Data Link Level Interconnection of Remote Fiber Distributed Data Interface Local Area Networks (FDDI LANs) Through the Critical Data Link (CDL)

by

Selcuk Karayakaylar

June 1994

Thesis Advisor: Shridhar B. Shukla

Approved for public release, distribution is unlimited.

94-29487
# Title and Subtitle
DATA LINK LEVEL INTERCONNECTION OF REMOTE FIBER DISTRIBUTED DATA INTERFACE LOCAL AREA NETWORKS (FDDI LANs) THROUGH THE CRITICAL DATA LINK (CDL)

# Authors
Karayakaylar, Selcuk

# Performing Organization Name and Address
Naval Postgraduate School
Monterey, CA 93943-5000

# Abstract
This thesis deals with the features and performance of a network interface device to interconnect two remote Fiber Distributed Data Interface (FDDI) Local Area Networks (LANs) through the Critical Data Link (CDL) which is a full-duplex, jam-resistant, point-to-point microwave communications system for use in imagery and signals intelligence collection systems. In particular, OPNET, a commercially available network engineering tool is used to model a medium access level remote bridge interface connecting two LANs. The effectiveness of two different load balancing techniques used to distribute traffic over the multiple channels of the CDL has been studied. Also, the effect of different jamming patterns on the bit error rate seen by the users has been studied.

# DTIC Quality
INSPECTED 3

# Subject Terms
FDDI, LAN, CDL, MAC, LLC, bridge, simulation

# Number of Pages
187

# Price Code
UL

# Security Classification
UNCLASSIFIED
Data Link Level Interconnection of Remote Fiber Distributed Data Interface Local Area Networks (FDDI LANs) Through the Critical Data Link (CDL)

by

Selcuk Karayakaylar
Lieutenant Junior Grade, Turkish Navy
B.S.E.E., Turkish Naval Academy, 1988

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL

June 1994

Author: Selcuk Karayakaylar

Approved by:

Shridhar B. Shukla, Thesis Advisor

Gilbert Lundy, Second Reader

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

This thesis deals with the features and performance of a network interface device to interconnect two remote Fiber Distributed Data Interface (FDDI) Local Area Networks (LANs) through the Critical Data Link (CDL) which is a full-duplex, jam-resistant, point-to-point microwave communications system for use in imagery and signals intelligence collection systems. In particular, OPNET, a commercially available network engineering tool is used to model a medium access level remote bridge interface connecting two LANs. The effectiveness of two different load balancing techniques used to distribute traffic over the multiple channels of the CDL has been studied. Also, the effect of different jamming patterns on the bit error rate seen by the users has been studied.
# TABLE OF CONTENTS

## I. INTRODUCTION

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

## II. INTERCONNECTION OF FDDI LOCAL AREA NETWORKS THROUGH CDL

A. OVERVIEW .......................................................... 3
B. ISSUES IN HIGH SPEED LAN INTERCONNECTION ............ 3
   1. Preliminary .................................................... 3
   2. Bridges versus Routers ...................................... 5
C. PROPOSED BRIDGING METHOD .................................. 6
   1. Current Standards ......................................... 6
   2. Remote Bridging ............................................. 6
D. UNIQUENESS OF THE CDL ENVIRONMENT .................... 8
   1. Description ................................................ 8
   2. System Requirements ...................................... 9
   3. Asymmetry between Command and Return Link .......... 9
   4. Considerations on Link Utilization ...................... 10
   5. The Effects of Jamming ................................... 11

## III. MODELING CDL NETWORK INTERFACE IN OPNET ......... 13

A. OVERVIEW .......................................................... 13
B. REVISED FDDI LAN MODEL .................................. 14
   1. FDDI LAN Model in Brief ................................. 14
   2. Modifications on FDDI Station Model ................. 18
      a. Preliminary ............................................. 18
      b. Source Node Modifications ......................... 18
      c. Sink Node Modifications ............................ 19
C. BRIDGE MODEL .................................................. 19
   1. Preliminary ................................................ 19
2. The CPNI ........................................ 19
   a. Station Model ................................ 19
   b. Source Node Modifications .................... 22
   c. MAC Node Modifications ........................ 22
   d. Sink Node Modifications ....................... 23
3. The SPNI ........................................ 24
   a. Station Model ................................ 24
   b. Source Node Modifications .................... 26
   c. MAC Node Modifications ........................ 26
   d. Sink Node Modifications ....................... 26
D. CDL MODEL ....................................... 27
   1. OPNET Model for Point-to-Point Links .......... 27
      a. Preliminary ................................ 27
      b. Transmitters ................................ 28
      c. Receivers .................................. 28
      d. Transceiver Pipeline Stages ................. 28
   2. Error Modeling over Multiple Links ............. 30
      a. Modifications on Error Allocation Pipeline Stage 30
      b. Jammer Implementations ...................... 31
E. LAN INTERCONNECTION ............................ 33
   1. Addressing through The Remote Bridge ........... 36
   2. Load Balancing over Multiple Links ............. 37
      a. Circular Allocation Algorithm ............... 43
      b. Empty Selection Algorithm .................. 43
F. FAITHFULNESS OF THE MODEL ....................... 44
IV. MODEL TESTING .................................... 47
   A. OVERVIEW ..................................... 47
   B. PERFORMANCE METRICS ......................... 47
   C. TRAFFIC MONITORING ........................... 48
      1. Overview ................................... 48
      2. Setup ..................................... 48
      3. Results ................................... 49
   D. RETURN LINK PERFORMANCE ....................... 52
      1. Overview ................................... 52
2. First Test ........................................ 52
   a. Setup ........................................ 52
   b. Results ..................................... 52
3. Second Test ..................................... 56
   a. Setup ........................................ 56
   b. Results ..................................... 56
E. COMMAND LINK PERFORMANCE ...................... 60
V. CONCLUSIONS AND RECOMMENDATIONS ...................... 63
   A. CONCLUSIONS .............................. 63
   B. RECOMMENDATIONS .......................... 63
APPENDIX A: CPNI SOURCE “C” CODE ....................... 65
APPENDIX B: CPNI MAC “C” CODE ........................ 75
APPENDIX C: CPNI SINK “C” CODE ....................... 115
APPENDIX D: SPNI SOURCE “C” CODE ....................... 133
APPENDIX E: SPNI MAC “C” CODE EXCERPT ............... 143
APPENDIX F: SPNI SINK “C” CODE ....................... 145
APPENDIX G: CDL MODEL ERROR ALLOCATION CODE ........ 161
APPENDIX H: SAMPLE ENVIRONMENT FILE FOR PULSED
JAMMER ............................................. 165
APPENDIX I: SAMPLE ENVIRONMENT FILE EXCERPT FOR
CHANNEL-SWEPT JAMMER ............................ 171
REFERENCES ........................................ 173
INITIAL DISTRIBUTION LIST ............................ 175
# LIST OF FIGURES

1. Protocols relayed by the bridges and routers ........................................ 4
3. FDDI LAN ring representation .......................................................... 15
4. Ten-station FDDI LAN, fddi_net_10 ................................................... 15
5. FDDI station model, fddi_station .......................................................... 16
6. Source process model, “fddi_gen” ....................................................... 16
7. MAC process model, “fddi_mac” ............................................................. 17
8. Sink process model, “fddi_sink” ............................................................. 17
9. Collection Platform LAN Network Interface (CPNI) ............................. 20
10. Surface Platform LAN Network Interface (SPNI) .................................... 25
11. The attributes used for jamming patterns ............................................. 32
12. Pulsed jammer representation .............................................................. 34
13. Channel-swept jammer representation .................................................. 34
14. Interconnected network model, fddi_cd1 ................................................ 35
15. Modified packet format, “fddi_llc_fr” .................................................. 35
17. Modified ICI packet format, “fddi_mac_req” ....................................... 40
18. Return link data rates and new attributes for llc_sink ............................ 41
19. Command link data rate ........................................................................ 42
20. Local throughput of surface LAN .......................................................... 50
21. Traffic directed to the command link ..................................................... 50
22. Local throughput of collection platform LAN ....................................... 51
23. Throughput directed to the return link .................................................. 51
24. Accumulation of packets on the CPNI buffers with circular allocation .... 53
25. Buffer overflows in the CPNI with circular allocation ............................ 54
26 Accumulation of packets in the CPNI buffers with empty selection . . 54
27 Queing delay of the CPNI buffers with circular allocation ............. 55
28 Queing delay of the CPNI buffers with empty selection ............. 55
29 Throughput at the transmit-end of the return link .................... 57
30 Average BER of the return link caused by pulsed jammer ............. 58
31 Average BER of the return link caused by channel-swept jammer .... 58
32 Effect of pulsed jammer on the return link throughput ............... 59
33 Effect of channel-swept jammer on the return link throughput ....... 59
34 Accumulation of packets in the SPNI buffer ........................... 61
35 Queing delay of the SPNI ............................................ 61
36 Throughput of the transmit and receive-ends of the command link .. 62
ACKNOWLEDGMENT

I wish to thank my thesis advisor Professor Shridhar B. Shukla for his guidance and encouragement in this research.

OPNET is a registered trademark of MIL 3 Inc.
I. INTRODUCTION

A. PROBLEM STATEMENT

This thesis deals with data link level interconnection of remote Local Area Networks (LANs) through Critical Data Link (CDL). The Critical Data Link (CDL) is a full-duplex, jam-resistant, point-to-point microwave communications system for use in imagery and signals intelligence collection systems. The CDL system is designed to provide a communications protocol between two or more Fiber Distributed Data Interface Local Area Networks (FDDI LAN). While the collection platform LAN is in the form of an airborne LAN that provides sensor information, the command information is provided by a ground-based LAN.

This study is concerned with the modeling of a CDL Network Interface (NI) using a commercially available network simulation program, MIL 3 Inc.'s Optimized Network Engineering Tool (OPNET).

B. SCOPE

The scope of this thesis is as follows:

- Model the interconnection of FDDI LANs in the unique environment of CDL deployment.

- Evaluate the efficiency of the LAN-CDL interface.

- Establish the necessary basis for running multilink point-to-point protocol in CDL deployment.
C. BENEFITS

This thesis continues the development of CDL related simulation/modeling programs started in [1]. In that study, the capabilities of OPNET's original FDDI LAN model were enhanced for the use in CDL deployment. We have added:

- A model of NI that performs load balancing over multiple links.

- A CDL model with appropriate jamming patterns that faithfully represents real conditions.

Using these, we have carried out several simulation experiments to determine the impact of load balancing and jamming on bit throughput.

D. ORGANIZATION

This thesis is organized in five chapters. A brief introduction, the discussion of FDDI LAN interconnection through CDL is provided in Chapter II. Chapter III presents the proposed NI model development in OPNET. The simulation results are supplied in Chapter IV. The conclusions and recommendations for future studies are in Chapter V.
II. INTERCONNECTION OF FDDI LOCAL AREA NETWORKS THROUGH CDL

A. OVERVIEW

This chapter addresses the issues related to the interconnection of FDDI LANs through the unique environment of the CDL. First, architectural alternatives available for implementing such a connectivity are discussed. Then, existing bridging mechanisms are considered to determine the ideal candidate for a NI for such deployment. The system specific features of CDL are also stated briefly in order to justify the proposed NI architecture.

B. ISSUES IN HIGH SPEED LAN INTERCONNECTION

1. Preliminary

Local area networks are limited in geography, traffic handling capability, and the number of stations. A single LAN is not likely to meet the needs of many organizations, specifically military applications which require sophisticated command, control and communication functions.

This makes the integration of any LAN topology with other LANs through a communication link necessary. The protocols used to achieve this integration can be at either the network or data link layers of ISO's Open Systems Interconnection (OSI) Model.

Interconnection at the network layer is achieved through routing devices. At the data link layer it is achieved using bridges. Figure 1 shows the layer hierarchies. In general, each layer of an architecture is associated with an additional
Figure 1: Protocols relayed by the bridges and routers.
level of addressing regardless of the functions performed within a layer. Furthermore, functional distinctions between them are major factors to determine the appropriate layer of interconnection.

2. Bridges versus Routers

Bridges are Medium Access Control (MAC) level store and forward devices that are independent of higher level protocols. They are transparent to communicating end stations. This is particularly desirable in many high speed applications. Bridges make simple forwarding or filtering decisions, based on addressing according to the standardized spanning tree algorithm [2]. This algorithm calls for the automatic discovery of topology changes in the whole network environment.

In a general bridge operation, frames are first received and then conditionally passed to the other LAN depending on a forwarding decision. Effectively, frames are filtered through the bridge. Thus, the frames having destination and source addresses in the same LAN are not forwarded. Although two LANs are connected in the frame level, they do not share each other’s local traffic, but only the cross traffic. The purpose of the bridge is to allow hosts attached to other LANs to communicate as if they were in the same LAN.

In contrast to bridges, routers are network layer store and forward devices that rely on an entire higher level protocol suite, and they are explicitly addressed by the communicating end stations [3]. They are capable of searching alternate routes having lower transmission delays and using the best path between two nodes in the network. Since they can tolerate failures in links and stations, routers are designed to enhance availability through the entire network with the penalty of additional processing overheads introduced by the network layer. Therefore, bridges become more important in a high speed network environment.
Recently, both technologies have begun to converge such that simultaneous operation of MAC level bridging and multiple protocol routing services are provided in a simple device. These devices are called as *brouters* [4]. Consequently, the individual attractive features of both architectures are supported with the brouters.

In this study, the NI developed for an FDDI LAN interconnection is intended to be a generic model. Other high speed interface models can be developed either for ATM or newer networking technologies, as they reach maturity.

C. PROPOSED BRIDGING METHOD

1. Current Standards

Throughout the networking evolution, bridging standardization procedures addressed the interconnection of separate LANs at geographically close points of attachment with local MAC bridges [2]. As LANs became more prevalent both in commercial and military applications, the need to interconnect geographically separated LANs proved inevitable.

An emerging IEEE standard is being developed to satisfy this fundamental necessity [5]. This draft standard specifies how a cluster of remote LANs can be bridged with remote bridges in the MAC level. Moreover, the proposed standard provides all functionalities of local MAC bridges to remote MAC bridges with special additions.

2. Remote Bridging

Remote MAC bridges, possibly not constrained to the same geographic area, interconnect locally situated LANs to the one or more remote MAC bridges by using a non-LAN communication medium as depicted in Figure 2. MAC service support of remote bridges performs the equivalent communication functions of the conventional local bridges across a non-LAN medium. This communication medium,
which may be any type of point-to-point link, is operated and controlled by the group communications entity within the remote MAC bridge.

The group communications entity, which is represented in the virtual port level, is an abstract communication functionality supported by protocols and procedures. As will be stated in the next section, a promising candidate protocol for CDL deployment is the Internet Standard Point-to-Point Protocol (PPP) and its imbedding into this entity is being addressed in further studies [6].
The spanning tree algorithm and its associated procedures are reemphasized in the draft standard for remote MAC bridges in order to preserve the active topology and provide automatic reconfiguration in the entire network.

Remote bridge functions enforced by the spanning tree topology also include forwarding process and learning process associated with filtering database as in the case of local MAC bridges. The forwarding process filters the frames coming from local LAN on the basis of permissible destination addresses contained in the filtering database, while forwarding the frames coming from other LANs to their destinations. The learning process updates the filtering database by observing the source addresses of received frames.

The developed model that will be discussed in the next chapter is evaluated in relation to the proposed remote MAC bridging functionality. However, it is not intended to be the implementation of a full-fledged remote MAC bridge. Since topology is restricted to a pair of FDDI LANs and probable topology changes are not modeled in this generic implementation, bridge functions are fitted into two separate stations existing in the interconnected network model. As a result, subsequent processes for these functions are preferred to be simple forward and filter decisions based on a priori knowledge of the addressing database maintained in these stations.

D. UNIQUENESS OF THE CDL ENVIRONMENT

1. Description

As noted before, CDL is a full-duplex, jam-resistant, point-to-point microwave communication system that provides real-time connectivity and interoperability between multiple collection platforms and surface terminals. In our study, CDL is assumed to be a bundle of unidirectional point-to-point bit pipes associated with certain data rates along the communication medium. The bit pipes carrying
information from surface terminals to the collection platform are grouped together as the command link. The link which performs the same functionality in the opposite direction is called the return link. The return link carries information such as voice, platform status, and data gathered by sensors from the collection platform.

2. System Requirements

The first set of requirements is simply related to NI architecture and its operation. Our model assumes the presence of FDDI LANs on both ends of the CDL link itself. In this thesis, the NI functions required when CDL link is established are addressed. Thus, the setup procedures such as link establishment and authentication are ignored in order to simplify the model simulation sequence. The second set of requirements refers to the typical CDL scenarios. The NI implementation should be generic so that different scenarios can be supported with a single type of NI. Some of the scenarios may require the LAN to be interconnected to multiple LANs permitting multicasting of collected data while others may require relaying of data from one LAN to another. The third set of requirements determines the issues related to data types and quality of service (QoS). QoS, associated with the link, is expected to be provided in the NI using different buffering schemes. Dynamic monitoring of link quality will be deployed as a subsequent subprotocol of PPP in the further developments of our model.

3. Asymmetry between Command and Return Link

The number of bit pipes, and consequently data rate of the return link depend upon the particular CDL configuration used. The information is assumed to be multiplexed, formatted, and transmitted as a composite stream to the surface LAN at various data rates from 10 Mbps to hundreds of Mbps in the full-capacity configuration. Several channel hierarchies are also assumed to be available for each data rate with each channel being an independent data stream.
As previously stated, the command link handles the transmission of user commands from the surface LAN to the collection platform LAN. Contrary to the return link, the command link is assumed to employ a fixed data rate of 200 Kbps. The asymmetry cited in the duplex point-to-point link, enforces separate performance evaluation of the links that will be examined in the Chapter IV.

4. Considerations on Link Utilization

The major issue to be addressed in this unique network architecture is efficient utilization of the communication link. In CDL deployment, the link is time-critical, bandwidth is expensive, and prone to errors due to the probable interference and jamming in the operational environment.

The implementation of an interconnection mechanism as multiple channels for the return link requires proper management of these channels. There are several studies about high speed protocol controllers which investigate and propose high performance [7–9].

One of these approaches is the realization of a distributed multilink system which offers a load balancing mechanism for protocol controllers in order to increase the total throughput of a high speed data link control system. The proposed method for distribution of frames to multiple links evaluates several transmission allocation algorithms. Our model employs two of the most efficient algorithms which are implemented in the NI. First, we modeled the circular allocation algorithm, which calls for the allocation of frames to the NI transmitters in a circular order regardless of the state of individual transmitter buffers [7]. Secondly, we implemented the empty selection algorithm, which requires the selection of an empty area in transmitting-waiting buffers for frame allocation [7]. The decision process, based on determining the next candidate for transmission allocation, is also carried out by NI in order to provide transparency to end stations.
Although load balancing is aimed towards the efficient multiple link utilization, side effects introduced by this procedure must be examined carefully. The slippage among multiple links due to different transmission capacities causes nondeterministic arrival times of data on the receiving end. For this reason, after the frames are received, reordering of them is unavoidable. Thus, the trade-off is increased receive-end buffer size necessary for this process or efficient multiple link utilization. Our model does not address the issues about resequencing of frames on the receiving end. Instead, a simple time division multiplexing procedure is performed for transmission such that consecutive frames coming from the same station are sent through the same transmission channel regardless of the load balancing algorithm in use. Thus, the reordering problem is solved automatically with this simplification.

5. The Effects of Jamming

Another feature intrinsic to CDL is varying bit error rate (BER) affecting links severely due to the nature of the deployment environment. A constant BER can not be assumed in the model during the existence of the link. The exact system performance must be evaluated in the presence of jamming for any typical military application. Therefore, the link efficiency must be investigated under two different types of jamming models which are described in the next chapter.
III. MODELING CDL NETWORK INTERFACE IN OPNET

A. OVERVIEW

As discussed in Chapter II, the necessity to integrate two FDDI LANs to each other in CDL scenario, enforces crucial changes in the modular structure of OPNET model. All of these changes are implemented in three phases:

- Applying the model modifications necessary to implement individual traffic, throughput etc. for separate FDDI LANs,

- Modeling the linking nodes in order to accommodate the generic remote MAC bridge features in relation to CDL, and

- Linking two FDDI LANs in CDL perspective.

This chapter provides the detailed model development in the order above. The explanation of FDDI protocol and model in OPNET will not be presented here.

The whole documentation for simulation development is available in the eleven volume set of manuals provided by MIL 3, Inc. In addition to these manuals, [1] serves as an extensive tutorial, particularly, for FDDI LAN development. The experiences gained through previous studies are readily documented in the aforementioned thesis.

The faithfulness of the developed model will also be discussed in the last section of the chapter.
B. REVISED FDDI LAN MODEL

1. FDDI LAN Model in Brief

OPNET provides a built-in model for FDDI LANs. The modifications achieved in a recent study for model development are focused on validating the FDDI protocol in OPNET [1]. It includes the synchronous and asynchronous transmission characteristics of the model. The individual throughput, mean delay and end-to-end delay features for the synchronous, prioritized asynchronous and total traffic in an FDDI LAN were modeled and examined. FDDI multicasting capability and a rudimentary linking node for the interface development are also introduced in [1].

OPNET’s FDDI LAN model is implemented in a hierarchy. Each LAN is represented as a ring of FDDI stations connected to each other. Figure 3 shows the complete ring representation of an FDDI LAN. The internal structure of a 10-station FDDI LAN ring is depicted in Figure 4.

Each station is represented as an FDDI station model which includes nodes connected to each other. An FDDI station model is illustrated in Figure 5. Each node is defined by process models that are represented as state transition diagrams. Figures 6–8 show this modular model structure. The detailed explanations of the models will not be reviewed here for the reasons mentioned before. A brief introduction to revised FDDI model in OPNET is intended to lead to a better exposition of the model enhancement made in this thesis.

The ultimate source that determines the behavior of a state in a process model, is "C" language codes embedded in that state. Thus, the modifications implemented throughout this thesis refer to code changes as well as the illustration of interconnected FDDI LAN model.
Figure 3: FDDI LAN ring representation.

Figure 4: Ten-station FDDI LAN, fddi_net_10.
Figure 5: FDDI station model, fddi_station.

Figure 6: Source process model, "fddi_gen".
Figure 7: MAC process model, “fddi_mac”.

Figure 8: Sink process model, “fddi_sink”.
2. Modifications on FDDI Station Model

a. Preliminary

The built-in FDDI LAN model in OPNET is designed such that only a single LAN's statistic can be obtained at the end of the model simulation. This restriction forces revision of the FDDI station model, and subsequently, its associated process models.

Some of the packet formats, used by all FDDI stations, are also modified in order to enhance the functionality of these models in relation to the interconnection.

Throughout the description of model development, OPNET objects are highlighted with a typewriter font as a technical convention. While node models are highlighted with a typewriter font, process models and simulation attributes are set off in double quotes with the same font. Similarly, packet formats are highlighted in italics and double quotes within the text.

b. Source Node Modifications

The message traffic, in form of packets is generated in llc_src which employs the “fddi_gen” process model. The information packet generated to be sent to mac is defined as “fddi lle_fr”. This packet format is also same for the packets sent from mac to llc sink. Moreover, the necessary interface between llc src and mac is provided by “fddi_mac_req” interface control information (ICI) packet format. After an information packet is received in mac, “fddi lle_fr” changes its form to “fddi_mac_fr”. Regardless of the functional differences between them, all of these packets are modified in order to be used in a bridge model. Consequently, three lines of code is added in the ARRIVAL state of “fddi_gen” process model to set the new fields of “fddi lle_fr” and “fddi_mac_req” packet formats.
The details about the modifications on packet formats will be stated in Section E of this chapter.

c. Sink Node Modifications

The packets are counted and the statistics are gathered in the llc\_sink, or in other words "fddi\_sink" process model. All statistics related to a LAN are updated through the global variables in this process model. In the case of a second LAN, this global nature must be restricted to individual LANs to provide a realistic representation of the whole network. Thus, all statistical attributes in INIT and STATS are renamed to make them private for the first FDDI LAN. A similar procedure is performed again for a second "fddi\_sink" process model. This process model is named as "fddi2\_sink", and it is embedded into llc\_sink nodes of the second FDDI LAN that will be interconnected. Furthermore, the DISCARD state is modified such that statistics are gathered only for local traffic on the basis of incoming frames' source addresses.

C. BRIDGE MODEL

1. Preliminary

The generic remote MAC bridge that interconnects two FDDI LANs is composed of two separate NIs. Since symmetric modifications are done for both LANs, for simplicity, the NI in the collection platform LAN is referred as CPNI, and the NI in the surface LAN is referred as SPNI in our FDDI-CDL model.

2. The CPNI

a. Station Model

This represents the FDDI station model functioning as an NI in the collection platform LAN. Figure 9 is an illustration of the CPNI employing
Figure 9: Collection Platform LAN Network Interface (CPNI).
simplified 137.088 Mbps mode return link hierarchy in the OPNET user interface window. When compared to the built-in FDDI station model of Figure 7, first major distinction seen is the implementation of llc_sink as a queue module. This alteration is the fundamental step in creation of a bridge link. The other differences from the original model include point-to-point transmitter and receiver nodes.

This particular station examines the destination addresses of frames coming from both LANs. Based on the addressing, frames are either destroyed or forwarded to destination. If the frames are destined for the local LAN, they are simply passed to the local MAC level as in the FDDI protocol. The frames destined for surface LAN are streamed through transmitters' buffers from llc_sink. Since the CPNI also has the full functionality of an ordinary FDDI station, any frame sourced from either LAN may be destined for this station address.

While the frames destined to the CPNI coming from its local LAN are treated within mac, the overhead of MAC access for the frames coming from remote LAN via pr_1, is prevented in Logical Link Control (LLC) level. These frames are evaluated in the llc_src and conditionally passed to the mac for transmission along the local LAN, or forwarded to llc_sink for destruction. Therefore, the frames destined for the CPNI do not need to be repeated. Instead, they are by-passed to llc_sink for higher layer's access.

The point-to-point transmitters, named pt_1 thru pt_4, are the return link transmission sources. Information gathered in collection platform LAN is conveyed to the stations on surface LAN by these transmitter nodes. Transmission allocation procedure is realized according to the load balancing algorithm in use.

Conversely, the point-to-point receiver node, pr_1 is connected to llc_src and it serves as the command link gate to the collection platform LAN.
b. Source Node Modifications

The source node of the CPNI includes “cp_fddi_gen” process model. This process model differs from “fddi_gen” due to the modifications implemented in its ARRIVAL state. In this state, frames coming from the surface LAN are determined first. If there are incoming frames, the CPNI postpones its own frame generation until no more frames are received. Then, the received frames’ destination addresses are checked. Any frame destined for the CPNI is simply forwarded to 11c_sink for destruction. Thus, these frames are not sent to mac like the rest of the incoming ones. The necessary interface along with the information packet is supplied to mac with “fddi_mac_req” ICI packet format. During this ICI packet transfer, “pri” field is set to the highest priority which is assigned to the synchronous transmission. This procedure is essential to keep the bridge model realistic. In FDDI, these priorities relate to LAN bandwidth allocation. Since priorities are only local to each LAN and FDDI frames do not contain an explicit priority field (unlike IEEE 802.5), there is no way to determine to which priority level the frame received from the other LAN belongs. Therefore, once the NI receives these frames, it always sends them as synchronous traffic on its LAN. The “.C” code for “cp_fddi_gen” is provided in Appendix A.

c. MAC Node Modifications

The mac of the CPNI employs “cp_fddi_mac” as its process model (Appendix B). The modifications made to the original model are categorized in four groups as below.

(1) “static” Declarations. Since we use common basic process models in stations with different names, the functions and variables used by the process models are made ‘ “static” to prevent name conflicts.
(2) **INIT State.** The token can be generated by any of the stations in the original OPNET FDDI model. To simplify the simulation sequence, the INIT state code is changed such that only the NI is capable of generating the first token. This takes effect only if the simulation environment file is set as described in Chapter IV.

(3) **FR_REPEAT State.** The frames coming from the physical layer are inspected on the basis of destination addresses in this state. The modifications in FR_REPEAT refer to the implementation of major bridge functions. While the frames addressed to the surface LAN are forwarded to llc_sink, the ones having a local destination address are propagated in the local LAN. Thus, the local traffic is effectively filtered by the NI.

(4) **ENCAP State.** The original source address of a frame coming from llc_src needs to be preserved in this state. Furthermore, a simple check is made for default values of source and destination addresses in "fddi_mac_req" ICI packet format. Thus, "fddi_mac_fr" packets containing the same source and destination addresses are not erroneously composed in this state.

d. **Sink Node Modifications**

Several modifications are required to the llc_sink of the NI as described below. This node uses the "cp_fddi_sink" process model and acts as an interface between the FDDI LAN and CDL. As a result, all three states of this process model are modified. The modified "C" code is in Appendix C.

(1) **INIT State.** All the global statistics' arrays which are used in the analysis tool are defined here. Consequently, the statistics array needed for the traffic monitoring of return link in each priority, is also defined in this state.

(2) **DISCARD State.** Notwithstanding its name, the "cp_fddi_sink" process model's DISCARD state does not discard all the frames.
The source and destination addresses of the frames are inspected here. The frames having the NI address as their destination are destroyed. Moreover, if they are generated by local stations, the related statistics are updated. The frames destined for the remote LAN are not destroyed. Instead, they are counted and enqueued in the subqueues of llc_sink for transmission. The transmitters' status are continuously monitored before and after the transmission with OPNET statistical interrupts described in Section D. The allocation of frames to transmitters using different load balancing algorithms is also performed in this state. The details of this procedure are provided in Section E.

(3) STATS State. This state refers to the documentation of statistics at the end of the simulation. It is this state that must generate the return link related statistics.

3. The SPNI
   a. Station Model

   This model is one of the FDDI stations located in the surface LAN. The SPNI employing the same return link hierarchy mentioned above, is depicted in Figure 10. This station has the same functionality of the NI as in collection platform LAN, but in the reverse direction. As clearly seen, there exists symmetry in the number of transmitters and receivers with respect to its correspondent in collection platform LAN. The node, llc_src accesses mac via pr_1 thru pr_4 that are the downstream gates of surface LAN. The frames destined for the SPNI are distinguished by llc_src and forwarded to the llc_sink directly. This process prevents the additional MAC access creating a significant overhead.

   Although the SPNI serves as a receive end-point for the return link, it is also capable of generating its own frames like other FDDI stations.
Figure 10: Surface Platform LAN Network Interface (SPNI).
The transmitter, pt-1 is command link access node. This node handles the frames coming from llc_sink and delivers them to the collection platform LAN. The forwarding and filtering decisions based on addressing, are implemented in llc_sink as in the CPNI.

Symmetric modifications are carried out in the process models of spni with respect to the CPNI. Since much of the details are provided before, only major distinctions for each node will be stated below.

b. Source Node Modifications

The source node of the SPNI employs “sp_fddi_gen” as the process model. The modifications implemented in this process model are the same as “cp_fddi_gen” with one exception. In the ARRIVAL state, the frames having destination addresses in collection platform LAN are passed to llc_sink. This inspection is based on the permanent database maintained in the NIs as can be seen in the “.C” code supplied in Appendix D.

c. MAC Node Modifications

The process model “sp_fddi_mac” has the same alterations as “cp_fddi_mac” process model’s case in INIT and ENCAP states. The same variables and all functions are also defined as “static”. The difference appears in FR_REPEAT state. Thus, “.C” code provided in Appendix E includes only this modified state. In this state, the frames belonging to the collection platform LAN or addressed to the SPNI are passed to llc_sink, while others are simply propagated. This decision is also made on the basis of a priori knowledge of addresses.

d. Sink Node Modifications

The changes in INIT and STATS states of “sp_fddi_sink” are symmetric to the ones in “cp_fddi_sink” process model. Furthermore, DISCARD
state modifications differ from "cp_fddi_sink" process model's DISCARD state. In this state, the frames destined to the SPNI are destroyed and related statistics are updated for the surface LAN. Since there is no multiple link deployment for the command link, transmission capacity allocation is based on FIFO queue of llc_sink. For this reason, load balancing algorithms are not employed here. The "C" code for "sp_fddi_sink" is provided in Appendix F.

D. CDL MODEL

1. OPNET Model for Point-to-Point Links
   a. Preliminary

   OPNET system models are composed of distributed subsystems which need a communication mechanism among them. There are several methods available in OPNET for the communication between two subsystems, the most prevalent being the packet-based one. In the OPNET environment, a packet is a data structure facilitating information transfer from one subsystem to another. While packet streams, represented as the physical connections, provide transmission between nodes in the same subsystem, the ultimate transfer of information to other subsystems is employed in the form of communication links.

   As mentioned in Chapter II, CDL deployment requires point-to-point link management because of the physical constraints of the environment. Point-to-point links, either simplex or duplex, allow packets to be transmitted between a single pair of nodes. Each link consists of several transmission channels between the source and destination node that it connects. In OPNET, these nodes are referred to as transmitters and receivers.
b. Transmitters

These nodes serve as the exit points of a station for packets forwarded on point-to-point data transmission links. They are composed of multiple channels, each of which is tied to a receiver channel in a remote station via point-to-point link.

Transmitters are built-in FIFO queues with infinite capacity per channel. These queues regulate the transmissions in a channel so that only one packet is transmitted on the link at any time.

The status of any transmitter can be monitored with OPNET statistical interrupts. These interrupts are represented via statistics wires in the OPNET user interface window. Particularly, the "but" statistic, which is fed back to a node from a transmitter plays the most important role for any transmission capacity allocation process.

c. Receivers

As opposed to transmitters, these nodes act as entry points of a station for packets received on point-to-point data transmission links. They employ the reverse functionality of the transmitters. After reception of packets on channels in the remote station, the packets are forwarded through output streams to the attached node of the receiver node for further processing.

d. Transceiver Pipeline Stages

In OPNET, point-to-point links can be configured to model packet transmission in several ways. Each link includes a series of default or user-definable submodels called pipeline stages. The source code for all the default models is provided in <opdir>/stdmod/base directory.
In any pipeline stage of the model, the data related to each packet is used in the various computations related to its transmission. These computations are performed in order to identify the reception time and whether or not a packet is received correctly.

Point-to-point links are based on a four stage pipeline that supports the transfer of packets from a transmitter to a receiver. These stages are described below in brief.

1. **Transmission Delay.** This is the first stage of the transceiver pipeline. In this stage, the amount of time required for the transmission of an entire packet is calculated. This computation is performed independently for each packet transmission on the basis of channel “data rate” and length of the packet. OPNET employs the default transmission delay model, “dpt_txdelay” for this pipeline stage.

2. **Propagation Delay.** After the first stage, packets are passed to the propagation delay pipeline stage. The purpose of this stage is to calculate the amount of time for the packet to reach the receiver in the destination node. The parameter necessary for this computation is the “delay” attribute of the point-to-point link. The default propagation delay model provided by OPNET is called “dpt_propdelay”.

3. **Error Allocation.** Since the packets are prone to errors during transmission, the third stage of the transceiver pipeline allocates errors for transmitted packets. The default model, “dpt_error”, uses the “ber” attribute of the point-to-point link for this process. The algorithm implemented in this model generates a random number of errors with a fixed BER during the whole period of the simulation.

4. **Error Detection and Correction.** This is the final stage of point-to-point transceiver pipeline. The purpose of this stage is to determine
whether or not the received packet can be accepted and forwarded to its destination. The comparison is based on the "ecc" attribute of the receiver node. This attribute represents the probability of bit error that can be tolerated for acceptability. If it is set to zero, default model "dpt_ecc" only accepts error-free packets. Setting a non-zero threshold for this attribute corresponds to error correction procedure for the received packet.

2. Error Modeling over Multiple Links

a. Modifications on Error Allocation Pipeline Stage

As stated earlier, error modeling over point-to-point links is carried out in the error allocation stage of transceiver pipeline. Since the default pipeline stage model uses a constant BER as the simulation progresses, a realistic model must be developed for the CDL link. For this purpose, default error model "dpt_error" is modified, and the new model is renamed as "cdl_pt_error".

In OPNET, the necessary modifications for a pipeline stage model are done in the "op_models" directory. And, after these modifications, the customized model file must be compiled separately. So, "cdl_pt_error" model is compiled with the following command:

```
cc -c -I/op_models/cdlpt.error.ps.c -I<opdir>/sys/include
```

This procedure is necessary to link other OPNET libraries. Otherwise, binding errors result during the generation of simulation file. Appendix G is the file "cdl_pt_error.ps.c", containing the modifications described in this section.

In contrast to the default model, "cdl_pt_error" performs error allocation with a varying BER. Thus, the link attribute "ber" is disregarded in this stage. Instead, new attributes are specified to accomplish a dynamic error allocation
Six different attributes are added for each channel in the extended attributes menu of the point-to-point link. These attributes are depicted in Figure 11, and defined below.

1. “jam_ber”.
   This attribute is the maximum BER on the transmission channel of a point-to-point link during jamming.

2. “ber_bet_jam_len”.
   This attribute corresponds to the maximum BER on the transmission channel of a point-to-point link in the duration between two consecutive jamming pulses.

3. “jam_length”.
   This is the duration of a jamming pulse affecting the transmission channel of a point-to-point link.

4. “interval_bet_jam_len”.
   This is the duration between two consecutive jamming pulses affecting the transmission channel of a point-to-point link.

5. “init_jam_offset”.
   This attribute is the initial offset time on the transmission channel of a point-to-point link. This offset is required for the channel-swept jammer which will be described later.

6. “jammer_type”.
   The last one of the extended attributes specifies the type of the jamming model of which CDL link is exposed to.

b. Jammer Implementations

There are two distinct jamming models implemented in this study. These are pulsed jammer and channel-swept jammer, respectively. Regardless of the jamming type in use, a transmitted packet is first time-stamped in simulation time domain. This time-stamp is used to determine whether or not a packet is subjected to jamming. After this decision, further procedures are carried out for each jammer type independently.
Figure 11: The attributes used for jamming patterns.
(1) **Pulsed Jammer.** In this model, all transmission channels are exposed to separate pulse trains in time. This configuration is achieved by setting up proper values for previously described extended attributes in the simulation environment file. Appendix H is a sample environment file configured for this particular model.

In order to provide a more realistic representation, the durations of jamming pulses and BERs during these periods are also randomized with uniform distribution. Because of this stochastic process, first four of the extended attributes are actually the maximum values that can occur during transmission. Consequently, each transmitted packet is subjected to a different BER according to its presence in simulation time domain. Figure 12 is an illustration of a pulsed jammer.

(2) **Channel-Swept Jammer.** The jammer implemented with this model sweeps each transmission channel consecutively in the simulation time domain. Again, the necessary configuration for this procedure is provided by the extended attributes' values which are specified in the simulation environment file. As opposed to the previous jammer model, the channel-swept jammer does not randomize the durations of jamming pulses to maintain consecutive pulse scheme in order. However, BERs are still randomized with uniform distribution. A proper offset must be specified in the “init_jam_offset” extended attribute of the point-to-point link to provide consecutive pulses as depicted in Figure 13.

**E. LAN INTERCONNECTION**

FDDI-CDL interconnection can be realized using the modifications described so far. Figure 14 shows the ultimate interconnected model. While the top simplex point-to-point link from surface LAN (ring1) to collection platform LAN (ring0) represents the command link, other four links in the reverse direction represent the
Figure 12: Pulsed jammer representation.

Figure 13: Channel-swept jammer representation.
Figure 14: Interconnected network model, fddi.cdl.
simplified return link hierarchy. In this section, specific features of this model of LAN interconnection are stated.

1. Addressing Through The Remote Bridge

The interconnection mechanism is based on the forwarding decision made by the inspection of station addresses. The stations that reside on LANs have unique addresses. Our model employs 10-station FDDI LANs on both sides of the CDL link. In order to simplify MAC bridge functions, these addresses are permanently installed in the filtering databases of the CPNI and the SPNI. These stations, acting as NIs, have the maximum station number of each LAN. Thus, filtering decision is based on the comparison of these NI addresses. If the number of stations or their addresses need to be changed, those NIs' filtering databases must be updated according to the new address assignments. Besides, the DISCARD state of process models of llc_sink nodes of the station models in both LANs must also be modified appropriately to obtain individual LAN statistics.

The address information is sent to the nodes with several packet formats in OPNET modeling environment. The packet format, "fddi_mac_fr", allocates 16 bits to represent station addresses. In current bridging standards [2, 5], the addresses are represented as 48 bits within a frame. Thus, the mentioned packet format is modified to include 48-bit source and destination addresses. Furthermore, the necessity to include source and destination addresses in "fddi_llc_fr" packet format results in the same modification. This packet format is used in llc_src nodes of NIs to prevent unnecessary MAC access as mentioned in Section C.

For interface control purposes, "src_addr" field is also added in the original "fddi_mac_req" ICI packet format. This address is used by the mac to keep the original source address of a frame unmodified during the data transfer between
LANs. Figures 15–17 show the resulting packet formats in the parameter editor of OPNET user interface window.

2. Load Balancing over Multiple Links

As stated before, the return link is composed of multiple channels. In our model, simplified return link hierarchy is represented as four simplex point-to-point links, each having one transmission channel. Data rates for each channel on both links are specified as in Figures 18 and 19.

This multiple channel hierarchy requires an algorithm for transmission capacity allocation. The selected algorithm eventually leads to load balancing over multiple links. In the OPNET model implemented, this procedure is carried out by 11c_sink of the CPNI. The node, 11c_sink, is in the form of a queue module consisting of four subqueues. Thus, the transmission capacity allocation process is basically allocating frames to these subqueues that are attached to the individual transmitters. In other words, the subqueues act as buffers. Instead of allocating frames to the transmitter buffers directly, this method is chosen to monitor the effects of load balancing algorithm in use. As stated in Section D, point-to-point transmitter buffers are built-in queues with infinite capacity. This approach would not be realistic for a bridge model. In a real bridge, insufficient transmission queue sizes can cause dropping of frames. In the CDL NI model, the frame currently being transmitted resides in the transmitter buffer until its transmission is completed. Other frames waiting for transmission reside in the finite subqueues of 11c_sink. Then, these frames are forwarded to the related transmitter when the "busy" signal goes low, meaning that the transmitter channel is idle. This is accomplished by OPNET statistical interrupts, and the allocation algorithm calls for the continuous monitoring of this signal.
Figure 15: Modified packet format, "fddi_llc_fr".
Figure 16: Modified packet format, "fddi_mac_fr".
Figure 17: Modified ICI packet format, "fddi_mac_req".
Figure 18: Return link data rates and new attributes for llc_sink.
Figure 19: Command link data rate.
The algorithm can be specified for the model from the simulation environment file. So, an extended attribute, "load balancing algorithm", is defined for the CPNI as in Figure 18. Regardless of the load balancing algorithm in use, a frame is allocated to a buffer, each time "cp_fddi_sink" process model is executed. Then, the order of subqueues is determined with an incrementing index according to the allocation algorithm employed. Every subqueue has an associated source address field, so that previously allocated frame's source address is maintained in that subqueue. Two algorithms are implemented as described below.

a. Circular Allocation Algorithm

Before allocating frames in a circular order to llc_sink subqueues, the source address of an incoming frame is inspected first. If that frame is sent from a station address which has been previously allocated to that subqueue, the new frame is also buffered in the same subqueue. Thus, consecutive frames from the same source are sent over the same transmission channel. Otherwise, allocation is still done circularly ignoring the state of individual subqueues. Because of the different channel data rates, some buffers can fill up and even lead to frame loss, unless the sizes of these buffers are selected adequately.

b. Empty Selection Algorithm

This algorithm refers to the allocation of frames to sub-queues having the maximum empty slots. The frequency of the occurrence of empty slots is directly proportional to the data rate of transmission channel. Thus, the buffer sizes needed for this algorithm are likely to be less than the ones needed for the previous algorithm. Allocation of consecutive frames sent from the same station is still implemented in the same manner as the circular allocation algorithm.
F. FAITHFULNESS OF THE MODEL

The CDL NI model is developed with the extensive features provided by OP-NET. Modifications to the original models are carried out step-by-step, so that the original modularity is not sacrificed with new enhancements.

The OPNET FDDI protocol model is studied and validated in [1]. Further model improvements of this thesis are realized on the basis of the results of this previous study. During the CDL model development, specific system requirements were primarily considered and adapted to OPNET modeling environment.

In a real CDL deployment, depending on the deployed channel hierarchy, an aggregate bit stream at the high frequency is obtained using multiplexing of the independent transmission channels. Then, at the receiving end, demultiplexing is carried out. Our CDL system implementation does not model the link between the output of the multiplexer and the input of the demultiplexer. Instead, the multiplexer input and the demultiplexer output are modeled and monitored. In other words, the link is not modeled at the aggregate bit stream level, but at the individual channel level. This permits us to view the link as a set of independent channels rather than a single stream of bits without loosing the realistic nature of the link.

The corruption of packets in a real jamming environment is of an irregular nature. The bits in a data stream may be inverted, or random sequences of bit patterns may be added to the information packet. In the OPNET modeling environment, error allocation is done by inverting the bits present within a packet. So, the jammer modeling is based on this constraint. In other words we can not model loss of the CDL framing synchronization in OPNET. However, this study is targetted at the data link layer and above. Therefore, this limitation does not present a problem. Despite this limitation, since our jamming models use a varying
BER within the random durations in the simulation time domain, the desired error allocation feature is still provided. In the model, the error allocation process within a link is such that a random number of errors are introduced in a randomly selected packet in each channel. This is essentially what real jamming of the aggregate CDL bitstream will result in. Two types of jamming models are investigated in order to evaluate system performance in a wider variety of scenarios.

Finally, interconnection is realized in terms of an upcoming bridging standard [5], so that dependability of the model is ensured.
IV. MODEL TESTING

A. OVERVIEW

This chapter provides several test results in relation to the enhanced capabilities of our model. These tests include the monitoring of traffic in the whole network for individual LANs and the evaluation of transmission capacity allocation algorithms for the return link. The effects of different jamming patterns on the return link are also demonstrated. Similarly, the command link performance is studied in brief.

B. PERFORMANCE METRICS

During the simulation tests, a moderate traffic load is offered for both LANs that are interconnected. The behavior of the OPNET’s FDDI model with varying traffic loads was studied previously [1]. Therefore, the tests are primarily run and evaluated for NI functional characteristics and the CDL link performance during jamming.

The transmission capacity allocation algorithms related to the load balancing over multiple links are monitored in the llc_sink node of the CPNI. The buffers within this node are observed in order to evaluate these algorithms’ efficiency with the “pksize” and “delay” probes in the OPNET analysis editor. These probes are simply used to determine the advantages and drawbacks of the load balancing implementations. The “pksize” attribute’s name is misleading in OPNET probe editor and it refers to the number of packets residing in a subqueue. Thus, while “pksize” probe is the means to determine the required buffer sizes, the transmission waiting periods of the packets in the buffers are monitored with the “delay” probe.
The average BER of the return link is monitored on the SPNI receivers with "avg_ber" probes for each transmission channel. In this study, the BER of the command link during jamming is not investigated. The utilization of multiple links under different jamming patterns is also examined on the transmit and receive-ends of the link. Results of the different simulation experiments follow in the next section.

C. TRAFFIC MONITORING

1. Overview

The modifications made within this thesis to the OPNET FDDI model provide the individual monitoring of two interconnected LANs. For CDL deployment, the collection platform LAN is likely to direct most of the traffic to the return link, although there may be still an amount of local traffic under consideration. On the other side, the surface LAN directs some of its traffic to the command link while introducing a significant amount of traffic locally. For the reasons specified above, each LAN must be monitored independently to provide a realistic representation of the entire network.

2. Setup

The simulation was set up to display local traffic within one LAN and the traffic directed to the remote LAN. Both LANs consist of ten FDDI stations, generating packets at a constant rate. The stations generated 20000-bit packets at an arrival rate of 250 packets per second, so that 50 Mbps of the traffic load is expected for a LAN. The stations f9 and f19 are specified as the CPNI and the SPNI, respectively. While the surface LAN can send packets to all stations in the interconnected network model, the collection platform LAN is designated to direct its total traffic only to the return link. Both LANs are capable of generating 90 percent asynchronous traffic at different priorities. Target Token Rotation Time
(TTRT) was set to 4ms, and necessary synchronous bandwidth was determined as 0.08955675 as the required OPNET parameter. This value is a function of TTRT; it is computed by following the necessary steps as described in [1] and amounts to 0.358227 ms/station.

As mentioned before, the initial tokens are launched by the NIs. Thus, in the simulation environment file, the "spawn station" attribute is disabled by setting it to 20. This number is chosen arbitrarily, but it must be greater than the maximum station number that exists in the entire network model. This is just an OPNET hack to keep the "fddi_mac" process model of original "fddi_station" unchanged.

Furthermore, the "accelerate_token" attribute is also disabled by setting it to zero. When this flag is enabled, the token is blocked in the station which has no packets to transmit. This procedure leads a faster simulation, and it is not part of the real FDDI protocol. The interconnected model is not allowed to use this shared flag, because of the tokens belonging to different LANs.

3. Results

Figure 20 shows the local throughput observed in the surface LAN. The random generation of destination addresses with uniform distribution led a considerable amount of traffic to the local stations. Figure 21 is the throughput directed to the command link from the surface LAN. In both cases, 90 percent of the traffic belongs to the asynchronous transmission as expected.

Figures 22 and 23 show the traffic contributed to the whole network by the collection platform LAN. While there is no local traffic appearing within this LAN, all traffic is directed to the return link with the specified proportion of asynchronous transmission.
Figure 20: Local throughput of surface LAN.

Figure 21: Traffic directed to the command link.
Figure 22: Local throughput of collection platform LAN.

Figure 23: Throughput directed to the return link.
D. RETURN LINK PERFORMANCE

1. Overview

The station acting as the CPNI in the collection platform LAN is capable of buffering packets in its llc_sink node. The buffering is realized in terms of a transmission capacity allocation algorithm for the efficient utilization of the return link. The tests run in this section are related to the evaluation of these algorithms and the link throughput under jamming.

Two tests are described here. The first test is monitoring of packets accumulated in the buffers with two distinct allocation algorithms. The second test is observing the multiple link throughput at both ends of the CDL link.

2. First Test

a. Setup

This test is based on the evaluation of different types of load balancing over the multiple links. Both FDDI LANs are configured with the same parameters as it is done for the traffic monitoring tests. In the simulation environment file, the “load balancing algorithm” attribute is also set appropriately for two different runs. This attribute is set to “0” for the circular allocation algorithm as it is set to “1” for the empty selection. Besides, the llc_sink subqueue capacities are set to hold a maximum of 200 packets. This was intended to observe the buffers’ saturation and overflow as the traffic is directed to the return link at a constant rate.

b. Results

Figure 24 shows the accumulation of packets in the buffers when the circular allocation algorithm is in use. As seen, the first transmission channel having the lowest data rate causes its buffer to fill up quickly. Consequently, the
Figure 24: Accumulation of packets on the CPNI buffers with circular allocation. Continuous overflow of this buffer is unavoidable with the specified buffer size as depicted in Figure 25.

Conversely, the empty selection algorithm results in equal utilization of buffers for the same traffic load without introducing any overflow as shown in Figure 26.

The queuing delays of the packets observed for each buffer using both algorithms are shown in Figures 27 and 28. As compared to the empty selection algorithm, the circular load balancing provides a faster transmission rate for the channels having higher data rates and introduce a longer queuing delay for
Figure 25: Buffer overflows in the CPNI with circular allocation.

Figure 26: Accumulation of packets in the CPNI buffers with empty selection.
Figure 27: Queing delay of the CPNI buffers with circular allocation.

Figure 28: Queing delay of the CPNI buffers with empty selection.
the slower channel. Primarily, the insufficient buffer sizes are the major drawback in the circular allocation algorithm, even though multiple links are still efficiently utilized.

As opposed to the circular allocation, empty selection algorithm does not require greater buffer sizes. It also offers an efficient utilization scheme with the penalty of more queuing delay for the channels with higher transmission capacities.

3. Second Test

a. Setup

This test is intended to verify the effects of jamming models on the return link. The same simulation configuration is used in this test as before. Moreover, as defined in Chapter III, the jamming related attributes are set up properly to model two different jammers. The environment file used for the pulsed jammer simulation is provided in Appendix H. For the channel-swept jammer simulation, the same type of jamming attributes are set to identical values except “init_jam_offset”. This attribute is doubled for each channel so that consecutive pulses can be created with proportional offsets. The jamming related attributes of the environment file for the channel-swept jammer simulation is in Appendix I.

The propagation delay of the CDL model is also specified as a typical value of 60 ms in the “delay” attribute of each link.

b. Results

Figure 29 shows the throughput of each transmitter channel in the CPNI end of the return link. As expected, identical number of bits are injected into the return link for the channels having the same data rates. The packets awaiting transmission in the buffers do not contribute to this statistic.
Figure 29: Throughput at the transmit-end of the return link.

The average BERs introduced by the pulsed jammer and the channel-swept jammer are depicted in Figures 30 and 31, respectively. As monitored in the receivers of the SPNI, all of the links are equally degraded in the channel-swept jammer's case due to the consecutive pulses. In the presence of pulsed jammer, the links exposed to the longer duration of jamming pulses are affected more severely.

Consequently, since the packets are corrupted due to jamming during the transmission, the throughput attained in the receive-end is not the same as the transmit-end of the return link. The effects of two different jamming models on the received throughput can be seen in Figures 32 and 33.
Figure 30:  Average BER of the return link caused by pulsed jammer.

Figure 31:  Average BER of the return link caused by channel-swept jammer.
Figure 32: Effect of pulsed jammer on the return link throughput.

Figure 33: Effect of channel-swept jammer on the return link throughput.
E. COMMAND LINK PERFORMANCE

Although the command link is modeled with respect to the CDL system requirements, its performance is not studied in as much detail as the return link in our simulations. As described before, the command link employs a relatively low fixed data rate in its channel hierarchy compared to the return link. Since multiple links are not modeled for this particular link, no algorithm for transmission allocation is developed in 1lc_sink node of the SPNI. As a result, the packets waiting for transmission to the collection platform LAN are accumulated in the single buffer of the SPNI.

Figure 34 shows the number of packets buffered over time. The accumulation is faster due to the lower transmission capacity of the command link. This is also emphasized with the buffer queuing delay as depicted in Figure 35. The asymmetric data rates existing in the CDL link require a greater buffer size for the SPNI. This buffer size must be evaluated with respect to the amount of traffic injected into the CDL by the surface LAN.

Similarly, the information transferred over the command link is monitored in both NIs of the interconnected model. Figure 36 is a simple illustration of the link throughput in transmit and receive-ends when the command link is not exposed to any jamming.
Figure 34: Accumulation of packets in the SPNI buffer.

Figure 35: Queing delay of the SPNI.
Figure 36: Throughput at the transmit and receive-ends of the command link.
V. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

In this thesis, we have presented a performance analysis of a network interface device that interconnects two remote FDDI LANs using the CDL. The efficiency of the CDL model is evaluated with two different jamming patterns that faithfully represent the real environmental conditions. The distribution of load over the multiple links is also investigated using different transmission capacity allocation algorithms so that the overall link utilization can be maximized. The need for such algorithms is based on the near-capacity return link traffic load that is expected in the interconnected network. It is seen that the circular allocation algorithm is able to provide a faster transmission for the channels having higher data rates with the penalty of increased buffer size for the slower channels. When the empty selection algorithm is in use, identical buffer sizes can be selected for any transmission channel, if the drawback of longer queuing delays for faster channels are acceptable.

This thesis lays the foundation for simulating the multilink point-to-point protocol over the CDL in the further development of the network interface features for the CDL project.

B. RECOMMENDATIONS

Further developments of our model will include:

1. Simulation of a link monitoring protocol over the return and command link. This capability will facilitate feedback to the LAN applications
(end-systems) about the link status and permit end-to-end performance analysis.

2. Incorporation of a forwarding/filtering database in the NI.

3. Incorporation of a traffic generation in the LAN end-systems that is closer to the characteristics of real traffic patterns.

4. Accurate estimation of NI buffer capacity for individual channels with various effective BERs over the link.

Since the tool-specific token acceleration feature is disabled during our simulations, a new mechanism need to be employed for a faster simulation in a larger network model.
APPENDIX A
CPNI SOURCE "C" CODE
"cp_fddi_gen.pr.c"

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

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

/* OPNET system definitions */
#include <opnet.h>
#include "cp_fddi_gen.pr.h"
FSM_EXT_DECLS

/* Header block */
#define MAC_LAYER_OUT_STREAM 0
#define LLC_SINK_OUT_STREAM 1 /*18APR94*/

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

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

/* State variable definitions */
typedef struct
{
    FFSM_SYS_STATE
    Distribution* sv_inter_dist_ptr;
    Distribution* sv_len_dist_ptr;
    Distribution* sv_dest_dist_ptr;
    Distribution* sv_pkt_priority_ptr;
    Objid sv_mac_objid;
    Objid sv_my_id;
    int sv_low_dest_addr;
    int sv_high_dest_addr;
}
int sv_station_addr;
int sv_src_addr;
int sv_low_pkt_priority;
int sv_high_pkt_priority;
double sv_arrival_rate;
double sv_mean_pk_len;
double sv_async_mix;
Icie sv_mac_iciptr;
Icie sv_mac_iciptr1;
Icie sv_llc_iciptr;
Packets sv_pkptr1;

Packet* cp_fddi_gen_state;

#define pr_state_ptr (cp_fddi_gen_state)_sim1_Mod_State_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 mac_objid pr_state_ptr->sv_mac_objid
#define my_id pr_state_ptr->sv_my_id
#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 src_addr pr_state_ptr->sv_src_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_pk_len pr_state_ptr->sv_async_mix
#define mac_iciptr pr_state_ptr->sv_mac_iciptr
#define mac_iciptr1 pr_state_ptr->sv_mac_iciptr1
#define llc_iciptr pr_state_ptr->sv_llc_iciptr
#define pkptr1 pr_state_ptr->sv_pkptr1

/* Process model interrupt handling procedure */

void cp_fddi_gen ()
{
  Packet* pkptr;
  int pklen;
  int dest_addr;
  int i, restricted;
  int pkt_prio;

  FSM_ENTER (cp_fddi_gen)
FSN_BLOCK_SWITCH
{
    /------------------------------------------------------------------------/
    /** state (INIT) enter executives */
    FSN_STATE_ENTER_UNFORCED (0, state0_enter_exec, "INIT")
    {
        /* determine id of own processor to use in finding attrs */
        my_id = op_id_self();

        /* determine address range for uniform destination assignment */
        op_ima_obj_attr_get (my_id, "low dest address", &low_dest_addr);
        op_ima_obj_attr_get (my_id, "high dest address", &high_dest_addr);

        /* determine object id of connected 'mac' layer process */
        mac_objid = op_topo_assoc (my_id, OPC_TOPO_ASSOC_OUT,
                                   OPC_OBJTYPE_MODULE,
                                   MAC_LAYER_OUT_STREAM);

        /* which is also the address of this station */
        op_ima_obj_attr_get (mac_objid, "station_address", &station_addr);

        /* set up a distribution for generation of addresses */
        dest_dist_ptr = op_dist_load ("uniform_int", low_dest_addr,
                                      high_dest_addr);

        /* added 26DEC93 */
        /* determine priority range for uniform traffic generation */
        op_ima_obj_attr_get (my_id, "high pkt priority", &high_pkt_priority);
        op_ima_obj_attr_get (my_id, "low pkt priority", &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 */

        /* 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 */

        /* 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 */

        /* 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 */

        /* 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 */

        /* 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 */

        /* 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 */

        /* 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 */

        /* 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 */

        /* 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 */

        /* 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 */
        "}
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 */
    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 (i, cp_fddi_gen)

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

/** state (INIT) transition processing **/
FSM_TRANSIT_FORCE (i, state1_enter_exec, )
/-------------------
/** state (ARRIVAL) enter executives **/
FSM_STATE_ENTER_UNFORCED (i, state1_enter_exec, "ARRIVAL")
{
    /* This station should receive frames from the other lan as long as */
    /* there are frames in the input streams addressed to this lan */
    /* check if the interrupt type is stream interrupt */
    if(op_intrpt_type() == OPCINTRPT_STRM)
    {
/* if it is, get the packet in the input stream causing interrupt */
pkptrl = op_pk_get(op_intrpt_strm());

/* get the destination address of the frame */
/* 16APR94 */
op_pk_nfd_get(pkptrl, "dest_addr", &dest_addr);
/* check if this frame destined for the local bridge station */
if(dest_addr == station_addr)
    /* if it is, send the packet to llc_sink directly */
    /* in order to prevent overhead of mac access */
    op_pk_send(pkptrl, LLC_SINK_OUT_STREAM); /* 19APR94 */
else
    /* this packet is to send to mac */
    {
        /* determine the source address of the frame */
        op_pk_nfd_get(pkptrl, "src_addr", &src_addr);
        /* set up an ICI structure to communicate parameters to */
        /* MAC layer process */
        mac_iciptrl = op_ici_create("fddi_mac_req");
        /* place the original source address into the ICI */
        /* 16APR94 */
        /* "fddi_mac_req" is modified so that it contains the original */
        /* source address from the remote lan */
        op_ici_attr_set(mac_iciptrl, "src_addr", src_addr);
        /* place the destination address into the ICI */
        /* 12APR94 */
        op_ici_attr_set(mac_iciptrl, "dest_addr", dest_addr);
        /* assign the service class and requested token class */
        /* At this moment the frames coming from the remote lan */
        /* are assumed to have */
        /* the same priority as synchronous frames in order not to */
        /* packets on the bridge station mac and instead to deliver */
        /* their destinations */
        /* as soon as possible */
        op_pk_nfd_set(pkptrl, "pri", 8);
        op_ici_attr_set(mac_iciptrl, "svc_class", FDDI_SVC_SYNC);
        op_ici_attr_set(mac_iciptrl, "pri", 8);
        op_ici_attr_set(mac_iciptrl, "tk_class", FDDI_TK_NONRESTRICTED);
        /* send the packet coupled with the ICI */
        op_ici_install(mac_iciptrl);
        op_pk_send(pkptrl, MAC_LAYER_OUT_STREAM);
    }
else
    /* otherwise, generate the frame : 12APR94 */
    { /* determine the length of the packet to be generated */
pklen = op_dist_outcome (len_dist_ptr);

/* determine the destination */
/* dont 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;

/* 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) */
op_ici_attr_set (mac_iciptr, "dest_addr", dest_addr);

/* place the source address into the ICI */
/* (17APR94 */
op_ici_attr_set (mac_iciptr, "src_addr", station_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. */
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);

/* Having determined priority, assign it; 26DEC93 */
/* op_ici_attr_set (mac_iciptr, "pri", pkt_prio); */
235 /* send the packet coupled with the ICI */
236 op_ici_install (mac_ici.ptr);
237 /* check if destination address is in the remote lan */
238 if(dest_addr > 9)
239    /* if it is, this packet is to send llc_sink directly */
240    op_pk_send (pkptr, LLC_SINK_OUT_STREAM);
241 /*18APR94*/
242 else
243    /* if not, the packet is destined for local lan, so send to mac */
244    op_pk_send (pkptr, MAC_LAYER_OUT_STREAM);
245 /* schedule the next arrival */
246   fddi_gen_schedule ();
247 }
248 }
249
250 /* blocking after enter executives of unforced state. */
251 FSM_EXIT (3, cp_fddi_gen)
252
253 /* state (ARRIVAL) exit executives */
254 FSM_STATE_EXIT_UNFORCED (1, state1_exit_exec, "ARRIVAL")
255 {
256 }
257
258 /* state (ARRIVAL) transition processing */
259 FSM_TRANSIT_FORCE (1, state1_enter_exec, ;)
260 /*--------------------------------------------------------*/
261
262 }
263
264 FSM_EXIT (0, cp_fddi_gen)
265 }
266
267 void
268 cp_fddi_gen_svar (prs.ptr, var_name, var_p.ptr)
269   cp_fddi_gen_state *prs.ptr;
270   char *var_name, **var_p.ptr;
271 {
272    FIN (cp_fddi_gen_svar (prs.ptr))
273
268  *var_p_ptr = VOS_WIL;
269  if (Vos_String_Equal ("inter_dist_ptr", var_name))
270      *var_p_ptr = (char *) (*prx_ptr->sv_inter_dist_ptr);
271  if (Vos_String_Equal ("len_dist_ptr", var_name))
272      *var_p_ptr = (char *) (*prx_ptr->sv_len_dist_ptr);
273  if (Vos_String_Equal ("dest_dist_ptr", var_name))
274      *var_p_ptr = (char *) (*prx_ptr->sv_dest_dist_ptr);
275  if (Vos_String_Equal ("pkt_priority_ptr", var_name))
276      *var_p_ptr = (char *) (*prx_ptr->sv_pkt_priority_ptr);
277  if (Vos_String_Equal ("mac_objid", var_name))
278      *var_p_ptr = (char *) (*prx_ptr->sv_mac_objid);
279  if (Vos_String_Equal ("my_id", var_name))
280      *var_p_ptr = (char *) (*prx_ptr->sv_my_id);
281  if (Vos_String_Equal ("low_dest_addr", var_name))
282      *var_p_ptr = (char *) (*prx_ptr->sv_low_dest_addr);
283  if (Vos_String_Equal ("high_dest_addr", var_name))
284      *var_p_ptr = (char *) (*prx_ptr->sv_high_dest_addr);
285  if (Vos_String_Equal ("station_addr", var_name))
286      *var_p_ptr = (char *) (*prx_ptr->sv_station_addr);
287  if (Vos_String_Equal ("src_addr", var_name))
288      *var_p_ptr = (char *) (*prx_ptr->sv_src_addr);
289  if (Vos_String_Equal ("low_pkt_priority", var_name))
290      *var_p_ptr = (char *) (*prx_ptr->sv_low_pkt_priority);
291  if (Vos_String_Equal ("high_pkt_priority", var_name))
292      *var_p_ptr = (char *) (*prx_ptr->sv_high_pkt_priority);
293  if (Vos_String_Equal ("arrival_rate", var_name))
294      *var_p_ptr = (char *) (*prx_ptr->sv_arrival_rate);
295  if (Vos_String_Equal ("mean_pk_len", var_name))
296      *var_p_ptr = (char *) (*prx_ptr->sv_mean_pk_len);
297  if (Vos_String_Equal ("async_mix", var_name))
298      *var_p_ptr = (char *) (*prx_ptr->sv_async_mix);
299  if (Vos_String_Equal ("mac_iciptr", var_name))
300      *var_p_ptr = (char *) (*prx_ptr->sv_mac_ICIPTR);  
301  if (Vos_String_Equal ("mac_icptr", var_name))
302      *var_p_ptr = (char *) (*prx_ptr->sv_mac_ICIPTR1);
303  if (Vos_String_Equal ("llc_ici_ptr", var_name))
304      *var_p_ptr = (char *) (*prx_ptr->sv_llc_ici_ptr);
305  if (Vos_String_Equal ("pkptr", var_name))
306      *var_p_ptr = (char *) (*prx_ptr->sv_pkptr);
307  FOUT;
308  }

309  void
310  cp_fddi_gen_diag()
311  {
312    Packet *pkptr;

72
313 int pklen;
314 int dest_addr;
315 int i, restricted;
316 int pkt_prio;
317 FIN (cp_fddi_gen_diag ()
318 FOUT;
319 }

320 void cp_fddi_gen_terminate ()
321 {
322 Packet *pkptr;
323 int pklen;
324 int dest_addr;
325 int i, restricted;
326 int pkt_prio;
327 FIN (cp_fddi_gen_terminate ())
328 FOUT;
329 }

331 Compcode
332 cp_fddi_gen_init (pr_state_pptr)
333 cp_fddi_gen_state = **pr_state_pptr;
334 {
335 static Vos_T_Cm_Obtype obtype = OPC_NIL;
336 FIN (cp_fddi_gen_init (pr_state_pptr))
337 if (obtype == OPC_NIL)
338 {
339   if (Vos_Catmem_Register ("proc state vars (cp_fddi_gen)",
340                           sizeof (cp_fddi_gen_state), Vos_Hop, &obtype) == VOSC_FAILURE)
341     FRET (OPC_COMPCODE_FAILURE)
342 }
343 if (**pr_state_pptr = (cp_fddi_gen_state*) Vos_Catmem_Alloc (obtype, 1)) ==
344   OPC_NIL
345   FRET (OPC_COMPCODE_FAILURE)
346 else
347 {

73
348 (*pr_state_pptr)->current_block = 0;
349  FRET (OPC_COMP_CODE_SUCCESS)
350 }
351 }

352 /* static added 2DEC93, on advice from MIL3 */
353 static
354 fddi_gen_schedule ()
355 {
356   double inter_time;
357   /* obtain an interarrival period according to the */
358   /* prescribed distribution */
359   inter_time = op_dist_outcome (inter_dist_ptr);
360   /* schedule the arrival of next generated packet */
361   op_intrpt_schedule_self (op_sim_time () + inter_time, 0);
362 }

74
APPENDIX B
CPNI MAC "C" CODE
"cp_fddi_mac.pr.c"

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

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

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

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 {
14     int enabled;
15     double start_time;
16     double accum;
17     double target_acc;
18 } FddiT_Timer;
19 /* Declare certain primitives dealing with timer.s */
20 double fddi_timer_remaining ();
21 FddiT_Timer* fddi_timer_create ();
22 double fddi_timer_value ();
23 /* Scratch strings for trace statements */
24 char str0 [512], str1 [512];
25 /* define constants particular to this implementation */
26 #define FDDI_MAX_STATIONS 512
27 /* define possible values for the frame control field */
28 #define FDDI_FC_FRAME 0

75
29  #define FDDI_FC_TOKEN 1

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

33  /* define input stream indices */
34  #define FDDI_LLCC_STMN_IN 1
35  #define FDDI_PHY_STMN_IN 0

36  /* define output stream indices */
37  #define FDDI_LLCC_STMN_OUT 1
38  #define FDDI_PHY_STMN_OUT 0

39  /* define token classes */
40  #define FDDI_TK_NONRESTRICTED 0
41  #define FDDI_TK_RESTRICTED 1

42  /* Ring Constants */
43  #define FDDI_TZRATE 1.0e+08
44  #define FDDI_SA_SCAN_TIME 26.0e-08

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

47  /* Codes used to differentiate remote interrupts */
48  #define FDDIC_TRT_EXPIRE 0
49  #define FDDIC_TRK_INJECT 1

50  /* Define symbolic expressions used on transition */
51  /* conditions and in executive statements. */
52  #define TRT_EXPIRE \ 
53  (op.intrpt.type () == OPC_INTRPT_REMOTE && op.intrpt.code () == FDDIC_TRT_EXPIRE)

54  #define TK_RECEIVED \ 
55  phy.arrival && frame.control == FDDI_FC_TOKEN

57  #define RC_FRAME \ 
58  phy.arrival && frame.control == FDDI_FC_FRAME

60  #define FRAME_ARRIVAL \ 
61  (op.intrpt.type () == OPC_INTRPT_STMN && \ 
62  op.intrpt.strm () == FDDI_LLCC_STMN_IN

63  #define STRIP my_address == src_addr

64  /* 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. */
static 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;
static 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_block_base_station [FDDI_MAX_RING_ID];
#define Fddi_Tk_Block_Base_Station (fddi_tk_block_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])
105 /* Below is part of the OPBUG 2081 patch; FB ended here, before. -Nix */

106 /* Event handles for the TRT are maintained at a global level to */
107 /* allow token acceleration mechanism to adjust these as necessary */
108 /* when blocking and reinjecting the token. TRT_handle simply */
109 /* represents the TRT for the local MAC */
110 #define fddi_trt_handle [FDDI_MAX_RING_ID][FDDI_MAX_STATIONS];
111 #define Fddi_Trtr_Handle (fddi_trt_handle [ring_id])
112 /*define TRT_handle Fddi_Trtr_Handle [my_address]

113 /* Similarly, the TRT data structure is maintained on a global level. */
114 FddiTimer* fddi_trt [FDDI_MAX_RING_ID] [FDDI_MAX_STATIONS];
115 #define Fddi_Trtr (fddi_trt [ring_id])
116 /*define TRT Fddi_Trtr [my_address]

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

121 /* the 'Late Ct' flag is declared on a global level so that it can be */
122 /* set at the time where the token is injected back into the ring. */
123 int fddi_late_ct [FDDI_MAX_RING_ID] [FDDI_MAX_STATIONS];
124 #define Fddi_Late.Ct (fddi_late_ct [ring_id])
125 #define Late.Ct Fddi_Late.Ct [my_address]

126 /* Convenient macro for setting TRT for a given station and absolute time. */
127 #define TRT_SET(station_id, abs_time)
128 fddi_timer_set (Fddi_Trtr [station_id], abs_time - op_sim_time());
129 Fddi_Trtr_Handle [station_id] = op_intrpt_schedule_remote (abs_time, 
130 FDDIC_TRT_EXPIRE, Fddi_Address_Table [station_id]);

131 /* State variable definitions */
132 typedef struct
133 {
134 int FSM_SYS_STATE
135 int Fddi_Timer*
136 double sv_ring_id;
137 double sv_T_Rq;
138 double sv_T_Pri [8];
139 ObjId sv_my_objid;
140 int sv_spawn_token;
141 int sv_my_address;
142 int sv_orig_src_addr;

78
141 Packet* sv_tk_pkptr;
142 double sv_sync_bandwidth;
143 double sv_sync_pc;
144 int sv_restricted;
145 int sv_res_peer;
146 int sv_tk_registered;
147 Ici* sv_to_llc_ici_ptr;
148 int sv_tk_trace_on;
149 } cp_fddi_mac_state;

150 #define pr_state_ptr ((cp_fddi_mac_state*) SimI_Mod_State_Ptr)
151 #define ring_id pr_state_ptr->sv_ring_id
152 #define T_Req pr_state_ptr->sv_T_Req
153 #define T_Pri pr_state_ptr->sv_T_Pri
154 #define my_objid pr_state_ptr->sv_my_objid
155 #define spawn_token pr_state_ptr->sv_spawn_token
156 #define my_address pr_state_ptr->sv_my_address
157 #define orig_src_addr pr_state_ptr->sv_orig_src_addr
158 #define tk_pkptr pr_state_ptr->sv_tk_pkptr
159 #define sync_bandwidth pr_state_ptr->sv_sync_bandwidth
160 #define sync_pc pr_state_ptr->sv_sync_pc
161 #define restricted pr_state_ptr->sv_restricted
162 #define res_peer pr_state_ptr->sv_res_peer
163 #define tk_registered pr_state_ptr->sv_tk_registered
164 #define to_llc_ici_ptr pr_state_ptr->sv_to_llc_ici_ptr
165 #define tk_trace_on pr_state_ptr->sv_tk_trace_on
166 */ Process model interrupt handling procedure */

167 void cp_fddi_mac ()
168 {
169 } /* Packets and ICI's */
170 { /* Packet Fields and Attributes */
171 Packet* mac_frame_ptr;
172 Packet* pdu_ptr;
173 Packet* pkptr;
174 Packet* data_pkptr;
175 Ici* ici_ptr;
176 int req_pri, svc_class, req_tk_class;
177 int frame_control, src_addr, dest_addr;
178 int pk_len, pri_level;
179 */ Token - Related */
180 }
182 int tk_usable, res_station, tk_class;
183 int current_tk_class;
184 double accum_sync;
185 /* Timer - Related */
186 double tx_time, timer_remaining, accum_bandwidth;
187 double tht_value;
188 /* Miscellaneous */
189 int i;
190 int spawn_station, phy_arrival;
191 char error_string [512];
192 int num_frames_sent, num_bits_sent;
193 /* 26DEC93: loop management variables, used in RCV_TK */
194 /* and ENCAP states. */
195 int NUM_PRIOS;
196 int punt;
197 int q_check;

198 FSM_ENTER (cp_fddi_mac)
199 FSM_BLOCK_SWITCH
200 {
201 /*-----------------------------------------------*/
202 /* state (INIT) enter executives */
203 FSM_STATE_ENTER_FORCED (0, state0_enter_exec, "INIT")
204 {
205 /* Obtain the station's address. This is an attribute */
206 /* of this process. Addressing is simplified by */
207 /* simply using integers, and only one mode. */
208 /* This mode is 16 bit addressing unless the */
209 /* packet format 'fddi_mac_fr' is modified. */
210 my_objid = op_id_self(); /* 29DEC93 */
211 op_ima_obj_attr_get (my_objid, "station_address", &my_address);
212 /* Register the station's object id in a global table. */
213 /* This table is used by the mechanism which improves */
214 /* simulation efficiency by 'jumping over' idle periods */
215 /* rather than circulating an unusable token. */
216 fddi_station_register (my_address, my_objid);
217 /* Obtain the station latency for tokens and frames. */
218 /* Default value is set at 100 nanoseconds. */
219 Fddi_St_Latency = 100.0e-09;
220 op_ima_sim_attr_get (OPC_INA_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;
op ima_sim_attr_get (OPC_IMA_DOUBLE, "prop-delay", &Fddi_Prop_Delay);

/* Derive the Delay for a 'hop' of a freely circulating packet. */
Fddi_Tk_Hop_Delay = Fddi_Prop_Delay + Fddi_Stat_Delay;

/* 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 */
/* Be aware of integer-double arithmetic conflicts ie, 1/8 = 0. -Nix */

op ima_obj_attr_get(my_objid, "T_Req", &T_Req);
for (i = 0; i < 8; i++)
{
    T_Pri[i] = ((double)(i + 1.0)/6.0) * Fddi_T_Opr;
    /* printf("MAC INIT: T_Pri[%d] is %lf; \
    Fddi_T_Opr is %lf\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 TRT */
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 */
initially restricted = 0;
Create an Interface Control Information structure */
/to use when delivering received frames to the LLC. */
to_llc_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 */
'teasure' 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_attr_get (OPC_INA_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_attr_get (my_objid, "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_attr_get (OPC_INA_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. */
14APR94: the bridges will spawn token in both rings */
-Harayakaylar */
spawn_token = 1;
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_TC_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_DEFAULT_COND
FSM_TEST_LOGIC ("INIT")
FSM_TRANSITION_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 Entered_exec, "IDLE")
{
}

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

/** 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_prg_odb_ltrace_active ("fddi_tk");
334 /* Trap packets arriving from physical layer so that their */
335 /* PC field can be extracted before evaluating conditions */
336 if (op_intrpt_type () == OPC_INTRPT_STRM & op_intrpt_strm () != FDDI_LLCC_STRM_IN)
337 {
338 /* Acquire the arriving packet. */
339 pkptr = op_pk_get (FDDI_PHY_STMTM_IN);
340 /* Determine the type of packet by extracting the */
341 /* the frame control field. */
342 op_pk_nfd_set (pkptr, "fc", &frame_control);
343 /* Physical layer arrival flag is set. */
344 phy_arrival = 1;
345 }
346 else{
347 /* The interrupt is not due to a physical layer arrival. */
348 phy_arrival = 0;
349 /* If the interrupt is a remote interrupt with specified code, it signifies */
350 /* the reinsertion of the token into the ring after an idle period. This only */
351 /* occurs if the token acceleration mechanism is active. */
352 if (op_intrpt_type () == OPC_INTRPT_REMOTE & op_intrpt_code () == FDDIC_TK_INJEXT)
353 {
354 /* create a new token */
355 tk_pkptr = op_pk_create_fmt ("fddi_mac tk");
356 /* assign its frame control field */
357 op_pk_nfd_set (tk_pkptr, "fc", FDDI_FC_TOKEN);
358 /* the token is non-restricted */
359 op_pk_nfd_set (tk_pkptr, "class", FDDI_TK_NOWRESTRICTED);
360 /* insert it into the ring */
361 op_pk_send (tk_pkptr, FDDI_PHY_STMTM_OUT);
362 }
363 }
364 } /* state (IDLE) transition processing */
365 FSM_INIT_COND (TK_RECEIVED)
366 FSM_TEST_COND (RC_FRAME)
367 FSM_TEST_COND (TRT_EXPIRE)
368 FSM_TEST_COND (FRAME_ARRIVAL)
369 FSM_DEFAULT_COND
370 FSM_TEST_LOGIC ("IDLE")
371 FSM_TRANSIT_SWITCH
372 {
373 FSM_CASE_TRANSIT (0, 3, state3_enter_exec, ;)
374 FSM_CASE_TRANSIT (1, 4, state4_enter_exec, ;)
375 FSM_CASE_TRANSIT (2, 7, state7_enter_exec, ;)
376 FSM_CASE_TRANSIT (3, 8, state8_enter_exec, ;)
377 FSM_CASE_TRANSIT (4, 1, state1_enter_exec, ;)
378 }
379 /\-----------------------------------------------\/

380 /** state (ISSUE_TK) enter executives **/
381 FSM_STATE_ENTER_FORCED (2, state2_enter_exec, "ISSUE_TK")
382 {
383 /* If the token is sent without having been used, and the station */
384 /* has no data to send, then indicate this fact to the */
385 /* token acceleration mechanism which may have an */
386 /* opportunity to block the token. */
387 if (!tk_usable && op_q_stat (OPC_QSTAT_PKSIZE) == 0.0)
388 {
389 /* Note that if the token cannot be blocked, */
390 /* this procedure will forward the token physically. */
391 fddi_tk_indicate_no_data (tk_pkptr, my_address, accum_bandwidth);
392 }
393 else{
394 if (tk_trace_on == OPC_TRUE)
395 {
396 sprintf (str0, "Issui-\r token. accum_bw (%.9f), prop_del (%.9f)"
397 accum_bandwidth, Fddi_Prop_Delay);
398 op_prv_odb_print_major (str0, OPC_WIL);
399 }
400 /* Send out the token packet using the accumulated */
401 /* consumed bandwidth as a scheduling base. */
402 /* In the case of the initial spawning of the token */
403 /* this will be zero; otherwise this variable will */
404 /* reflect the bandwidth consumed since the last capture */
405 /* of the usable token. Propagation delay is also accounted for. */
406 op_pk_send_delayed (tk_pkptr, FDDI_PHY_STRM_OUT,
407 accum_bandwidth + Fddi_Prop_Delay);
408 }
409 }

410 /** state (ISSUE_TK) exit executives **/
411 FSM_STATE_EXIT_FORCED (2, state2_exit_exec, "ISSUE_TK")
412 {
413 }
414 /**< state (ISSUE_TK) transition processing */
415 FSM_TRANSIT_FORCE (I, state1_enter_exec, ;)
416 */-----------------------------*/

417 /**< state (RCV_TK) enter executives */
418 FSM_STATE_ENTER_FORCED (3, state3_enter_exec, "RCV_TK")
419 {
420 /* The arriving packet, when received in the IDLE state */
421 /* is placed in the variable 'pkptr'. Since it is now */
422 /* known that it is a token, it can be placed in 'tk_pkptr'. */
423 tk_pkptr = pkptr;
424 
425 /* Load the token's class into the temporary variable 'tk_class.' */
426 op_pk_nfd_get (pkptr, "class", &tk_class);

427 /* If the token is restricted, determine for which station. */
428 if (tk_class == FDDI_TK_RESTRICTED)
429 {
430 /* Place the station address in the variable 'res_station' */
431 /* which may factor into the determination of token usability. */
432 op_pk_nfd_get (tk_pkptr, "res_station", &res_station);
433 }

434 /* Determine if the token is usable: */
435 /* assume by default that it is not */
436 /* Subsequent conditions may override this. */
437 tk_usable = 0;

438 /* The token can only be usable if there are frames enqueued */
439 /* 27DEC93: the entire bank of subqueues must be checked, */
440 /* starting at the highest priority (corresponding to */
441 /* synchronous traffic), and stopping when a packet is */
442 /* found. Then the loop is broken. */
443 NUM_PRIOS = 9;
444 for (i = NUM_PRIOS - 1; i > -1; i--)
445 {
446   if (op_subq_stat (i, OPC_QSTAT_PKSIZE) > 0.0)
447     {
448       /* examine the attributes of the packet at the */
449       /* head of the queue. */
450       fddi_load_frame_attrs (&dest_addr, &svc_class, &pri_level);
451     /* If synchronous data is queued, the token is */
452     /* necessarily usable, regardless of timing conditions. */
453     if (svc_class == FDDI_SVC_SYNC)
454     {

86
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)
            if (res_station == my_address)
                
    } /* closes the "if (op-subq-stat (OPC_QSTAT_PKSIZE) > 0.0" statement */
    /* closes the "for" loop */

    /* 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);
        /* clears the "for" loop */
        TRT_SET (my_address, op_sim_time () + Fddi_T_Opr);
    /* closes the "if (tk_usable)" */
}
/* 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. */
/* Test the frame's priority against the current TRT */
if (T_Pri [pri_level] >= fddi_timer_value (TRT))
{
    tk_usable = 1;
    break;
}
/* opens the "if (op-subq-stat (OPC_QSTAT_PKSIZE) > 0.0" statement */
/* closes the "for" loop */
else {
    /* If the token is late, set the TRT 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 TRT remains before TRT expires again. */
lateCt = 0;
}

else{
    /* If the token is not usable, different adjustments are made. */
    /* Again, the adjustments depend on the lateness of the token */
    if (lateCt == 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 TRT remains before TRT expires again. */
        lateCt = 0;
    }
}

/* If the token is not usable, different adjustments are made. */
/* Again, the adjustments depend on the lateness of the token */
if (lateCt == 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 TRT remains before TRT expires again. */
    lateCt = 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
541 FSM_TEST_LOGIC ("RCV_TK"),

542 FSM_TRANSIT_SWITCH
543 {
544 FSM_CASE_TRANSIT (0, 9, state9_enter_exec, ;)
545 FSM_CASE_TRANSIT (1, 2, state2_enter_exec, ;)
546 }
547 /***************************************************/

548 /** state (FR_RCV) enter executives **/
549 FSM_STATE_ENTER_FORCED (4, state4_enter_exec, "FR_RCV")
550 {
551 /* A frame has been received from the physical layer. Note that */
552 /* at this time, only the leading edge of the frame has arrived. */
553 /* Extract the frame's source address (this will be used to */
554 /* determine whether or not to strip the frame from the ring). */
555 op_pk_nfd_get (pkptr, "src_addr", &src_addr);
556 }

557 /** state (FR_RCV) exit executives **/
558 FSM_STATE_EXIT_FORCED (4, state4_exit_exec, "FR_RCV")
559 {
560 }

561 /** state (FR_RCV) transition processing **/
562 FSM_INIT_COND (STRIP)
563 FSM_DFLT_COND
564 FSM_TEST_LOGIC ("FR_RCV")

565 FSM_TRANSIT_SWITCH
566 {
567 FSM_CASE_TRANSIT (0, 5, state5_enter_exec, ;)
568 FSM_CASE_TRANSIT (1, 6, state6_enter_exec, ;)
569 }
570 /***************************************************/

571 /** state (FR_STRIP) enter executives **/
572 FSM_STATE_ENTER_FORCED (5, state5_enter_exec, "FR_STRIP")
573 {
574 /* Destroy the frame which has now circulated the entire ring. */
575 op_pk_destroy (pkptr);
576 }

89
577 /* state (FR_STRIP) exit executives */
578 FSM_STATE_EXIT_FORCED (5, state6_exit_exec, "FR_STRIP")
579 {
580 }
581 /* state (FR_STRIP) transition processing */
582 FSM_TRANSIT_FORCED (1, state1_enter_exec, ;)
583 /*------------------------------------------*/
584 /* state (FR_REPEAT) enter executives */
585 FSM_STATE_ENTER_FORCED (6, state6_enter_exec, "FR_REPEAT")
586 {
587 /* Extract the destination address of the frame. */
588 op_pk_nfd_get (pkptr, "dest_addr", &dest_addr);
589 /* If the frame is for this station, make a copy */
590 /* of the frame's data field and forward it to */
591 /* the higher layer. */
592 /* 14APR94: In order to send the frames which are */
593 /* addressed to the remote lan, check the address database */
594 /* of remote lan. Frames addressed to the remote lan shouldn't */
595 /* be repeated in the local ring -- This is a simple forwarding */
596 /* decision algorithm, one of the bridge's function */
597 /* Karayakaylar */
598 if((dest_addr == my_address)||(dest_addr > my_address))
599 {
600 /* record total size of the frame (including data) */
601 pk_len = op_pk_total_size_get (pkptr);
602 /* decapsulate the data contents of the frame */
603 /* 29JAN94: a new field, "pri", has been added to */
604 /* the fddi_llc_fr packet format in the Parameters */
605 /* Editor, so that output statistics can be */
606 /* generated by class and priority. -Six */
607 op_pk_nfd_get (pkptr, "info", &data_pkptr);
608 op_pk_nfd_get (pkptr, "pri", &pri_level);
609 /* The source and destination address are placed in the */
610 /* LLC's ICI before delivering the frame's contents. */
611 op_icl_attr_set (to_llc_ici_ptr, "src_addr", src_addr);
612 op_icl_attr_set (to_llc_ici_ptr, "dest_addr", dest_addr);
613 op_icl_install (to_llc_ici_ptr);
614 /* Because, as noted in the FR_RCV state, only the */
615 /* frame's leading edge has arrived at this time, the */
616 /* complete frame can only be delivered to the higher */
617 /* layer after the frame's transmission delay has elapsed. */
618 /* (since decapsulation of the frame data contents has occurred, */
tx-time = (double) pk_len / FDDI_TX_RATE;

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 changed so that the frame is copied and the original repeated. Logic is already present for stripping the frame at the origin. Thus, local traffic is constrained. -- This is filtering decision.

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 station can strip the frame).

The timer is reset and allowed to continue running. The late token counter is incremented. This will

state (FR_REPEAT) exit executives

The timer is reset and allowed to continue running.
660 /\* prevent this station from making any asynchronous */
661 /\* transmissions when it next captures the token. */
662 Late_Ct++;
663 }
664 /\* state (TRT_EXP) exit executives */
665 FSN_STATE_EXIT_FORCED (7, state7_exit_exec, "TRT_EXP")
666 {
667 }
668 /\* state (TRT_EXP) transition processing */
669 FSN_TRANSIT_FORCE (1, state1_enter_exec, ;)
670 /-------------------------*-

671 /\* state (ENCAP) enter executives */
672 FSN_STATE_ENTER_FORCED (8, state8_enter_exec, "ENCAP")
673 {
674 /\* A frame has arrived from a higher layer; place it in 'pdu_ptr'. */
675 pdu_ptr = op_pk_get (op_intrpt_strm ());

676 /\* Also get the interface control information */
677 /\* associated with the new frame. */
678 ici_ptr = op_intrpt_ici ();
679 if (ici_ptr == OPCNIL)
680 {
681 sprintf (error_string, "Simulation aborted; error in object (%d)",
682 op_id_self ());
683 op_sim_end (error_string, "fddi_mac: required ICI not received", " ", " ");
684 }
685 /\* Extract the requested service class */
686 /\* (e.g., synchronous or asynchronous). */
687 if (op_ici_attr_exists (ici_ptr, "svc_class"))
688 op_ici_attr_get (ici_ptr, "svc_class", &svc_class);
689 else svc_class = FDDI_SVC_ASYNC;
690 /\* Extract the destination address. */
691 op_ici_attr_get (ici_ptr, "dest_addr", &dest_addr);
692 /\* Extract the original source address from ICI :16APR94 */
693 op_ici_attr_get (ici_ptr, "src_addr", &orig_src_addr);
694 /\* If the frame is asynchronous, the priority and */
695 /\* requested token class parameter may be specified. */
696 if (svc_class == FDDI_SVC_ASYNC)
if (op_ici_attr_exists (ici_ptr, "pri"))
    op_ici_attr_get (ici_ptr, "pri", &req_pri);
else req_pri = 0;

if (op_ici_attr_exists (ici_ptr, "tk_class"))
    op_ici_attr_get (ici_ptr, "tk_class", &req(tk_class);
else tk_class = FDDI_TK_NONRESTRICTED;

/* Check for the default ICI values; if they are not present */

/* compose the frame:21APR94 */
if( dest_addr != orig_src_addr){

    mac_frame_ptr = op_pk_create frat ("fddi_mac_frame");
    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", orig_src_addr);

    printf("\ndest_addr = %Sd\n",dest_addr);
    printf("orig_src_addr= %Sd\n",orig_src_addr);

    if (svc_class == FDDI_SVC_ASYNC)
    {
        op_pk_nfd_set (mac_frame_ptr, "tk_class", req(tk_class);
        op_pk_nfd_set (mac_frame_ptr, "pri", req_pri);
    }

    if (svc_class == FDDI_SVC_SYNC)
    {
        op_pk_nfd_set (mac_frame_ptr, "pri", 8);
    }

    op_pk_nfd_set (mac_frame_ptr, "fc", FDDI_FC_FRAME);

    /* Enqueue the frame at the tail of the queue. */
    op_pk_nfd_set (mac_frame_ptr, "fc", FDDI_FC_FRAME);

    if (svc_class == FDDI_SVC_ASYNC)
    {

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_QPOS_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;
}
/* end of if(dest_addr != orig_src_addr) statement */

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

/** state (ENCAP) transition processing **/
FSM_TRANSIT_FORCE (1, state1_enter_exec, ;)
/e---------------------------------------------------------------*/

/** 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 */

I 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 'tht_value' is used to emulate the timer's progress. */
tht_value = fddi_timer_value (THT);

************
30MAR94: print T_Pri[i]. THT data */
for (i = 0; i < 8; i++)
{
    printf("TX_DATA: T_Pri[%d] = %d, THT = %d, Fddi_T_Opr = %d\n", i, T_Pri[i], tht_value */
}

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 accomodate 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_QSTAT_PFSIZE) > 0.0)
    {
        /* Remove the next frame for transmission. */
        pkptr = op_subq_pk_remove (i, OPC_QPOS_HEAD);
        /* Obtain the frame's service class. */
        op_pk_nfd_get (pkptr, "svc_class", &svc_class);
        /* Synchronous and asynchronous frames are treated differently. */
        if (svc_class == FDDI_SVC_SYNC)
        {
            /* Obtain the frame's length, and compute */
            /* the time required to transmit it. */
            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_QPOS_HEAD);
}
/* Exit the transmission loop since the frame */
/* transmission request cannot be honored. */
    punt = 1;
    break;
}
else{
    /* 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_STRM_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", apri_level);

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

    }
870 /* after replacing the packet in the queue. */
871 if (TPri [pri_level] < tht_value)
872 {
873     op_subq_pk_insert (i, pkptr, OPC_QPOS_HEAD);
874     punt = i;
875     break;
876 }
877 /* Obtain the frame's length, and compute the time */
878 /* which would be required to transmit it. */
879     pk_len = op_pk_total_size_get (pkptr);
880     tx_time = (double) pk_len / FDDI_TX_RATE;
881 /* Determine the requested token class to be */
882 /* released after this frame is transmitted. */
883     op_pk_nfd_get (pkptr, "tk_class", &tk_class);
884 /* If the station is in restricted mode, then it may */
885 /* exit this mode if the class is now nonrestricted */
886 /* or if the restricted peer is not the addressee. */
887 if (restricted)
888 {
889     /* Determine the destination address for the new packet. */
890     op_pk_nfd_get (pkptr, "dest_addr", &dest_addr);
891     if (tk_class == FDDI_TK_NOWRESTRICTED ||
892         res_peer != dest_addr)
893 {
894         /* Exit restricted mode */
895         restricted = 0;
896         /* Modify the token to reflect the mode change. */
897         op_pk_nfd_set (tk_pkptr, "class", FDDI_TK_NOWRESTRICTED);
898     }
899 }
900 }
901 else{
902     /* Determine the class of the current captured token. */
903     op_pk_nfd_get (tk_pkptr, "class", &current_tk_class);
904     /* When not in restricted mode, this mode may be entered */
905     /* if the passed packet has the appropriate token class requested, */
906     /* and the token is not already restricted. */
907     if (tk_class == FDDI_TK_RESTRICTED &&
908         current_tk_class != FDDI_TK_RESTRICTED)
909 {
910     /* Enter restricted mode. */
911     restricted = 1;
912     /* Store the address of the restricted peer station. */
913     op_pk_nfd_get (pkptr, "dest_addr", &res_peer);
Modify the token to reflect the mode change. */
op.pk.nfd.set (tk.pkptr, "class", FDDI_TK_RESTRICTED);
op.pk.nfd.set (tk.pkptr, "res_station", res_peer);
}
}

Send the frame once previous transmissions have completed. */
Account for propagation delay as well. */
op.pk.send_delayed (pkptr, FDDI_PHY_STRM_OUT, accum_bandwidth + Fddi_Prop_Delay);

Increment THT emulation variable, and consumed bandwidth accumulator. */

Increase counters for transmitted bits and frames. */
num_frames_sent++;
um_bits_sent += pk_len;
}
}

/* closes the 'while' loop */
if (punt == 1) /* If the 'while' loop was broken, */
{
punt = 0; /* then reset the 'break' marker, */
break; /* and break out of the 'for' loop too. */
}

/* closes the 'for' loop. */

Since the token is about to be sent, its transmission time */

must be reflected in the accumulated bandwidth. This is not */
done in the ISSUE_TK state because when the token is merely */
repeated, full transmission delay is not required, only */
a small delay for repeating. */
accum_bandwidth += FDDIC_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")

my_objid = op_id_self();

op_ima_obj_attr_get (my_objid, "ring.id", &ring_id);

Fddi_Tk_Block = 0;
Fddi_Num_Stations = 0;
Fddi_Num_Registered = 0;

op_ima_obj_attr_get (my_objid, "T_R.q", &TReq);

Fddi_T_Opr = 500;
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 */
    Fddi_Claim_Start = 0;
}

/\ 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,cp_fddi_mac)

/** state (CLAIM) exit executives **/
FSM_STATE_EXIT_UNFORCED (10, state10_exit_exec, "CLAIM")
{
}

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

FSM_EXIT (10,cp_fddi_mac)
FSM_EXIT (10,cp_fddi_mac)
void cp_fddi_mac_svar (prs_ptr, var_name, var_p_ptr)
cp_fddi_mac_state *prs_ptr;
char *var_name, **var_p_ptr;
{
FIN (cp_fddi_mac_svar (prs_ptr))

*var_p_ptr = VOS_NIL;
if (Vos_String_Equal ("ring_id", var_name))
*var_p_ptr = (char *) (prs_ptr->sv_ring_id);
if (Vos_String_Equal ("THT", var_name))
*var_p_ptr = (char *) (prs_ptr->sv_THT);
if (Vos_String_Equal ("T_Req", var_name))
*var_p_ptr = (char *) (prs_ptr->sv_T_Req);
if (Vos_String_Equal ("T_Pri", var_name))
*var_p_ptr = (char *) (prs_ptr->sv_T_Pri);
if (Vos_String_Equal ("my_obj_id", var_name))
*var_p_ptr = (char *) (prs_ptr->sv_my_objid);
if (Vos_String_Equal ("spawn_token", var_name))
*var_p_ptr = (char *) (prs_ptr->sv_spawn_token);
if (Vos_String_Equal ("my_address", var_name))
*var_p_ptr = (char *) (prs_ptr->sv_my_address);
if (Vos_String_Equal ("orig_src_addr", var_name))
*var_p_ptr = (char *) (prs_ptr->sv_orig_src_addr);
if (Vos_String_Equal ("tk_pkptr", var_name))
*var_p_ptr = (char *) (prs_ptr->sv=tk_pkptr);
if (Vos_String_Equal ("sync_bandwidth", var_name))
*var_p_ptr = (char *) (prs_ptr->sv_sync_bandwidth);
if (Vos_String_Equal ("sync_pc", var_name))
*var_p_ptr = (char *) (prs_ptr->sv_sync_pc);
if (Vos_String_Equal ("restricted", var_name))
*var_p_ptr = (char *) (prs_ptr->sv_restricted);
if (Vos_String_Equal ("res_peer", var_name))
*var_p_ptr = (char *) (prs_ptr->sv_res_peer);
if (Vos_String_Equal ("tk_registered", var_name))
*var_p_ptr = (char *) (prs_ptr->sv=tk_registered);
if (Vos_String_Equal ("to_llc_ici_ptr", var_name))
*var_p_ptr = (char *) (prs_ptr->sv=to_llc_ici_ptr);
if (Vos_String_Equal ("tk_trace_on", var_name))
*var_p_ptr = (char *) (prs_ptr->sv=tk_trace_on);

POUT;
}

void cp_fddi_mac_diag ()
{
1067 /* Packets and ICI's */
1068 Packet* mac_frame_ptr;
1069 Packet* pdu_ptr;
1070 Packet* pkptr;
1071 Packet* data_pkptr;
1072 Ici* ici_ptr;

1073 /* Packet Fields and Attributes */
1074 int req_pri, svc_class, req_tk_class;
1075 int frame_control, src_addr, dest_addr;
1076 int pk_len, pri_level;

1077 /* Token - Related */
1078 int tk_usable, res_station, tk_class;
1079 int current_tk_class;
1080 double accum_sync;

1081 /* Timer - Related */
1082 double tx_time, timer_remaining, accum_bandwidth;
1083 double tht_value;

1084 /* Miscellaneous */
1085 int i;
1086 int spawn_station, phy_arrival;
1087 char error_string [612];
1088 int num_frames_sent, num_bits_sent;

1089 /* 26DEC93: loop management variables, used in RCV_TK */
1090 /* and ENCAP states. -Nix */
1091 int NUM_PRIOS;
1092 int punt;
1093 int q_check;

1094 FIN (cp_fddi_mac_diag ());

1095 /* Print out values of timers, and late token counter. */
1096 /* Also print out data about restricted mode. */
1097 /* (This code may be executed by the simulation debugger */
1098 /* by invoking the command 'modprint'). */

1099 sprintf (strO, "Timers (count upwards): TRT (%.9g), THT (%.9g)",
1100 fddi_timer_value (TRT), fddi_timer_value (THT));
1101 sprintf (strl, "Late_c (Xd)", Late_c);
1102 op_prf_odb.print_major (strO, strl, OPC_NIL);

1103 if (restricted)
1104 sprintf (strO, "token is in restricted dialog with (Xd)n", res_peer);
1105 else sprintf (strO, "token is unrestricted\n");

102
1106 op_prg_odb_print_major (str0, OPC_NIL);

1107 FOUT;
1108 }

1109 void
1110 cp_fddi_mac_terminate ()
1111 {
1112 /* Packets and ICI's */
1113 Packet* mac_frame_ptr;
1114 Packet* pdu_ptr;
1115 Packet* pkptr;
1116 Packet* data_pkptr;
1117 Ici* ici_ptr;

1118 /* Packet Fields and Attributes */
1119 int req_pri, svc_class, req_tk_class;
1120 int frame_control, src_addr, dest_addr;
1121 int pk_len, pri_level;

1122 /* Token - Related */
1123 int tk_usable, res_station, tk_class;
1124 int current_tk_class;
1125 double accum_sync;

1126 /* Timer - Related */
1127 double tx_time, timer_remaining, accum_bandwidth;
1128 double tht_value;

1129 /* Miscellaneous */
1130 int i;
1131 int spawn_station, phy_arrival;
1132 char error_string [612];
1133 int num_frames_sent, num_bits_sent;

1134 /* 26DEC93: loop management variables, used in RCV_TK */
1135 /* and ENCAP states. -Nix */
1136 int NUM_PAIOS;
1137 int punt;
1138 int q_check;

1139 FIN (cp_fddi_mac_terminate ())
```
1140 POUT;
1141 }

1142 Compcode
1143 cp_fddi_mac_init (pr_state_pptr)
1144 cp_fddi_mac_state **pr_state_pptr;
1145 {
1146 static VosT_Cm_Obtype obtype = OPC_NIL;
1147 FIN (cp_fddi_mac_init (pr_state_pptr))
1148 if (obtype == OPC_NIL)
1149 {
1150 if (Vos_Catmem_Register ("proc state vars (cp_fddi_mac)",
1151 sizeof (cp_fddi_mac_state), Vos_Nop, &obtype) == VOSC_FAILURE)
1152 FRET (OPC_COMPCODE_FAILURE)
1153 }
1154 if (((*pr_state_pptr) = (cp_fddi_mac_state*) Vos_Catmem_Alloc (obtype, 1)) == OPC_NIL)
1155 FRET (OPC_COMPCODE_FAILURE)
1156 else
1157 {
1158 (*pr_state_pptr)->current_block = 20;
1159 FRET (OPC_COMPCODE_SUCCESS)
1160 }
1161 }

1162 /** The procedures defined in this section serve **/
1163 /** to simplify the code in the main body of the **/
1164 /** process model by providing primitives for timer **/
1165 /** manipulation.. **/
1166 static
1167 fddi_timer_disable (timer_ptr)
1168 FddiT_Timer* timer_ptr;
1169 {
1170 /* if the timer is already disabled, do nothing */
1171 if (timer_ptr->enabled)
1172 {
1173 /* disable the timer */
1174 timer_ptr->enabled = 0;
1175 /* reassign the accumulated time so far */
1176 timer_ptr->accum = op_sim_time () - timer_ptr->start_time;
1177 }
1178 }
```
1179 static
defdi_timer_enable (timer_ptr)
defdi_Timer* timer_ptr;
1181 {
1182  /* if the timer is already enabled, simply return */
1183  if (!timer_ptr->enabled)
1184  {
1185     /* re-enable the timer */
1186     timer_ptr->enabled = 1;
1187  }
1188  /* set the start time to the current time */
1189  /* less the accumulated time so far */
1190  timer_ptr->start_time = op_sim_time () - timer_ptr->accum;
1191 }
1192 }
1193 static
defdi_timer_expired (timer_ptr)
defdi_Timer* timer_ptr;
1196 {
1197  if (defdi_timer_remaining (timer_ptr) <= 0.0)
1198     return 1;
1199  else return 0;
1200 }
1201 static
defdi_timer_remaining (timer_ptr)
defdi_Timer* timer_ptr;
1204 {
1205  /* if the timer is enabled, update the accumulated time */
1206  if (timer_ptr->enabled)
1207  {
1208      timer_ptr->accum = op_sim_time () - timer_ptr->start_time;
1209  }
1210  /* return the timer remaining before expiration */
1211  /* a non-positive value indicates an expired timer */
1212  return (timer_ptr->target_accum - timer_ptr->accum);
1214 }
1215 static
defdi_timer_value (timer_ptr)
defdi_Timer* timer_ptr;
1219 {
1220  /* if the timer is enabled, update the accumulated time */
1221  if (timer_ptr->enabled)
1222  {
1223      timer_ptr->accum = op_sim_time () - timer_ptr->start_time;
1224  }
return (timer_ptr->accum);
}

static
fddi_timer_set_value (timer_ptr, value)
FddiT_Timer* timer_ptr;
double value;
{
timer_ptr->accum = value;
}

static
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));
}

static
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_accum = duration;

/* assign the current time */
timer_ptr->start_time = op_sim_time ();

/* enable the timer */
timer_ptr->enabled = 1;
}

static
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);
1264 /* return the timer's address */
1265 return (timer_ptr);
1266 }

1267 static
1268 fddi_timer_init (timer_ptr)
1269 FddiTTTiaer* timer_ptr;
1270 {
1271 /* the timer is initially disabled */
1272 timer_ptr->enabled = 0;
1273 /* the accumulated time is zero */
1274 timer_ptr->accum = 0.0;
1275 /* the target accumulated time is infinite */
1276 timer_ptr->target_accu = VOS_DOUBLE_INFINITY;
1277 /* the start time is now */
1278 timer_ptr->start_time = op_sim_time();
1279 }
1280 static
1281 fddi_station_register (address, objid)
1282 Objid objid;
1283 int address;
1284 {
1285 /* Fill an entry in the table which maps station */
1286 /* addresses to OPNET object ids */
1287 FNN (fddi_station_register (address, objid))
1288 Fddi_Address_Table[address] = objid;
1289 /* Keep track of total number of stations on the ring */
1290 Fddi_Num_Stations++;
1291 FOUT
1292 }

1293 static
1294 fddi_tk_register ()
1295 {
1296 /* Register the station's intent to use the token. */
1297 /* This should be done whenever an unregistered */
1298 /* station obtains new data to transmit. */
1299 FNN (fddi_tk_register ())
1300 /* increase the number of registered stations */
1301 Fddi_Num_Registered++;
1302 /* if the token is currently blocked, unblock it */
1303 if (Fddi_Tk<Blocked && Fddi_Tk_Accelerate)
1304 {

107
1305 fddi_tk_unblock();
1306 }

1307 FOUT
1308 }
1309 static
1310 fddi_tk_deregister ()
1311 {
1312 /* Cancel the station's intent to use the token. */
1313 /* This should be done whenever a registered */
1314 /* station exhausts its transmittable data. */
1315 FOUT (fddi_tk_deregister ())
1316 /* decrease the number of registered stations */
1317 Fddi_Num_Registered--;
1318 FOUT
1319 }

1320 static
1321 fddi_tk_indicate_no_data (token, address, delay)
1322 Packet* token;
1323 int address;
1324 double delay;
1325 {
1326 FOUT (fddi_tk_indicate_no_data (token, address, delay))
1327 /* The calling station is indicating that it has captured */
1328 /* the token, but has no data to send. If no other stations */
1329 /* have data to send either, the token may be blocked to gain */
1330 /* simulation efficiency. */
1331 if (Fddi_Num_Registered == 0 && Fddi_Tk_Accelerate)
1332 {
1333 fddi_tk_block (token, address);
1334 }
1335 else{
1336 /* If the token cannot be blocked, send it into the ring. */
1337 op_pk_send_delayed (token, FDDI_PHY_STRM_OUT,
1338 delay + Fddi_Prop_Delay);
1339 }
1340 }

1341 static
1342 fddi_tk_block (token, address)
1343 Packet* token;
1344 int address;
1345 {
1346 int i;

108
FIN (fddi_tk_block (token, address))

1348 // Record the address of the blocking station and blocking time. */
1349 Fddi_Tk_Block_Base_Time = op_sim_time ();
1350 Fddi_Tk_Block_Base_Station = address;

1351 if (tk_trace_on == OPC_TRUE)
1352 {
1353 sprintf (str0, "Blocking Token: station (%d), time (%.9f)",
1354 Fddi_Tk_Block_Base_Station, Fddi_Tk_Block_Base_Time);
1355 op_prg_odb_print_major (str0, OPC_WIL);
1356 }
1357 // Indicate that the token is blocked */
1358 Fddi_Tk_Blocked = 1;

1359 // discard the token packet; another one will be */
1360 // created when the token is unblocked. */
1361 op_pk_destroy (token);

1362 // Cancel TAT timers at all MAC interfaces; otherwise these */
1363 // timers may continue to expire during the idle period, */
1364 // generating unnecessary events. */
1365 if (tk_trace_on == OPC_TRUE)
1366 {
1367 sprintf (str0, "Canceling timers for (%d) stations", Fddi_Num_Stations);
1368 op_prg_odb_print_major (str0, OPC_WIL);
1369 }

1370 for (i = 0; i < Fddi_Num_Stations; i++)
1371 {
1372 // Retain the time at which the TAT would have expired; */
1373 // this is used for calculations when the token is */
1374 // reinjected into the ring. */
1375 Fddi_Trt_Exp_Time [i] = op_ev_time (Fddi_Trt_Handle [i]);

1376 // Cancel the TAT expiration event. */
1377 op_ev_cancel (Fddi_Trt_Handle [i]);
1378 }
1379}
1380
1381 static
1382 fddi_tk_unblock ()
1383 {
1384 double elapsed_time, first_tk_rx, last_tk_rx;
1385 double tk_lap_time, next_time, current_time;
1386 double dbl_num_hops, num_tk_rx, floor (), ceil ();
1387 int i, num_hops, next_station;

109
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 (str0, "Unblocking token for ring (%d)", ring_id);
    op_prg_odb_print_major (str0, 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 (str0, "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_tkn_rx = Fddi_Tk_Block_Base_Time + FDDIC_TOKEN_RX_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_tkn_rx);
        op_prg_odb_print_minor (str0, str1, OPC_NIL);
    }

    /* Case 1: the token would not yet have been received by station i. */
    if (first_tkn_rx > current_time)

110
1430 {
1431 /* Case 1a: the TRT at station i would not yet have expired. */
1432 if (Fddi_Trt_Exp_Time [i] > current_time)
1433 {
1434 /* Late_Ct remains at its original value; only the TRT needs */
1435 /* to be started again, with the same expiration time. */
1436 TRT_SET (i, Fddi_Trt_Exp_Time [i])
1437 if (tk_trace_on == OPC_TRUE)
1438 {
1439 sprintf (strO, "Restoring TRT to previous exp. time (%.9f)", Fddi_Trt_Exp_Time [i]);
1440 op_prg_odb_print_minor ("Token would not be received and TRT not expired", strO, OPC_NIL);
1441 }
1442 }
1443 /* Case 1b: the TRT at station i would have expired. */
1444 else
1445 {
1446 /* Late_Ct would have been set; also the timer would have been rescheduled */
1447 /* for an entire TTAT at the time of expiration. */
1448 Fddi_Late_Ct [i] = 1;
1449 TRT_SET (i, (Fddi_T_Opr + Fddi_Trt_Exp_Time [i]))
1450 if (tk_trace_on == OPC_TRUE)
1451 {
1452 sprintf (strO, "Restoring TRT to proper exp. time (%.9f)", Fddi_T_Opr + Fddi_Trt_Exp_Time [i]);
1453 op_prg_odb_print_minor ("Token would not be received and TRT would have expired", strO, OPC_FALSE);
1454 }
1455 }
1456 }
1457 /* Case 2: the token would have been received (perhaps more than once). */
1458 else
1459 {
1460 /* Calculate the number of times the token would have been received */
1461 /* not including the first receipt. */
1462 tk_lap_time = Fddi_Tk_Hop_Delay * Fddi_Num_Stations;
1463 num_tx_rx = floor ((current_time - first_tk_rx) / tk_lap_time);
1464 /* Calculate the latest time at which the token would have been received. */
1465 last_tk_rx = first_tk_rx + (num_tx_rx * tk_lap_time);
1466 /* Clear Late_Ct and schedule timer to expire at last receipt of token */
1467 /* plus one full TTAT. */
1468 Fddi_Late_Ct [i] = 0;
1469 TRT_SET (i, (last_tk_rx + Fddi_T_Opr))
1470 if (tk_trace_on == OPC_TRUE)
1471 {
1472 sprintf (strO, "token received (%d) times, last receipt at (%.9f)",
1473 num_tx_rx + 1.0, last_tk_rx);
1474 sprintf (str1, "Restoring TRT to proper exp. time (%.9f)"),
1475 111
1476 last_tk_rx + Fddi_T_Opr);  
1477 opKeyPressed ("Token would have been received; LateCt is cleared",  
1478 str1, str0, OPC_NIL);  
1479 }  
1480 }  
1481 }  
1482 /* compute the time since the token was blocked */  
1483 elapsed_time = current_time - Fddi_Tk_Block_Base_Time;  
1484  
1485 /* compute the number of hops completed on the ring. For the first hop */  
1486 /* the token is transmitted directly, not repeated. For all remaining */  
1487 /* hops, the delay is the station latency plus the propagation delay. */  
1488 /* Thus, the first hop is assumed, and the remaining time for additional*/  
1489 /* hops is computed beginning at the time where the token enters the */  
1490 /* base station's downstream neighbor. */  
1491 dbl_num_hops = 1.0 +  
1492 (elapsed_time - FDDI_TOKEN_TX_TIME - Fddi_Prop_Delay) / Fddi_Tk_Hop_Delay;  
1493  
1494 /* If the token was unblocked in less time than it would have taken to */  
1495 /* be fully transmitted by the base station, dbl_num_hops will be */  
1496 /* negative. However, 1 full hop would still be required before the */  
1497 /* token could be used, since the station had already committed to */  
1498 /* issuing the token. Thus, the actual number of hops should never */  
1499 /* be less than 1. If it is, round it to 1. */  
1500 if (dbl_num_hops < 1.0)  
1501 dbl_num_hops = 1.0;  
1502 else  
1503 {  
1504 dbl_num_hops = ceil (dbl_num_hops);  
1505 }  
1506  
1507 /* Obtain an integer equivalent of dbl_num_hops. */  
1508 num_hops = dbl_num_hops;  
1509  
1510 /* Based on the number of hops and the base station, compute the */  
1511 /* next station where the token will appear. */  
1512 next_station = (num_hops + Fddi_Tk_Block_Base_Station) % Fddi_Num_Stations;  
1513  
1514 /* Compute the time at which the token will appear there. */  
1515 /* Again, assume the first hop occurred, and measure time */  
1516 /* from there forward. */  
1517 next_time = Fddi_Tk_Block_Base_Time + (FDDI_TOKEN_TX_TIME + Fddi_Prop_Delay) +  
1518 (dbl_num_hops - 1.0) * Fddi_Tk_Hop_Delay;  
1519  
1520 if (tk_trace_on == OPC_TRUE)  
1521 {  
1522 sprintf (str0, "Re-introducing token at station (Yd), at time (X.9f)",  
1523 next_station, next_time);  
1524}
1520 op_prd_odb_print_minor (str0, OPC_NIL);
1521 }
1522
1523 fddi tk_inject (next_station, next_time);
1524 FOUT
1525 }
1526
1527 fddi tk_inject (address, arv_time)
1528 int address;
1529 double arv_time;
1530 {
1531 /* Reinsert the token into the ring after an idle period. */
1532 FIN (fddi tk_inject (address, arv_time))
1533 /* The token is recreated and reinserted onto the ring */
1534 /* at the specified station which is not necessarily the */
1535 /* station now requesting the token. */
1536 /* The station which will reinsert the token is */
1537 /* asked to do so by means of a remote interrupt. */
1538 op_intrpt_schedule_remote (arv_time, FDDIC_TK_INJECT,
1539 Fddi_Address_Table [address]);
1540 FOUT
1541 }
1542
1543 fddi_load_frame_attrs (dest_addr_ptr, svc_class_ptr, pri_level_ptr)
1544 int *dest_addr_ptr, *svc_class_ptr, *pri_level_ptr;
1545 {
1546 int NUM_PRIOS, i; /* 26JAN94 */
1547 Packet *epkptr;
1548 FIN (fddi_load_frame_attrs (dest_addr_ptr, svc_class_ptr, pri_level_ptr))
1549 /* remove next packet in queue */
1550 /* 27DEC94: loop structure superimposed to handle a bank of subqueues. */
1551 /* Extract the packet with the highest priority, that is, the packet */
1552 /* at the head of the highest-numbered subqueue containing packets. */
1553 /* Note that the C language vector numbering convention numbers the */
1554 /* subqueues from 0 to 7, while FDDI convention is to number the */
1555 /* corresponding asynchronous priorities from 1 to 8. This is */
1556 /* reconciled in the statistical outputs available in the Analysis */
1557 /* Editor, where labels are assigned accordingly. Also note that */
1558 /* synchronous traffic is assigned priority 8 as an artifice to allow */
1559 /* routing through a separate subqueue, by which statistics may be */
1560 /* gathered for traffic by class and by priority. -Mix */
1561 NUM_PRIOS = 9;
1562 for (i = NUM_PRIOS - 1; i > -1; i--)
1563 .
1563     
1564     if (op_subq_stat (i, OPC_QSTAT_PKSIZE) > 0.0)
1565     
1566     pkptr = op_subq_pk_remove (i, OPC_QPOS_HEAD);
1567     break;
1568     
1569 
1570     /* extract the fields of interest */
1571     op_pk_nfd_get (pkptr, "dest_addr", dest_addr_ptr);
1572     op_pk_nfd_get (pkptr, "svc_class", svc_class_ptr);

1573     /* only read priority level if frame is asynchronous */
1574     if (*svc_class_ptr == FDDI_SVC_ASYNC)
1575     op_pk_nfd_get (pkptr, "pri", pri_level_ptr);

1576     /* replace the packet on the proper subqueue */
1577     op_subq_pk_insert (i, pkptr, OPC_QPOS_HEAD);

1578     }
APPENDIX C
CPNI SINK "C" CODE
“cp_fddi_sink.pr.c”

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

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

2 /* OPNET system definitions */
3 #include <opnet.h>
4 #include "cp_fddi_sink.pr.h"
5 FSM_EXIT_DECS

6 /* Header block */
7 /* Globals */
8 /* array format installed 20JAN94; positions 0-7 represent the async priority levels, PRIORI
9 /* represents synch traffic, and grand totals are as given in the original.

10 #define PRIORITIES 8 /* 20JAN94 */
11 #define XMITTER_ONE 0 /*10MAY94*/
12 #define XMITTER_TWO 1
13 #define XMITTER_THREE 2
14 #define XMITTER_FOUR 3

15 static /* 05FEB94 */
16 double fddi_sink_accum_delay = 0.0;
17 static /* 05FEB94 */
18 double fddi_sink_accum_delay_a[PRIORITIES+1]={0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
19 static /* 05FEB94 */
20 int fddi_sink_total_pkts = 0;
21 static /* 05FEB94 */
22 int fddi_sink_total_pkts_a[PRIORITIES + 1] = {0, 0, 0, 0, 0, 0, 0, 0};
23 static /* 05FEB94 */
24 double fddi_sink_total_bits = 0.0;
25 static /* 05FEB94 */
26 double fddi_sink_total_bits_a[PRIORITIES+1]={0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
27 static /* 05FEB94 */
28 double fddi_sink_peak_delay = 0.0;
29 static /* 05FEB94 */
29 double fddi_sink_peak_delay_a[PRIORITIES+2]={0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};

115
30 static /* 05FEB94 */
31 int fddi_sink_scalar_write = 0;
32 static /* 05FEB94 */
33 int pri_set = 20; /* 20JAN94 */
34 static
35 int subq_index = 0; /* 5APR94 */
36 static
37 int prev_src_addr[4] = {0, 1, 2, 3}; /* 25APR94 */
38 double buffer[4] = {0.0, 0.0, 0.0, 0.0}; /* 10MAY94 */

39 /* statistics used for CDL throughput */
40 static /* 20APR94 */
41 int fddil1_total_pkts = 0;
42 static
43 int fddil1_total_pkts_a[PRIORITIES + 1] = {0, 0, 0, 0, 0, 0, 0, 0};
44 static
45 double fddil1_total_bits = 0.0;
46 static
47 double fddil1_total_bits_a[PRIORITIES + 1] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};

48 /* Externally defined globals. */
49 extern double fddi_t_opr;

50 /*12JAN94: attributes from the Environment file */
51 double Offered_Load; /* 12JAN94 */
52 double Asynch_Offered_Load; /* 12JAN94 */

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

55 /* State variable definitions */
56 typedef struct
57 {
58 FSM_STS_STATE
59 Ghandle
60 Ghandle
61 Ghandle
62 Ghandle
63 Ghandle
64 Ghandle
65 Ghandle
66 Ghandle
67 Objid
68 } cp_fddi_sink_state;
69 #define pr_state_ptr ((cp_fddi_sink_state*) Sim1_Mod_State_Ptr)
70 #define thru_gshandle
71 #define m_delay_gshandle
72 #define ete_delay_gshandle
73 #define thru_gshandle_a
74 #define m_delay_gshandle_a
75 #define ete_delay_gshandle_a
76 #define t_gshandle
77 #define t_gshandle_a
78 #define my_id

79 /* Process model interrupt handling procedure */

80 void
81 cp_fddi_sink ()
82 {
83 double delay, create_time;
84 Packet* pkptr;
85 Packet* pkptr1; /*5APR94*/
86 int src_addr, my_addr;
87 int dest_addr; /*14APR94*/
88 Ici* from_mac_i pci_ptr;
89 double fddi_sink_ttrt;
90 int xmit_subq_index; /*5APR94*/
91 int load_balance_code; /*6APR94*/
92 int i, subq_no; /*25APR94*/
93 int index; /*1OMAY94*/

94 FSM_ENTER (cp_fddi_sink)

95 FSM_BLOCK_SWITCH
96 {
97 /*----------------------------------*/
98 /* FSM_STATE_ENTER_UNFORCED (0, state0_enter_exec, "DISCARD") */
99 {
100 /* determine type of interrupt:1OMAY94 */
101 switch (op_intrpt_type())
102 {
103 case OPC_INTRPT_STAT:
104 /* the interrupt is caused by the transmitters' status */
105 {
106 index = op_intrpt_stat();
107 switch(index)
108 {
109 case XMITTER_ONE:
110 {
111 buffer[0] = op_stat_local_read(XMITTER_ONE);
112 break;
113 }
114 case XMITTER_TWO:
115 {
116 buffer[1] = op_stat_local_read(XMITTER_TWO);
117 break;
118 }
119 case XMITTER_THREE:
120 {
121 buffer[2] = op_stat_local_read(XMITTER_THREE);
122 break;
123 }
124 case XMITTER_FOUR:
125 {
126 buffer[3] = op_stat_local_read(XMITTER_FOUR);
127 break;
128 }
129 default:
130 {
131 op_sim_end("*** FDDI-CDL : FATAL ERROR","Unexpected stat interrupt","");
132 }
133 break;
134 }
135 case OPC_INTRPT_STRM:
136 /* the interrupt is caused by the incoming packets */
137 {
138 pkptr = op_pk_get (op_intrpt_strm ());
139 from_mac_ici_ptr = op_intrpt_ici ();
140 /* 20JAN94: get the packet’s priority level, which */
141 /* will be used to index arrays of thruput and delay */
142 /* computations. */
143 /* pri_set = op_pk_priority_get (pkptr); doesn’t work here */
144 op_pk_nfd_get (pkptr, "pri", &pri_set); /* 29JAN94 */
145 /* determine the time of creation of the packet */
146 op_pk_nfd_get (pkptr, "cr_time", &creat_time);
147 /* 18APR94: determine the destination address of the packet */
148 op_pk_nfd_get (pkptr, "dest_addr", &dest_addr);
149 /* 20APR94: determine the source address of the packet */
150 op_pk_nfd_get (pkptr, "src_addr", &src_addr);
151 /* 7APR94: determine id of own processor to use in finding */
152 /* load balancing attribute and station address of the bridge node */
my_id = op_id_self();

/* 7APR94: determine which load balancing algorithm is in use */
op_iaa_obj_attr_get ( my_id, "load balancing algorithm", &load_balance_code );

/* 14APR94: also get my own address */
op_iaa_obj_attr_get ( my_id, "station_address", &my_addr);

/* destroy the packet */
op_pk_destroy (pkptr); /*
/* 03FEB94: rather, enqueue the packet. This will be the */
/* first step toward developing a LAN bridging structure. */
/* -Nix */
/* op_subq_pk_insert (pri_set, pkptr, OPC_QPOS_TAIL); */

/* 14APR94: check the frame passed to "llc" is destined for */
/* this station. If it is destroy the packet and update the local traffic */
/* statistics; if not, allocate the packets */
/* to the transmitters since they are destined for the remote lan */
/* update also incoming return link statistics for the frames */
/* which will be queued in llc_sink to be sent to remote lan. */
/* -Karayakalar */

if((dest_addr <= my_addr) & (src_addr < my_addr))
{
    /* 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); /* 20JAN-20APR94 */

    /* accumulate delays */
    delay = op_sim_time () - creat_time;
    fddi_sink_accu_delay += delay;
    fddi_sink_accu_delay_a[pri_set] += delay; /* 20JAN-20APR94 */

    /* 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 23JAN-20APR94 */
    if (delay > fddi_sink_peak_delay_a[pri_set])
        fddi_sink_peak_delay_a[pri_set] = delay;

    op_pk_destroy (pkptr);

    /* 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 asynch priorities 1->8, */
    /* respectively; [8] represents synchronous traffic, */
    /* 20JAN94 */
195 /* and [0] represents overall asynchronous traffic. -Nix */
196 if (fddi_sink_total_pkts % 25 == 0)
197 {
198  op_stat_global_write (thru_gshandle,
199       fddi_sink_total_bits / op_sim_time());
200  op_stat_global_write (thru_gshandle_a[0],
201       fddi_sink_total_bits_a[0] / op_sim_time());
202  op_stat_global_write (thru_gshandle_a[1],
203       fddi_sink_total_bits_a[1] / op_sim_time());
204  op_stat_global_write (thru_gshandle_a[2],
205       fddi_sink_total_bits_a[2] / op_sim_time());
206  op_stat_global_write (thru_gshandle_a[3],
207       fddi_sink_total_bits_a[3] / op_sim_time());
208  op_stat_global_write (thru_gshandle_a[4],
209       fddi_sink_total_bits_a[4] / op_sim_time());
210  op_stat_global_write (thru_gshandle_a[5],
211       fddi_sink_total_bits_a[5] / op_sim_time());
212  op_stat_global_write (thru_gshandle_a[6],
213       fddi_sink_total_bits_a[6] / op_sim_time());
214  op_stat_global_write (thru_gshandle_a[7],
215       fddi_sink_total_bits_a[7] / op_sim_time());
216  op_stat_global_write (thru_gshandle_a[8],
217       fddi_sink_total_bits_a[8] / op_sim_time());
218
219  /* 30JAN94: gather all async stats into one overall figure */
220  op_stat_global_write (thru_gshandle_a[0],
221       (fddi_sink_total_bits - fddi_sink_total_bits_a[8]) /
222       op_sim_time());
223
224  /* fddi_sink_total_bits_a[0] + fddi_sink_total_bits_a[1] + */
227  /* fddi_sink_total_bits_a[6] + fddi_sink_total_bits_a[7]) / */
228  /* op_sim_time(); */
229
229  op_stat_global_write (m_delay_gshandle,
230       fddi_sink_accum_delay / fddi_sink_total_pkts);
231  op_stat_global_write (m_delay_gshandle_a[0],
232       fddi_sink_accum_delay_a[0] / fddi_sink_total_pkts_a[0]);
233  op_stat_global_write (m_delay_gshandle_a[1],
234       fddi_sink_accum_delay_a[1] / fddi_sink_total_pkts_a[1]);
235  op_stat_global_write (m_delay_gshandle_a[2],
236       fddi_sink_accum_delay_a[2] / fddi_sink_total_pkts_a[2]);
237  op_stat_global_write (m_delay_gshandle_a[3],
238       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],
  fddi_sink_accum_delay_a[7] / fddi_sink_total_pkts_a[7]);
op_stat_global_write (m_delay_gshandle_a[8],
  fddi_sink_accum_delay_a[8] / fddi_sink_total_pkts_a[8]);

op_stat_global_write (m_delay_gshandle_a[9],
  (fddi_sink_accum_delay - fddi_sink_accum_delay_a[9]) /
  (fddi_sink_total_pkts - fddi_sink_total_pkts_a[9]));

op_stat_global_write (ete_delay_gshandle_a[pri_set], delay);
}
} /* end of if(dest_addr == my_addr) && (src_addr < my_addr) statement */

op_pk_destroy(pkptr);

/* 20APR94: check the frame passed to "llc" is destined for remote lan */
/* This will allow only the packets to be counted for CDL traffic. */
/* -Karayakaylar */
else
{
  /* also record actual delay values */
  op_stat_global_write (ete_delay_gshandle, delay);
  op_stat_global_write (ete_delay_gshandle_a[pri_set], delay);
}

/ * 20APR94: destroy the packets coming from the remote lan destined for this */
/ * station. These packets are not counted for local traffic. */
else if(dest_addr == my_addr)
  op_pk_destroy(pkptr);

/* add in its size */
fddipl_total_bits += op_pk_total_size_get (pkptr);
fddipl_total_bits_a[pri_set] += op_pk_total_size_get (pkptr); /* 20APR94 */

/* increment packet counter; 20APR94 */
fddipl_total_pkts++;
280 fddilpi_total_pkts_a[pri_set]++; 

281 /* if a multiple of 25 packets is reached, update stats */
282 /* [0]->[7] represent async priorities 1->8, */
283 /* respectively; [8] represents synchronous traffic, */
284 /* [9] represents overall asynchronous traffic. */
285 if (fddilpi_total_pkts % 25 == 0)
286 {
287  op_stat_global_write (t_gshandle,
288    fddilpi_total_bits / op_sim_time ());
289   op_stat_global_write (t_gshandle_a[pri_set],
290    fddilpi_total_bits_a[0] / op_sim_time());
291   op_stat_global_write (t_gshandle_a[0],
292    fddilpi_total_bits_a[1] / op_sim_time());
293   op_stat_global_write (t_gshandle_a[1],
294    fddilpi_total_bits_a[pri_set] / op_sim_time());
295   op_stat_global_write (t_gshandle_a[2],
296    fddilpi_total_bits_a[2] / op_sim_time());
297   op_stat_global_write (t_gshandle_a[3],
298    fddilpi_total_bits_a[3] / op_sim_time());
299   op_stat_global_write (t_gshandle_a[4],
300    fddilpi_total_bits_a[4] / op_sim_time());
301   op_stat_global_write (t_gshandle_a[5],
302    fddilpi_total_bits_a[5] / op_sim_time());
303   op_stat_global_write (t_gshandle_a[6],
304    fddilpi_total_bits_a[6] / op_sim_time());
305   op_stat_global_write (t_gshandle_a[7],
306    fddilpi_total_bits_a[7] / op_sim_time());
307   op_stat_global_write (t_gshandle_a[8],
308    fddilpi_total_bits_a[8] / op_sim_time());
309 /* gather all async stats into one overall figure */
310  op_stat_global_write (t_gshandle_a[9],
311    (fddilpi_total_bits - fddilpi_total_bits_a[8]) /
312    op_sim_time());
313 /* (fddilpi_total_bits_a[0] + fddilpi_total_bits_a[1]) */
317 /* op_sim_time()); */
318 }

319 /* 14APR94 : allocate the packets to llc_sink subqueues */
320 /* 6APR94 - Karayakalar*/
321 /* check if load balancing algorithm is circular */
322 /* zero(0) is the circular load balancing code */
323 if (load_balance_code == 0)
324 {
325    /* 5APR94 */
326    /* Apply load balancing to insert the packets in the */
327    /* subqueues, in a circular order */
328    i = subq_index % 4;
329    /* check if previous source address is allocated to this queue */
330    /* if so, allocate the packet to that subqueue so that consecutive packets */
331    /* coming from the same station follow the same channel */
332    /* otherwise, allocate the packet to the next queue for transmission */
333    if (prev_src_addr[i] == src_addr)
334    {
335       op_subq_pk_insert(i, pkptr, OPC_QPOS_TAIL);
336       prev_src_addr[i] = src_addr;
337    }
338    else
339    {
340       subq_index++;
341       subq_no = subq_index % 4;
342       op_subq_pk_insert(subq_no, pkptr, OPC_QPOS_TAIL);
343       prev_src_addr[subq_no] = src_addr;
344    }
345 }
346 /* 25APR94 */
347 /* check if load balancing algorithm is empty allocation */
348 /* one(1) is the empty allocation load balancing code */
349 if (load_balance_code == 1)
350 {
351    i = subq_index % 4;
352    /* check if previous source address is allocated to this queue */
353    /* if so, allocate the packet to that subqueue so that consecutive packets */
354    /* coming from the same station follow the same channel */
355    /* otherwise, allocate the packet to the next queue for transmission */
356    if (prev_src_addr[i] == src_addr)
357    {
358       op_subq_pk_insert(i, pkptr, OPC_QPOS_TAIL);
359       prev_src_addr[i] = src_addr;
360    }
361    else
362    {
363       subq_no = op_subq_index_map(OPC_QSEL_MAX_FREE_PKSIZE);
364       op_subq_pk_insert(subq_no, pkptr, OPC_QPOS_TAIL);
365       prev_src_addr[subq_no] = src_addr;
366       subq_index++;
367    }
368    }
369 }
369 /* send the packets to the transmitters */
370 xmit_subq_index = i;
371 /* check if this subqueue is empty and transmitter is not busy */
372 if (((op_subq_empty(xmit_subq_index))&& (buffer[xmit_subq_index] == 0.0)))
373 {
374  /* access the first packet in the subqueue */
375  pkptr1 = op_subq_pk_remove (xmit_subq_index, OPC_QPOS_HEAD);
376  /* forward it to the destination xmitter */
377  /* associated with the subqueue index */
378  op_pk_send (pkptr1, xmit_subq_index);
379 }
380 } /* if(dest_addr > my_addr) statement */
381 break;
382 } /* end of case OPC_INTRPT_STMT statement */
383 } /* end of switch */
384 
385 /** blocking after enter executives of unforced state. **/
386 FSH_EXIT (1, cp_fddi_sink)
387 */
388 FSH_STATE_EXIT_UNFORCED (0, state0_exit_exec, "DISCARD")
389 {
390 }
391 /** state (DISCARD) exit executives **/
392 FSH_INIT_COND (END_OF_SIM)
393 FSH_DFLT_COND
394 FSH_TEST_LOGIC ("DISCARD")
395 FSH_TRANSIT_SWITCH
396 {
397 FSH_CASE_TRANSIT (0, 1, state1_enter_exec, ;)
398 FSH_CASE_TRANSIT (1, 0, state0_enter_exec, ;)
399 }
400 } /*-----------------------------------*/
401 /** state (STATS) enter executives **/
402 FSH_STATE_ENTER_UNFORCED (1, state1_enter_exec, "STATS")
403 {
404 /* At end of simulation, scalar performance statistics */
405 /* and input parameters are written out. */
406 op_stat_scalar_write ("RL Throughput (bps), Priority 1", 
407 fddilp1_total_bits_a[0] / op_sim_time ();
408 /*20APR94*/
409 */
124
op_stat_scalar_write ("RL Throughput (bps), Priority 2", fddilpi_total_bits_a[1] / op_sim_time());

op_stat_scalar_write ("RL Throughput (bps), Priority 3", fddilpi_total_bits_a[2] / op_sim_time());

op_stat_scalar_write ("RL Throughput (bps), Priority 4", fddilpi_total_bits_a[3] / op_sim_time());

op_stat_scalar_write ("RL Throughput (bps), Priority 5", fddilpi_total_bits_a[4] / op_sim_time());

op_stat_scalar_write ("RL Throughput (bps), Priority 6", fddilpi_total_bits_a[5] / op_sim_time());

op_stat_scalar_write ("RL Throughput (bps), Priority 7", fddilpi_total_bits_a[6] / op_sim_time());

op_stat_scalar_write ("RL Throughput (bps), Priority 8", fddilpi_total_bits_a[7] / op_sim_time());

op_stat_scalar_write ("RL Throughput (bps), Asynchronous", (fddilpi_total_bits - fddilpi_total_bits_a[8]) / op_sim_time());

op_stat_scalar_write ("RL Throughput (bps), Synchronous", fddilpi_total_bits_a[8] / op_sim_time());

op_stat_scalar_write ("RL Throughput (bps), Total", fddilpi_total_bits / op_sim_time()); /*20APR94*/

/* 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-0 (sec.), Priority 1", fddi_sink_accum_delay_a[0] / fddi_sink_total_pkts_a[0]);
op_stat_scalar_write ("Mean End-to-End Delay-0 (sec.), Priority 2", fddi_sink_accum_delay_a[1] / fddi_sink_total_pkts_a[1]);

op_stat_scalar_write ("Mean End-to-End Delay-0 (sec.), Priority 3", fddi_sink_accum_delay_a[2] / fddi_sink_total_pkts_a[2]);

op_stat_scalar_write ("Mean End-to-End Delay-0 (sec.), Priority 4", fddi_sink_accum_delay_a[3] / fddi_sink_total_pkts_a[3]);

op_stat_scalar_write ("Mean End-to-End Delay-0 (sec.), Priority 5", fddi_sink_accum_delay_a[4] / fddi_sink_total_pkts_a[4]);

op_stat_scalar_write ("Mean End-to-End Delay-0 (sec.), Priority 6", fddi_sink_accum_delay_a[5] / fddi_sink_total_pkts_a[5]);

op_stat_scalar_write ("Mean End-to-End Delay-0 (sec.), Priority 7", fddi_sink_accum_delay_a[6] / fddi_sink_total_pkts_a[6]);

op_stat_scalar_write ("Mean End-to-End Delay-0 (sec.), Priority 8", fddi_sink Accum_delay_a[7] / fddi_sink_total_pkts_a[7]);

op_stat_scalar_write ("Mean End-to-End Delay-0 (sec.), Asynchronous", (fddi_sink_accum_delay - fddi_sink_accum_delay_a[8]) / (fddi_sink_total_pkts - fddi_sink_total_pkts_a[8]));

op_stat_scalar_write ("Mean End-to-End Delay-0 (sec.), Synchronous", fddi_sink_accum_delay_a[8] / fddi_sink_total_pkts_a[8]);

op_stat_scalar_write ("Mean End-to-End Delay-0 (sec.), Total", fddi_sink_accum_delay / fddi_sink_total_pkts);

op_stat_scalar_write ("Throughput-0 (bps), Priority 1", fddi_sink_total_bits_a[0] / op_sim_time ());

op_stat_scalar_write ("Throughput-0 (bps), Priority 2", fddi_sink_total_bits_a[1] / op_sim_time ());

op_stat_scalar_write ("Throughput-0 (bps), Priority 3", fddi_sink_total_bits_a[2] / op_sim_time ());

op_stat_scalar_write ("Throughput-0 (bps), Priority 4", fddi_sink_total_bits_a[3] / op_sim_time ());

op_stat_scalar_write ("Throughput-0 (bps), Priority 5", fddi_sink_total_bits_a[4] / op_sim_time ());

op_stat_scalar_write ("Throughput-0 (bps), Priority 6", fddi_sink_total_bits_a[5] / op_sim_time ());

op_stat_scalar_write ("Throughput-0 (bps), Priority 7", fddi_sink_total_bits_a[6] / op_sim_time ());

op_stat_scalar_write ("Throughput-0 (bps), Priority 8", fddi_sink_total_bits_a[7] / op_sim_time ());

op_stat_scalar_write ("Throughput-0 (bps), Asynchronous", (fddi_sink_total_bits - fddi_sink_total_bits_a[8]) / (fddi_sink_total_pkts - fddi_sink_total_pkts_a[8]));

op_stat_scalar_write ("Throughput-0 (bps), Synchronous", fddi_sink_total_bits_a[8] / fddi_sink_total_pkts_a[8]);

op_stat_scalar_write ("Throughput-0 (bps), Total", fddi_sink_total_bits / fddi_sink_total_pkts);

op_stat_scalar_write ("Throughput-0 (bps), Asynchronous", (fddi_sink_total_bits - fddi_sink_total_bits_a[8]) / (fddi_sink_total_pkts - fddi_sink_total_pkts_a[8]));

op_stat_scalar_write ("Throughput-0 (bps), Synchronous", fddi_sink_total_bits_a[8] / fddi_sink_total_pkts_a[8]);

op_stat_scalar_write ("Throughput-0 (bps), Total", fddi_sink_total_bits / fddi_sink_total_pkts);
476 \texttt{fddi\_sink\_total\_bits\_a[3] / op\_sim\_time (});}

477 \texttt{op\_stat\_scalar\_write ("Throughput-0 (bps), Priority 5",}

478 \texttt{fddi\_sink\_total\_bits\_a[4] / op\_sim\_time (});}

479 \texttt{op\_stat\_scalar\_write ("Throughput-0 (bps), Priority 6",}

480 \texttt{fddi\_sink\_total\_bits\_a[5] / op\_sim\_time (});}

481 \texttt{op\_stat\_scalar\_write ("Throughput-0 (bps), Priority 7",}

482 \texttt{fddi\_sink\_total\_bits\_a[6] / op\_sim\_time (});}

483 \texttt{op\_stat\_scalar\_write ("Throughput-0 (bps), Priority 8",}

484 \texttt{fddi\_sink\_total\_bits\_a[7] / op\_sim\_time (});}

485 \texttt{op\_stat\_scalar\_write ("Throughput-0 (bps), Asynchronous",}

486 \texttt{(fddi\_sink\_total\_bits - fddi\_sink\_total\_bits\_a[8]) / op\_sim\_time (});}

487 \texttt{/* (fddi\_sink\_total\_bits\_a[0] + fddi\_sink\_total\_bits\_a[1] + */

488 \texttt{/* fddi\_sink\_total\_bits\_a[2] + fddi\_sink\_total\_bits\_a[3] + */

489 \texttt{/* fddi\_sink\_total\_bits\_a[4] + fddi\_sink\_total\_bits\_a[5] + */

490 \texttt{/* fddi\_sink\_total\_bits\_a[6] + fddi\_sink\_total\_bits\_a[7]) / */

491 \texttt{/* op\_sim\_time ());}

492 \texttt{op\_stat\_scalar\_write ("Throughput-0 (bps), Synchronous",}

493 \texttt{fddi\_sink\_total\_bits\_a[8] / op\_sim\_time (});}

494 \texttt{op\_stat\_scalar\_write ("Throughput-0 (bps), Total",}

495 \texttt{fddi\_sink\_total\_bits / op\_sim\_time (});}

496 \texttt{op\_stat\_scalar\_write ("Peak End-to-End Delay-0 (sec.), Priority 1",}

497 \texttt{fddi\_sink\_peak\_delay\_a[0]);}

498 \texttt{op\_stat\_scalar\_write ("Peak End-to-End Delay-0 (sec.), Priority 2",}

499 \texttt{fddi\_sink\_peak\_delay\_a[1]);}

500 \texttt{op\_stat\_scalar\_write ("Peak End-to-End Delay-0 (sec.), Priority 3",}

501 \texttt{fddi\_sink\_peak\_delay\_a[2]);}

502 \texttt{op\_stat\_scalar\_write ("Peak End-to-End Delay-0 (sec.), Priority 4",}

503 \texttt{fddi\_sink\_peak\_delay\_a[3]);}

504 \texttt{op\_stat\_scalar\_write ("Peak End-to-End Delay-0 (sec.), Priority 5",}

505 \texttt{fddi\_sink\_peak\_delay\_a[4]);}

506 \texttt{op\_stat\_scalar\_write ("Peak End-to-End Delay-0 (sec.), Priority 6",}

507 \texttt{fddi\_sink\_peak\_delay\_a[5]);}

508 \texttt{op\_stat\_scalar\_write ("Peak End-to-End Delay-0 (sec.), Priority 7",}

127
fddi_sink_peak_delay_a[0]);

op_stat_scalar_write("Peak End-to-End Delay-O (sec.), Priority 8",
                    fddi_sink_peak_delay_a[7]);

op_stat_scalar_write("Peak End-to-End Delay-O (sec.), Synchronous",
                    fddi_sink_peak_delay_a[8]);

op_stat_scalar_write("Peak End-to-End Delay-O (sec.), Overall",
                    fddi_sink_peak_delay);

515 /* Write the TTRT value for ring 0. This preserves */
516 /* the old behavior for single-ring simulations. */
517 op_stat_scalar_write("TTRT (sec.) - Ring 0",
                    fddi_t_opr [0]);

519 /* 12JAN94: obtain offered load information from the Environment */
520 /* file; this will be used to provide abscissa information that */
521 /* can be plotted in the Analysis Editor (see "fddi_sink" STATS */
522 /* state. To the user: it's your job to keep these current in */
523 /* the Environment File. -Mrs
524 op_ima_sim_attr_get (OPCIMA_DOUBLE, "total_offered_load_0", &Offered_Load);
525 op_ima_sim_attr_get (OPCIMA_DOUBLE, "asynch_offered_load_0", &Asynch_Offered_Load);

526 /* 12JAN94: write the total offered load for this run */
527 op_stat_scalar_write("Total Offered Load-O (Nbps)",
                    Offered_Load);

529 op_stat_scalar_write("Asynchronous Offered Load-O (Mbps)",
                    Asynch_Offered_Load);

531 }
532 }

533 /** blocking after enter executives of unforced state. **/
534 FSM_EXIT (3,cp_fddi_sink)

535 /** state (STATS) exit executives **/
536 FSM_STATE_EXIT_UNFORCED (1, state1_exit_exec, "STATS")
537 {
538 }

539 /** state (STATS) transition processing **/
540 FSM_TRANSIT_MISSING ("STATS")
541 /e--------------------------------------------------*/
542 /* state (INIT) enter executives */
543 FN_STATE_ENTER_FORCED (2, state2_enter_exec, "INIT")
544 {
545 /* get the gshandles of the global statistic to be obtained */
546 /* 20JAN94: set array format */

547 thru_gshandle_a[0] = op_stat_global_reg ("pri 1 throughput-0 (bps)");
548 thru_gshandle_a[1] = op_stat_global_reg ("pri 2 throughput-0 (bps)");
549 thru_gshandle_a[2] = op_stat_global_reg ("pri 3 throughput-0 (bps)");
550 thru_gshandle_a[3] = op_stat_global_reg ("pri 4 throughput-0 (bps)");
551 thru_gshandle_a[4] = op_stat_global_reg ("pri 5 throughput-0 (bps)");
552 thru_gshandle_a[5] = op_stat_global_reg ("pri 6 throughput-0 (bps)");
553 thru_gshandle_a[6] = op_stat_global_reg ("pri 7 throughput-0 (bps)");
554 thru_gshandle_a[7] = op_stat_global_reg ("pri 8 throughput-0 (bps)");
555 thru_gshandle_a[8] = op_stat_global_reg ("synch throughput-0 (bps)");
556 thru_gshandle_a[9] = op_stat_global_reg ("async throughput-0 (bps)");
557 thru_gshandle = op_stat_global_reg ("total throughput-0 (bps)");

558 m_delay_gshandle_a[0] = op_stat_global_reg ("pri 1 mean delay-0 (sec.)");
559 m_delay_gshandle_a[1] = op_stat_global_reg ("pri 2 mean delay-0 (sec.)");
560 m_delay_gshandle_a[2] = op_stat_global_reg ("pri 3 mean delay-0 (sec.)");
561 m_delay_gshandle_a[3] = op_stat_global_reg ("pri 4 mean delay-0 (sec.)");
562 m_delay_gshandle_a[4] = op_stat_global_reg ("pri 5 mean delay-0 (sec.)");
563 m_delay_gshandle_a[5] = op_stat_global_reg ("pri 6 mean delay-0 (sec.)");
564 m_delay_gshandle_a[6] = op_stat_global_reg ("pri 7 mean delay-0 (sec.)");
565 m_delay_gshandle_a[7] = op_stat_global_reg ("pri 8 mean delay-0 (sec.)");
566 m_delay_gshandle_a[8] = op_stat_global_reg ("synch mean delay-0 (sec.)");
567 m_delay_gshandle_a[9] = op_stat_global_reg ("async mean delay-0 (sec.)");
568 m_delay_gshandle = op_stat_global_reg ("total mean delay-0 (sec.)");

569 ete_delay_gshandle_a[0] = op_stat_global_reg ("pri 1 end-to-end delay-0 (sec.)");
570 ete_delay_gshandle_a[1] = op_stat_global_reg ("pri 2 end-to-end delay-0 (sec.)");
571 ete_delay_gshandle_a[2] = op_stat_global_reg ("pri 3 end-to-end delay-0 (sec.)");
572 ete_delay_gshandle_a[3] = op_stat_global_reg ("pri 4 end-to-end delay-0 (sec.)");
573 ete_delay_gshandle_a[4] = op_stat_global_reg ("pri 5 end-to-end delay-0 (sec.)");
574 ete_delay_gshandle_a[5] = op_stat_global_reg ("pri 6 end-to-end delay-0 (sec.)");
575 ete_delay_gshandle_a[6] = op_stat_global_reg ("pri 7 end-to-end delay-0 (sec.)");
576 ete_delay_gshandle_a[7] = op_stat_global_reg ("pri 8 end-to-end delay-0 (sec.)");
577 ete_delay_gshandle_a[8] = op_stat_global_reg ("synch end-to-end delay-0 (sec.)");
578 ete_delay_gshandle = op_stat_global_reg ("total end-to-end delay-0 (sec.)");

579 t_gshandle_a[0] = op_stat_global_reg ("pri 1 RL throughput (bps)"); /*20APR94*/
580 t_gshandle_a[1] = op_stat_global_reg ("pri 2 RL throughput (bps)");
581 t_gshandle_a[2] = op_stat_global_reg ("pri 3 RL throughput (bps)");
582 t_gshandle_a[3] = op_stat_global_reg ("pri 4 RL throughput (bps)");
583 t_gshandle_a[4] = op_stat_global_reg ("pri 5 RL throughput (bps)");
584 t_gshandle_a[5] = op_stat_global_reg ("pri 6 RL throughput (bps)");
585 t_gshandle_a[6] = op_stat_global_reg ("pri 7 RL throughput (bps)");
586 t_gshandle_a[7] = op_stat_global_reg ("pri 8 RL throughput (bps)");
587 t_gshandle_a[8] = op_stat_global_reg ("synch RL throughput (bps)");

129
588 t_gchandle_a[0] = op_stat_global_reg ("async RL throughput (bps)");
589 t_gchandle = op_stat_global_reg ("total RL throughput (bps)");

590 }

591 /** state (INIT) exit executives **/
592 FS_M_STATE_EXIT_FORCED (2, state2_exit_exec, "INIT")
593 {
594
595 /** state (INIT) transition processing **/
596 FS_M_INIT_COND (END_OF_SIM)
597 FS_M_DFLT_COND
598 FS_M_TEST_LOGIC ("INIT")
599 FS_M_TRANSIT_SWITCH
600 {
601 FS_M_CASE_TRANSIT (0, 1, state1_enter_exec, ;)
602 FS_M_CASE_TRANSIT (1, 0, state0_enter_exec, ;)
603 }
604 /***********************************************************/
605 }

606 FS_M_EXIT (2, cp_fddi_sink)
607 }

608 void
cp_fddi_sink_svar (pr_s_ptr, var_name, var_p_ptr)
609 cp_fddi_sink_state *pr_s_ptr;
610 char *var_name, **var_p_ptr;
611 {
612
613 FIN (cp_fddi_sink_svar (pr_s_ptr))
614 *var_p_ptr = VOS_NIL;
615 if (Vos_String_Equal ("thru_gchandle", var_name))
616 *var_p_ptr = (char *) (pr_s_ptr->sv_thru_gchandle);
617 if (Vos_String_Equal ("m_delay_gchandle", var_name))
618 *var_p_ptr = (char *) (pr_s_ptr->sv_m_delay_gchandle);
619 if (Vos_String_Equal ("ste_delay_gchandle", var_name))
if (Vos_String_Equal("thru_gshandle_a", var_name))
   *var_p_ptr = (char *) (pkt_p->sv_thru_gshandle_a);
if (Vos_String_Equal("m_delay_gshandle_a", var_name))
   *var_p_ptr = (char *) (pkt_p->sv_m_delay_gshandle_a);
if (Vos_String_Equal("ete_delay_gshandle_a", var_name))
   *var_p_ptr = (char *) (pkt_p->sv_ete_delay_gshandle_a);
if (Vos_String_Equal("t_gshandle", var_name))
   *var_p_ptr = (char *) (pkt_p->sv_t_gshandle);
if (Vos_String_Equal("t_gshandle_a", var_name))
   *var_p_ptr = (char *) (pkt_p->sv_t_gshandle_a);
if (Vos_String_Equal("my_id", var_name))
   *var_p_ptr = (char *) (pkt_p->sv_my_id);

FOUT;
}

void cp_fddi_sink_diag()
{
   double delay, creat_time;
   Packet* pktpr;
   Packet* pktpr1; /*5APR94*/
   int src_addr, my_addr;
   int dest_addr; /*1APR94*/
   Int from_mac_ici_ptr;
   double fddi_sink_ttrt;
   int xmit_subq_index; /*5APR94*/
   int load_balance_code; /*5APR94*/
   int i_subq_no; /*25APR94*/
   int index; /*10JAY94*/

   FIN (cp_fddi_sink_diag())
}

void cp_fddi_sink_terminate()
{
   double delay, creat_time;
   Packet* pktpr;
   Packet* pktpr1; /*5APR94*/
int src_addr, my_addr;
int dest_addr; /* 14APR94 */
int icmp from_mac_icl_ptr;
double fddi_sink_ptr;
int xmit_subq_index; /* 14APR94 */
int load_balance_code; /* 84APR94 */
int i, subq_no; /* 25APR94 */
int index; /* 10MAY94 */

FIN (cp_fddi_sink_terminate())

FOUT;
}

Compcode

cp_fddi_sink_init (pr_state_pptr)
cp_fddi_sink_state *pr_state_pptr;
{
static Vos_T_Catm_Type obtype = OPC_NIL;

FIN (cp_fddi_sink_init (pr_state_pptr))

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

if ((*pr_state_pptr = (cp_fddi_sink_state*) Vos_Catmem_Alloc (obtype, 1)) == OPC_NIL)
FRET (OPC_COMPCODE_FAILURE)
else
{
(*pr_state_pptr)->current_block = 4;
FRET (OPC_COMPCODE_SUCCESS)
}

132
APPENDIX D
SPNI SOURCE "C" CODE
"sp_fddi_gen.pr.c"

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

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

2 /* OPNET system definitions */
3 #include <opnet.h>
4 #include "sp_fddi_gen.pr.h"
5 FSN_EXT_DECLS

6 /* Header block */
7 #define MAC_LAYER_OUT_STREAM 0
8 #define LLC_SINK_OUT_STREAM 1 /*18APR94*/

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_TOKEN_NONRESTRICTED 0
14 #define FDDI_TOKEN_RESTRICTED 1

15 /* State variable definitions */
16 typedef struct
17 {
18   FSN_SYS_STATE
19   Distribution*      sv_inter_dist_ptr;
20   Distribution*      sv_len_dist_ptr;
21   Distribution*      sv_dest_dist_ptr;
22   Distribution*      sv_pkt_priority_ptr;
23   Objid              sv_mac_objid;
24   Objid              sv_my_id;
25   int                sv_low_dest_addr;
26   int                sv_high_dest_addr;

133
27 int sv_station_addr;
28 int sv_src_addr;
29 int sv_low_pkt_priority;
30 int sv_high_pkt_priority;
31 double sv_arrival_rate;
32 double sv_mean_pk_len;
33 double sv_async_mix;
34 int sv_mac_iciptr;
35 int sv_mac_icipt1;
36 int sv_llc_ici_ptr;
37 Packet* sv_pkptr1;
38 }
39 #define pr_state_ptr 
40 #define inter_dist_ptr pr_state_ptr->sv_inter_dist_ptr
41 #define len_dist_ptr pr_state_ptr->sv_len_dist_ptr
42 #define dest_dist_ptr pr_state_ptr->sv_dest_dist_ptr
43 #define pkt_priority_ptr pr_state_ptr->sv_pkt_priority_ptr
44 #define mac_objid pr_state_ptr->sv_mac_objid
45 #define my_id pr_state_ptr->sv_my_id
46 #define low_dest_addr pr_state_ptr->sv_low_dest_addr
47 #define high_dest_addr pr_state_ptr->sv_high_dest_addr
48 #define station_addr pr_state_ptr->sv_station_addr
49 #define src_addr pr_state_ptr->sv_src_addr
50 #define low_pkt_priority pr_state_ptr->sv_low_pkt_priority
51 #define high_pkt_priority pr_state_ptr->sv_high_pkt_priority
52 #define arrival_rate pr_state_ptr->sv_arrival_rate
53 #define mean_pk_len pr_state_ptr->sv_mean_pk_len
54 #define async_mix pr_state_ptr->sv_async_mix
55 #define mac_iciptr pr_state_ptr->sv_mac_iciptr
56 #define mac_icipt1 pr_state_ptr->sv_mac_icipt1
57 #define llc_ici_ptr pr_state_ptr->sv_llc_ici_ptr
58 #define pkptrl
59 /* Process model interrupt handling procedure */
60 void sp_fddi_gen ()
61 {
62 Packet* pkptr;
63 int phlen;
64 int dest_addr;
65 int i, restricted;
66 int pkt_prio;
68 FSM_ENTER (sp_fddi_gen)

69 FSM_BLOCK_SWITCH
70 {
71 /*-----------------------------------------*/
72 /* state (INIT) enter executives */
73 FSM_STATE_ENTER_UNFORCED (0, state0_enter_exec, "INIT")
74 {
75 /* determine id of own processor to use in finding attr */
76 my_id = op_id_self();
77 /* determine address range for uniform destination assignment */
78 op_ima_obj_attr_get (my_id, "low dest address", &low_dest_addr);
79 op_ima_obj_attr_get (my_id, "high dest address", &high_dest_addr);
80 /* determine object id of connected 'mac' layer process */
81 mac_objid = op_topo_assoc (my_id, OPC_TOPO_ASSOC_OUT, 
82 OPC_OBJINTYPE_MODULE, MAC_LAYER_OUT_STREAM);
83 /* determine the address assigned to it */
84 /* which is also the address of this station */
85 op_ima_obj_attr_get (mac_objid, "station_address", &station_addr);

86 /* set up a distribution for generation of addresses */
87 dest_dist_ptr = op_dist_load ("uniform_int", low_dest_addr, 
88                        high_dest_addr);

89 /* added 26DEC93 */
90 /* determine priority range for uniform traffic generation */
91 op_ima_obj_attr_get (my_id, "high pkt priority", &high_pkt_priority);
92 op_ima_obj_attr_get (my_id, "low pkt priority", &low_pkt_priority);
93 /* set up a distribution for generation of priorities */
94 pkt_priority_ptr = op_dist_load ("uniform_int", low_pkt_priority, high_pkt_priority);
95 /* above added 26DEC93 */

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

98 /* determine the mix of asynchronous and synchronous */
99 /* traffic. This is expressed as the proportion of */
100 /* asynchronous traffic. i.e a value of 1.0 indicates */
101 /* that all the produced traffic shall be asynchronous. */
102 op_ima_obj_attr_get (my_id, "async_mix", &async_mix);
103 /* set up a distribution for arrival generations */
104 if (arrival_rate != 0.0)
arrivals are exponentially distributed, with given mean e/

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

/* determine the distribution for packet size e/

op_ima_obj_attr_get (my_id, "mean pk length", &mean_pk_len);

/* set up corresponding distribution e/

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

/* designate the time of first arrival e/

fddi_gen_schedule ();

/* set up an interface control information (ICI) structure e/

to communicate parameters to the mac layer process e/
as a state variable than to allocate one on each packet xfer) e/

mac_ici = op_ici_create ("fddi_mac_req");

} 

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

/** 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, state1_enter_exec, "ARRIVAL")

This station should receive frames from the other lan as long as e/
there are frames in the input streams addressed to this lan e/
check if the interrupt type is stream interrupt e/
if(op_intrpt_type() == OPC_INTRPT_STRM)

if it is, get the packet in the input stream causing interrupt e/
pkptr = op_pk_get(op_intrpt_strm());

get the destination address of the frame : 16APR94 e/
op_pk_nfd_get(pkptr, "dest_addr", &dest_addr);
142 /* check if this frame is for the remote bridge station (bridge in surface lan) */
143 if(dest_addr == station_addr)
144 /* if it is, send the packet to LLC_sink directly */
145 /* in order to prevent overhead of MAC access */
146 op_pk_send(pkptr1, LLC_SINK_OUT_STREAM); /*19APR94 */
147 else
148 /* this packet is to send to MAC */
149 {
150 /* determine the source address of the frame */
151 op_pk_nfd_get(pkptr1, "src_addr", &src_addr);
152 /* set up an ICI structure to communicate parameters to */
153 /* MAC layer process */
154 mac_iciptr1 = op_ili_create("fddi_mac_req");
155 /* place the original source address into the ICI */ /*16APR94 */
156 /* "fddi_mac_req" is modified so that it contains the original */
157 /* source address from the local lan(collection platform) */
158 op_ici_attr_set(mac_iciptr1, "src_addr", src_addr);
159 /* place the destination address into the ICI */ /*12APR94 */
160 op_ici_attr_set(mac_iciptr1, "dest_addr", dest_addr);
161 /* assign the service class and requested token class */
162 /* At this moment the frames coming from the remote lan are assumed */
163 /* to have the same priority as synchronous frames in order not to accumulate */
164 /* packets on the bridge station mac and instead to deliver their destinations */
165 /* as soon as possible */
166 op_pk_nfd_set(pkptr1, "pri", 8);
167 op_ici_attr_set(mac_iciptr1, "svc_class", FDDI_SVC_SYNC);
168 op_ici_attr_set(mac_iciptr1, "pri", 8);
169 op_ici_attr_set(mac_iciptr1, "tk_class", FDDI_TK_NONRESTRICTED);
170 /* send the packet coupled with the ICI */
171 op_ici_install(mac_iciptr1);
172 op_pk_send(pkptr1, MAC_LAYER_OUT_STREAM);
173 }
174 }
175 /* otherwise, generate the frame */
176 else
177 {
178 /* determine the length of the packet to be generated */
179 pklen = op_dist_outcome(len_dist_ptr);

180 /* determine the destination */
181 /* don't allow this station's address as a possible outcome */
182 gen_packet:
183 dest_addr = op_dist_outcome(dest_dist_ptr);
184 if (dest_addr != -1 && dest_addr == station_addr)
185 goto gen_packet;

186 /* 26DEC94 & 29JAN94: determine its priority */
187 pkt_prio = op_dist_outcome(pkt_priority_ptr);

188 /* create a packet to send to MAC */
189 pkptr = op_pk_create_fmt ("fddi_llc_fmt");
190 /* assign its overall size */
191 op_pk_total_size_set (pkptr, pklen);
192
193 /* assign the time of creation */
194 op_pk_nfd_set (pkptr, "cr_time", op_sim_time ());
195
196 /* place the destination address into the ICI */
197 / (the protocol_type field will default) */
198 op_ici_attr_set (mac_iciptr, "dest_addr", dest_addr);
199
200 /* place the source address into the ICI */
201 op_ici_attr_set (mac_iciptr, "src_addr", station_addr);
202
203 assign the priority, and requested token class */
204 /* also assign the service class; 29JAN94: the fddi_llc_tr */
205 /* format is modified to include a "pri" field. */
206 if (op_dist_uniform (1.0) <= async_mix)
207 { 208
209   op_pk_nfd_set (pkptr, "pri", pkt_prio); /* 29JAN94 */
210   op_ici_attr_set (mac_iciptr, "svc_class", FDDI_SVC_ASYNC);
211   op_ici_attr_set (mac_iciptr, "pri", pkt_prio); /* 29JAN94 */
212
213 }
214
215 /* Request only nonrestricted tokens after transmission */
216 op_ici_attr_set (mac_iciptr, "tk_class", FDDI_TK_NONRESTRICTED);
217
218 /* Having determined priority, assign it; 26DEC93 */
219 op_ici_attr_set (mac_iciptr, "pri", pkt_prio);
220
221 /* send the packet coupled with the ICI */
222 op_ici_install (mac_iciptr);
223 /* check if destination address is in the local lan(collection platform) */
224 if (dest_addr <= 9)
225 { 226 /* if it is, this packet is to send llc_sink directly */
227   op_pk_send (pkptr, LLC_SINK_OUT_STREAM); /* 18APR94 */
228   if (dest_addr <= 9)
229   { 230   /* if not, the packet is destined for remote lan (surface stations) */
231     op_pk_send (pkptr, MAC_LAYER_OUT_STREAM);
232     fddi_gem_schedule ();
233 } 234 } 235

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

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

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

void
sp_fddi_gen_svar (prs_ptr, var_name, var_p_ptr)
sp_fddi_gen_state *prs_ptr;
char *var_name, **var_p_ptr;
{
FIN (sp_fddi_gen_svar (prs_ptr))

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))
268 *var_p_ptr = (char *) (&prs_ptr->sv_station_addr);
269 if (Vos_String_Equal ("src_addr", var_name))
270 *var_p_ptr = (char *) (&prs_ptr->sv_src_addr);
271 if (Vos_String_Equal ("low_pkt_priority", var_name))
272 *var_p_ptr = (char *) (&prs_ptr->sv_low_pkt_priority);
273 if (Vos_String_Equal ("high_pkt_priority", var_name))
274 *var_p_ptr = (char *) (&prs_ptr->sv_high_pkt_priority);
275 if (Vos_String_Equal ("arrival_rate", var_name))
276 *var_p_ptr = (char *) (&prs_ptr->sv_arrival_rate);
277 if (Vos_String_Equal ("mean_pk_len", var_name))
278 *var_p_ptr = (char *) (&prs_ptr->sv_mean_pk_len);
279 if (Vos_String_Equal ("async_mix", var_name))
280 *var_p_ptr = (char *) (&prs_ptr->sv_async_mix);
281 if (Vos_String_Equal ("mac_iciptr", var_name))
282 *var_p_ptr = (char *) (&prs_ptr->sv_mac_iciptr);
283 if (Vos_String_Equal ("mac_iciptri", var_name))
284 *var_p_ptr = (char *) (&prs_ptr->sv_mac_iciptri);
285 if (Vos_String_Equal ("llc_ici_ptr", var_name))
286 *var_p_ptr = (char *) (&prs_ptr->sv_llc_ici_ptr);
287 if (Vos_String_Equal ("pkptr1", var_name))
288 *var_p_ptr = (char *) (&prs_ptr->sv_pkptr1);

289 FOUT;
290 }
291
292 void sp_fddi_gen_diag ()
293 {
294 Packet *pkptr;
295 int pklen;
296 int dest_addr;
297 int i, restricted;
298 int pkt_prio;
299 FIN (sp_fddi_gen_diag ());

300 FOUT;
301 }

302 void sp_fddi_gen_terminate ()
303 {
304 Packet *pkptr;
305 int pklen;
int dest_addr;
int i, restricted;
int pkt_prio;

FIN (sp_fddi_gen_terminate ())

FOUT;
}

Compcode
sp_fddi_gen_init (pr_state_pptr)
sp_fddi_gen_state **pr_state_pptr;
{
static Vos_Cmd_Obtype obtype = OPC_NIL;

FIN (sp_fddi_gen_init (pr_state_pptr))

if (obtype == OPC_NIL)
{
if (Vos_Cmd_Register ("proc state vars (sp_fddi_gen)",
sizeof (sp_fddi_gen_state), Vos_Cmd, &obtype) == VOSC_FAILURE)
FRET (OPC_COMP_CODE_FAILURE)
}

if ((*pr_state_pptr = (sp_fddi_gen_state*) Vos_Cmd_Alloc (obtype, 1)) == OPC_NIL)
FRET (OPC_COMP_CODE_FAILURE)
else
{
(*pr_state_pptr)->current_block = 0;
FRET (OPC_COMP_CODE_SUCCESS)
}

/* static added 2DECG93, on advice from NILS */
static
fddi_gen_schedule ()
{
double inter_time;

/* obtain an interarrival period according to the */
/* prescribed distribution */
inter_time = op_dist_outcome (inter_dist_ptr);

/* schedule the arrival of next generated packet */
op_intrpt_schedule_self (op_sim_time () + inter_time, 0);
APPENDIX E
SPNI MAC "C" CODE EXCERPT
“sp_fddi_mac.pr.c”

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

```c
1 /** state (FR_REPEAT) enter executives **/
2 FSM_STATE_ENTER_FORCED (6, state6_enter_exec, "FR_REPEAT")
3 {
4 /* Extract the destination address of the frame. */
5 op_pk_nfd_get (pkptr, "dest_addr", &dest_addr);

6 /* If the frame is for this station, make a copy */
7 /* of the frame's data field and forward it to */
8 /* the higher layer. */
9 /* 14APR94: In order to send the frames which are */
10 /* addressed to the remote lan, check the address database */
11 /* of remote lan. Frames addressed to the remote lan shouldn't */
12 /* be repeated in the local ring -- This is a simple forwarding */
13 /* decision algorithm, one of the bridge's function */
14 /* - Karayakaylar */
15 if((dest_addr == my_address) || (dest_addr <= 9))
16 {
17 /* record total size of the frame (including data) */
18 pk_len = op_pk_total_size_get (pkptr);

19 /* decapsulate the data contents of the frame */
20 /* 29JAN94: a new field, "pri", has been added to */
21 /* the fddi_llc_fr packet format in the Parameters */
22 /* Editor, so that output statistics can be */
23 /* generated by class and priority. -Mit */
24 op_pk_nfd_get (pkptr, "info", &data_pkptr);
25 op_pk_nfd_get (pkptr, "pri", &pri_level);

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

31 /* Because, as noted in the FR_RCV state, only the */
32 /* frame's leading edge has arrived at this time, the */
33 /* complete frame can only be delivered to the higher */
34 /* layer after the frame's transmission delay has elapsed. */
35 /* (since decapsulation of the frame data contents has occurred, */
36 /* the original MAC frame length is used to calculate delay) */
37 tx_time = (double) pk_len / FDDI_TX_RATE;
38 op_pk_send_delayed (data_pkptr, FDDI_LLC_STRM_OUT, tx_time);

39 /* Note that the standard specifies that the original */
40 /* frame should be passed along until the originating station */
41 /* receives it, at which point it is stripped from the ring. */
42 /* However, in the simulation model, there is no interest */
43 /* in letting the frame continue past its destination unless */
44 /* group addresses are used, so that the same frame could be */
45 /* destined for several stations. Here the frame is stripped */
46 /* for efficiency as it reaches the destination; if the model */
47 /* is modified to include group addresses, this should be changed */
48 /* so that the frame is copied and the original repeated. */
49 /* Logic is already present for stripping the frame at the origin. */
50 op_pk_destroy (pkptr);
51 }
52 /* 14APR94: the frames belong to this ring should be repeated. */
53 /* Thus, local traffic is constrained. -- This is filtering decision */
54 /* One of the bridge's function -- Karayakaylar */
55 else{
56 /* Repeat the original frame on the ring and account for */
57 /* the latency through the station and the propagation delay */
58 /* for a single hop. */
59 /* (Only the originating station can strip the frame). */
60 op_pk_send_delayed (pkptr, FDDI_PHY_STRM_OUT,
61 Fddi_Stat_Latency + Fddi_Prop_Delay);
62 }
63 }

64 /** state (FR_REPEAT) exit executives */
65 FSM_STATE_EXIT_FORCED (6, state6_exit_exec, "FR_REPEAT")
66 {
67 }
68 }
APPENDIX F
SPNI SINK “C” CODE
“sp_fdii_sink.pr.c”

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

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

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

7 /* Header block */
8 /* Globals */
9 /* positions 0-7 represent the asynch priority levels, PRIORITIES + 1 */
10 /* represents synch traffic, and grand totals are as given in the original. */

11 #define PRIORITIES 8 /* 20JAN94 */
12 #define XMITTER_BUSY 0 /*10MAY94 */

13 static /* 05FEB94 */
14 double fdii2_sink_accum_delay = 0.0;
15 static /* 05FEB94 */
16 double fdii2_sink_accum_delay_a[PRIORITIES + 1] = {0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};
17 static /* 05FEB94 */
18 int fdii2_sink_total_pkts_a = 0;
19 static /* 05FEB94 */
20 int fdii2_sink_total_pkts_a[PRIORITIES + 1] = {0, 0, 0, 0, 0, 0, 0, 0};
21 static /* 05FEB94 */
22 double fdii2_sink_total_bits = 0.0;
23 static /* 05FEB94 */
24 double fdii2_sink_total_bits_a[PRIORITIES + 1] = {0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};
25 static /* 05FEB94 */
26 double fdii2_sink_peak_delay = 0.0;
27 static /* 05FEB94 */
28 double fdii2_sink_peak_delay_a[PRIORITIES + 2] = {0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};
29 static /* 05FEB94 */
30 int fdii2_sink_scalar_write = 0;
31 static /* 05FEB94 */
32 int pri2_set = 20; /* 20JAN94 */

145
double busy = 0.0; /* 10MAY94 */

/* Statistics used for command link:21APR94 */
static int fddilp2_total_pkts = 0;
static int fddilp2_total_pkts_a[PRIORITIES + 1] = {0, 0, 0, 0, 0, 0, 0, 0};
static double fddilp2_total_bits = 0.0;
double fddilp2_total_bits_a[PRIORITIES + 1] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};

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

/*12JAN94:attributes from the Environment file */
do...ffered_Load; /* 12JAN94 */
Async_Offered_Load; /* 12JAN94 */

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

/* State variable definitions */
typedef struct 
{
    FSN_STS_STATE
    Ghandle sv_thru2_ghandle;
    Ghandle sv_m2_delay_ghandle;
    Ghandle sv_ete2_delay_ghandle;
    Ghandle sv_thru2_ghandle_a[10];
    Ghandle sv_m2_delay_ghandle_a[10];
    Ghandle sv_ete2_delay_ghandle_a[10];
    Ghandle sv_t2_ghandle;
    Ghandle sv_t2_ghandle_a[10];
    Objid sv_my_id;
} ap_fddi_sink_state;

#define pr_state_ptr
#define thru2_ghandle ((ap_fddi_sink_state) SimI_Mod_State_Ptr)
#define m2_delay_ghandle pr_state_ptr->sv_thru2_ghandle
#define ete2_delay_ghandle pr_state_ptr->sv_m2_delay_ghandle
#define thru2_ghandle_a pr_state_ptr->sv_ete2_delay_ghandle
#define m2_delay_ghandle_a pr_state_ptr->sv_thru2_ghandle_a
#define ete2_delay_ghandle_a pr_state_ptr->sv_m2_delay_ghandle_a
#define t2_ghandle pr_state_ptr->sv_ete2_delay_ghandle_a
#define t2_ghandle_a pr_state_ptr->sv_thru2_ghandle_a
#define my_id pr_state_ptr->sv_my_id

146
74 /* Process model interrupt handling procedure */

75 void
76 sp_fddi_sink ()
77 {
78 double delay, creat_time;
79 Packet* pkptr;
80 Packet* pkptr1 ; /*5APR94*/
81 int src_addr, my_addr;
82 int dest_addr; /*14APR94*/
83 Ici* from_mac_ici_ptr;
84 double fddi_sink_ttrt;

85 FSN_ENTER (sp_fddi_sink)
86 FSN_BLOCK_SWITCH
87 {
88 /*----------------------------------------*/
89 /* state (DISCARD) enter executives */
90 FSN_STATE_ENTER_UNFORCED (0, state0_enter_exec, "DISCARD")
91 {
92 /* determine the type of interrupt */
93 switch(op_intrpt_type ())
94 {
95 /* check if transmitter is busy */
96 case OPC_INTRPT_STAT:
97 {
98 busy = op_stat_local_read (EMITTER_BUSY);
99 break;
100 }
101 /* check if a packet has arrived */
102 case OPC_INTRPT_STRM:
103 {
104 /* get the packet and the interface control info */
105 pkptr = op_pk_get (op_intrpt_strm ());
106 from_mac_ici_ptr = op_intrpt_ici ();

107 /* 20JAN94: get the packet's priority level, which */
108 /* will be used to index arrays of throughput and delay */
109 /* computations. */
110 /* pri2_set = op_pk_priority_get (pkptr); doesn't work here */
111 op_pk_nfd_get (pkptr, "pri", &pri2_set); /* 20JAN94 */

112 /* determine the time of creation of the packet */

147
113 op_pk_mfd_get (pkptr, "cr_time", &create_time);

114 /* determine the dest address of the packet */ 18APR94
115 op_pk_mfd_get (pkptr, "dest_addr", &dest_addr);

116 /* 7APR94: determine id of own processor to use in finding */
117 /* station address of the bridge node */
118 my_id = op_id_self();

119 /* 14APR94: also get my own address */
120 op_id_obj_attr_get ( my_id, "station_address", &my_addr);

121 /* destroy the packet */
122 /* op_pk_destroy (pkptr); */
123 /* 03FEB94: rather, enqueue the packet. This will be the */
124 /* first step toward developing a LAN bridging structure. */
125 /* -Nix */
126 /* op_subq_pk_insert (pri_set, pkptr, OPC_QPOS_TAIL); */

127 /* 14APR94: check the frame passed to "llc" is destined for */
128 /* this station. If it is destroy the packet; if not, allocate the packets */
129 /* to the command link transmitter since they are destined for the remote lan */
130 /* -Karayaklar */
131 /* determine the packets coming from surface stations, this will */
132 /* be counted for local traffic */
133 /* 9(nine) is model specific, this is the "station_number" of */
134 /* collection platform bridge station */
135 if((dest_addr == my_addr)&&(src_addr > 9))
136 {
137 /* add in its size */
138 fddi2_sink_total_bits += op_pk_total_size_get (pkptr);
139 fddi2_sink_total_bits_a[pri2_set] += op_pk_total_size_get (pkptr); /* 20JAN-20APR94 */

140 /* accumulate delays */
141 delay = op_sim_time () - create_time;
142 fddi2_sink_accum_delay += delay;
143 fddi2_sink_accum_delay_a[pri2_set] += delay; /* 20JAN-20APR94 */

144 /* keep track of peak delay value */
145 if (delay > fddi2_sink_peak_delay)
146 fddi2_sink_peak_delay = delay;

147 /* 20JAN94: keep track by priority levels as well 23JAN-20APR94 */
148 if (delay > fddi2_sink_peak_delay_a[pri2_set])
149 fddi2_sink_peak_delay_a[pri2_set] = delay;

150 op_pk_destroy (pkptr);

151 /* increment packet counter; 20JAN94 */
152 fddi2_sink_total_pkts++;
153 fddi2_sink_total_pkts_a[pri2_set]++;

148
if a multiple of 25 packets is reached, update states

/03FEB94: [0]->[7] represent async priorities 1->8, *
/ respectively; [8] represents synchronous traffic, *
/ and [9] represents overall asynchronous traffic. Mix *
if (fddi2_sink_total_pkts % 25 == 0)
{
    op_stat_global_write (thru2_gshandle,
    fddi2_sink_total_bits / op_sim_time());

    op_stat_global_write (thru2_gshandle_a[0],
    fddi2_sink_total_bits_a[0] / op_sim_time());
    op_stat_global_write (thru2_gshandle_a[1],
    fddi2_sink_total_bits_a[1] / op_sim_time());
    op_stat_global_write (thru2_gshandle_a[2],
    fddi2_sink_total_bits_a[2] / op_sim_time());
    op_stat_global_write (thru2_gshandle_a[3],
    fddi2_sink_total_bits_a[3] / op_sim_time());
    op_stat_global_write (thru2_gshandle_a[4],
    fddi2_sink_total_bits_a[4] / op_sim_time());
    op_stat_global_write (thru2_gshandle_a[5],
    fddi2_sink_total_bits_a[5] / op_sim_time());
    op_stat_global_write (thru2_gshandle_a[6],
    fddi2_sink_total_bits_a[6] / op_sim_time());
    op_stat_global_write (thru2_gshandle_a[7],
    fddi2_sink_total_bits_a[7] / op_sim_time());
    op_stat_global_write (thru2_gshandle_a[8],
    fddi2_sink_total_bits_a[8] / op_sim_time());

    /* 30JAN94: gather all async stats into one overall figure */
    op_stat_global_write (thru2_gshandle_a[9],
    (fddi2_sink_total_bits - fddi2_sink_total_bits_a[8]) /
    op_sim_time());

    /*
    fddi2_sink_total_bits_a[0] + fddi2_sink_total_bits_a[1] + * /
    fddi2_sink_total_bits_a[6] + fddi2_sink_total_bits_a[7]) / * /
    op_sim_time());
*/

    op_stat_global_write (m2_delay_gshandle,
    fddi2_sink_accum_delay / fddi2_sink_total_pkts);

    op_stat_global_write (m2_delay_gshandle_a[0],
    fddi2_sink_accum_delay_a[0] / fddi2_sink_total_pkts_a[0]);
    op_stat_global_write (m2_delay_gshandle_a[1],
    fddi2_sink_accum_delay_a[1] / fddi2_sink_total_pkts_a[1]);

    op_stat_global_write (m2_delay_gshandle_a[2],
    fddi2_sink_accum_delay_a[2] / fddi2_sink_total_pkts_a[2]);
    op_stat_global_write (m2_delay_gshandle_a[3],
    fddi2_sink_accum_delay_a[3] / fddi2_sink_total_pkts_a[3]);
    op_stat_global_write (m2_delay_gshandle_a[4],
    fddi2_sink_accum_delay_a[4] / fddi2_sink_total_pkts_a[4]);
    op_stat_global_write (m2_delay_gshandle_a[5],
    fddi2_sink_accum_delay_a[5] / fddi2_sink_total_pkts_a[5]);
    op_stat_global_write (m2_delay_gshandle_a[6],
    fddi2_sink_accum_delay_a[6] / fddi2_sink_total_pkts_a[6]);
    op_stat_global_write (m2_delay_gshandle_a[7],
    fddi2_sink_accum_delay_a[7] / fddi2_sink_total_pkts_a[7]);
    op_stat_global_write (m2_delay_gshandle_a[8],
    fddi2_sink_accum_delay_a[8] / fddi2_sink_total_pkts_a[8]);
op_stat_global_write (m2_delay_gshandle_a[2],
  fddi2_sink_accm_delay_a[2] / fddi2_sink_total_pkts_a[2]);

op_stat_global_write (m2_delay_gshandle_a[3],
  fddi2_sink_accm_delay_a[3] / fddi2_sink_total_pkts_a[3]);

op_stat_global_write (m2_delay_gshandle_a[4],
  fddi2_sink_accm_delay_a[4] / fddi2_sink_total_pkts_a[4]);

op_stat_global_write (m2_delay_gshandle_a[5],
  fddi2_sink_accm_delay_a[5] / fddi2_sink_total_pkts_a[5]);

op_stat_global_write (m2_delay_gshandle_a[6],
  fddi2_sink_accm_delay_a[6] / fddi2_sink_total_pkts_a[6]);

op_stat_global_write (m2_delay_gshandle_a[7],
  fddi2_sink_accm_delay_a[7] / fddi2_sink_total_pkts_a[7]);

op_stat_global_write (m2_delay_gshandle_a[8],
  fddi2_sink_accm_delay_a[8] / fddi2_sink_total_pkts_a[8]);

op_stat_global_write (m2_delay_gshandle_a[9],
  fddi2_sink_accm_delay_a[9] / fddi2_sink_total_pkts_a[9]);

(fddi2_sink_accm_delay - fddi2_sink_accm_delay_a[8]) /
(fddi2_sink_total_pkts - fddi2_sink_total_pkts_a[8]));

(fddi2_sink_accm_delay_a[0] + fddi2_sink_accm_delay_a[1] + */
(fddi2_sink_accm_delay_a[6] + fddi2_sink_accm_delay_a[7]) / */
(fddi2_sink_total_pkts_a[0] + fddi2_sink_total_pkts_a[1] + */
(fddi2_sink_total_pkts_a[6] + fddi2_sink_total_pkts_a[7]));

op_stat_global_write (ete2_delay_gshandle, delay);

op_stat_global_write (ete2_delay_gshandle_a[pri2_set], delay);
}
227 } /*end of if(dest_addr==my_addr) && (src_addr > 9) statement */

228 /* 20APR94: destroy the packets coming from the first lan destined */
229 /* for this station. These packets are not counted for local traffic.*/
230 else if (dest_addr == my_addr)
231 op_pk_destroy(pkptr);

232 /* Other frames passed to "llc" should be destined for other lan */
233 /* 18APR94: allocate the packets to transmitter of command link */
234 else
235 {

236 /* add in its size */
237 fddi2lp2_total_bits += op_pk_total_size_get (pkptr);
238 fddi2lp2_total_bits_a[pri2_set] += op_pk_total_size_get (pkptr); /* 20JAN-20APR94 */
239 /* increment packet counter; 20APR94 */
240 fddilp2_total_pkts++;
241 fddilp2_total_pkts_a[pr2_set]++;

242 /* if a multiple of 25 packets is reached, update stats */
243 /* [0]->[7] represent async priorities 1->8, */
244 /* respectively; [8] represents synchronous traffic, */
245 /* and [9] represents overall asynchronous traffic. */
246 if (fddilp2_total_pkts % 25 == 0)
247 {
248 op_stat_global_write (t2_gshandle,
249 fddilp2_total_bits / op_sim_time ());

250 op_stat_global_write (t2_gshandle_a[pr2_set],
251 fddilp2_total_bits_a[0] / op_sim_time());
252 op_stat_global_write (t2_gshandle_a[0],
253 fddilp2_total_bits_a[1] / op_sim_time());
254 op_stat_global_write (t2_gshandle_a[1],
255 fddilp2_total_bits_a[pr2_set] / op_sim_time());
256 op_stat_global_write (t2_gshandle_a[2],
257 fddilp2_total_bits_a[2] / op_sim_time());
258 op_stat_global_write (t2_gshandle_a[3],
259 fddilp2_total_bits_a[3] / op_sim_time());
260 op_stat_global_write (t2_gshandle_a[4],
261 fddilp2_total_bits_a[4] / op_sim_time());
262 op_stat_global_write (t2_gshandle_a[5],
263 fddilp2_total_bits_a[5] / op_sim_time());
264 op_stat_global_write (t2_gshandle_a[6],
265 fddilp2_total_bits_a[6] / op_sim_time());
266 op_stat_global_write (t2_gshandle_a[7],
267 fddilp2_total_bits_a[7] / op_sim_time());
268 op_stat_global_write (t2_gshandle_a[8],
269 fddilp2_total_bits_a[8] / op_sim_time());

270 /* gather all async stats into one overall figure */
271 op_stat_global_write (t2_gshandle_a[9],
272 (fddilp2_total_bits - fddilp2_total_bits_a[8]) /
273 op_sim_time());

274 /* (fddilp2_total_bits_a[0] + fddilp2_total_bits_a[1] +
277 fddilp2_total_bits_a[6] + fddilp2_total_bits_a[7]) */
278 op_sim_time());

279 }

280 /* 21APR94: allocate packets to the command link transmitter */
261 op_subq_pk_insert(0, pkptr, OPC_QPOS_TAIL);

262 /* check if this subqueue is empty and transmitter is not busy */
263 if (!op_subq_empty(0) && (busy == 0.0))
264 {
265  /* access the first packet in the subqueue */
266  pkptr1 = op_subq_pk_remove (0, OPC_QPOS_HEAD);
267  /* forward it to the transmitter of command link */
268  op_pk_send (pkptr1, 0);
269 }
270 }/* end of if */
271 break;
272 }/* end of case OPC_INTRPT_STRM statement */
273 }/* end of switch */
274 }

275 }/* end of state (DISCARD) exit of unforced state. */
276 FSN_EXIT (1, sp_fddi_sink)

277 }/* end of state (DISCARD) transition processing */
278 FSN_INIT_COND (END_OF_SIM)
279 FSN_DEF_COND
280 FSN_TEST_LOGIC ("DISCARD")
281 FSN_TRANSIT_SWITCH
282 {
283  FSN_CASE_TRANSIT (0, 1, state1_enter_exec, ;)
284  FSN_CASE_TRANSIT (1, 0, state0_enter_exec, ;)
285 }
286 }/*--------------------------------------*/

287 }/* end of state (STATS) enter of unforced state. */
288 FSN_STATE_ENTER_UNFORCED (1, state1_enter_exec, "STATS")
289 {
290 }/* At end of simulation, scalar performance statistics */
291 }/* and input parameters are written out. */
292 }/* This is for command link throughput :21APR94*/
293 op_stat_scalar_write ("CL Throughput (bps), Priority 1", 
294 fddilp2_total_bits_a[0] / op_sim_time ()

152
op_stat_scalar_write ("CL Throughput (bps), Priority 2", fddilp2_total_bits_a[1] / op_sim_time ());

op_stat_scalar_write ("CL Throughput (bps), Priority 3", fddilp2_total_bits_a[2] / op_sim_time ());

op_stat_scalar_write ("CL Throughput (bps), Priority 4", fddilp2_total_bits_a[3] / op_sim_time ());

op_stat_scalar_write ("CL Throughput (bps), Priority 5", fddilp2_total_bits_a[4] / op_sim_time ());

op_stat_scalar_write ("CL Throughput (bps), Priority 6", fddilp2_total_bits_a[5] / op_sim_time ());

op_stat_scalar_write ("CL Throughput (bps), Priority 7", fddilp2_total_bits_a[6] / op_sim_time ());

op_stat_scalar_write ("CL Throughput (bps), Priority 8", fddilp2_total_bits_a[7] / op_sim_time ());

op_stat_scalar_write ("CL Throughput (bps), Asynchronous", (fddilp2_total_bits - fddilp2_total_bits_a[8]) / op_sim_time ());

/* (fddilp2_total_bits_a[0] + fddilp2_total_bits_a[1]) */
/* op_sim_time (); */

op_stat_scalar_write ("CL Throughput (bps), Synchronous", fddilp2_total_bits_a[8] / op_sim_time ());

op_stat_scalar_write ("CL Throughput (bps), Total", fddilp2_total_bits / op_sim_time ());

/* Only one station needs to do this for the second ring(Ring 1)*/
if (!fddi2_sink_scalar_write)
{
  /* set the scalar write flag */
  fddi2_sink_scalar_write = 1;

  op_stat_scalar_write ("Mean End-to-End Delay-i (sec.), Priority 1", fddi2_sink_accum_delay_a[0] / fddi2_sink_total_pkts_a[0]);
op_stat_scalar_write ("Mean End-to-End Delay-1 (sec.), Priority 2",
    fddi2_sink_accum_delay_a[1] / fddi2_sink_total_pkts_a[1]);

op_stat_scalar_write ("Mean End-to-End Delay-1 (sec.), Priority 3",
    fddi2_sink_accum_delay_a[2] / fddi2_sink_total_pkts_a[2]);

op_stat_scalar_write ("Mean End-to-End Delay-1 (sec.), Priority 4",
    fddi2_sink_accum_delay_a[3] / fddi2_sink_total_pkts_a[3]);

op_stat_scalar_write ("Mean End-to-End Delay-1 (sec.), Priority 5",
    fddi2_sink_accum_delay_a[4] / fddi2_sink_total_pkts_a[4]);

op_stat_scalar_write ("Mean End-to-End Delay-1 (sec.), Priority 6",
    fddi2_sink_accum_delay_a[5] / fddi2_sink_total_pkts_a[5]);

op_stat_scalar_write ("Mean End-to-End Delay-1 (sec.), Priority 7",
    fddi2_sink_accum_delay_a[6] / fddi2_sink_total_pkts_a[6]);

op_stat_scalar_write ("Mean End-to-End Delay-1 (sec.), Priority 8",
    fddi2_sink_accum_delay_a[7] / fddi2_sink_total_pkts_a[7]);

op_stat_scalar_write ("Mean End-to-End Delay-1 (sec.), Asynchronous",
    (fddi2_sink_accum_delay - fddi2_sink_accum_delay_a[8]) /
    (fddi2_sink_total_pkts - fddi2_sink_total_pkts_a[8]));

/* (fddi2_sink_accum_delay_a[0] + fddi2_sink_accum_delay_a[1] + */
/* fddi2_sink_accum_delay_a[6] + fddi2_sink_accum_delay_a[7]) / */
/* (fddi2_sink_total_pkts_a[0] + fddi2_sink_total_pkts_a[1] + */
/* fddi2_sink_total_pkts_a[6] + fddi2_sink_total_pkts_a[7])); */

op_stat_scalar_write ("Mean End-to-End Delay-1 (sec.), Synchronous",
    fddi2_sink_accum_delay_a[8] / fddi2_sink_total_pkts_a[8]);

op_stat_scalar_write ("Mean End-to-End Delay-1 (sec.), Total",
    fddi2_sink_accum_delay / fddi2_sink_total_pkts);
387  fddi2_sink_total_bits_a[3] / op_sim_time();
388  op_stat_scalar_write("Throughput-1 (bps), Priority 5",
389            fddi2_sink_total_bits_a[4] / op_sim_time());
390  op_stat_scalar_write("Throughput-1 (bps), Priority 6",
391            fddi2_sink_total_bits_a[5] / op_sim_time());
392  op_stat_scalar_write("Throughput-1 (bps), Priority 7",
393            fddi2_sink_total_bits_a[6] / op_sim_time());
394  op_stat_scalar_write("Throughput-1 (bps), Priority 8",
395            fddi2_sink_total_bits_a[7] / op_sim_time());
396  op_stat_scalar_write("Throughput-1 (bps), Asynchronous",
397            (fddi2_sink_total_bits - fddi2_sink_total_bits_a[8]) / op_sim_time());
398  /* (fddi2_sink_total_bits_a[0] + fddi2_sink_total_bits_a[1] + */
401  /* fddi2_sink_total_bits_a[6] + fddi2_sink_total_bits_a[7]) / */
402  /* op_sim_time());
403  op_stat_scalar_write("Throughput-1 (bps), Synchronous",
404            fddi2_sink_total_bits_a[8] / op_sim_time());
405  op_stat_scalar_write("Throughput-1 (bps), Total",
406            fddi2_sink_total_bits / op_sim_time());
407  op_stat_scalar_write("Peak End-to-End Delay-1 (sec.), Priority 1",
408            fddi2_sink_peak_delay_a[0]);
409  op_stat_scalar_write("Peak End-to-End Delay-1 (sec.), Priority 2",
410            fddi2_sink_peak_delay_a[1]);
411  op_stat_scalar_write("Peak End-to-End Delay-1 (sec.), Priority 3",
412            fddi2_sink_peak_delay_a[2]);
413  op_stat_scalar_write("Peak End-to-End Delay-1 (sec.), Priority 4",
414            fddi2_sink_peak_delay_a[3]);
415  op_stat_scalar_write("Peak End-to-End Delay-1 (sec.), Priority 5",
416            fddi2_sink_peak_delay_a[4]);
417  op_stat_scalar_write("Peak End-to-End Delay-1 (sec.), Priority 6",
418            fddi2_sink_peak_delay_a[5]);
op_stat_scalar_write ("Peak End-to-End Delay-1 (sec.), Priority 7",
    fddi2_sink_peak_delay_a[6]);

op_stat_scalar_write ("Peak End-to-End Delay-1 (sec.), Priority 8",
    fddi2_sink_peak_delay_a[7]);

op_stat_scalar_write ("Peak End-to-End Delay-1 (sec.), Synchronous",
    fddi2_sink_peak_delay_a[8]);

op_stat_scalar_write ("Peak End-to-End Delay-1 (sec.), Overall",
    fddi2_sink_peak_delay);

/* Write the TTRT value for ring 0. This preserves */
/* the old behavior for single-ring simulations. */
op_stat_scalar_write ("TTRT (sec.) - Ring 1",
    fddi_t_opr [1]);

/* 12JAN94: obtain offered load information from the Environment */
/* file; this will be used to provide abscissa information that */
/* can be plotted in the Analysis Editor (see "fddi_sink" STATS */
/* state. To the user: it's your job to keep these current in */
/* the Environment File. -Nix */
op_ima_sim_attr_get (OPC_IMA_DOUBLE, "total_offered_load_1", &Offered_Load);

op_ima_sim_attr_get (OPC_IMA_DOUBLE, "asynch_offered_load_1", &Asynch_Offered_Load);

/* 12JAN94: write the total offered load for this run */
op_stat_scalar_write ("Total Offered Load-1 (Mbps)",
    Offered_Load);

op_stat_scalar_write ("Asynchronous Offered Load-1 (Mbps)",
    Asynch_Offered_Load);

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

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

/* state (STATS) transition processing */
FSM_TRANSIT_MISSING ("STATS")
454 /** state (INIT) enter executives */
455 FSM_STATE_ENTER_FORCED (2, state2_enter_exec, "INIT")
456 {
457 /* get the ghandles of the global statistic to be obtained */
458 /* 20JAN94: set array format */

459 thru2_gshandle_a[0] = op_stat_global_reg ("pri 1 throughput-1 (bps)");
460 thru2_gshandle_a[1] = op_stat_global_reg ("pri 2 throughput-1 (bps)");
461 thru2_gshandle_a[2] = op_stat_global_reg ("pri 3 throughput-1 (bps)");
462 thru2_gshandle_a[3] = op_stat_global_reg ("pri 4 throughput-1 (bps)");
463 thru2_gshandle_a[4] = op_stat_global_reg ("pri 5 throughput-1 (bps)");
464 thru2_gshandle_a[5] = op_stat_global_reg ("pri 6 throughput-1 (bps)");
465 thru2_gshandle_a[6] = op_stat_global_reg ("pri 7 throughput-1 (bps)");
466 thru2_gshandle_a[7] = op_stat_global_reg ("pri 8 throughput-1 (bps)");
467 thru2_gshandle_a[8] = op_stat_global_reg ("synch throughput-1 (bps)");
468 thru2_gshandle_a[9] = op_stat_global_reg ("async throughput-1 (bps)");
469 thru2_gshandle = op_stat_global_reg ("total throughput-1 (bps)");

470 m2_delay_gshandle_a[0] = op_stat_global_reg ("pri 1 mean delay-1 (sec.)");
471 m2_delay_gshandle_a[1] = op_stat_global_reg ("pri 2 mean delay-1 (sec.)");
472 m2_delay_gshandle_a[2] = op_stat_global_reg ("pri 3 mean delay-1 (sec.)");
473 m2_delay_gshandle_a[3] = op_stat_global_reg ("pri 4 mean delay-1 (sec.)");
474 m2_delay_gshandle_a[4] = op_stat_global_reg ("pri 5 mean delay-1 (sec.)");
475 m2_delay_gshandle_a[5] = op_stat_global_reg ("pri 6 mean delay-1 (sec.)");
476 m2_delay_gshandle_a[6] = op_stat_global_reg ("pri 7 mean delay-1 (sec.)");
477 m2_delay_gshandle_a[7] = op_stat_global_reg ("pri 8 mean delay-1 (sec.)");
478 m2_delay_gshandle_a[8] = op_stat_global_reg ("synch mean delay-1 (sec.)");
479 m2_delay_gshandle_a[9] = op_stat_global_reg ("async mean delay-1 (sec.)");
480 m2_delay_gshandle = op_stat_global_reg ("total mean delay-1 (sec.)");

481 ete2_delay_gshandle_a[0] = op_stat_global_reg ("pri 1 end-to-end delay-1 (sec.)");
482 ete2_delay_gshandle_a[1] = op_stat_global_reg ("pri 2 end-to-end delay-1 (sec.)");
483 ete2_delay_gshandle_a[2] = op_stat_global_reg ("pri 3 end-to-end delay-1 (sec.)");
484 ete2_delay_gshandle_a[3] = op_stat_global_reg ("pri 4 end-to-end delay-1 (sec.)");
485 ete2_delay_gshandle_a[4] = op_stat_global_reg ("pri 5 end-to-end delay-1 (sec.)");
486 ete2_delay_gshandle_a[5] = op_stat_global_reg ("pri 6 end-to-end delay-1 (sec.)");
487 ete2_delay_gshandle_a[6] = op_stat_global_reg ("pri 7 end-to-end delay-1 (sec.)");
488 ete2_delay_gshandle_a[7] = op_stat_global_reg ("pri 8 end-to-end delay-1 (sec.)");
489 ete2_delay_gshandle_a[8] = op_stat_global_reg ("synch end-to-end delay-1 (sec.)");
490 ete2_delay_gshandle = op_stat_global_reg ("total end-to-end delay-1 (sec.)");

491 t2_gshandle_a[0] = op_stat_global_reg ("pri 1 CL throughput (bps)");
492 t2_gshandle_a[1] = op_stat_global_reg ("pri 2 CL throughput (bps)");
493 t2_gshandle_a[2] = op_stat_global_reg ("pri 3 CL throughput (bps)");
494 t2_gshandle_a[3] = op_stat_global_reg ("pri 4 CL throughput (bps)");
495 t2_gshandle_a[4] = op_stat_global_reg ("pri 5 CL throughput (bps)");
496 t2_gshandle_a[5] = op_stat_global_reg ("pri 6 CL throughput (bps)");
497 t2_gshandle_a[6] = op_stat_global_reg ("pri 7 CL throughput (bps)");
498 t2_gshandle_a[7] = op_stat_global_reg ("pri 8 CL throughput (bps)");
t2_gchandle_a[0] = op_stat_global_reg ("synch CL throughput (bps)");
t2_gchandle_a[0] = op_stat_global_reg ("async CL throughput (bps)");
t2_gchandle = op_stat_global_reg ("total CL throughput (bps)");

/** 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, ;)
}

void
sp_fddi_sink_svar (prs_ptr, var_name, var_p_ptr)
sp_fddi_sink_state = prs_ptr;
char *var_name, **var_p_ptr;
{
FIN (sp_fddi_sink_svar (prs_ptr))

*var_p_ptr = VOS_NIL;
if (Vos_String_Equal ("thru2_gchandle", var_name))
*var_p_ptr = (char *) (prs_ptr->sv_thru2_gchandle);
if (Vos_String_Equal ("m2_delay_gchandle", var_name))
*var_p_ptr = (char *) (prs_ptr->sv_m2_delay_gchandle);
if (Vos_String_Equal ("etc2_delay_gchandle", var_name))
532 *var_p_ptr = (char *) (prs_p->sv_ete2_delay_gshandle);
533 if (Vos_String_Equal ("thru2_gshandle_a", var_name))
534 *var_p_ptr = (char *) (prs_p->sv_thru2_gshandle_a);
535 if (Vos_String_Equal ("m2_delay_gshandle_a", var_name))
536 *var_p_ptr = (char *) (prs_p->sv_m2_delay_gshandle_a);
537 if (Vos_String_Equal ("ete2_delay_gshandle_a", var_name))
538 *var_p_ptr = (char *) (prs_p->sv_ete2_delay_gshandle_a);
539 if (Vos_String_Equal ("t2_gshandle", var_name))
540 *var_p_ptr = (char *) (prs_p->sv_t2_gshandle);
541 if (Vos_String_Equal ("t2_gshandle_a", var_name))
542 *var_p_ptr = (char *) (prs_p->sv_t2_gshandle_a);
543 if (Vos_String_Equal ("my_id", var_name))
544 *var_p_ptr = (char *) (prs_p->sv_my_id);
545 };
546

547 void
548 sp_fddi_sink_diag ()
549 {
550 double delay, creat_time;
551 Packet* pkptr;
552 Packet* pkptr1; /*5APR94*/
553 int src_addr, my_addr;
554 int dest_addr; /*14APR94*/
555 Ici* from_mac_ici_ptr;
556 double fddi_sink_ttrt;

557 FIN (sp_fddi_sink_diag ()
558 };
559

560 void
561 sp_fddi_sink_terminate ()
562 {
563 double delay, creat_time;
564 Packet* pkptr;
565 Packet* pkptr1; /*5APR94*/
566 int src_addr, my_addr;
567 int dest_addr; /*14APR94*/
568 Ici* from_mac_ici_ptr;
569 double fddi_sink_ttrt;
570 FIN (sp_fddi_sink_terminate())

571 FOUT;
572 }

573 Compcode
574 sp_fddi_sink_init (pr_state_pptr)
575 sp_fddi_sink_state = pr_state_pptr;
576 {
577 static VosT_Cmsk_Obtype obtype = OPC_NIL;

578 FIN (sp_fddi_sink_init (pr_state_pptr))

579 if (obtype == OPC_NIL)
580 {
581 if (Vos_Catmem_Register("proc state vars (sp_fddi_sink)",
582 sizeof(sp_fddi_sink_state), Vos_Nop, &obtype) == VOSC_FAILURE)
583 FRET (OPC_COMPCODE_FAILURE)
584 }

585 if (*((pr_state_pptr = (sp_fddi_sink_state*)) Vos_Catmem_Alloc (obtype, 1)) == OPC_NIL)
586 FRET (OPC_COMPCODE_FAILURE)
587 else
588 {
589 (*pr_state_pptr)->current_block = 4;
590 FRET (OPC_COMPCODE_SUCCESS)
591 }
592 )
APPENDIX G

CDL MODEL ERROR ALLOCATION CODE

"cdl_pt_error.ps.c"

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

```c
#include <opnst.h>
#include <math.h>

#define log_factorial(n) lgamma ((double) n + 1.0)

void cdl_pt_error (pkptr) 

Packet* pkptr;
{

Objid link_objid;

double pe, r, p_accum, p_exact;
double log_pi, log_p2, log_arrange;
double duty_cycle; /* 31MAR94 */
double jam_length, jam_ber, int_bet_jamlen, ber_bet_jamlen; /*29MAR94*/
double time_stamp; /* time stamp for the packet arriving time */
double offset; /* 1APR94 */

int invert_errors = OPC_FALSE, seg_size, num_errs;
int jammer_type; /*26APR94*/

/*int channel_index;*/ /* 4APR94 */

/** Compute the number of errors assigned to the given packet **/
/** based on its length and the bit error probability. **/
FIN (cdl_pt_error (pkptr))

/*Make a time stamp to see whether the packet is in jamming period or not */
time_stamp = op_sim_time();
/*printf("time_stamp = %16.12f\n", time_stamp);*/

/* Obtain object id of point-to-point link carrying transmission. */
link_objid = op_td_get_int (pkptr, OPC_TDA_PT_LINK_OBJID);
```
/* Obtain the channel index for the particular link */
/* Determine which channel the packet is on */
/* channel_index = op_sd_get_int(pkptr, OPC_TDA_PT_CH_INDEX); */

/* Obtain the bit-error probability of the channel. */
/* op_dma_obj_attr_get (link_objid, "ber", &pbt); */
/* ignore this attribute 31MAR94 */

/* Obtain the extended attributes for the point-to-point link */
/* These attributes are appended in order to simulate jamming features */
/* 29MAR94 */

op_dma_obj_attr_get (link_objid, "jmlength", &jmlength);
op_dma_obj_attr_get (link_objid, "jamber", &jamber);
op_dma_obj_attr_get (link_objid, "interval_bet_jmlen", &int_bet_jmlen);
op_dma_obj_attr_get (link_objid, "ber_bet_jmlen", &ber_bet_jmlen);
op_dma_obj_attr_get (link_objid, "init_jam_offset", &offset);
op_dma_obj_attr_get (link_objid, "jammer_type", &jammer_type);

/* Obtain the length of the packet. */
seg_size = op_pk_total_size_get (pkptr);

/* Determine the jammer type in use: 28APR94 */
/* Check if pulsed jammer is in use */
if (jammer_type == 0)
{
    /* Randomize the jamming durations */
    /* These durations are randomized with uniform distribution */
    /* in range [0, duration]. User should be aware of these */
    /* attributes specified in the environment file. They are max values */
    /* for those particular durations */
    jmlength = op_dist_uniform(jmlength);
    int_bet_jmlen = op_dist_uniform(int_bet_jmlen);
}
/* Otherwise, channel swept jammer is in use. Jamming durations */
/* should not be randomized to keep consecutive pulses in order. */

/* Compute duty cycle for jamming */
duty_cycle = jmlength + int_bet_jmlen;

/* Check time stamp if it is in the initial jam offset period */
/* All BER's are uniformly distributed in range [0, ber], so that */
/* realistic representation is provided; User should be aware of */
/* these attributes specified in the environment file. They are max values */
/* for those particular berse */
if (time_stamp < offset)
    pe = op_dist_uniform(ber_bet_jmlen); /* the packet is still not in the jamming */
    /* period */
else
{
    /* Check packet is in jamming period */
if ( fmod(time_stamp, duty_cycle) <= jam_length )
    pe = op_dist_uniform(jam_ber); /* the packet is in jamming period */
    /* random "pe" to be computed as jam_ber */
else
    pe = op_dist_uniform(ber_best_jamlen); /* packet is in unjammed period */
    /* random "pe" to be computed as ber_best_jamlen */
}

/* This part computes num_errs for the packet */
/* Case 1: if the bit error rate is zero, so is the number of errors. */
if (pe == 0.0 || seg_size == 0)
    num_errs = 0;
/* Case 2: if the bit error rate is 1.0, then all the bits are in error. */
/* (note however, that bit error rates should not normally exceed 0.5). */
else if (pe >= 1.0)
    num_errs = seg_size;
/* Case 3: The bit error rate is not zero or one. */
else {
    /* If the bit error rate is greater than 0.5 and less than 1.0, invert */
    /* the problem to find instead the number of bits that are not in error */
    /* in order to accelerate the performance of the algorithm. Set a flag */
    /* to indicate that the result will then have to be inverted. */
    if (pe > 0.5)
        { pe = 1.0 - pe;
          invert_errors = OPC_TRUE;
        }
    /* The error count can be obtained by mapping a uniform random number */
    /* in [0, 1] via the inverse of the cumulative mass function (CMF) */
    /* for the bit error count distribution. */
    /* Obtain a uniform random number in [0, 1] to represent */
    /* the value of the CDF at the outcome that will be produced. */
    r = op_dist_uniform(1.0);
    /* Integrate probability mass over possible outcomes until r is exceeded. */
    /* The loop iteratively corresponds to "inverting" the CMF since it finds */
    /* the bit error count at which the CMF first meets or exceeds the value r. */
    for (p_accum = 0.0, num_errs = 0; num_errs <= seg_size; num_errs++)
        { /* Compute the probability of exactly 'num_errs' bit errors occurring. */
          /* The probability that the first 'num_errs' bits will be in error */
          /* is given by pow (pe, num_errs). Here it is obtained in logarithmic */
          /* form to avoid underflow for small 'pe' or large 'num_errs_jam'. */
          log_pi = (double) num_errs * log (pe);
          /* Base case: if the bit error rate is zero, so is the number of errors. */
        }
Similarly, obtain the probability that the remaining bits will not * be in error. The combination of these two events represents one * possible configuration of bits yielding a total of 'num_errs' errors. *

\[
\log_{\text{p}2} = (\text{double}) (\text{seg}_{\text{size}} - \text{num}_{\text{errs}}) \times \log (1.0 - \text{pe});
\]

/* Compute the number of arrangements that are possible with the same */
/* number of bits in error as the particular case above. Again obtain */
/* this number in logarithmic form (to avoid overflow in this case). */
/* This result is expressed as the logarithmic form of the formula for */
/* the number \( N \) of combinations of \( k \) items from \( n \): \( N = n!/(n-k)!k! \) */
log_arrange = log_factorial (seg_size) -
    log_factorial (num_errs) -
    log_factorial (seg_size - num_errs);

/* Compute the probability that exactly 'num_errs' are present */
/* in the segment of bits, in any arrangement. */
p_exact = exp (log_arrange + log_p1 + log_p2);

/* Add this to the probability mass accumulated so far for previously */
/* tested outcomes to obtain the value of the CMF at outcome='num_errs'*/
p_accum += p_exact;

/* 'num_errs' is the outcome for this trial if the CMF meets or exceeds */
/* the uniform random value selected earlier. */
if (p_accum >= r)
    break;

/* If the bit error rate was inverted to compute correct bits instead, then */
/* reinvert the result to obtain the number of bits in error. */
if (invert_errors == OPC_TRUE)
    num_errs = seg_size - num_errs;

/* printf("num_of_errors = %5d\n", num_errs); */
/* Set number of bit errors in packet transmission data attribute. */
op_td_set_int (pkptr, OPC_TDA_PT_NUM_ERRORS, num_errs);
FOUT
}
APPENDIX H
SAMPLE ENVIRONMENT FILE FOR
PULSED JAMMER

```bash
# cd14_lbojam0.ef
# sample simulation configuration file for
# two interconnected 10 station network in the
# existence of pulsed jammer interference (137.088 Mbps channel hierarchy)
# with circular allocation load balancing algorithm

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

```bash
# station addresses
*.ring0.f0.mac.station_address: 0
*.ring0.f1.mac.station_address: 1
*.ring0.f2.mac.station_address: 2
*.ring0.f3.mac.station_address: 3
*.ring0.f4.mac.station_address: 4
*.ring0.f5.mac.station_address: 5
*.ring0.f6.mac.station_address: 6
*.ring0.f7.mac.station_address: 7
*.ring0.f8.mac.station_address: 8
*.ring0.f9.mac.station_address: 9
*.ring1.f0.mac.station_address: 10
*.ring1.f1.mac.station_address: 11
*.ring1.f2.mac.station_address: 12
*.ring1.f3.mac.station_address: 13
*.ring1.f4.mac.station_address: 14
*.ring1.f5.mac.station_address: 15
*.ring1.f6.mac.station_address: 16
*.ring1.f7.mac.station_address: 17
*.ring1.f8.mac.station_address: 18
*.ring1.f9.mac.station_address: 19
*.ring0.\.mac.ring_id:0
*.ring1.\.mac.ring_id:1

# Specific stations may be tailored by specifying the full name:
# for example, top.ring0.f10.llc_src.async_mix : .5
# This means all stations must be specified, or individuals
# may be named after the generic is specified.
# destination addresses for random message generation
# "top.ring0.f0.llc_src.low dest address"
# "top.ring0.f0.llc_src.high dest address" :
```
".ring0.*.llc_src.low dest address": 10
".ring0.*.llc_src.high dest address": 19
".ring1.*.llc_src.low dest address": 0
".ring1.*.llc_src.high dest address": 19

---

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.

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

---

arrival rate(frames/sec), and message size (bits) for random message generation at each station on the ring.

"e.*.arrival rate": 250
"e.*.mean pk length": 20000

---

7APR94 - S.Karayakaylar
User should specify the algorithm before simulation.

0 (zero) -----> circular load balancing algorithm (default)
1 (one) -----> empty allocation algorithm

"top.ring0.f9.llc_sink.load balancing algorithm": 0

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

total_offered_load_0: 50
asynch_offered_load_0: 45
total_offered_load_1: 50
asynch_offered_load_1: 45

set the proportion of asynchronous traffic
a value of 1.0 indicates all asynchronous traffic
"e.*.async_mix": 0.9
### 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*: 0.08955675
    - "top.ring0.f9.mac.sync bandwidth": 0.0

- Target Token Rotation Time (one half of maximum synchronous response time)
  - This is commented out for compatibility with the fddi_script, which sets $T_{\text{Req}}$ on the simulation command line; remove the comment pound-sign below to make this environment file self-sufficient.

  - $\text{SUM(SA)} + D_{\text{Max}} + F_{\text{Max}} + \text{Token_Time} \leq T_{\text{TRT}}$
  - Powers gives $T_{\text{TRT}} = 10$ ms as necessary for voice transmission; in "BONeS", $D_{\text{Max}} + F_{\text{Max}} + \text{Token_Time} = 1.97888$ ms.
  - "*.*mac.$T_{\text{Req}}$": .004

- Index of the station which initially launches the token
  - 17APR94: -Karayakaylar
  - This index should be greater than the maximum station number
  - Bridge stations spawns token for interconnected simulation by default.
    - "spawn station": 20

- Delay incurred by packets as they traverse a station's ring interface
  - See Powers, p. 351 for a discussion of this (Powers gives $\mu\text{sec}$, but $60.0e-08$ agrees with Dykeman & Bux)
  - station_latency: 60.0e-08

- Propagation Delay separating stations on the ring.
  - If propagation delay is 5.085 microsec/km, this corresponds to
    - a 50 station ring with a circumference of 50 km.
  - (The value given for propagation delay corresponds to Powers, and to
    - Dykeman & Bux)
  - prop_delay: 5.085e-06

- CDL link related attributes -Karayakaylar 7APR94
  - The attributes below are specified with respect to the jammer type
  - There are two types of jamming models which the CDL is exposed to.
    - (1) Pulsed jammer
      - $(\text{jammer.type} = 0)$
    - (2) Channel-swept jammer
      - $(\text{jammer.type} = 1)$

  - NOTE: For pulsed jammer init_jam_offset may be zero, whereas a proper offset should be specified for channel-swept jammer.
# jam_length, jam_ber, interval_bet_jam_len, ber_bet_jam_len are maximum values in the case of pulsed jammer since they are randomized in the error allocation pipeline stage.
# For channel-swept jammer only jam_ber and ber_bet_jam_len attributes are maximum values to be randomized.
# return link ls_0 to ls_3
# command link ls_4
* ls_0.jam_length: 0.05
* ls_1.jam_length: 0.02
* ls_2.jam_length: 0.01
* ls_3.jam_length: 0.09
* ls_4.jam_length: 0.06
* ls_0.jam_ber: 2e-3
* ls_1.jam_ber: 2e-3
* ls_2.jam_ber: 2e-3
* ls_3.jam_ber: 2e-3
* ls_4.jam_ber: 0.0
* ls_0.interval_bet_jam_len: 0.03
* ls_1.interval_bet_jam_len: 0.03
* ls_2.interval_bet_jam_len: 0.03
* ls_3.interval_bet_jam_len: 0.03
* ls_4.interval_bet_jam_len: 0.03
* ls_0.ber_bet_jam_len: 2e-6
* ls_1.ber_bet_jam_len: 2e-6
* ls_2.ber_bet_jam_len: 2e-6
* ls_3.ber_bet_jam_len: 2e-6
* ls_4.ber_bet_jam_len: 2e-6
* ls_0.init_jam_offset: 0.0
* ls_1.init_jam_offset: 0.0
* ls_2.init_jam_offset: 0.0
* ls_3.init_jam_offset: 0.0
* ls_4.init_jam_offset: 0.0
* ls_0.jammer_type: 0
* ls_1.jammer_type: 0
* ls_2.jammer_type: 0
* ls_3.jammer_type: 0
* ls_4.jammer_type: 0

# Return and command link propagation delays are specified as 60 msec.
* ls_0.delay: 0.06
* ls_1.delay: 0.06
* ls_2.delay: 0.06
* ls_3.delay: 0.06
* ls_4.delay: 0.06

#### Simulation related attributes

# Token Acceleration Mechanism enabling flag.
# It is recommended that this mechanism be enabled for most situations
# 16APR94 : for bridged fddi cd1_interconnection network this flag must be zero. Otherwise, program fault occurs. -Karayakaylar
accelerate_token: 0

# Run control attributes
seed: 10
duration: 1
verbose_sim: TRUE
upd_int: .1
os_file: cdl4_lb0jam0

# (This is commented out for compatibility with the fddi_script, which
# sets the output vector file on the simulation command line; remove the
# comment pound-sign below to make this environment file self-sufficient.)
# ov_file: cdl4_lb0jam0

# Opnet Debugger (odb) enabling attribute
# debug: TRUE
APPENDIX I
SAMPLE ENVIRONMENT FILE EXCERPT
FOR CHANNEL-SWEPT JAMMER

# cd14_1b1jam1.ef
# sample simulation configuration file for
# two interconnected 10 station network in the
# existence of channel-swept jammer interference (137.088 Mbps channel hierarchy)
# with empty selection load balancing algorithm
...
...
...
...
...
...
...

# CDL link related attributes - Karayaklar 7APR94
# The attributes below are specified with respect to the jammer type
# There are two types of jamming models which the CDL is exposed to.
#
# (1) Pulsed jammer (jammer_type = 0)
# (2) Channel-swept jammer (jammer_type = 1)
#
# NOTE: For pulsed jammer init_jam_offset may be zero, whereas a proper
# offset should be specified for channel-swept jammer.
# jam_length, jam_ber, interval_bet_jam_len, ber_bet_jam_len are maximum
# values in the case of pulsed jammer since they are randomized in the
# error allocation pipeline stage.
# For channel-swept jammer only jam_ber and ber_bet_jam_len attributes are
# maximum values to be randomized.
# return link ls_0 to ls_3
# command link ls_4
  * ls_0.jam_length: 0.02
  * ls_1.jam_length: 0.02
  * ls_2.jam_length: 0.02
  * ls_3.jam_length: 0.02
  * ls_4.jam_length: 0.02
  * ls_0.jam_ber: 2e-3
  * ls_1.jam_ber: 2e-3
  * ls_2.jam_ber: 2e-3
  * ls_3.jam_ber: 2e-3
  * ls_4.jam_ber: 0.0
  * ls_0.interval_bet_jam_len: 0.06
  * ls_1.interval_bet_jam_len: 0.06
  * ls_2.interval_bet_jam_len: 0.06
  * ls_3.interval_bet_jam_len: 0.06
  * ls_4.interval_bet_jam_len: 0.06
  * ls_0.ber_bet_jam_len: 2e-6
  * ls_1.ber_bet_jam_len: 2e-6
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ls_2.ber_bet_jam_len</td>
<td>2e-6</td>
</tr>
<tr>
<td>ls_3.ber_bet_jam_len</td>
<td>2e-6</td>
</tr>
<tr>
<td>ls_4.ber_bet_jam_len</td>
<td>0.0</td>
</tr>
<tr>
<td>ls_0.init_jam_offset</td>
<td>0.0</td>
</tr>
<tr>
<td>ls_1.init_jam_offset</td>
<td>0.02</td>
</tr>
<tr>
<td>ls_2.init_jam_offset</td>
<td>0.04</td>
</tr>
<tr>
<td>ls_3.init_jam_offset</td>
<td>0.06</td>
</tr>
<tr>
<td>ls_4.init_jam_offset</td>
<td>0.0</td>
</tr>
<tr>
<td>ls_0.jammer_type</td>
<td>1</td>
</tr>
<tr>
<td>ls_1.jammer_type</td>
<td>1</td>
</tr>
<tr>
<td>ls_2.jammer_type</td>
<td>1</td>
</tr>
<tr>
<td>ls_3.jammer_type</td>
<td>1</td>
</tr>
<tr>
<td>ls_4.jammer_type</td>
<td>1</td>
</tr>
</tbody>
</table>

...
LIST OF REFERENCES


## INITIAL DISTRIBUTION LIST

<table>
<thead>
<tr>
<th>No. Copies</th>
<th>Name and Address</th>
</tr>
</thead>
</table>
| 2          | Defense Information Center  
Cameron Station  
Alexandria, VA 22304-6145 |
| 2          | Library Code 052  
Naval Postgraduate School  
Monterey, CA 93943-5000 |
| 1          | Chairman, Code EC  
Department of Electrical and Computer Engineering  
Naval Postgraduate School  
Monterey, CA 93943-5121 |
| 2          | Professor Shirdhar B. Shukla, Code EC/Sh  
Department of Electrical and Computer Engineering  
Naval Postgraduate School  
Monterey, CA 93943-5121 |
| 2          | Professor Gilbert Lundy, Code CS/Ln  
Department of Computer Science  
Naval Postgraduate School  
Monterey, CA 93943-5118 |
| 1          | Professor Paul Moose, Code EC/Me  
Department of Electrical and Computer Engineering  
Naval Postgraduate School  
Monterey, CA 93940-5121 |
| 1          | Mark Russon, UNISYS  
Mail Station F2-G14  
640 North 2000 West  
Salt Lake City, UT 84116-2988 |
| 1          | CDL Program Manager  
Defense Support Project Office  
Washington DC 20330-1000 |
| No. Copies | 9. Deniz Kuvvetleri Komutanligi  
Personel Daire Baskanligi  
Bakanliklar, Ankara, Turkey | 2 |
| 10. Golcuk Tersanesi Komutanligi  
Golcuk, Kocaeli, Turkey | 1 |
| 11. Deniz Harp Okulu Komutanligi  
81704 Tuzla, Istanbul, Turkey | 1 |
| 12. Taskizak Tersanesi Komutanligi  
Kasimpasa, Istanbul, Turkey | 1 |
| 13. Selcuk Karayakaylar  
Kardesler sk. Huzur Apt.  
A-Blok 11/10  
80700 Dikilitas, Istanbul, Turkey | 1 |
| Department of Electrical and Computer Engineering | |