &pri_level); */
	fddi_load_frameattrs (dest_addr, &svc_class, &pri_level);

	/* If synchronous data is queued, the token is */
	/* necessarily usable, regardless of timing conditions. */
	if (svc_class == FDDI_SVC_SYNC)
	{
		tk usable = 1;
		break;
	}
	else{
	/* Otherwise, if asynchronous data is queued, it must */
	/* meet several criteria for the token to be usable. */
	/* The token is only usable only if it is early. */
	if (Late_Ct == 0)
	{
	/* The token's class must be nonrestricted, unless */
	/* this station is involved in the restricted transfer. */
	if (tk_class == FDDI_TK_NONRESTRICTED || res_station == my_address || restricted)
	{
	/* Test the frame's priority assignment against the current TRT */
	/* This test uses the priority indirection table T_Pri */
	/* so that only packets whose T_Pri[pri_level] exceeds */
	/* the TRT can be transmitted. In other words, by */
	/* assigning lower values to T_Pri for a given priority */
	}
/* level, packets of that level will be further restricted */
/* from using the ring bandwidth. */
if (T_Pri [pri_level] >= fddi_timer_value (TRT))
{
    tk_usable = 1;
    break;
}
/* closes the *if (op_subq_stat (OPC_QSTAT_PKSIZE) > 0.0*
statmen */

/* If the token is usable, timers must be readjusted. */
if (tk_usable)
{
    /* The timer adjustment depends on whether the token is early or late. */
    if (Late_Ct == 0)
    {
        /* Transfer the contents of TRT into THT. */
        fddi_timer_copy (TRT, THT);
        /* Disable the THT timer. */
        fddi_timer_disable (THT);
        /* Reset TRT to time the next rotation. */
        op_ev_cancel (TRT_handle);
        TRT_SET (my_address, op_sim_time () + Fddi_T_Opr);
    }
    else
    {
        /* If the token is late, set the THT to its expired */
        /* value, and disable it. This will prevent any */
        /* asynchronous transmissions from occurring. */
        fddi_timer_set_value (THT, Fddi_T_Opr);
        fddi_timer_disable (THT);
        /* clear the Late token counter (note that TRT is not modified, */
        /* so that less than a full TTRT remains before TRT expires again. */
        Late_Ct = 0;
    }
}

/* If the token is not usable, different adjustments are made. */
else
{
    /* Again, the adjustments depend on the lateness of the token */
    if (Late_Ct == 0)
    {
/* If the token is not late, the TRT is reset to time the next rotation. */

    op_ev_cancel (TRT_handle);
    TRT_SET (my_address, op_sim_time () + Fddi_T_Opr);
  }
else{
  /* clear the Late token counter (note that TRT is not modified. */
  /* so that less than a full TTRT remains before TRT expires again. */
  Late.Ct = 0;
}

/* also, account for the time needed by the token */
/* to traverse the station, since it is about to be sent. */
/* Note: station latency is not inclusive of token */
/* transmission time, but only of the time required to */
/* process and repeat the token's symbols. */
accum_bandwidth = Fddi_St_Latency;

/** state (RCV_TK) exit executives **/
  FSM_STATE_EXIT_FORCED (3. state3_exit_exec, "RCV_TK")

/** state (RCV_TK) transition processing **/
  FSM_INIT_COND (tk_usable)
  FSM_DFLT_COND
  FSM_TEST_LOGIC ("RCV_TK")
  FSM_TRANSIT_SWITCH
  {  
    FSM_CASE_TRANSIT (0. 9. state9_enter_exec, :)
    FSM_CASE_TRANSIT (1. 2. state2_enter_exec, :)
  }

/** state (FR_RCV) enter executives **/
  FSM_STATE_ENTER_FORCED (4. state4_enter_exec, "FR_RCV")
  {  
    /* A frame has been received from the physical layer. Note that */
    /* at this time, only the leading edge of the frame has arrived. */

    /* Extract the frame's source address (this will be used to */
    /* determine whether or not to strip the frame from the ring). */
    op_pk_nfd_get (pkptr. "src_addr", &src_addr);
  }

/** state (FR_RCV) exit executives **/
  FSM_STATE_EXIT_FORCED (4. state4_exit_exec, "FR_RCV")

146
/* state (FRRCV) transition processing **/
FSM_INIT_COND (STRIP)
FSM_DEFAULT_COND
FSM_TEST_LOGIC ("FR_RCVC")

FSM_TRANSITION_SWITCH
{
    FSM_CASE_TRANSIT (0, 5, state5_enter_exec, ;)
    FSM_CASE_TRANSIT (1, 6, state6_enter_exec, ;)
}

/* state (FRSTRIP) enter executives **/
FSM_STATE_ENTERFORCED (5, state5_enter_exec, "FRSTRIP")

/* Destroy the frame which has now circulated the entire ring. */
op_pk_destroy (pkptr);

/* state (FRSTRIP) exit executives **/
FSM_STATE_EXIT_FORCED (5, state5_exit_exec, "FRSTRIP")

/* state (FRSTRIP) transition processing **/
FSM_TRANSITION_FORCE (1, state1_enter_exec, ;)

/*--------------------------------------------------------*/

/* state (FRREPEAT) enter executives **/
FSM_STATE_ENTERFORCED (6, state6_enter_exec, "FR_REPEAT")

/* Extract the destination address of the frame. */
/* 20FEB94: use a pointer to the array dest-addr. */
/* since referring to dest_addr directly produces */
/* unexpected results. -Nix */
op_pk_nfnd_get (pkptr, "dest_addr", &da_ptr);

/* printf("%d; %d; %d\n", *da_ptr, &da_ptr, &da_ptr); */

for (i = 0; i < NUM_STATIONS+1; i++):
    dest_addr[i] = da_ptr[i];

/* 02MAR94: print out the address, and the contents. */

147
/* for (i = 0; i < NUM_STATIONS + 1; i++) */
/* printf("1.FR_REPEAT:element: %d, address: %s/%d, content: %d\n", */
/* i, &dest_addr[i]), &dest_addr[i], dest_addr[i]); */
/* */
/***................................................................*****/

/* 08FEB94: re-initialize counters. -Nix */
for me = 0:
count_addees = 0:

/* 08FEB94: inspect the address field: interested in */
/* whether this packet is sent here only, or here and */
/* to others, or to others only. Note that a real packet */
/* would carry all the addresses: the simulation refers */
/* to memory locations. -Nix */
for (i = 1; i < NUM_STATIONS + 1; i++)
{
    if (dest_addr[i] == 1)
        count_addees += 1;
}

/* If the frame is for this station, make a copy */
/* of the frame's data field and forward it to */
/* the higher layer. */
/* if (dest_addr == my_address) */
/* 08FEB94: if this packet is addressed only to this */
/* station, make a copy of the frame's data field and */
/* and forward it to the higher layer. -Nix */
/* (a) If the packet is addressed to me only... */
/* (note offset applied) */
/* if (dest_addr[my_address - 1] == 1 && count_addees == 1)
{

/* *************/
/* printf("Here is Case 1.\n"); */
/* ******************** */

/* record total size of the frame (including data) */
pk_len = op_pk_total_size_get (pkptr);

/* decapsulate the data contents of the frame */
/* 29JAN94: a new field, "pri", has been added to */
/* the fddi_llc_fr packet format in the Parameters */
/* Editor, so that output statistics can be */
/* generated by class and priority. -Nix */
/* op_pk_nfd_get (pkptr, "info", data_pkptr);
*/
/* op_pk_nfd_get (pkptr, "pri", &pri_level); */

148
/* The source and destination address are placed in the */
/* LLC's ICI before delivering the frame's contents. */
op_ici_attr_set (to_llc_ici_ptr, "src_addr", src_addr);
op_ici_attr_set (to_llc_ici_ptr, "dest_addr", da_ptr);
op_ici_install (to_llc_ici_ptr);

/* 18FEB94: print out the address, and the contents. */
for (i = 0; i < NUM_STATIONS+1; i++)
{
dest_addr[i] = da_ptr[i];

printf("2.FR_REPEAT:element: %d. address: %X/id. content: %d\n", *
i, &dest_addr[i], &dest_addr[i], dest_addr[i]); */
printf("*************************/

/* Because, as noted in the FR_RCV state, only the */
/* frame's leading edge has arrived at this time, the */
/* complete frame can only be delivered to the higher */
/* layer after the frame's transmission delay has elapsed. */
/* (since decapsulation of the frame data contents has occurred, */
/* the original MAC frame length is used to calculate delay) */

/* Note that the standard specifies that the original */
/* frame should be passed along until the originating station */
/* receives it, at which point it is stripped from the ring. */
/* However, in the simulation model, there is no interest */
/* in letting the frame continue past its destination unless */
/* group addresses are used, so that the same frame could be */
/* destined for several stations. Here the frame is stripped */
/* for efficiency as it reaches the destination; if the model */, /* is modified to include group addresses, this should be change. */
/* so that the frame is copied and the original repeated. */
/* Logic is already present for stripping the frame at the origin. */

else if (dest_addr[my_address+1] --)
{

/* OBFEB94: (b)...or if this packet is not for me at all... -Nix */
}

/* 08FEB94: (b)...or if this packet is not for me at all... -Nix */
else if (dest_addr[my_address+1] == 0)
{

/* Repeat the original frame on the ring and account for */
/* the latency through the station and the propagation delay */
/* for a single hop. */
/* (Only the originating stat can strip the frame). */
    op_pk_send_delayed (pkptr, FDDI_PHY_STRM_OUT,
                      Fddi_St_Latency - Fddi_Prop_Delay);
}

/* 08FEB94: (c)...or if this packet is for me and for others, will */
/* need to send the contents to the SINK, then re-encapsulate the */
/* packet for further transmission. Much of this code is */
/* duplicated from the above. -Nix */

    else if (dest_addr[my_address+1] == 1 && count_addees > 1)
    {
    /* **************************** */
    /* printf("Here is Case 3.\n"); */
    /* **************************** */

    /* record total size of the frame (including data) */
    pk_len = op_pk_total_size_get (pkptr);

    /* decapsulate the data contents of the frame */
    op_pk_nfd_get (pkptr, "info", &data_pkptr);
    op_pk_nfd_get (pkptr, "pri", &pri_level);

    /* *********************************
    /* Print out the address of the "info" field information */
    /* printf("Case 3: 'info' is located at address %X\n", &data_pkptr); */
    /* *********************************/

    /* 08MAR94: copy the "info" address into a local variable, so that */
    /* it may be held for re-installation. -Nix */
    /* info_ptr = op_pk_copy(data_pkptr); */

    /* The source and destination address are placed in the */
    /* LLC's ICI before delivering the frame's contents. */
    op_ici_attr_set (to_llc_ici_ptr, "src_addr", src_addr);
    op_ici_attr_set (to_llc_ici_ptr, "dest_addr", dest_addr);
    op_ici_install (to_llc_ici_ptr);

    /* Because, as noted in the FR_RCV state, only the */
    /* frame's leading edge has arrived at this time. the */
    /* complete frame can only be delivered to the higher */
    /* layer after the frame's transmission delay has elapsed. */
    /* (since decapsulation of the frame data contents has occurred, */
    /* the original MAC frame length is used to calculate delay) */
    tx_time = (double) pk_len / FDDI_TX_RATE;
    op_pk_send_delayed (data_pkpt, FDDI_LLCC_STRM_OUT, tx_time);
/* 08FEB94: remove this station from the dest_addr array, reassemble */
/* the packet, and send the packet on its way. -Nix */
dest_addr[my_address+1] = 0;
op_pk_nfd_set(pkptr,"src_addr", src_addr);
op_pk_nfd_set(pkptr,"dest_addr", dest_addr);
op_pk_nfd_set(pkptr,"pri", pri_level);
op_pk_nfd_set(pkptr,"info", info_ptr);
    op_pk_send_delayed (pkptr, FDDI_PHY_STRM_OUT,
    Fddi_St_Latency + Fddi_Prop_Delay);

/** state (FR_REPEAT) exit executives **/
FSM_STATE_EXIT_FORCED (6, state6_exit_exec, "FR_REPEAT")
{
}
/** state (FR_REPEAT) transition processing **/
FSM_TRANSIT_FORCE (1, statel_enter_exec, :)

/** state (TRT_EXP) enter executives **/
FSM_STATE_ENTER_FORCED (7, state7_enter_exec, "TRT_EXP")
{
/* The timer is reset and allowed to continue running. */
    TRT_SET (my_address, op_sim_time () + Fddi_T_Opr);
/* The late token counter is incremented. This will */
/* prevent this station from making any asynchronous */
/* transmissions when it next captures the token. */
    Late_Ct++;}
/** state (TRT_EXP) exit executives **/
FSM_STATE_EXIT_FORCED (7, state7_exit_exec, "TRT_EXP")
{
}
/** state (TRT_EXP) transition processing **/
FSM_TRANSIT_FORCE (1, statelenterexec, :)

/** state (ENCAP) enter executives **/
FSM_STATE_ENTER_FORCED (8, state8_enter_exec, "ENCAP")
{
/* A frame has arrived from a higher layer; place it in 'pdu_ptr'. */
    pdu_ptr = op_pk_get (op_intrpt_strm());
/* Also get the interface control information */
also_intrupt_ici ();
    if (ici_ptr == OPC_NIL)
        }
        printf (error_string, "Simulation aborted; error in object
            (from %d). op_id_self ()");
        op_sim_end (error_string, "fddi_mac: required ICI not
            received", " ", ");
    }

/* Extract the requested service class */
/* (e.g. synchronous or asynchronous). */
if (op_ici_attr_exists (ici_ptr, "svc_class"))
    op_ici_attr_get (ici_ptr, "svc_class", &svc_class);
else svc_class = FDDI_SVCASYNC;

/****************************************************************************
for (i=0; i<NUM_STATIONS+1; i++)
    printf("ENCAP a.Field:%d, Address(dec/hex):%d/%X, Contents:%d
",
        *i, &dest_addr[i], &dest_addr[i], dest_addr[i]); */
/****************************************************************************

/* Extract the destination address. */
/* 20FEB94: use a pointer to the array, since the */
/* use of dest_array as its own pointer causes */
/* unexpected results. -Nix */
op_ici_attr_get (ici_ptr, "dest_addr", &da_ptr);

/****************************************************************************
for (i=0; i<NUM_STATIONS+1; i++)
    printf("%d: &da_ptr: %d/%X: da_ptr: %d
", i, &da_ptr, &da_ptr, da_ptr);
/****************************************************************************

for (i=0; i<NUM_STATIONS+1; i++)
    dest_addr[i] = da_ptr[i];

/****************************************************************************
for (i=0; i<NUM_STATIONS+1; i++)
    printf("ENCAP b.Field:%d, Address(dec/hex):%d/%X, Contents:%d
",
        *i, &dest_addr[i], &dest_addr[i], dest_addr[i]); */

152
/* If the frame is asynchronous, the priority and */
/* requested token class parameter may be specified. */
if (svc_class == FDDI_SVCASYNC)
{
/* Extract the requested priority level. */
if (op_ici_attr_exists(ici_ptr, "pri"))
    op_ici_attr_get(ici_ptr, "pri", &req_pri);
else req_pri = 0;
/* Extract the token class (restricted or non-restricted). */
if (op_ici_attr_exists(ici_ptr, "tk_class"))
    op_ici_attr_get(ici_ptr, "tk_class", &req_tk_class);
else req_tk_class = FDDI_TK_NONRESTRICTED;
}

/* Compose a mac frame from all these elements. */
mac_frame_ptr = op_pk_create_fmt("fddi_mac_fr");
op_pk_nfd_set(mac_frame_ptr, "svc_class", svc_class);
op_pk_nfd_set(mac_frame_ptr, "dest_addr", dest_addr);
op_pk_nfd_set(mac_frame_ptr, "src_addr", my_address);
op_pk_nfd_set(mac_frame_ptr, "info", pdu_ptr);

if (svc_class == FDDI_SVCASYNC)
{
op_pk_nfd_set(mac_frame_ptr, "tk_class", req_tk_class);
op_pk_nfd_set(mac_frame_ptr, "pri", req_pri);
}

/* 04JAN94: if the frame is synchronous, assign it a separate */
/* priority so that it may be assigned its own subqueue, and */
/* thereby be assigned its own probe for monitoring. -Nix */
if (svc_class == FDDI_SVC_SYNC)
{
op_pk_nfd_set(mac_frame_ptr, "pri", 8);
}

/* Assign the frame control field, which in the model */
/* is used to distinguish between tokens and ordinary */
/* frames on the ring. */
op_pk_nfd_set(mac_frame_ptr, "fc", FDDI_FC_FRAME);

/* Enqueue the frame at the tail of the queue. */
/* 27DEC93: at the tail of the prioritized queue. */
/* 04JAN94: must distinguish between synch & async. */
if (svc_class == FDDI_SVCASYNC)
{
op_subq_pk_insert(req_pri, mac_frame_ptr, OPC_QPOS_TAIL);
}
if (svc_class == FDDI_SVC_SYNC)
{
    op_subq_pk_insert (8, mac_frame_ptr, OPC_OPOS_TAIL);
}

/* if this station has not yet registered its intent to */
/* use the token, it may do so now since it has data to send */
if (!tk_registered)
{
    fddi(tk_register ());
    tk_registered = 1;
}

/** state (ENCAP) exit executives **/
FSM_STATE_EXIT_FORCED (8, state8_exit_exec, "ENCAP")

/** state (ENCAP) transition processing **/
FSM_TRANSIT_FORCE (1, statel_enter_exec, :)

/** state (TX_DATA) enter executives **/
FSM_STATE_ENTER_FORCED (9, state9_enter_exec, "TX_DATA")

/* In this state, frames are transmitted until the */
/* token is no longer usable. Frames are taken from */
/* the single input queue in FIFO order. */

/* Reset the accumulator used to keep track of bandwidth */
/* consumed by the transmissions. Because all the transmissions */
/* are scheduled to happen at the appropriate times, but */
/* these schedulings occur instantly, this accumulator serves */
/* as the scheduling base for the transmissions. */
/* In other words, each successively transmitted frame */
/* is delayed relative to the previous one by the time which */
/* the latter took to send. At the end of transmission (e.g. *)
/* when the token is no longer usable), this accumulator */
/* serves to delay the forwarding of the token. */
accum_bandwidth = 0.0;

/* Note that, because all transmissions are */
/* scheduled, the value of the THT timer will not progress */
/* between schedulings (these all happen in zero time), and so */
/* the variable 'tth_value' is used to emulate the timer's progress. */
/* tth_value = fddi_timer_value (THT); */

/* Reset an accumulator which reflects the consumed */
/* synchronous bandwidth. */
accum_sync = 0.0;

/* Reset counters for transmitted frames and bits. */
num_frames_sent = 0;
num_bits_sent = 0;

/* The transmission sequence must end if the input queue */
/* becomes exhausted. Other termination conditions are */
/* embedded in the loop. */
/* 27DEC93: modify the loop to accommodate subqueue structure. */
/* A "for" loop is imposed over the original "while" loop. */
/* First, reset the break marker. "punt". -Nix */
punt = 0;
for (i = NUM_PRIOS - 1; i > -1; i--)
{
  while (op_subq_stat (i. OPC_OSTAT_PKSIZE) > 0.0)
    {
    /* Remove the next frame for transmission. */
    pkptr = op_subq_pk_remove (i, OPC_OPOS_HEAD);
    /* Obtain the frame's service class. */
    op_pk_nfd_get (pkptr, "svc_class", &svcclass);
    /* Synchronous and asynchronous frames are treated differently. */
    if (svcclass == FDDI_SVC_SYNC)
      {
      /* Obtain the frame's length, and compute */
      pk_len = op_pk_total_size_get (pkptr);
      tx_time = (double) pk_len / FDDI_TX_RATE;
      /* Check if synchronous bandwidth allocation for this */
      /* station would be exceeded if the transmission were to occur. */
      if (accum_sync + tx_time > sync_bandwidth)
        {
        /* The frame could not be sent without exceeding */
        /* the allocated synchronous bandwidth. */
        /* so it is replaced on the queue. */
        /* 27DEC93: in this case, i is the highest priority. */
        /* which is reserved for synchronous traffic. -Nix */
        op_subq_pk_insert (i, pkptr, OPC_OPOS_HEAD);
        /* Exit the transmission loop since the frame */
        /* transmission request cannot be honored. */
        punt = 1;
        break;
      }
      /* Send the frame into the ring after other frames have completed. */
/* Also, account for its propagation delay; because the token propagation */
/* delay and the frame propagation delay must be consistent, and the */
/* token propagation delay is specified as a ring parameter (i.e. */
/* stations */
/* are assumed to be equally spaced), the ring is intended to run with */
/* */
/* the "delay" attributes of point-to-point links set at zero. */
op_pk_send_delayed (pkptr, FDDI_PHY_STREAM_OUT,
accum_bandwidth + Fddi.Prop_Delay);

/* increase the consumed bandwidth to reflect this */
/* transmission. Also increase synchronous consumption. */
accum_bandwidth += tx_time;
accum_sync += tx_time;

/* Increase counters for transmitted bits and frames. */
num_frames_sent++;
num_bits_sent += pk_len;
}
else{
/* The request enqueued at the head of the queue is */
/* asynchronous. It may only be honored if the THT timer */
/* has not expired. */
if (tht_value >= Fddi_T_Opr)
{
/* replace the packet on the queue and exit the transmission loop. */
op_subq_pk_insert (i, pkptr, OPC_QPOS_HEAD);
punt = 1;
break;
}
else{
/* Obtain the priority assignment of the frame. */
op_pk_nfd_get (pkptr, "pri", &pri_level);

/* If the packet's assigned priority level */
/* is too low for it to be serviced, then exit the loop */
/* after replacing the packet in the queue. */

/* ************************************************************ */
/* 08MAR94: print the values to be compared. -Nix */
/* printf("T_TXDATA: T_Pri[%d] < tht_value ?\n", i); */
/* printf("%d < %d ?\n", T_Pri[i], tht_value); */
/* ************************************************************ */

if (T_Pri [pri_level] < tht_value)
{
op_subq_pk_insert (i, pkptr, OPC_QPOS_HEAD);
punt = 1;
break;
}

/* Obtain the frame's length, and compute the time */
/* which would be required to transmit it. */

pk_len = op_pk_total_size_get (pkptr);
tx_time = (double) pk_len / FDDI_TX_RATE;

/* Determine the requested token class to be */
/* released after this frame is transmitted. */
op_pk_nfd_get (pkptr, "tk_class", &tk_class);

/* If the station is in restricted mode, then it may */
/* exit this mode if the class is now nonrestricted */
/* or if the restricted peer is not the addressee. */
if (restricted)
{
/* Determine the destination address for the new packet. */
op_pk_nfd_get (pkptr, "dest_addr", dest_addr);

/* if (tk_class == FDDI_TK_NONRESTRICTED || */
/* res_peer != dest_addr */

if (tk_class == FDDI_TK_NONRESTRICTED ||
    dest_addr[res_peer] != 1)
{
/* Exit restricted mode */
    restricted = 0;

/* Modify the token to reflect the mode change. */
op_pk_nfd_set (tk_pkptr, "class", FDDI_TK_NONRESTRICTED);
}
else
/* Determine the class of the current captured token. */
op_pk_nfd_get (tk_pkptr, "class", &current_tk_class);

/* When not in restricted mode, this mode may be entered */
/* if the passed packet has the appropriate token class requested. */
/* and the token is not already restricted. */
if (tk_class == FDDI_TK_RESTRICTED &&
current_tk_class != FDDI_TK_RESTRICTED)
{
/* Enter restricted mode. */
    restricted = 1;

/* Store the address of the restricted peer station. */
1038 /* op_pk_nfd_get (pkptr, "dest_addr", &res_peer); */
1039 op_pk_nfd_get (pkptr, "dest_addr", &dest_addr[res_peer]);
1040
1041 /* Modify the token to reflect the mode change. */
1042 op_pk_nfd_set (tk_pkptr, "class", FDDI_TK_RESTRICTED);
1043 op_pk_nfd_set (tk_pkptr, "res_station", res_peer);
1044 }  
1045
1046 /* Send the frame once previous transmissions have completed. */
1047 /* Account for propagation delay as well. */
1048 op_pk_send_delayed (pkptr, FDDI_PHY_STRM_OUT.
1049 accum_bandwidth + Fddi_Prop_Delay);
1050
1051 /* Increment THI emulation variable, and consumed bandwidth accumulator.
1052 */
1053 /* 08MAR94: note that tht_value is incrementing, not decrementing. -Nix */
1054 tht_value += tx_time;
1055 accum_bandwidth += tx_time;
1056
1057 /* *******************************************************  */
1058 /* 08MAR94: print the Token Holding Time value. -Nix */
1059 /* printf("2. TXDATA: tht_value is %d\n", tht_value); */
1060 /* *******************************************************  */
1061
1062 /* Increase counters for transmitted bits and frames. */
1063 num_frames_sent++;
1064 num_bits_sent += pk_len;
1065 }
1066 } /* closes the 'while' loop */
1067 if (punt == 1) /* If the 'while' loop was broken. */
1068 {
1069 punt = 0; /* then reset the 'break' marker. */
1070 break; /* and break out of the 'for' loop too. */
1071 }
1072 /* closes the 'for' loop. */
1073
1074 /* Since the token is about to be sent, its transmission time */
1075 /* must be reflected in the accumulated bandwidth. This is not */
1076 /* done in the ISSUE_TK state because when the token is merely */
1077 /* repeated, full transmission delay is not required, only */
1078 /* a small delay for repeating. */
1079 accum_bandwidth += FDDI_TOKEN_TX_TIME;
/* If the station has no more data to send (synchronous or */
/* asynchronous), it should indicate this to the token acceleration */
/* mechanism by deregistering its interest in the token. */
/* 27DEC94: the original code must be modified to include a check */
/* of subqueues. -Nix */
q_check = 1;
for (i = NUM_PRIOS - 1; i < -1; i--)
{
    if (op_subq_stat (i, OPC_QSTAT_PKSIZE) == 0.0)
    {
        q_check = 0;
    }
    else {
        q_check = 1;
        break;
    }
}
if (tk_registered && q_check == 0)
{
    tk_registered = 0;
    fddi_tk_deregister ();
}

/** state (TX_DATA) exit executives **/
FSM_STATE_EXIT_FORCED (9, state9_exit_exec, "TX_DATA")

/** state (TX_DATA) transition processing **/
FSM_TRANSIT_FORCE (2, state2_enter_exec, ;)

/** state (CLAIM) enter executives **/
FSM_STATE_ENTER_UNFORCED (10, state10_enter_exec, "CLAIM")

/* Obtain this station's object id which is used */
/* to access the station's attribute assignments. */
my_objid = op_id_self ();

/* Using the object id, obtain the ring id. */
/* The ring id is used by macros defined in the */
/* header block to obtain "ring-global" values. */
/* values shared by all stations on a ring. */
op_ima_obj_attr_get (my_objid, "ring_id", &ring_id);

/* Initialize global variable values. */
Fddi_TkBlocked = 0;
Fddi_Num_Stations = 0;
Fddi_Num_Registered = 0;

/* Using the object id, obtain the value of 'T_Req'. */
/* the value of TTRT requested by this station. */
op ima_obj_attr_get (my_objid, "T_Req", &T_Req);

/* The lowest value of T_Req becomes T_Opr for the ring as a whole. */
if (T_Req < Fddi_T_Opr || Fddi_Claim_Start)
    { /* The T_Req for this station is lower than any other to date */
      /* so it is installed in the T_Opr variable. */
      Fddi_T_Opr = T_Req;
    /* The flag indicating that the claim process is just */
    /* beginning may now be cleared. */
    Fddi_Claim_Start = 0;
}

/* Request a self interrupt from the Simulation Kernel at the current */
/* time so that after all stations have executed their claim states, */
/* they can proceed with initializations. This is necessary */
/* because some initializations are based in the value of T_Opr */
/* and it must therefore be known that all stations have settled */
/* on a final value. */
    op_intrpt_schedule_self (op_sim_time (), 0);

/* blocking after enter executives of unforced state. */
FSM_EXIT (21.fddi_mac_mult)

/** state (CLAIM) exit executives **/
FSM_STATE_EXIT_UNFORCED (10, state0_exit_exec, "CLAIM")

/** state (CLAIM) transition processing **/
FSM_TRANSIT_FORCE (0, state0_enter_exec, ;)

FIN (fddi_mac_mult_svar (prs_ptr, var_name, var_p_ptr))

void fddi_mac_mult_svar (prs_ptr, var_name, var_p_ptr)
    fddi_mac_mult_state  *prs_ptr;
    char                *var_name, **var_p_ptr;

FIN (fddi_mac_mult_svar (prs_ptr))
*var_p_ptr = VOS_NIL;
if (Vos_String_Equals (*ring_id", var_name))
  *var_p_ptr = (char *) (&prs_ptr->sv_ring_id);
if (Vos_String_Equals ("THT", var_name))
  *var_p_ptr = (char *) (&prs_ptr->sv_THT);
if (Vos_String_Equals ("T_Req", var_name))
  *var_p_ptr = (char *) (&prs_ptr->sv_T_Req);
if (Vos_String_Equals ("T_Pri", var_name))
  *var_p_ptr = (char *) (prs_ptr->sv_T_Pri);
if (Vos_String_Equals ("my_objid", var_name))
  *var_p_ptr = (char *) (prs_ptr->sv_my_objid);
if (Vos_String_Equals ("spawn_token", var_name))
  *var_p_ptr = (char *) (prs_ptr->sv_spawn_token);
if (Vos_String_Equals ("my_address", var_name))
  *var_p_ptr = (char *) (prs_ptr->sv_my_address);
if (Vos_String_Equals ("tk_pkptr", var_name))
  *var_p_ptr = (char *) (&prs_ptr->sv=tk_pkptr);
if (Vos_String_Equals ("sync_bandwidth", var_name))
  *var_p_ptr = (char *) (&prs_ptr->sv_sync_bandwidth);
if (Vos_String_Equals ("sync_pc", var_name))
  *var_p_ptr = (char *) (&prs_ptr->sv_sync_pc);
if (Vos_String_Equals ("restricted", var_name))
  *var_p_ptr = (char *) (prs_ptr->sv=restricted);
if (Vos_String_Equals ("res_peer", var_name))
  *var_p_ptr = (char *) (prs_ptr->sv=res_peer);
if (Vos_String_Equals ("tk_registered", var_name))
  *var_p_ptr = (char *) (prs_ptr->sv=tk_registered);
if (Vos_String_Equals ("to_llc_ici_ptr", var_name))
  *var_p_ptr = (char *) (prs_ptr->sv=to_llc_ici_ptr);
if (Vos_String_Equals ("tk_trace_on", var_name))
  *var_p_ptr = (char *) (prs_ptr->sv=tk_trace_on);
FOUT;
}

void
fddi_mac_mult_diag ()
{
  /* Packets and ICI's */
  Packet* mac_frame_ptr;
  Packet* pdu_ptr;
  Packet* pkptr;
  Packet* data_pkptr;
  Ici* ici_ptr;

  /* Packet Fields and Attributes */
  int req_pri, svc_class, req tk_class;
  int frame_control, src_addr;
  int pk_len, pri_level;
  static
int *da_ptr, dest_addr[];

/* Token - Related */
int tk_usable, res_station, tk_class;
int current_tk_class;
double accum_sync;

/* Timer - Related */
double tx_time, timer_remaining, accum_bandwidth;
double tht_value;

/* Miscellaneous */
int i;
int spawn_station, phy_arrival;
char error_string[512];
int num_frames_sent, num_bits_sent;

/* 26DEC93: loop management variables, used in RCV_TK */
/* and ENCAP states. -Nix */
int NUM_PRIOS;
int punt;
int q_check;

/* 08FEB94: case management variables, used in FR_REPEAT. -Nix */
int for_me;
int count_addrees;

/* 08MAR94: "field holding" variables, used in FR_REPEAT. -Nix */
Packet* info_ptr;

FIN (fddi_mac_multdiag ())

/* Print out values of timers, and late token counter. */
/* Also print out data about restricted mode. */
/* (This code may be executed by the simulation debugger */
/* by invoking the command 'modprint'). */

sprintf (strmers (count upwards): TRT (%.9g), THT (%.9g) ",
fddi_timer_value (TRT), fddi_timer_value (THT));
sprintf (strl, "Late Ct (%d)", Late_Ct);
op_prd_odb_print_major (str0, str1, OPC_NIL);
if (restricted)
    sprintf (str0, "token is in restricted dialog with (%d)\n", res.peer);
else sprintf (str0, "token is unrestricted\n");
op_prd_odb_print_major (str0, OPC_NIL);
FOUT;
}
fddi_mac_multTerminate()
{
  /* Packets and ICI's */
  Packet* mac_frame_ptr;
  Packet* pdu_ptr;
  Packet* pkptr;
  Packet* data_pkptr;
  Ici* ici_ptr;

  /* Packet Fields and Attributes */
  int req_pri, svc_class, req_tk_class;
  int frame_control, src_addr;
  int pk_len, pri_level;
  static *da_ptr, dest_addr[];

  /* Token - Related */
  int tk_usable, res_station, tk_class;
  int current_tk_class;
  double accum_sync;

  /* Timer - Related */
  double tx_time, timer_remaining, accum_bandwidth;
  double tht_value;

  /* Miscellaneous */
  int i;
  int spawn_station, phy_arrival;
  char error_string[512];
  int num_frames_sent, num_bits_sent;

  /* 26DEC93: loop management variables, used in RCV_TK */
  /* and ENCAP states. -Nix */
  int NUM_PRIOS;
  int punt;
  int q_check;

  /* 08FEB94: case management variables, used in FR_REPEAT. -Nix */
  int for_me;
  int count_addees;

  /* 08MAR94: "field holding" variables, used in FR_REPEAT. -Nix */
  Packet* info_ptr;

  FIN (fddi_mac_multTerminate())
  FOUT;
}

Compcode
fddi_mac_multInit (pr_state_pptr)
```c
fddi_mac_mult_state  **pr_state_pptr;
{
    static VosTCm_Obtype  obtype = OPC_NIL;

    FIN (fddi_mac_mult_init (pr_state_pptr))

    if (obtype == OPC_NIL)
    {
        if (Vos_Catmem_Register ("proc state vars (fddi_mac_mult)", sizeof (fddi_mac_mult_state), Vos_Nop, &obtype) == VOSC_FAILURE)
            FRET (OPC_COMPCODE_FAILURE)
    }

    if (**pr_state_pptr == (fddi_mac_mult_state*) Vos_Catmem_Alloc (obtype, 1)) == OPC_NIL)
        FRET (OPC_COMPCODE_FAILURE)
    else
    {
        (**pr_state_pptr)->current_block = 20;
        FRET (OPC_COMPCODE_SUCCESS)
    }
}

/** The procedures defined in this section serve **/
/** to simplify the code in the main body of the **/
/** process model by providing primitives for timer **/
/** manipulation... **/

fddi_timer_disable (timer_ptr)
    FddlTTimer*  timer_ptr;
{
    /* if the timer is already disabled, do nothing */
    if (timer_ptr->enabled)
        { /* disable the timer */
            timer_ptr->enabled = 0;

            /* reassign the accumulated time so far */
            timer_ptr->accum = op_sim_time () - timer_ptr->start_time;
        }
}

fddi_timer_enable (timer_ptr)
    FddlTTimer*  timer_ptr;
{
    /* if the timer is already enabled, simply return */
    if (!timer_ptr->enabled)
        { /* reenable the timer */
            timer_ptr->enabled = 1;
            return;
        }
```

164
1335 /* set the start time to the current time */
1336 /* less the accumulated time so far */
1337     timer_ptr->start_time = op_sim_time () - timer_ptr->accum;
1338 }
1339 }
1340 
fddi_timer_expired (timer_ptr)
1341     FddiT_Timer*      timer_ptr;
1342 {
1343     if (fddi_timer_remaining (timer_ptr) <= 0.0)
1344         return 1;
1345     else return 0;
1346 }
1347 
1348 double
1349 fddi_timer_remaining (timer_ptr)
1350     FddiT_Timer*      timer_ptr;
1351 {
1352     /* if the timer is enabled, update the accumulated time */
1353     if (timer_ptr->enabled)
1354         {  
1355             timer_ptr->accum = op_sim_time () - timer_ptr->start_time;
1356         }
1357     /* return the timer remaining before expiration */
1358     /* a non-positive value indicates an expired timer */
1359     return (timer_ptr->targetAccum - timer_ptr->accum);
1360 }
1361 
1362 double
1363 fddi_timer_value (timer_ptr)
1364     FddiT_Timer*      timer_ptr;
1365 {
1366     /* if the timer is enabled, update the accumulated time */
1367     if (timer_ptr->enabled)
1368         {  
1369             timer_ptr->accum = op_sim_time () - timer_ptr->start_time;
1370         }
1371     return (timer_ptr->accum);
1372 }
1373 
1374 fddi_timer_set_value (timer_ptr, value)
1375     FddiT_Timer*      timer_ptr;
1376     double            value;
1377 {
1378     timer_ptr->accum = value;
1379 }
1380 
1381 fddi_timer_copy (from_timer_ptr, to_timer_ptr)
FddiT_Timer* from_timer_ptr;
FddiT_Timer* to_timer_ptr;
{
  Vos_Copy_Memory(from_timer_ptr, to_timer_ptr, sizeof
  (FddiT_Timer));
}

fddi_timer_set(timer_ptr, duration)
FddiT_Timer* timer_ptr;
{
  /* clear out accumulated time */
  timer_ptr->accum = 0.0;
  /* assign the timer duration */
  timer_ptr->target_accumm = duration;
  /* assign the current time */
  timer_ptr->start_time = op_sim_time();
  /* enable the timer */
  timer_ptr->enabled = 1;
}

FddiT_Timer* fddi_timer_create()
{
  FddiT_Timer* timer_ptr;

  /* allocate memory for a timer structure */
  timer_ptr = (FddiT_Timer*) malloc(sizeof (FddiT_Timer));

  /* initialize the timer in the disabled mode */
  fddi_timer_init(timer_ptr);

  /* return the timer's address */
  return (timer_ptr);
}

fddi_timer_init(timer_ptr)
FddiT_Timer* timer_ptr;
{
  /* the timer is initially disabled */
  timer_ptr->enabled = 0;

  /* the accumulated time is zero */
  timer_ptr->accum = 0.0;

  /* the target accumulated time is infinite */
  timer_ptr->target_accumm = VOS_DOUBLE_INFINITY;
/* the start time is now */
timer_ptr->start_time = op_sim_time();

fddi_station_register (address, objid)
Objid objid;
int address;
{
/* Fill an entry in the table which maps station addresses to OPNET object ids */
FddiAddressTable [address] = objid;

/* Keep track of total number of stations on the ring */
Fddi_Num_Stations++;

fddi_tk_register ()
{
/* Register the station's intent to use the token. */
/* This should be done whenever an unregistered station obtains new data to transmit. */
FIN (fddi_tk_register ())

/* increase the number of registered stations */
Fddi_Num_Registered++;

/* if the token is currently blocked, unblock it */
if (Fddi_Tk_Block & Fddi_Tk_Accelerate)
{
    fddi_tk_unblock ();
}

fddi_tk_deregister ()
{
/* Cancel the station's intent to use the token. */
/* This should be done whenever a registered station exhausts its transmittable data. */
FIN (fddi_tk_deregister ())

/* decrease the number of registered stations */
Fddi_Num_Registered--;

FOUT
fddi_tk_indicate_no_data (token, address, delay)

fddi_tk_indicate_no_data (token, address, delay)

FIN (fddi_tk_indicate_no_data (token, address, delay))

/* The calling station is indicating that it has captured */
/* the token, but has no data to send. If no other stations */
/* have data to send either, the token may be blocked to gain */
/* simulation efficiency. */
if (Fddi_NumRegistered == 0 && Fddi_Tk_Accelerate)
{
    fddi_tk_block (token, address);
}
else{
/* If the token cannot be blocked, send it into the ring. */
op_pk_sendDelayed (token, FDDI_PHY_STRM_OUT, delay + Fddi_Prop_Delay);
}
FOUT

fddi_tk_block (token, address)

fddi_tk_block (token, address)

FIN (fddi_tk_block (token, address))

/* Record the address of the blocking station and blocking time. */
Fddi_Tk_Block_Base_Time = op_sim_time();
Fddi_Tk_Block_Base_Station = address;
if (tk_trace_on == OPC_TRUE)
{
    sprintf (str0, "Blocking Token: station (%d), time (%.9f)",
            Fddi_Tk_Block_Base_Station, Fddi_Tk_Block_Base_Time);
    op_prsg_odb_print_major (str0, OPC_NIL);
}
/* Indicate that the token is blocked */
Fddi_Tk Blocked = 1;
/* discard the token packet; another one will be */
/* created when the token is unblocked. */
op_pk_destroy (token);
/* Cancel TRT timers at all MAC interfaces; otherwise these */
/* timers may continue to expire during the idle period. */
/* generating unnecessary events. */
if (tk_trace_on == OPC_TRUE)
{
    sprintf (strO, "Canceling timers for (%d) stations", Fddi_Num_Stations);
    op_prd_odb_print_major (strO, OPC_NIL);
}

for (i = 0; i < Fddi_Num_Stations; i++)
{
    /* Retain the time at which the TRT would have expired; */
    /* this is used for calculations when the token is */
    /* reinjected into the ring. */
    Fddi_Trt_Exp_Time [i] = op_ev_time (Fddi_Trt_Handle [i]);

    /* Cancel the TRT expiration event. */
    op_ev_cancel (Fddi_Trt_Handle [i]);
}
FOUT
FIN (fddi_tk_unblock ())

/* reset the blocking indicator */
Fddi_Tk_Blocked = 0;

/* Get the current time, used for many calculations below */
current_time = op_sim_time ();

if (tk_trace_on == OPC_TRUE)
{
    sprintf (strO, "Unblocking token for ring (%d)", ring_id);
    op_prd_odb_print_major (strO, OPC_NIL);
}

/* For all stations on the ring, adjust TRT timer and Late Ct flag. */
for (i = 0; i < Fddi_Num_Stations; i++)
{
    if (tk_trace_on == OPC_TRUE)
    {
        sprintf (strO, "adjusting state of station (%d)", i);
op_prg_odb_print_minor("", str0, OPC_NIL);

/* Calculate number of hops separating station i from block base station. */
/* In special case where i is the base station, the token must run a full */
/* lap before returning. */
if (i != Fddi_Tk_Block_Base_Station)
{
    num_hops = (i - Fddi_Tk_Block_Base_Station) % Fddi_Num_Stations;
    if (num_hops < 0)
        num_hops = Fddi_Num_Stations + num_hops;
}
else num_hops = Fddi_Num_Stations;

/* Calculate first time at which token would have been received by station i. */
/* Note that initial release of token from base station takes a different */
/* amount of time than repeating of token by other stations. Thus, the first */
/* hop is assumed, and the base time is augmented by the time required */
/* to complete it. */
first_tkrx = Fddi_Tk_Block_Base_Time + FDDIC_TOKEN_TX_TIME + Fddi_Prop_Delay + (num_hops - 1) * Fddi_Tk_Hop_Delay;

if (tk_trace_on == OPC_TRUE)
{
    sprintf (str0, "station is (%d) hops from base", num_hops);
    sprintf (str1, "first receipt of token would be at (%.9f)",
            first_tkrx);
    op_prg_odb_print_minor (str0, str1, OPC_NIL);
}

/* Case 1: the token would not yet have been received by station i. */
if (first_tkrx > current_time)
{
    /* Case 1a: the TRT at station i would not yet have expired. */
    if (Fddi_Trt_Exp_Time [i] > current_time)
    {
        /* Late Ct remains at i original value; only the TRT needs */
        /* to be started again, with the same expiration time. */
        TRT_SET (i, Fddi_Trt_Exp_Time [i])
    }
    if (tk_trace_on == OPC_TRUE)
    {
        sprintf (str0, "Restoring TRT to previous exp. time (%.9f)", Fddi_Trt_Exp_Time [i]);
    }
op_pragma_ordbrob_print_minor ("Token would not be received
and TRT not expired", strO, OPC_NIL);

/* Case 1b: the TRT at station i would have expired. */
else
{
/* Late_Ct would have been set; also the timer would have been
rescheduled */
/* for an entire TTRT at the time of expiration. */
Fddi_Late_Ct [i] = 1;
TRT_SET (i, (Fddi_T_Opr + Fddi_Trt_Exp_Time [i]))

if (tk_trace_on == OPC_TRUE)
{
    sprintf (strO, "Restoring TRT to proper exp. time
(%.9f)", Fddi_T_Opr + Fddi_Trt_Exp_Time [i]);
    op_pragramordbrob_print_minor ("Token would not be received
and TRT would have expired", strO, OPC_NIL);
}

/* Case 2: the token would have been received (perhaps more than once).
*/
else
{
/* Calculate the number of times the token would have been received */
/* not including the first receipt. */
tk_lap_time = Fddi_Tk_Hop_Delay * Fddi_Num_Stations;
    num_tkrx = floor ((current_time - first_tkrx) /
tk_lap_time);

/* Calculate the latest time at which the token would have been
received. */
    last_tkrx = first_tkrx + (num_tkrx * tk_lap_time);

/* Clear Late_Ct and schedule timer to expire at last receipt of token
*/
/* plus one full TTRT. */
Fddi_Late_Ct [i] = 0;
    TRT_SET (i, (last_tkrx + Fddi_T_Opr))
    if (tk_trace_on == OPC_TRUE)
    {
        sprintf (strO, "token received (%g) times, last receipt
at (%.9f)", num_tkrx + 1.0, last_tkrx);
        sprintf (strl, "Restoring TRT to proper exp. time
(%.9f)", last_tkrx + Fddi_T_Opr);
        op_pragramordbrob_print_minor ("Token would have been received;
Late_Ct is cleared", strl, strO, OPC_NIL);

171
/* compute the time since the token was blocked */
elapsed_time = current_time - Fddi_Tk_Block_Base_Time;

/* compute the number of hops completed on the ring. For the first hop */
/* the token is transmitted directly, not repeated. For all remaining */
/* hops, the delay is the station latency plus the propagation delay. */
/* Thus, the first hop is assumed, and the remaining time for */
/* hops is computed beginning at the time where the token enters the */
/* base station's downstream neighbor */
/* If the token was unblocked in less time than it would have taken to */
/* be fully transmitted by the base station, dbl_num_hops will be */
/* negative. However, 1 full hop would still be required before the */
/* token could be used, since the station had already committed to */
/* issuing the token. Thus, the actual number of hops should never */
/* be less than 1. If it is, round it to 1. */
if (dbl_num_hops < 1.0)
    dbl_num_hops = 1.0;
else
    [missing code]
/* In all other cases, round the number of hops up to the nearest */
/* integer value. If already an integer, then leave as is. */
dbl_num_hops = ceil (dbl_num_hops);

/* Obtain an integer equivalent of dbl_num_hops. */
um_hops = dbl_num_hops;

/* Based on the number of hops and the base station, compute the */
/* next station where the token will appear. */
next_station = (num_hops + Fddi_Tk_Block_Base_Station) %
Fddi_Num_Stations;

/* Compute the time at which the token will appear there. */
/* Again, assume the first hop occurred, and measure time */
/* from there forward. */
next_time = Fddi_Tk_Block_Base_Time + (FDDIC_TOKEN_TX_TIME +
Fddi_TkHop_Delay) + (dbl_num_hops - 1.0) * Fddi_TkHop_Delay;
if (tk_trace_on == OPC_TRUE)
{
    sprintf (strO, "Re-introducing token at station (%d), at time
(%9f) at next_station. next_time):
    op_prg_odb_print_minor (strO, OPC_NIL);
}

/* reinject the token at that station */
fddi_tk_inject (next_station, next_time):
    FOUT
}

fddi_tk_inject (address, arv_time)
int address;
double arv_time;
{
    /* Re-insert the token into the ring after an idle period. */
    FOUT
}

fddi_tk_inject (address, arv_time)
int address;
double arv_time;
{
    /* Re-insert the token at that station */
    FOUT
}

fddi_load_frame_attrs (dest_addr_ptr, svc_class_ptr, pri_level_ptr)
int *dest_addr_ptr, *svc_class_ptr, *pri_level_ptr;
{
    int NUM_PRIOS, i; /* 26JAN94 */
    Packet *pkptr;
    FIN (fddi_load_frame_attrs (dest_addr_ptr, svc_class_ptr,
        pri_level_ptr))

    /* remove next packet in queue */
    /* 27DEC94: loop structure superimposed to handle a bank of subqueues.*/
    /* Extract the packet with the highest priority, that is, the packet */
    /* at the head of the highest-numbered subqueue containing packets. */
    /* Note that the C language vector numbering convention numbers the */
    /* subqueues from 0 to 7, while FDDI convention is to number the */

/* corresponding asynchronous priorities from 1 to 8. This is */
/* reconciled in the statistical outputs available in the Analysis */
/* Editor, where labels are assigned accordingly. Also note that */
/* synchronous traffic is assigned priority 8 as an artifice to allow */
/* routing through a separate subqueue, by which statistics may be */
/* gathered for traffic by class and by priority. -Nix */

NUM_PRIOS = 9;

for (i = NUM_PRIOS - 1; i > -1; i--)
{
    if (op_subq_stat (i, OPC_QSTAT_PKSIZE) > 0.0)
    {
        pkptr = op_subq_pk_remove (i, OPC_QPOS_HEAD);
        break;
    }
}

/* extract the fields of interest */
op_pk_nfd_get (pkptr, "dest_addr", dest_addr_ptr);
op_pk_nfd_get (pkptr, "svc_class", svc_class_ptr);

/* only read priority level if frame is asynchronous */
if (*svc_class_ptr == FDDI_SVC_ASYNC)
op_pk_nfd_get (pkptr, "pri", pri_level_ptr);

/* replace the packet on the proper subqueue */
op_subq_pk_insert (i, pkptr, OPC_QPOS_HEAD);

FOUT
APPENDIX F. SOURCE "C" CODE:

"fddi_gen_mult.pr.c"

The line numbering in this appendix is used for reference within this thesis only, and
does not correspond with that seen in OPNET’s text editors.

```
1 /* Process model C form file: fddi_gen_mult.pr.c */
2 /* Portions of this file Copyright (C) MIL 3, Inc. 1992 */

3 /* OPNET system definitions */
4 #include <opnet.h>
5 #include "fddi_gen_mult.pr.h"
6 FSM_EXT_DECLS

7 /* Header block */
8 #define MAC.Layer.OUT_STREAM 0

9 /* define possible service classes for frames */
10 #define FDDI_SVC_ASYNC 0
11 #define FDDI_SVC_SYNC 1

12 /* define token classes */
13 #define FDDI_TK_NONRESTRICTED 0
14 #define FDDI_TK_RESTRICTED 1

15 /* 07FEB94: define the number of stations */
16 #define NUM_STATIONS 50

17 /* a global counting variable */
18 /* nt genARRIVAL = 0; */

19 /* State variable definitions */
20 typedef struct
21 {
22   FSM_SYS_STATE
23   Distribution* sv_inter_dist_ptr;
24   Distribution* sv_len_dist_ptr;
25   Distribution* sv_dest_dist_ptr;
26   Distribution* sv_pkt_priority_ptr;
27   Objid sv_mac_objid;
```

175
ObjId sv_my_id;
int sv_low_dest_addr;
int sv_high_dest_addr;
int sv_station_addr;
int sv_low_pkt_priority;
int sv_high_pkt_priority;
double sv_arrival_rate;
double sv_mean_pkt_len;
double sv_async_mix;
Ici* svjiiac_iciptr;
Distribution* sv_num_addees_dist_ptr;
int sv_num_addees;
int sv_min_num_addees;
int sv_max_num_addees;
int sv_dest_addr[NUM_STATIONS+1];

#define pr_state_ptr ((fddi_gen_mult_state*))
#define inter_dist_ptr pr_state_ptr->sv_inter_dist_ptr
#define len_dist_ptr pr_state_ptr->sv_len_dist_ptr
#define dest_dist_ptr pr_state_ptr->sv_dest_dist_ptr
#define pkt_priority_ptr pr_state_ptr->sv_pkt_priority_ptr
#define my_id pr_state_ptr->sv_mac_objid
#define low_dest_addr pr_state_ptr->sv_low_dest_addr
#define high_dest_addr pr_state_ptr->sv_high_dest_addr
#define station_addr pr_state_ptr->sv_station_addr
#define low_pkt_priority pr_state_ptr->sv_low_pkt_priority
#define high_pkt_priority pr_state_ptr->sv_high_pkt_priority
#define arrival_rate pr_state_ptr->sv_arrival_rate
#define mean_pkt_len pr_state_ptr->sv_mean_pkt_len
#define async_mix pr_state_ptr->sv_async_mix
#define mac_objid pr_state_ptr->sv_mac_objid
#define mac_iciptr pr_state_ptr->sv_mac_iciptr
#define num_addees_dist_ptr pr_state_ptr->sv_num_addees_dist_ptr
#define num_addees pr_state_ptr->sv_num_addees
#define min_num_addees pr_state_ptr->sv_min_num_addees
#define max_num_addees pr_state_ptr->sv_max_num_addees
#define dest_addr pr_state_ptr->sv_dest_addr

/* Process model interrupt handling procedure */

void fddi_gen_mult()
{
    Packet *pkptr;
    int pklen;
    int *da_ptr;
    int i, restricted;
}
74    int    pkt_prio;
75    int    nix;
76
77    FSM_ENTER (fdci_gen_mult)
78
79    FSM BLOCK SWITCH
80    {
81        /******************************
82        /** state (INIT) enter executives **/
83        FSM_STATE_ENTER_UNFORCED (0. state0_enter_exec, "INIT")
84        {
85            /* determine id of own processor to use in finding attrs */
86            my_id = op_id_self ();
87
88        /* 07FEB94: determine the upper and lower limits for multiple */
89        /* addressing from this station. -Nix */
90        op_ima_obj_attr_get (my_id, "min num addees", &min_num_addees);
91        op_ima_obj_attr_get (my_id, "max num addees", &max_num_addees);
92
93        /* 07FEB94: set up a distribution for the number of stations */
94        /* receive this packet. -Nix */
95        num_addees_dist_ptr = op_dist_load ("uniform_int",
96            min_num_addees, max_num_addees);
97
98        /* determine address range for uniform destination assignment */
99        op_ima_obj_attr_get (my_id, "low dest address",
100           &low_dest_addr);
101        op_ima_obj_attr_get (my_id, "high dest address",
102           &high_dest_addr);
103
104        /* determine object id of connected 'mac' layer process */
105        mac_objid = op_topo_assoc (my_id, OPC_TOPO_ASSOC_OUT.
106            OPC_OBJMTYPE_MODULE. MAC_LAYER_OUT_STREAM);
107
108        /* determine the address assigned to it */
109        /* which is also the address of this station */
110        op_ima_obj_attr_get (mac_objid, "station address",
111           &station_addr);
112
113        /* set up a distribution for generation of addresses */
114        dest_dist_ptr = op_dist_load ("uniform_int", low_dest_addr,
115            high_dest_addr);
116
117        /* added 26DEC93 */
118        /* determine priority range for uniform traffic generation */
119        op_ima_obj_attr_get (my_id, "high pkt priority",
120            &high_pkt_priority);
121        op_ima_obj_attr_get (my_id, "low pkt priority",
122            &low_pkt_priority);
/* set up a distribution for generation of priorities */
pkt_priority_ptr = op_dist_load("uniform_int",
    low_pkt_priority, high_pkt_priority);

/* above added 26DEC93 */

/* also determine the arrival rate for packet generation */
op_ima_obj_attr_get(my_id, "arrival rate", &arrival_rate);

/* determine the mix of asynchronous and synchronous */
/* traffic. This is expressed as the proportion of */
/* asynchronous traffic. i.e a value of 1.0 indicates */
/* that all the produced traffic shall be asynchronous. */
op_ima_obj_attr_get(my_id, "async_mix", &async_mix);

/* set up a distribution for arrival generations */
if (arrival_rate != 0.0)
{
/* arrivals are exponentially distributed, with given mean */
    inter_dist_ptr = op_dist_load("constant", 1.0 /
        arrival_rate, 0.0);

/* determine the distribution for packet size */
op_ima_obj_attr_get(my_id, "mean pk length", &mean_pk_len);

/* set up corresponding distribution */
    len_dist_ptr = op_dist_load("constant", mean_pk_len, 0.0);

/* designate the time of first arrival */
fddi_gen_schedule();

/* set up an interface control information (ICI) structure */
/* to communicate parameters to the mac layer process */
/* (it is more efficient to set one up now and keep it */
/* as a state variable than to allocate one on each packet xfer) */
mac_iciptr = op_ici_create("fddi-mac-req");
}

/** blocking after enter executives of unforced state. **/
FSM_EXIT(1, fddi_gen_mult)

/** state (INIT) exit executives **/
FSM_STATE_EXIT_UNFORCED(0, state0_exit_exec, "INIT")

/** state (INIT) transition processing **/
FSM_TRANSIT_FORCE(1, state1_enter_exec, ;)

/*---------------------------*/
/** state (ARRIVAL) enter executives **/
FSM_STATE_ENTER_UNFORCED (1, statel_enter_exec, "ARRIVAL")

/* determine the length of the packet to be generated */
pklen = op_dist_outcome (len_dist_ptr);

/* 07FEB94: re-initialize the destination address array */
/* to zeros. -Nix */
for (i = 0; i < NUM_STATIONS+1; i++)
{ 
    dest_addr[i] = 0;
}

/* determine the destination */
/* don't allow this station's address as a possible outcome */
/* gen_packet: */
/* dest_addr = op_dist_outcome (dest_dist_ptr); */
/* if (dest_addr != -1 && dest_addr == station_addr) */
/* goto gen_packet: */
/* 07FEB94: determine the destinations. -Nix */

/* Determine the number of stations to receive this packet */
num_addees = op_dist_outcome (num_addees_dist_ptr);

/* Find these stations, using num_addees as a counter. -Nix */
for (i = num_addees; i > 0; i--)
{ 
    gen_packet:
    nix = op_dist_outcome (dest_dist_ptr);
    if (dest_addr[nix] == 1 || nix == station_addr)
    { 
        goto gen_packet;
    }
    dest_addr[nix] = 1;
}

/* 05MAR94: because the op_pk_nfd_get() command in FR_REPEAT */
/* overwrites the first field with the array address, an */
/* offset needs to be applied so that the dest_array[0] */
/* contents aren't lost; that is, one field more than the */
/* number of stations is included to allow a one-step shift */
/* that will preserve the address array. In fddi_mac. all */
/* references to dest_addr must allow for this shift. -Nix */
for (i=NUM_STATIONS; i>0; i--)
dest_addr[i] = dest_addr[i-1];

/* 26DEC94 & 29JAN94: determine its priority */
pkt_prio = op_dist_outcome (pkt_priority_ptr);
/* create a packet to send to mac */
    pkptr = op_pk_create_fmt ("fddi_llc_fr");
/* assign its overall size. */
    op_pk_total_size_set (pkptr, pklen);
/* assign the time of creation */
    op_pk_nfd_set (pkptr, "cr_time", op_sim_time ());
/* place the destination address into the ICI */
/* (the protocol_type field will default) */
/* 15MAR94: note that dest_addr now serves as a */
/* pointer to an array in memory, as it is the */
/* name of an array of what will be Os and ls. -Nix */
    op_ici_attr_set (mac_iciptr, "dest_addr", dest_addr);
/* assign the priority, and requested token class */
/* also assign the service class */
/* 29JAN94: the fddi_llc_fr format is modified */
/* to include a "pri" field. -Nix */
    if (op_dist_uniform (1.0) <= async_mix)
        
        op_pk_nfd_set (pkptr, "pri", pkt_prio); /* 29JAN94 */
    op_ici_attr_set (mac_iciptr, "svc_class", FDDI_SVC_ASYNC);
    op_ici_attr_set (mac_iciptr, "pri", pkt_prio); /* 29JAN94 */
    
    } else{
    op_pk_nfd_set (pkptr, "pri", 8); /* 29JAN94 */
    op_ici_attr_set (mac_iciptr, "svc_class", FDDI_SVC_SYNC);
    op_ici_attr_set (mac_iciptr, "pri", 8); /* 29JAN94 */
    }
/* Request only nonrestricted tokens after transmission */
    op_ici_attr_set (mac_iciptr, "tk_class", FDDI_TK_NONRESTRICTED);
/* send the packet coupled with the ICI */
    op_ici_install (mac_iciptr);
    op_pk_send (pkptr, MAC_LAYER_OUT_STREAM);
/* 17MAR94: count and report the running total number */
/* of packets generated. -Nix */
/* genARRIVAL +=; */
/* printf("Packets generated: %d\n", genARRIVAL); */
/* schedule the next arrival */
fddi_gen_schedule();

/* ****************************************************** */
/* 18FEB94: print out the address, and the contents. */
/* for (i=0; i<NUM_STATIONS+1; i++) */
/* printf("ARRIVAL: %d. address: %x; contents: %d\n", */
/* i, &(dest_addr[i]), dest_addr[i]); */
/* ****************************************************** */

/** blocking after enter executives of unforced state. **/
FSM_EXIT (3, fddi_gen_mult)

/** state (ARRIVAL) exit executives **/
FSM_STATE_EXIT_UNFORCED (1, state1_exit_exec, "ARRIVAL")

/** state (ARRIVAL) transition processing **/
FSM_TRANSIT_FORCE (1, state1_enter_exec, ;)

FIN (fddi_gen_mult_svar (prs_ptr))

*var_p_ptr = VOS_NIL;
if (Vos_String_Equal ("inter_dist_ptr", var_name)) *var_p_ptr =
(char*) (&prs_ptr->sv_inter_dist_ptr);
if (Vos_String_Equal ("len_dist_ptr", var_name)) *var_p_ptr =
(char*) (&prs_ptr->sv_len_dist_ptr);
if (Vos_String_Equal ("dest_dist_ptr", var_name)) *var_p_ptr =
(char*) (&prs_ptr->sv_dest_dist_ptr);
if (Vos_String_Equal ("pkt_priority_ptr", var_name)) *var_p_ptr =
(char*) (&prs_ptr->sv_pkt_priority_ptr);
if (Vos_String_Equal ("mac_objid", var_name)) *var_p_ptr = (char*)
(&prs_ptr->sv_mac_objid);
if (Vos_String_Equal ("my_id", var_name)) *var_p_ptr = (char*)
(&prs_ptr->sv_my_id);
if (Vos_String_Equal ("low_dest_addr", var_name)) *var_p_ptr =
(char*) (&prs_ptr->sv_low_dest_addr);
if (Vos_String_Equal ("high_dest_addr", var_name)) *var_p_ptr =
(char*) (&prs_ptr->sv_high_dest_addr);
if (Vos_String_Equal ("station_addr", var_name)) *var_p_ptr =
(char*) (&prs_ptr->sv_station_addr);
if (Vos_String_Equal ("low_pkt_priority", var_name)) *var_p_ptr =
(char*) (&prs_ptr->sv_low_pkt_priority);
if (Vos_String_Equal ("high_pkt_priority", var_name)) *var_p_ptr
= (char*) (&prs_ptr->sv_high_pkt_priority);
if (Vos_String_Equal ("arrival_rate", var_name)) *var_p_ptr =
(char*) (&prs_ptr->sv_arrival_rate);
if (Vos_String_Equal ("mean_pkt_len", var_name)) *var_p_ptr = (char*)
(&prs_ptr->sv_mean_pkt_len);
if (Vos_String_Equal ("async_mix", var_name)) *var_p_ptr = (char*)
(&prs_ptr->sv_async_mix);
if (Vos_String_Equal ("mac_iciptr", var_name)) *var_p_ptr = (char*)
(&prs_ptr->sv_mac_iciptr);
if (Vos_String_Equal ("num_addees_dist_ptr", var_name)) *var_p_ptr =
(char*) (&prs_ptr->sv_num_addees_dist_ptr);
if (Vos_String_Equal ("num_addees", var_name)) *var_p_ptr = (char*)
(&prs_ptr->sv_num_addees);
if (Vos_String_Equal ("min_num_addees", var_name)) *var_p_ptr =
(char*) (&prs_ptr->sv_min_num_addees);
if (Vos_String_Equal ("max_num_addees", var_name)) *var_p_ptr =
(char*) (&prs_ptr->sv_max_num_addees);
if (Vos_String_Equal ("dest_addr", var_name)) *var_p_ptr = (char*)
(&prs_ptr->sv_dest_addr);
FOUT;

void
fddi_gen_mult_diag ()
{
Packet *pkptr;
int pklen;
int *da_ptr;

int i, restricted;
int pkt_prio;
int nix;

FIN (fddi_gen_mult_diag ()
FOUT;
}

void
fddi_gen_mult_terminate ()
{
Packet *pkptr;
int pklen;
324     int     *da_ptr;
325     int     i, restricted;
326     int     pkt_prio;
327     int     nix;
328
329     FIN (fddi_gen_mult_terminate ()
330     FOUT;
331 }
332
333 Compcode
334 fddi_gen_mult_init (pr_state_pptr)
335     fddi_gen_mult_state **pr_state_pptr;
336 {
337     static VosT_Cm_Obtype obtype = OPC_NIL;
338
339     FIN (fddi_gen_mult_init (pr_state_pptr))
340     if (obtype == OPC_NIL)
341     {
342     if (Vos_Catmem_Register ("proc state vars (fddi_gen_mult)*",
343     sizeof (fddi_gen_mult_state), Vos_Nop, &obtype) ==
344     VOSC_FAILURE)
345     FRET (OPC_COMPCODE_FAILURE)
346     }
347     else
348     {
349     (*pr_state_pptr)->current_block = 0;
350     FRET (OPC_COMPCODE_SUCCESS)
351     }
352 }
353
354 static added 2DEC93, on advice from MIL 3, Inc. */
355 fddi_gen_schedule ()
356 {
357     double     inter_time:
358     /* obtain an interarrival period according to the */
359     /* prescribed distribution */
360     inter_time = op_dist outcome (inter_dist_ptr);
361     /* schedule the arrival of next generated packet */
362     op_intrpt_schedule_self (op_sim_time () + inter_time, 0);
APPENDIX G. SINK "C" CODE:

"fddi_sink_mult.pr.c"

The line numbering in this appendix is used for reference within this thesis only, and does not correspond with that seen in OPNET's text editors.

/* Process model C form file: fddi_sink_mult.pr.c */
/* Portions of this file Copyright (C) MIL 3, Inc. 1992 */

/* OPNET system definitions */
#include <opnet.h>
#include "fddi_sink_mult.pr.h"
FSM_EXT_DECS

/* Header block */
/* Globals */
/* array format installed 20JAN94; positions 0-7 represent the asynch priority levels, PRIORITIES + 1 */
/* represents synch traffic, and grand totals are as given in the original. */

#define PRIORITIES 8 /* 20JAN94 */
double fddi_sink_accum_delay = 0.0;
double fddi_sink_accum_delay_a[PRIORITIES + 1] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
int fddi_sink_total_pkts = 0;
int fddi_sink_total_pkts_a[PRIORITIES + 1] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};

int fddi_sink_total_bits = 0;
double fddi_sink_total_bits_a[PRIORITIES + 1] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
double fddi_sink_peak_delay = 0.0;
double fddi_sink_peak_delay_a[PRIORITIES + 2] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
int fddi_sink_scalar_write = 0;
int pri_set = 20; /* 20JAN94 */

/* Externally defined globals. */
extern double fddi_t_opr [];
/* Attributes fromEnvironment file. */

double Offered_Load; /* 12JAN94 */
double Asynch_Offered_Load; /* 12JAN94 */

/* transition expressions */
#define END_OF_SIM op_intrpt_type() == OPC_INTRPT_ENDSIM

/* State variable definitions */
typedef struct
{
  FSM_SYS_STATE
  Gshandle sv_thru_gshandle;
  Gshandle sv_m_delay_gshandle;
  Gshandle sv_ete_delay_gshandle;
  Gshandle sv_thru_gshandle_a[10];
  Gshandle sv_m_delay_gshandle_a[10];
  Gshandle sv_ete_delay_gshandle_a[9];
} fddi_sink_mult_state;

#define pr_state_ptr ((fddi_sink_mult_state*)
  SimMod_State_Ptr)
#define thru_gshandle pr_state_ptr->sv_thru_gshandle
#define m_delay_gshandle pr_state_ptr->sv_m_delay_gshandle
#define ete_delay_gshandle pr_state_ptr->sv_ete_delay_gshandle
#define thru_gshandle_a pr_state_ptr->sv_thru_gshandle_a
#define m_delay_gshandle_a pr_state_ptr->sv_m_delay_gshandle_a
#define ete_delay_gshandle_a pr_state_ptr->sv_ete_delay_gshandle_a

/* Process model interrupt handling procedure */

void fddi_sink_mult ()
{
  double delay, creat_time;
  Packet* pkptr;
  int src_addr, my_addr;
  Ici* from_mac_ici_ptr;
  double fddi_sink_ttrt;

  FSM_ENTER (fddi_sink_mult)
  FSM_BLOCK_SWITCH
  {
    /*************************/
    /** state (DISCARD) enter executives /**
    FSM_STATE_ENTER_UNFORCED (0, state0_enter_exec, "DISCARD")
    {
    /* get the packet and the interface control info */
    pkptr = op_pk_get (op_intrpt_strm ());
    from_mac_ici_ptr = op_intrpt_ici ();
/* 20JAN94: get the packet's priority level, which */
/* will be used to index arrays of thruput and delay */
/* computations. */
/* pri_set = op_pk_priority_get (pkptr); doesn't work here */
op_pk_nfdd_get (pkptr, "pri", &pri_set); /* 29JAN94 */
/* add in its size */
fddi_sink_total_bits += op_pk_total_size_get (pkptr);
fddi_sink_total_bits_a[pri_set] += op_pk_total_size_get
(pkptr); /* 20JAN94 */
/* determine the time of creation of the packet */
op_pk_nfdd_get (pkptr, "cr_time", &creat_time);
/* accumulate delays */
delay = op_sim_time () - creat_time;
fddi_sink_accum_delay += delay;
fddi_sink_accum_delay_a[pri_set] += delay; /* 20JAN94 */
/* keep track of peak delay value */
if (delay > fddi_sink_peak_delay)
fddi_sink_peak_delay = delay;
/* 20JAN94: keep track by priority levels as well 23JAN94 */
if (delay > fddi_sink_peak_delay_a[pri_set])
fddi_sink_peak_delay_a[pri_set] = delay;
/* *********************************************** */
/* printf("DISCARD: pri_set is %d\n", pri_set); */
/* *********************************************** */
/* destroy the packet */
op_pk_destroy (pkptr);
/* 03FEB94: To convert this to the "fddi_sink_link" */
/* model, deactivate the 'destroy' code, and activate */
/* the following 'enqueue' code. This is a first */
/* step toward developing a LAN bridging structure. */
/* -Nix */
/* op_subq_pk_insert (pri_set, pkptr, OPC_QPOS_TAIL); */
/* increment packet counter; 20JAN94 */
fddi_sink_total_pkts++;
fddi_sink_total_pkts_a[pri_set]++;
/* if a multiple of 25 packets is reached, update stats */
/* 03FEB94: [0]->[7] represent async priorities 1->8, */
/* respectively; [8] represents synchronous traffic, */
/* and [9] represents overall asynchronous traffic. -Nix */
if (fddi_sink_total_pkts % 1 == 0)
op_stat_global_write (thru_gshandle, fddi_sink_total_bits / op_sim_time());

op_stat_global_write (thru_gshandle_a[0], fddi_sink_total_bits_a[0] / op_sim_time());

op_stat_global_write (thru_gshandle_a[1], fddi_sink_total_bits_a[1] / op_sim_time());

op_stat_global_write (thru_gshandle_a[2], fddi_sink_total_bits_a[2] / op_sim_time());

op_stat_global_write (thru_gshandle_a[3], fddi_sink_total_bits_a[3] / op_sim_time());

op_stat_global_write (thru_gshandle_a[4], fddi_sink_total_bits_a[4] / op_sim_time());

op_stat_global_write (thru_gshandle_a[5], fddi_sink_total_bits_a[5] / op_sim_time());

op_stat_global_write (thru_gshandle_a[6], fddi_sink_total_bits_a[6] / op_sim_time());

op_stat_global_write (thru_gshandle_a[7], fddi_sink_total_bits_a[7] / op_sim_time());

op_stat_global_write (thru_gshandle_a[8], fddi_sink_total_bits_a[8] / op_sim_time());

op_stat_global_write (m_delay_gshandle, fddi_sink_accum_delay / fddi_sink_total_pkts);

op_stat_global_write (m_delay_gshandle_a[0], fddi_sink_accum_delay_a[0] / fc_sink_total_pkts_a[0]);

op_stat_global_write (m_delay_gshandle_a[1], fddi_sink_accum_delay_a[1] / fcg1_sink_total_pkts_a[1]);

op_stat_global_write (m_delay_gshandle_a[2], fddi_sink_accum_delay_a[2] / fddi_sink_total_pkts_a[2]);

op_stat_global_write (m_delay_gshandle_a[3], fddi_sink_accum_delay_a[3] / fddi_sink_total_pkts_a[3]);

op_stat_global_write (m_delay_gshandle_a[4], fddi_sink_accum_delay_a[4] / fddi_sink_total_pkts_a[4]);

op_stat_global_write (m_delay_gshandle_a[5], fddi_sink_accum_delay_a[5] / fddi_sink_total_pkts_a[5]);

op_stat_global_write (m_delay_gshandle_a[6], fddi_sink_accum_delay_a[6] / fddi_sink_total_pkts_a[6]);
op_stat_global_write(m_delay_gshandle_a[7],
  fddl_sink_accum_delay_a[7] / fddl_sink_total_pkts_a[7]);
op_stat_global_write(m_delay_gshandle_a[8],
  fddl_sink_accum_delay_a[8] / fddl_sink_total_pkts_a[8]);

/* 30JAN94: gather all asynch stats into one figure */
  op_stat_global_write(m_delay_gshandle_a[9],
    (fddl_sink_accum_delay_a[0] + fddl_sink_accum_delay_a[1] +
     fddl_sink_accum_delay_a[2] + fddl_sink_accum_delay_a[3] +
     fddl_sink_accum_delay_a[6] + fddl_sink_accum_delay_a[7]) /
    (fddl_sink_total_pkts_a[0] + fddl_sink_total_pkts_a[1] +
     fddl_sink_total_pkts_a[2] + fddl_sink_total_pkts_a[3] +
     fddl_sink_total_pkts_a[4] + fddl_sink_total_pkts_a[5] +
     fddl_sink_total_pkts_a[6] + fddl_sink_total_pkts_a[7]));

/* also record actual delay values */
opt_stat_global_write(ete_delay_gshandle, delay);
opt_stat_global_write(ete_delay_gshandle_a[pri_set], delay);

/** blocking after enter executives of unforced state. **/
  FSM_EXIT (1, fddl_sink_muit)

/** state (DISCARD) exit executives **/
  FSM_STATE_EXIT_UNFORCED (0, state0_exit_exec, "DISCARD")
  [ ]

/** state (DISCARD) transition processing **/
  FSM_INIT_COND (END_OF_SIM)
  FSM_DFLT_COND
  FSM_TEST_LOGIC ("DISCARD")
  FSM_TRANSIT_SWITCH
  [ ]
  FSM_CASE_TRANSIT (0, 1, statel_enter_exec, ;)
  FSM_CASE_TRANSIT (1, 0, state0_exit_exec, ;)
  ]
  */-----------------------------------------------*

/** state (STATS) enter executives **/
  FSM_STATE_ENTER_UNFORCED (1, statel_enter_exec, "STATS")
  [ ]

188
/* At end of simulation, scalar performance statistics */
/* and input parameters are written out. */

/* Only one station needs to do this */
if (fddi_sink_scalar_write)
{
/* set the scalar write flag */
fddi_sink_scalar_write = 1;
op_stat_scalar_write("Mean End-to-End Delay (sec.),
Priority 1", fddi_sink_accum_delay_a[0] / fddi_sink_total_pkts_a[0]);
op_stat_scalar_write("Mean End-to-End Delay (sec.),
Priority 2", fddi_sink_accum_delay_a[1] / fddi_sink_total_pkts_a[1]);
op_stat_scalar_write("Mean End-to-End Delay (sec.),
Priority 3", fddi_sink_accum_delay_a[2] / fddi_sink_total_pkts_a[2]);
op_stat_scalar_write("Mean End-to-End Delay (sec.),
Priority 4", fddi_sink_accum_delay_a[3] / fddi_sink_total_pkts_a[3]);
op_stat_scalar_write("Mean End-to-End Delay (sec.),
Priority 5", fddi_sink_accum_delay_a[4] / fddi_sink_total_pkts_a[4]);
op_stat_scalar_write("Mean End-to-End Delay (sec.),
Priority 6", fddi_sink_accum_delay_a[5] / fddi_sink_total_pkts_a[5]);
op_stat_scalar_write("Mean End-to-End Delay (sec.),
Priority 7", fddi_sink_accum_delay_a[6] / fddi_sink_total_pkts_a[6]);
op_stat_scalar_write("Mean End-to-End Delay (sec.),
Priority 8", fddi_sink_accum_delay_a[7] / fddi_sink_total_pkts_a[7]);
op_stat_scalar_write("Mean End-to-End Delay (sec.),
Asynchronous", (fddi_sink_accum_delay - fddi_sink_accum_delay_a[8]) / (fddi_sink_total_pkts - fddi_sink_total_pkts_a[8]));

/* (fddi_sink_accum_delay_a[0] + fddi_sink_accum_delay_a[1] + */
/* fddi_sink_accum_delay_a[6] + fddi_sink_accum_delay_a[7]) / */
/* (fddi_sink_total_pkts_a[0] + fddi_sink_total_pkts_a[1] + */
/* fddi_sink_total_pkts_a[6] + fddi_sink_total_pkts_a[7]); */

op_stat_scalar_write("Mean End-to-End Delay (sec.),
Synchronous", fddi_sink_accum_delay_a[8] / fddi_sink_total_pkts_a[8]);
op_stat_scalar_write("Mean End-to-End Delay (sec.), Total", fddi_sink_accum_delay / fddi_sink_total_pkts);
op_stat_scalar_write("Throughput (bps), Priority 1", fddi_sink_total_bits_a[0] / op_sim_time ());
op_stat_scalar_write("Throughput (bps), Priority 2", fddi_sink_total_bits_a[1] / op_sim_time ());
op_stat_scalar_write("Throughput (bps), Priority 3", fddi_sink_total_bits_a[2] / op_sim_time ());
op_stat_scalar_write("Throughput (bps), Priority 4", fddi_sink_total_bits_a[3] / op_sim_time ());
op_stat_scalar_write("Throughput (bps), Priority 5", fddi_sink_total_bits_a[4] / op_sim_time ());

op_stat_scalar_write("Throughput (bps), Priority 6", fddi_sink_total_bits_a[5] / op_sim_time ());
op_stat_scalar_write("Throughput (bps), Priority 7", fddi_sink_total_bits_a[6] / op_sim_time ());
op_stat_scalar_write("Throughput (bps), Priority 8", fddi_sink_total_bits_a[7] / op_sim_time ());

op_stat_scalar_write("Throughput (bps), Asynchronous", (fddi_sink_total_bits - fddi_sink_total_bits_a[8]) / op_sim_time ());

/* (fddi_sink_total_bits_a[0] + fddi_sink_total_bits_a[1]) + */
/* fddi_sink_total_bits_a[6] + fddi_sink_total_bits_a[7]) / */
/* op_sim_time ()); */

op_stat_scalar_write("Throughput (bps), Synchronous", fddi_sink_total_bits_a[8] / op_sim_time ());
op_stat_scalar_write("Throughput (bps), Total", fddi_sink_total_bits / op_sim_time ());

op_stat_scalar_write("Peak End-to-End Delay (sec.), Priority 1", fddi_sink_peak_delay_a[0]);
op_stat_scalar_write("Peak End-to-End Delay (sec.), Priority 2", fddi_sink_peak_delay_a[1]);
op_stat_scalar_write("Peak End-to-End Delay (sec.), Priority 3", fddi_sink_peak_delay_a[2]);
op_stat_scalar_write("Peak End-to-End Delay (sec.), Priority 4", fddi_sink_peak_delay_a[3]);
op_stat_scalar_write("Peak End-to-End Delay (sec.), Priority 5", fddi_sink_peak_delay_a[4]);
op_stat_scalar_write("Peak End-to-End Delay (sec.), Priority 6", fddi_sink_peak_delay_a[5]);
op_stat_scalar_write("Peak End-to-End Delay (sec.), Priority 7", fddi_sink_peak_delay_a[6]);
op_stat_scalar_write("Peak End-to-End Delay (sec.), Priority 8", fddi_sink_peak_delay_a[7]);
295 \texttt{op\_stat\_scalar\_write ("Peak End-to-End Delay (sec.),}
296 \texttt{Synchronous", fddi\_sink\_peak\_delay\_a[8]);}
297 \texttt{op\_stat\_scalar\_write ("Peak End-to-End Delay (sec.), Overall",}
298 \texttt{fddi\_sink\_peak\_delay);}
299
300 /* Write the TTRT value for ring 0. This preserves */
301 /* the old behavior for single-ring simulations. */
302 \texttt{op\_stat\_scalar\_write ("TTRT (sec.) - Ring 0", fddi\_t\_opr}
303 \texttt{[0]);}
304
305 /* 12JAN94: obtain offered load information from the Environment */
306 /* file; this will be used to provide abscissa information that */
307 /* can be plotted in the Analysis Editor (see "fddi\_sink" STATS */
308 /* state. To the user: it's your job to keep these current in */
309 /* the Environment File. -Nix */
310 \texttt{op\_ima\_sim\_attr\_get (OPC\_IMA\_DOUBLE, "total\_offered\_load",}
311 \texttt{&OfferedLoad);
312 \texttt{op\_ima\_sim\_attr\_get (OPC\_IMA\_DOUBLE, "asynch\_offered\_load",}
313 \texttt{&Asynch\_Offered\_Load);}
314
315 /* 12JAN94: write the total offered load for this run */
316 \texttt{op\_stat\_scalar\_write ("Total Offered Load}
317 \texttt{(Mbps), Offered\_Load);
318 \texttt{op\_stat\_scalar\_write ("Asynchronous Offered Load (Mbps)",}
319 \texttt{Asynch\_Offered\_Load);}
320 \texttt{)}
321
322 /** blocking after enter executives of unforced state. **/}
323 \texttt{FSM\_EXIT (3, fddi\_sink\_mult)}
324
325 /** state (STATS) exit executives **/}
326 \texttt{FSM\_STATE\_EXIT\_UNFORCED (1, state\_exit\_exec, "STATS")}
327 \texttt{[}
328 \texttt{]}
329
330 /** state (STATS) transition processing **/}
331 \texttt{FSM\_TRANSIT\_MISSING ("STATS")}
332 \texttt{-----------------------------*/}
333
334 /** state (INIT) enter executives **/}
335 \texttt{FSM\_STATE\_ENTER\_FORCED (2, state\_enter\_exec, "INIT")}
336 \texttt{[}
337 \texttt{]}
338 /* get the gshandles of the global statistic to be obtained */
339 /* 20JAN94: set array format */
340
341 \texttt{thru\_gshandle\_a[0] = op\_stat\_global\_reg ("pri 1 throughput}
342 \texttt{(bps));}
343
344 191
thru_gshandle_a[1] = op_stat_global_reg ("pri 2 throughput (bps)");
thru_gshandle_a[2] = op_stat_global_reg ("pri 3 throughput (bps)");
thru_gshandle_a[3] = op_stat_global_reg ("pri 4 throughput (bps)");
thru_gshandle_a[4] = op_stat_global_reg ("pri 5 throughput (bps)");
thru_gshandle_a[5] = op_stat_global_reg ("pri 6 throughput (bps)");
thru_gshandle_a[6] = op_stat_global_reg ("pri 7 throughput (bps)");
thru_gshandle_a[7] = op_stat_global_reg ("pri 8 throughput (bps)");
thru_gshandle_a[8] = op_stat_global_reg ("synch throughput (bps)");
thru_gshandle_a[9] = op_stat_global_reg ("async throughput (bps)");
thru_gshandle = op_stat_global_reg ("total throughput (bps)");
m_delay_gshandle_a[0] = op_stat_global_reg ("pri 1 mean delay (sec.)");
m_delay_gshandle_a[1] = op_stat_global_reg ("pri 2 mean delay (sec.)");
m_delay_gshandle_a[2] = op_stat_global_reg ("pri 3 mean delay (sec.)");
m_delay_gshandle_a[3] = op_stat_global_reg ("pri 4 mean delay (sec.)");
m_delay_gshandle_a[4] = op_stat_global_reg ("pri 5 mean delay (sec.)");
m_delay_gshandle_a[5] = op_stat_global_reg ("pri 6 mean delay (sec.)");
m_delay_gshandle_a[6] = op_stat_global_reg ("pri 7 mean delay (sec.)");
m_delay_gshandle_a[7] = op_stat_global_reg ("pri 8 mean delay (sec.)");
m_delay_gshandle_a[8] = op_stat_global_reg ("synch mean delay (sec.)");
m_delay_gshandle_a[9] = op_stat_global_reg ("async mean delay (sec.)");
m_delay_gshandle = op_stat_global_reg ("total mean delay (sec.)");
ete_delay_gshandle_a[0] = op_stat_global_reg ("pri 1 end-to-end delay (sec.)");
ete_delay_gshandle_a[1] = op_stat_global_reg ("pri 2 end-to-end delay (sec.)");
ete_delay_gshandle_a[2] = op_stat_global_reg ("pri 3 end-to-end delay (sec.)");
ete_delay_gshandle_a[3] = op_stat_global_reg ("pri 4 end-to-end delay (sec.)");

ete_delay_gshandle_a[4] = op_stat_global_reg("pri 5 end-to-end
delay (sec.)");
ete_delay_gshandle_a[5] = op_stat_global_reg("pri 6 end-to-end
delay (sec.)");
ete_delay_gshandle_a[6] = op_stat_global_reg("pri 7 end-to-end
delay (sec.)");
ete_delay_gshandle_a[7] = op_stat_global_reg("pri 8 end-to-end
delay (sec.)");
ete_delay_gshandle_a[8] = op_stat_global_reg("synch end-to-end
delay (sec.)");
ete_delay_gshandle = op_stat_global_reg("total end-to-end
delay (sec.)");
}

/** state (INIT) exit executives **/
FSM_STATE_EXIT_FORCED (2, state2_exit_exec, "INIT")
{
}

/** state (INIT) transition processing **/
FSM_INIT_COND (END_OF_SIM)
FSM_DFLT_COND
FSM_TEST_LOGIC ("INIT")

FSM_TRANSIT_SWITCH
{
    FSM_CASE_TRANSIT (0, 1, state1_enter_exec, ,)
    FSM_CASE_TRANSIT (1, 0, state0_enter_exec, ,)
}

/*------------------------------------------------------------------*/
FSM_EXIT (2.fddi_sink_mult)
}

void
fddi_sink_mult_svar (prs_ptr, var_name, var_p_ptr)
{
    fddi_sink_mult_state *prs_ptr;
    char *var_name, **var_p_ptr;

    FIN (fddi_sink_mult_svar (prs_ptr))

    *var_p_ptr = VOS NIL;
    if (Vos_String_Equal ("thru_gshandle" , var_name))
        *var_p_ptr = (char *) (&prs_ptr->sv_thru_gshandle);
    if (Vos_String_Equal ("m_delay_gshandle" , var_name))
        *var_p_ptr = (char *) (&prs_ptr->sv_m_delay_gshandle);
    if (Vos_String_Equal ("ete_delay_gshandle" , var_name))
        *var_p_ptr = (char *) (&prs_ptr->sv_ete_delay_gshandle);

if (Vos_String_Equal("thru_gshandle_a", var_name))
    *var_p_ptr = (char *) (prs_ptr->sv_thru_gshandle_a);
if (Vos_String_Equal("m_delay_gshandle_a", var_name))
    *var_p_ptr = (char *) (prs_ptr->sv_m_delay_gshandle_a);
if (Vos_String_Equal("ete_delay_gshandle_a", var_name))
    *var_p_ptr = (char *) (prs_ptr->sv_ete_delay_gshandle_a);
FOUT;
}

void
fddi_sink_mult_diag ()
{
    double delay, creat_time;
    Packet* pkptr;
    int src_addr, my_addr;
    Ici* from_mac_ici_ptr;
    double fddi_sink_ttrt;

    FIN (fddi_sink_mult_diag ()
    FOUT;
}

void
fddi_sink_mult_terminate ()
{
    double delay, creat_time;
    Packet* pkptr;
    int src_addr, my_addr;
    Ici* from_mac_ici_ptr;
    double fddi_sink_ttrt;

    FIN (fddi_sink_mult_terminate ()
    FOUT;
}

Compcode
fddi_sink_mult_init (pr_state_pptr)
    fddi_sink_mult_state **pr_state_pptr;
{
    static VosT_Cm_Obtype obtype = OPC NIL;

    FIN (fddi_sink_mult_init (pr_state_pptr))
    if (obtype == OPC NIL)
    {
        if (Vos_Catmem_Register ("proc state vars (fddi_sink_mult)",
            sizeof (fddi_sink_mult_state), Vos_Nop, &obtype) ==
            VOSC_FAILURE)
            FRET (OPC_COMPCODE_FAILURE)
if (**pr_state_pptr == (fddi sink mult state*) Vos_Catmem_Alloc
    (obtype, 1)) == OPCNIL)
    FRET (OPC_COMPCODE_FAILURE)
    else
    {
    (**pr_state_pptr)->current_block = 4;
    FRET (OPC_COMPCODE_SUCCESS)
    }
APPENDIX H. ENVIRONMENT FILE FOR
50-STATION MULTICAST CAPABLE FDDI LAN

# fddi50mult.ef
# sample simulation configuration file for fddi example model
# 50 station network with multiple addressing capability

### Attributes related to loading used by "fddi_gen" ###

```plaintext
# station addresses
.*.f0.mac.station_address: 0
.*.f1.mac.station_address: 1
.*.f2.mac.station_address: 2
.*.f3.mac.station_address: 3
.*.f4.mac.station_address: 4
.*.f5.mac.station_address: 5
.*.f6.mac.station_address: 6
.*.f7.mac.station_address: 7
.*.f8.mac.station_address: 8
.*.f9.mac.station_address: 9
.*.f10.mac.station_address: 10
.*.f11.mac.station_address: 11
.*.f12.mac.station_address: 12
.*.f13.mac.station_address: 13
.*.f14.mac.station_address: 14
.*.f15.mac.station_address: 15
.*.f16.mac.station_address: 16
.*.f17.mac.station_address: 17
.*.f18.mac.station_address: 18
.*.f19.mac.station_address: 19
.*.f20.mac.station_address: 20
.*.f21.mac.station_address: 21
.*.f22.mac.station_address: 22
.*.f23.mac.station_address: 23
.*.f24.mac.station_address: 24
.*.f25.mac.station_address: 25
.*.f26.mac.station_address: 26
.*.f27.mac.station_address: 27
.*.f28.mac.station_address: 28
.*.f29.mac.station_address: 29
.*.f30.mac.station_address: 30
.*.f31.mac.station_address: 31
.*.f32.mac.station_address: 32
.*.f33.mac.station_address: 33
```
# Range number of stations that may receive this packet if more
# than one is designated (model defaults are both 1)
# Note that the code does not allow the originating station to
# address a packet to itself, so max_num_addees is less than
# the number of stations.
"llc_src.min_num_addees" : 1
"llc_src.max_num_addees" : 1

# destination addresses for random message generation
"llc_src.low_dest_address" : 0
"llc_src.high_dest_address" : 49

# range of priority values that can be assigned to packets; FDDI
# standards allow for 8 priorities of asynchronous traffic. MIL3's
# original model is modified to allow each station to generate
# multiple priorities, within a specified range. (Note that while
# research literature refers to asynchronous priorities ranging
# from 1 to 8, the corresponding numbering here is 0 to 7, in
# keeping with the C language array element numbering convention.)
"llc_src.high_pkt_priority" : 7
"llc_src.low_pkt_priority" : 0

# arrival rate(frames/sec), and message size (bits) for random
# message generation at each station on the ring.
"arrival_rate" : 750
"mean_pk_length" : 1000

# These are the synchronous transmitters
**.f0.*.arrival rate" : 6000
**.f0.*.mean pk length" : 512
**.f5.*.arrival rate" : 6000
**.f5.*.mean pk length" : 512
**.f10.*.arrival rate" : 6000
**.f10.*.mean pk length" : 512
**.f15.*.arrival rate" : 6000
**.f15.*.mean pk length" : 512
**.f20.*.arrival rate" : 600000
**.f20.*.mean pk length" : 512
**.f25.*.arrival rate" : 6000
**.f25.*.mean pk length" : 512
**.f30.*.arrival rate" : 6000
**.f30.*.mean pk length" : 512
**.f35.*.arrival rate" : 6000
**.f35.*.mean pk length" : 512
**.f40.*.arrival rate" : 6000
**.f40.*.mean pk length" : 512
**.f45.*.arrival rate" : 600000
**.f45.*.mean pk length" : 512

# 12DEC93: total offered load is the sum of all stations' loads (Mbps). Compute this by hand; this value is used in the sink process model for generating scalar plots where offered load is the abscissa.

```
total_offered_load = 60.72
asynch_offered_load = 30.00
```

# set the proportion of asynchronous traffic
# a value of 1.0 indicates all asynchronous traffic
**.f0.*.async_mix" : 1.0
**.f5.*.async_mix" : 0.0
**.f10.*.async_mix" : 0.0
**.f15.*.async_mix" : 0.0
**.f20.*.async_mix" : 0.0
**.f25.*.async_mix" : 0.0
**.f30.*.async_mix" : 0.0
**.f35.*.async_mix" : 0.0
**.f40.*.async_mix" : 0.0
**.f45.*.async_mix" : 0.0

### Ring configuration attributes used by "fddi_mac" ###

# allocate percentage of synchronous bandwidth to each station
# this value should not exceed 1 for all stations combined; OPNET does not enforce this; 01FEB94: this must be less than 1; see equation below
**.f0.*.mac.sync bandwidth" : 0.0


**.f0.mac.sync bandwidth**: 0.09358
**.f5.mac.sync bandwidth**: 0.09358
**.f10.mac.sync bandwidth**: 0.09358
**.f15.mac.sync bandwidth**: 0.09358
**.f20.mac.sync bandwidth**: 0.09358
**.f25.mac.sync bandwidth**: 0.09358
**.f30.mac.sync bandwidth**: 0.09358
**.f35.mac.sync bandwidth**: 0.09358
**.f40.mac.sync bandwidth**: 0.09358
**.f45.mac.sync bandwidth**: 0.09358

# Target Token Rotation Time (one half of maximum synchronous response time)

**.*.mac.T_Req**: 0.0107

# Index of the station which initially launches the token

"spawn station": 0

# Delay incurred by packets as they traverse a station’s ring interface (see Powers, p. 351 for a discussion of this (Powers gives lusec, but 60.0e-08 agrees with Dykeman & Bux)

station_latency: 60.0e-08

# Propagation Delay separating stations on the ring.

prop_delay: 5.085e-06

# Simulation related attributes

# Token Acceleration Mechanism enabling flag.
# It is recommended that this mechanism be enabled for most situations

accelerate_token: 1

seed: 10

# Run control attributes

duration: 0.5

verbose_sim: TRUE

upd_int: 0.1

os_file: fddi50mult

ov_file: fddi50mult

# Opnet Debugger (odb) enabling attribute

debug: TRUE
APPENDIX I. CONVENTIONS

One of the purposes of this report is that it will be used both as a teaching tool and a springboard for future researchers and assessors of fiber optic network simulations implementing OPNET®. Throughout the writing of this report, the author has kept these goals in sight and the resulting narrative contains technical stylistic conventions in keeping with the projected use of this material in a teaching, reference, and research environment. These conventions, implemented in the narrative portion of this report only, are briefly described here.

All excerpted programming code fragments are isolated on their own lines within the text and highlighted by a standard san-serif font. Variable names, function names, and names of programming objects referred to within the text of the report are also highlighted in this manner, with a standard san-serif font. Messages from the computer or responses to be made to computer queries are set off in double quotes and a "bold standard san-serif font." Single keystrokes are highlighted in capitalized italics, (e.g. \texttt{CTRL+S}), while parameters are also set off in the same manner (e.g. \texttt{<number of nodes>}).
# APPENDIX J. GLOSSARY

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BONeS*</td>
<td>Block Oriented Network Simulator</td>
</tr>
<tr>
<td>CDL</td>
<td>Common Data Link</td>
</tr>
<tr>
<td>DSPO</td>
<td>Defense Support Project Office</td>
</tr>
<tr>
<td>Environment file</td>
<td>A command file containing descriptors and values utilized by a system to define the operating parameters. Sometimes this file is referred to as the &quot;configuration file.&quot;</td>
</tr>
<tr>
<td>FDDI LAN</td>
<td>Fiber Distributed Data Interface Local Area Network</td>
</tr>
<tr>
<td>MAC</td>
<td>Medium Access Control</td>
</tr>
<tr>
<td>OPNET®</td>
<td>Optimized Network Engineering Tool</td>
</tr>
<tr>
<td>THT</td>
<td>Token Holding Timer</td>
</tr>
<tr>
<td>TRT</td>
<td>Token Rotation Timer</td>
</tr>
<tr>
<td>TTRT</td>
<td>Target Token Rotation Time</td>
</tr>
</tbody>
</table>
LIST OF REFERENCES


## INITIAL DISTRIBUTION LIST

<table>
<thead>
<tr>
<th>No.</th>
<th>Copies</th>
<th>Name and Address</th>
</tr>
</thead>
</table>
| 1.  | 2      | Defense Technical Information Center  
Cameron Station  
Alexandria, VA 22304-6145 |
| 2.  | 2      | Library, Code 52  
Naval Postgraduate School  
Monterey, CA 93943-5101 |
| 3.  | 1      | Chairman, Code EC  
Department of Electrical and Computer Engineering  
Naval Postgraduate School  
Monterey, CA 93943-5121 |
| 4.  | 2      | Professor Shridhar Shukla, Code EC/Sh  
Department of Electrical and Computer Engineering  
Naval Postgraduate School  
Monterey, CA 93943-5121 |
| 5.  | 2      | Professor Gilbert Lundy, Code CS/Ln  
Department of Computer Science  
Naval Postgraduate School  
Monterey, CA 93943-5118 |
| 6.  | 1      | Professor Paul Moose, Code EC/Me  
Department of Electrical and Computer Engineering  
Naval Postgraduate School  
Monterey, CA 93943-5121 |
| 7.  | 1      | Mark Russon, UNISYS  
Mail Station F2-G14  
640 North 2200 West  
Salt Lake City, UT 84116-2988 |
8. CDL Program Manager  
   Defense Support Project Office  
   Washington D.C. 20330-1000

9. LT Ernest E. Nix, Jr., USN  
   502 Paris View Dr.  
   Travelers Rest, SC 29690