STARS

University of Central Florida
STARS

Institute for Simulation and Training

Digital Collections

1-1-1989

Simulation Networks Modeling And Monitoring: Final Project Report

Michael Bassiouni

Find similar works at: https://stars.library.ucf.edu/istlibrary
University of Central Florida Libraries http://library.ucf.edu

This Research Report is brought to you for free and open access by the Digital Collections at STARS. It has been accepted for inclusion in Institute for Simulation and Training by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

Recommended Citation
Bassiouni, Michael, "Simulation Networks Modeling And Monitoring: Final Project Report" (1989). *Institute for Simulation and Training*. 167.
https://stars.library.ucf.edu/istlibrary/167

University of Central Florida
STARS
Showcase of Text, Archives, Research & Scholarship

INSTITUTE FOR SIMULATION AND TRAINING

Contract Number N61339-88-G-0002 Order 0008
PM TRADE

August 1, 1988 through July 31, 1989

Simulation Networks Modeling and Monitoring

Final Project Report

Institute for Simulation and Training
12424 Research Parkway, Suite 300
Orlando FL 32826

University of Central Florida
Division of Sponsored Research

IST-CR-89-15

B 108

INSTITUTE FOR SIMULATION AND TRAINING

Contract Number N61339-88-G-0002 Order 0008
PM TRADE

August 1, 1988 through July 31, 1989

Simulation Networks Modeling and Monitoring

Final Project Report

Author:
M. Bassiouni

Institute for Simulation and Training
12424 Research Parkway, Suite 300
Orlando FL 32826

University of Central Florida
Division of Sponsored Research

IST-CR-89-15

TABLE OF CONTENTS

	Page

Organization of Final Report	1
Section 1: Chronological Listing of Activities	3
Section 2: Network System Configuration Models	8
Section 3: The SIMNET/ETHERNET Simulation Model	11
Section 4: The Token-Ring Simulation Model	23
Section 5: Performance Results	25
Section 6: Conclusions	46
Acknowledgements	50

Appendix A: Source Code Listing of the ETHERNET Simulation Model
Appendix B: Source Code Listing of the Token-ring Simulation Model
Appendix C: Copy of Published Paper (IST Networking Conference)
Appendix D: Copy of Published Paper (I/ITSC Conference)
Appendix E: Literature Search Summary

Organization of Final Report

This report contains six sections (1-6) and five appendices (A-E) as follows:

* Section 1 gives a chronological listing of the activities and progress made during the one year period of the project.
* Section 2 describes the two network system models considered in this project, namely, the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) and the token-ring medium access protocols.
* Section 3 gives a description of the logical design of the simulation model for SIMNET with the ETHERNET CSMA/CD protocol.
* Section 4 describes the logical design of the simulation model for SIMNET with the token-ring protocol.
* Section 5 contains performance results and miscellaneous statistics gathered from the simulation models.
* Section 6 contains summary of the project and concluding remarks.
* Appendix A gives the complete listing of the Concurrent-C code of the ETHERNET simulation model.
* Appendix B gives the complete listing of the Concurrent-C code of the token-ring simulation model.

* Appendix C contains a copy of the paper entitled "Simulation Networking and Protocol Alternatives" (by M. Bassiouni, M. Georgiopoulos, and J. Thompson) published in the Proceedings of the First IST Networking Conference, April 1989.

* Appendix D contains a copy of the paper entitled "Real-Time Simulation Networking- Network Modeling and Protocol Alternatives" (by M. Bassiouni, M. Georgiopoulos, and J. Thompson) accepted for publication in the Proceedings of the 11th I/ITSC Conference, November 1989.

* Appendix E contains a summary of the literature search performed during this project.

1. Chronological Listing of Activities and Progress

The following is a brief description of the progress made and the activities carried out during the one-year period of this project. The progress and activities are listed in chronological order and are grouped into two-month periods.

Months 1-2:

* Detailed workplan was developed and graduate students were hired.
* Various technical information and articles about SIMNET were gathered.
* A study to characterize the SIMNET networking approach was performed and basic features of the protocols were examined.
* Experience on the capabilities and method of operation of the HP-4972A network analyzer (at IST SIMNET Laboratory) was gained.
* Review of literature on analytical models for ETHERNET was started and several technical articles were studied.
* Preliminary work on the design of the simulation models for SIMNET was started and appropriate available simulation languages (e.g., Concurrent-C, SLAM, SIMSCRIPT, Concurrent-Euclid, etc.) were examined and evaluated.

Months 3-4

* More technical information and articles about SIMNET were gathered and studied. We also obtained copies of the latest release of the IEEE Standards for the

CSMA/CD access method and its physical layer specifications.

* The effort for the design of the simulation models for SIMNET with ETHERNET continued. Methods for generating traffic load (both by stochastic distributions and trace data) were finalized. Design details for simulating the CSMA/CD collision handling mechanism were extensively examined and discussed during our weekly meetings at IST.

* The evaluation process of several available programming languages was completed and the Concurrent-C language was selected for the implementation of the simulation models in this project.

* We continued gaining experience on the capabilities and method of operation of the HP-4972A network analyzer. Tests to measure the SIMNET traffic load using the HP-4972A analyzer were conducted.

* Literature review on token-ring local area networks was started.

Months 5-6

* The high-level design of the simulation models for SIMNET under ETHERNET was completed and the implementation (coding) phase was started. The simulation program (written in Concurrent-C) was designed to be a collection of parallel processes where each SIMNET vehicle simulator is mapped to an independent Concurrent-C process. The ETHERNET bus was simulated by a group of Concurrent-C processes representing the points of contacts with the TCL boxes.

Auxiliary server processes were used to accomplish the transfer of packets and simulate the propagation delay of the transmission medium.

* The Concurrent-C code of a general purpose time-scheduler for event-driven simulation (an AT&T product) was acquired. The code of this scheduler was extensively studied and the modifications needed for its adaptation to our SIMNET simulation were identified.

* The detailed review of the latest release of the IEEE 802.3 standards for the CSMA/CD access methods and protocols was completed. Our simulation programs were made to adhere to all specifications and protocols of the IEEE 802.3 standards.

* The review of the token-ring local area networks continued.

Months 7-8

* The coding of the simulation models for SIMNET under ETHERNET was completed and the effort for testing and collecting results was started. A copy of the documented listing of the Concurrent-C code of the SIMNET/ETHERNET simulation program is attached (Appendix A).

* A paper was written and presented in the first IST Networking Conference, April 1989. The paper is entitled "Simulation Networking and Protocol Alternatives" (authored by: M. Bassiouni, M. Georgiopoulos, and J. Thompson). The paper was also published in the Conference Proceedings. Copy of this paper is attached

(Appendix C).

* An extended abstract of a second paper was submitted to the 11th Interservice/Industry Training Systems (I/ITSC) Conference. The abstract was selected for paper preparation and we were invited to submit the complete paper by June 1989 for final review/selection.

Months 9-10

* Performance results of the SIMNET network under the CSMA/CD ETHERNET protocol were collected from the simulation models. Several runs covering a wide range of traffic load conditions were conducted and appropriate statistics were gathered.

* The coding of the simulation models for token-ring LANs was started. The Concurrent-C language was also selected for the implementation of simulation models of token-ring LANs and each node on the ring was mapped to an independent Concurrent-C process. Auxiliary server processes were used to accomplish the transfer of packets and simulate the propagation delay of the point-to-point connections of the ring.

* The full paper for the I/ITSC Conference was written and submitted for final review.

Months 11-12

* More performance results were collected from the ETHERNET simulation models.

* The development of the simulation models for token-ring networks was completed and performance results were collected. A copy of the documented listing of the Concurrent-C code of the token-ring SIMNET simulation program is attached (Appendix B).

* Our I/ITSC paper was accepted for presentation and publication in the 11th I/ITSC Conference Proceedings, November 1989. The paper is entitled "Real-Time Simulation Networking- Network Modeling and Protocol Alternatives" authored by M. Bassiouni, M. Georgiopoulos, and J. Thompson. Copy of this paper is attached (Appendix D).

2. Network System Configuration Models

Various choices exist for the implementation of a local area network (LAN), (e.g., transmission medium, topology, access protocols, etc.) to interconnect simulation devices. In this project, we have performed research aiming at modeling and analyzing the performance of two network configurations having different topologies and protocol access methods. The first configuration has a bus topology and uses the ETHERNET protocol which belongs to the class of Carrier Sense Multiple Access with Collision Detection (CSMA/CD) protocols. The second configuration has a loop topology and employs a non-contention protocol that avoids collision by a token-passing mechanism.

ETHERNET is a CSMA/CD protocol used for LANs with bus topology and is based on a distributed network control whereby each node on the bus determines its own channel access time based on the information collected by monitoring the traffic activities on the common bus. Figure 1 gives the bus network configuration for an ETHERNET-Based LAN. In this implementation, up to eight nodes can be connected through a multi-port transceiver to a single point on the coaxial cable (via a medium access unit). If a node on the bus has a packet ready for transmission, it first monitors the network to determine whether any transmission is in progress. If a transmission is in progress, the network is said to be "busy", otherwise it is "idle". If the node finds the network bus busy, transmission of its data packet is deferred until the bus becomes idle. The node waits a certain time interval (inter-frame gap) after the bus becomes idle and then starts the transmission of its packet. If multiple nodes attempt to transmit at the same time, their transmissions interfere resulting in "packet collision".

The collision is acknowledged by each transmitting node by it sending out a bit sequence referred to as a "jam signal". After the jam signal has been transmitted, the nodes involved in the collision schedule a retransmission attempt at a randomly selected time in the future. A packet is discarded if its transmission does not succeed (due to collisions) after sixteen consecutive transmission attempts. The performance of the ETHERNET protocol is directly related to how efficiently nodes avoid collisions and handle retransmissions. The problem of data collision is directly related to the network traffic load.

Figure 2 gives a block diagram of the basic configuration of the token-ring LAN. Simply stated, a token passing ring is a LAN with a loop topology in which a token (a unique bit sequence in a data packet) is passed around the network in a round-robin fashion from one node to the next. Contention for transmission is resolved by stipulating that only the node currently in possession of the token is allowed to transmit a frame or a sequence of frames onto the ring. When the transmission is finished, the token is passed to the node downstream which then gains the right to transmit. Since there is a single token on the ring, only one node can be transmitting at a time. Other (non-transmitting) nodes, however, continuously receive the bit stream, examine it and repeat it onto the network (i.e., place it on the medium to the next station). A station repeating the bit stream may copy it into local buffers or modify some control bits if appropriate.

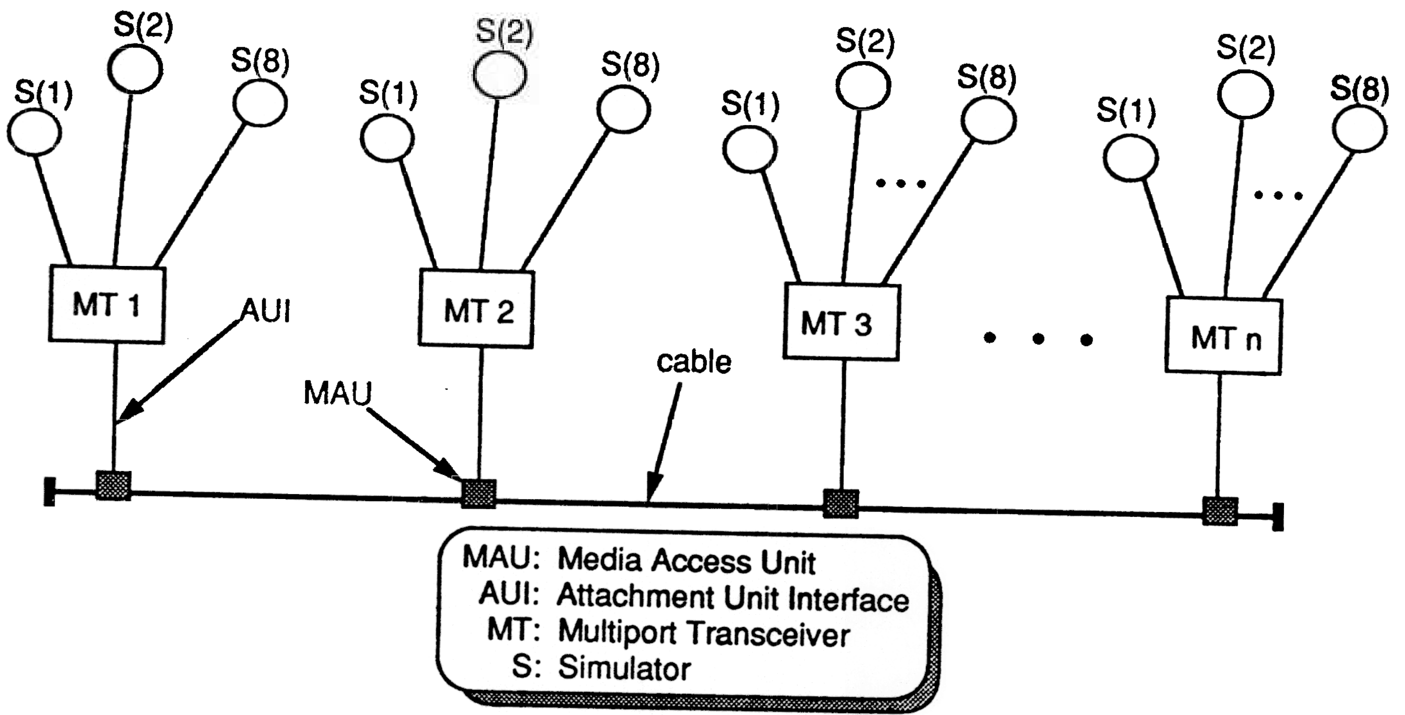


FIGURE 1. Bus Network Topology System Configuration

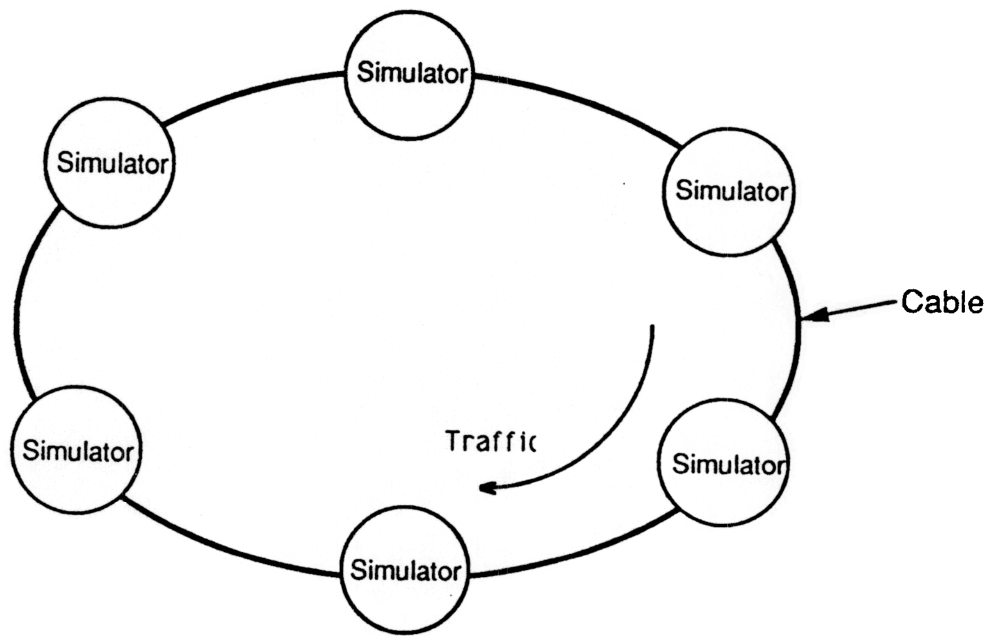


FIGURE 2. Ring Network Topology System Configuration

3. The SIMNET/ETHERNET Simulation Model

Below, we give a high-level description of the simulation model used in evaluating and predicting the performance of SIMNET with ETHERNET. The simulation model is written in Concurrent-C (an extension of the C language with concurrent programming facilities based on the "rendezvous" concept). The powerful synchronization and concurrency aspects of Concurrent-C have provided us with a notationally convenient and conceptually elegant tool for modeling the parallel activities of SIMNET nodes and the underlying networking layer. The complete listing of the simulation program for SIMNET with ETHERNET is given in Appendix A.

The process interaction model of Concurrent-C has been used in our simulation to map the different entities and activities of the simulated network to its corresponding Concurrent-C processes. The following process types are the major generic entities used in our simulation.

* Process "Simnode" is used to represent a vehicle simulator on the network. A process of this type is created for each such simulator.

* Process "Busnode" is used to represent the point of contact of each SIMNET node with the ETHERNET bus. A process of this type is created for each such point of contact on the bus.

* Process "Lserver" is used to implement and control the flow of data (packets and jam signals) in the direction from right to left for each SIMNET node. A process of this type is created for each SIMNET node.

* Process "Rserver" is analogously defined for traffic flowing in the direction from left to right.

* Process "Scheduler" is used to order time events and control the sequencing of activities of the entire simulation.

Figure 3 gives a block diagram showing the interactions among the different processes used in the SIMNET/ETHERNET simulation model. Typically, eight simulators connect to the coaxial transmission cable at a single point via a multi-port transceiver. Each of the simulators is modeled as a Simnode process. A Busnode process for each point of contact is created to accept and transmit local traffic from any one of the eight SIMNET nodes as well as retransmit any external messages arriving at this node. For this purpose, we use two separate processes called Rserver and Lserver. The Rserver process implements the transfer of data from its left Busnode process to its right Busnode process. This transmission is actually simulated by calling the Scheduler process to wait for the propagation delay (the time needed by the message to travel from one SIMNET node to the next). The Lserver similarly carries data signals from the right Busnode to its left neighbor. The Busnode process detects collision by checking for the existence of local traffic, left traffic or right traffic.

Below, we give a brief description of the different processes mentioned above. Although the complete code is given in Appendix A, we use high-level pseudo-code in this section to illustrate the important features and main activities of each process.

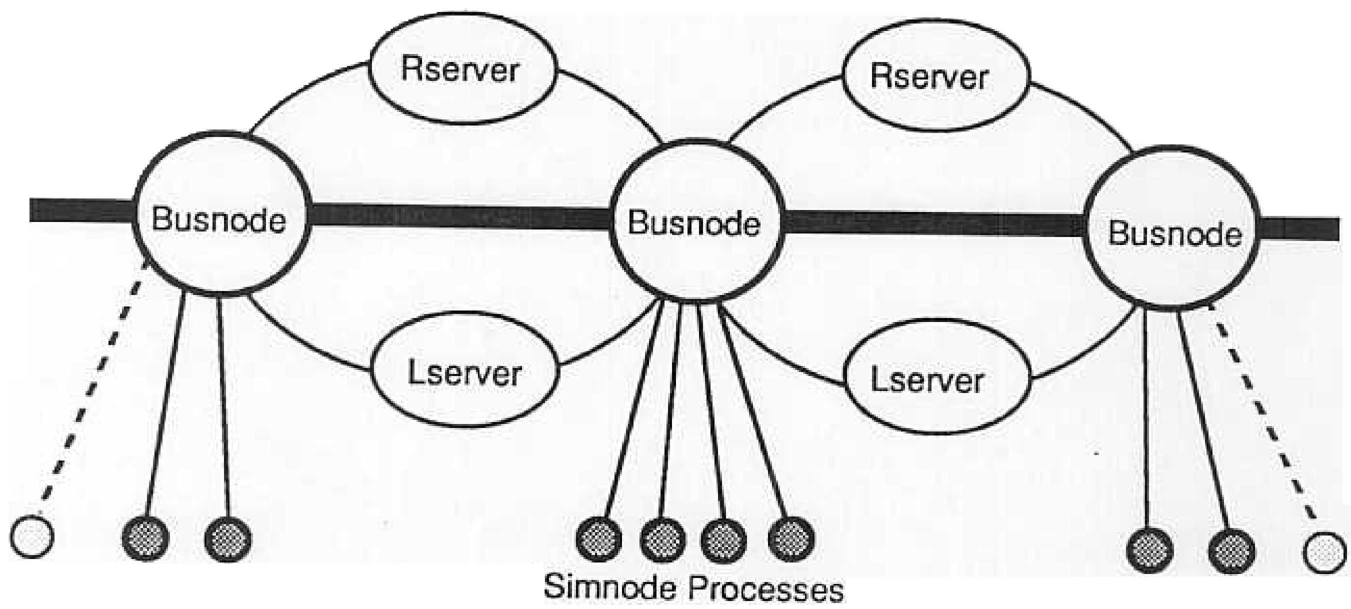


Figure 3. Simulation Model for Bus Topology Networks

The Simnode Process:

This process is the source of local traffic. It generates packets according to a specified input method such as using traces of real data or stochastically generated interarrival times (e.g., exponential, uniform, fixed with jitters, etc.) Upon the arrival of a local packet, the Simnode process makes a request to the corresponding Busnode process in order to transmit the new packet. This is done by calling a specific transaction in the Busnode process as illustrated by the pseudo code given later. At this point, the Busnode process checks for a carrier flag. If the flag has been off for at least the interframe gap, the Simnode process can proceed with its transmission. If the carrier flag is on, the Simnode process has to wait for the interframe gap then retry its request. When a collision is detected during transmission, the Simnode process sends a jam signal and increments the collision counter. This is followed by invoking a backoff algorithm for retransmission. A packet is discarded after 16 unsuccessful attempts for transmission. Also, if a new update message arrives before successful transmission, the old message is simply replaced by the new message. The specification and major activities of the Simnode process are described by the following code:

```
Process spec simnode (process sched s,
                   process bus bid,
          long meanIit, name_t name)

Process body simnode (s, bid, meanIit, name)
   /* Initialization phase */
```

```
    c_setname(c_mypid(), name.str);
    s.adduser(); bid.addProd();

/* Main processing phase */
while (simulation time not expired) do
  {
  /* Get next arrival time */
  t= erand(meanIit)
  /* Call Scheduler to wait for arrival */
  arrive =  s.wait(s.reqDelay(t));
Interrupt
  /* Attempt transmission */
  while ((dt = bid.transReq(c_mypid())!= 0)
       s.transDelay(dt);
  /* Code for collision check and
  subsequent backoff algorithm */
  collision_handler (collis_counter);
  /* Increment counter for successful
  or discarded packets */
  sst.success += 1;
  }
/* Termination phase */
statistic_fun();
```

The Busnode Process

The Busnode process acts like a server process ready to accept transaction calls from the local Simnode processes, the Lserver processes or the Rserver processes. The Busnode is responsible for detecting collisions and it continuously monitors the carrier flag to see if it is busy. In the case of a collision, the Busnode process calls the Scheduler to awaken the transmitting Simnode process which then stops the transmission and sends the jam signal. The following code gives the Concurrent-C specification and body of the Busnode process.

```
Process spec busnode (process sched s,
                      process statProc stp,
                      float simt,
                        name_t name)

Process body busnode (s, stp, simt, name)
  {
   /* Initialization phase */
   s.addUser(); s.passive();
   /Main Processing Phase */

   while (# of producers + consumers > 0)
    select {
```

```
/* Handle request from left or right servers */
accept putReq();
/* Make sure that a Simnode which detect
collision gets a slot in the waiting
queue of the scheduler */
or accept insure();
/* Take request from left or right server */
or accept takeReq();
or accept takeWait();
/* Local transmission request from Simnode */
or accept transReq();
/* Terminate local transmission */
or accept stopTran();
/* Increment consumer or producer count */
or accept addCons();
or accept add Prod();
/* Decrement consumer or producer count */
or accept dropCons();
or accept dropProd();
/* Update local Simnode count */
or accept addLoProd();
or accept dropLoProd();
} /* end of while */
```

```
/* Get performance statistics */
s.active(); s.dropUser();
} /* end of Busnode body */
```

The Rserver and Lserver Processes

These processes transmit the traffic delivered to the Busnode process by any transmitting Simnode process to the right and left, respectively. The specification and body of the Rserver process are given below.

```
Process spec Rserver(process sched s,
                     process bus inbus,
                     process bus outbus)

Process body Rserver(s, inbus, outbus, name)
{
typedef struct{ /* data submitted by simnode */
    /* Time of arrival */
    long arrive;
    /* Packet length */
    int packet_length ;
    /* No. of update messages per sec */
    int update_num;
    /* No. of attempts to transmit */
    int attempt_index;
```

```
        }       local_traffic

    /* Initialization phase  */
        c_setname(c_mypid(), name.str);
        s.adduser(); inbus.addCons();
        outbus.addProd();
   /* Main processing phase */
      while (takereq(1){
         /* Wait for propagation delay */
         t = arrivetime + propagation delay;
         ts = s.wait(s.reqDelay(t));
         /* Deliver message */
         put_FLTR(type);
         }
```

The body and specification of the Lserver process are similar to those of Rserver.

The Scheduler Process

Delays in the simulated network (such as transmission delays) are handled by the Scheduler process. This process maintains the simulated clock and advances it appropriately. For each delay request from a process, the Scheduler determines the time when the process needs to be reactivated and saves this time in an "activation request" list. When all processes are waiting, the scheduler picks the next process to run, advances the simulated clock and reactivates the process. The simulated clock

advances only when all processes are waiting; thus any (non-delay) computation done by a process takes place in zero simulated time. At any given moment, each client process is in one of the following three states:

* Waiting: for an explicit delay request from the scheduler;
* Active: computing in zero simulated time;
* Passive: waiting for an event other than a delay request from the Scheduler.

The specification and the body of the Scheduler are given below:

```
process spec sched() {
/* Return current simulated time */
  trans long now();
  /* Request a delay */
  trans long reqDelay(long);
  /* Wait for reqDelay */
  trans long wait(long);
  /* Add or delete client process */
  trans void adduser(),dropuser();
  /* Change client to new state */
  trans void passive(), active()

  /* Handle collision */
  trans void collision(id);
      };
```

```
typedef struct {
   /* Structure describing a delay request */
   long ts;   /* time stamp */
   int next; /* index of next entry or -1 */
  /* Number of clients waiting for this time */
   int nwait;
       }   reqent ;

  static reqent Rtab[MAXREQ];
  static int Ifree; /* first free entry */
  /* Ihead is entry with lowest timestamp */
  static int Ihead =-1;

  process body sched()
{ int nclients, nactive, i;
   long curtime = 0;
  /* Initialization phase */
    c_setname(c_mypid(),"sched");
    rqInit();
    accept adduser() {nclients = nactive = 1}
  /* Main processing phase: accept requests while
     clients exist */
   while (nclients >0) {
```

```
    select{
     accept adduser() {nclients += 1; nactive += 1;}
     or
     accept dropuser() {nclients -= 1; nactive -= 1;}
     or
     accept passive()  {nactive -= 1;}
     or
     accept active() {nactive += 1;}
     or
     accept now()     {treturn curtime;}
     or
     accept reqDelay(x)
      {nactive -= 1; treturn addreq(curtime+x);}
     or
     accept jam(id)
      {change timestamp of record with this id}
          }
/* If all clients are waiting, find the first event */
/* and allow all clients waiting for it to proceed */
if (nactive == 0 && Ihead != -1){
    curtime = Rtab[Ihead].ts;
    nactive = Rtab[Ihead].nwait;
While (--Rtab[Ihead].nwait >= 0)
```

```
accept wait(key) such that (key == Ihead)
    {treturn curtime;}
```

4. The Token-Ring Simulation Model

The simulation model for ring LAN topology is also written in Concurrent-C. A functional diagram of the simulation model for SIMNET with the token-ring LAN is shown in Figure 4. The following process types are the major generic entities used in our simulation for the ring structure.

* Process "Simnode" is used to represent a vehicle simulator on the network. A process of this type is created for each such simulator. This process is the source of local traffic and is capable of generating packets according to a specified input method (e.g., using traces of real data or random stochastically generated interarrival times such as exponential, uniform, fixed with jitters, etc.)

* Process "Ringnode" is used to monitor the ring traffic at the location of each node on the ring. A process of type Ringnode is created for each node on the ring. This process is responsible for implementing the token-based medium access protocol.

* Processes "Server" is used to simulate the propagation delay between each pair of LAN nodes. A process of this type is created for each pair of adjacent nodes on the ring.

Figure 4. Simulation Model for Ring Topology Networks

5. Performance Results

The detailed simulation models described earlier have been used to gain insight into the performance of simulation networks under both the ETHERNET and token-ring protocols. In what follows, we summarize the results of the numerous performance tests and the statistics gathered by the simulation models.

5.1. Performance of ETHERNET

The tables given on the following pages present some selected statistics gathered when the CSMA/CD ETHERNET configuration is driven by 80 SIMNET simulators. The assumptions made for the configuration used in these tables are given below.

* Each simulator uses an exponential stochastic distribution for packet arrivals with a cumulative rate that drives the network at various levels of total traffic load for the entire LAN.
* The simulation uses a total of ten multi-port transceivers. A multi-port transceiver is used to connect eight SIMNET nodes to a single point on the coaxial cable.
* The speed of data propagation in the coaxial cable is assumed to be the speed of light.

ETHERNET STATISTICS

80 Simulators

Measure	Value
Average LAN load	5000 frames/sec
Packets discarded	0.0%
Max. Attempt	9
Avg. Attempt	1.47
Throughput	5000 frames/sec
Utilization	52.5%
Avg. Packet Delay	270.7 microsec

ETHERNET TATISTIC:

80 Simulators

Measure	Value
Average LAN load	6667 frames/sec
Packets discarded	0.0%
Max. Attempt	16
Avg. Attempt	2.45
Throughput	6667 frames/sec
Utilization	74.3%
Avg. Packet Delay	5498.0 microsec

ETHERNET STATISTICS

80 Simulators

Measure	Value
Average LAN load	7200 frames/sec
Packets discarded	0.026%
Max. Attempt	16
Avg. Attempt	2.81
Throughput	6912 frames/sec
Utilization	76.3%
Avg. Packet Delay	7842.3 microsec

ETHERNET STATISTICS

80 Simulators

Measure	Value
Average LAN load	8000 frames/sec
Packets discarded	0.283%
Max. Attempt	16
Avg. Attempt	3.11
Throughput	7044 frames/sec
Utilization	78.8%
Avg. Packet Delay	21469.9 microsec

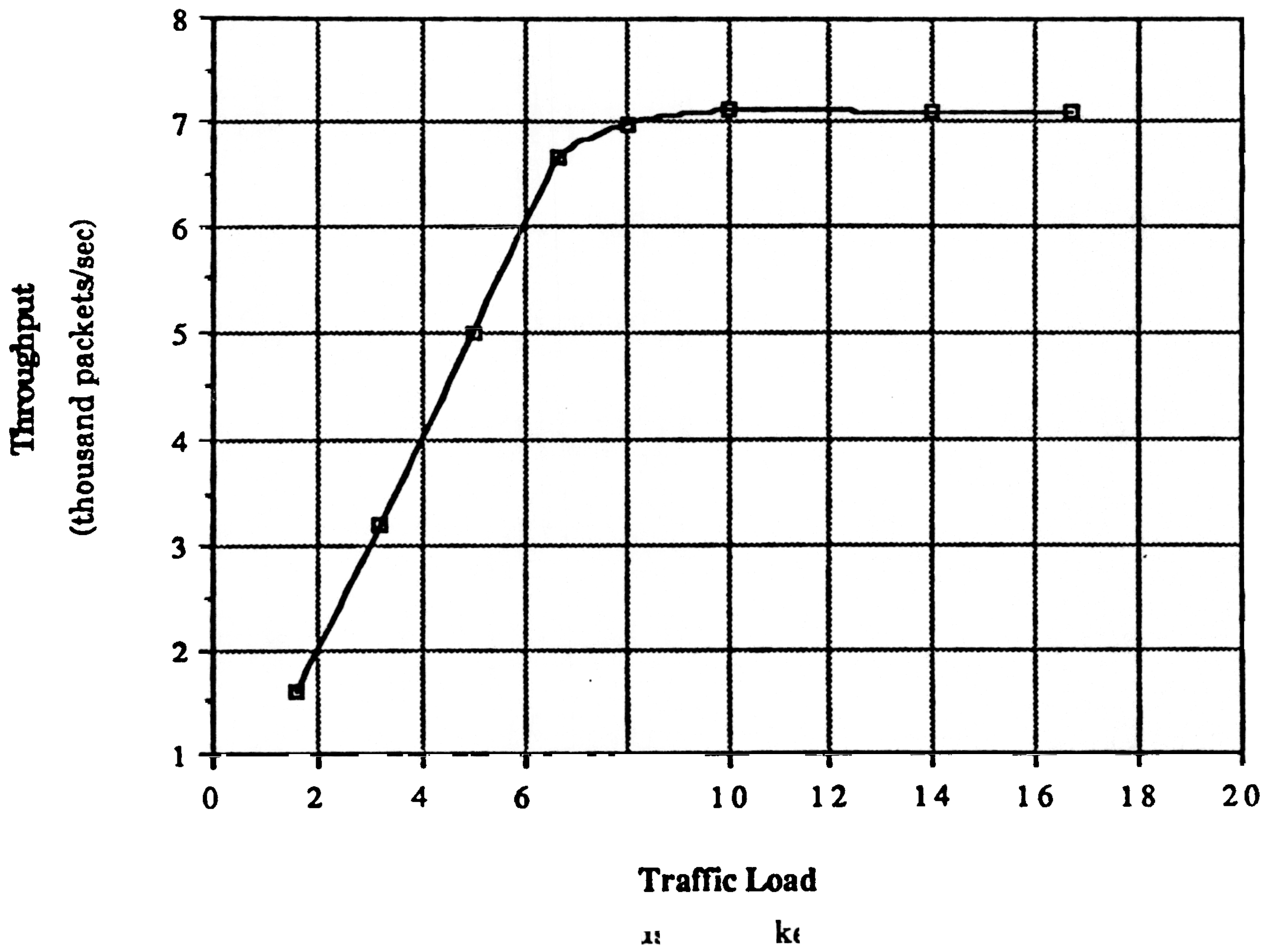
ETHERNET TATISTIC

80 Simulators

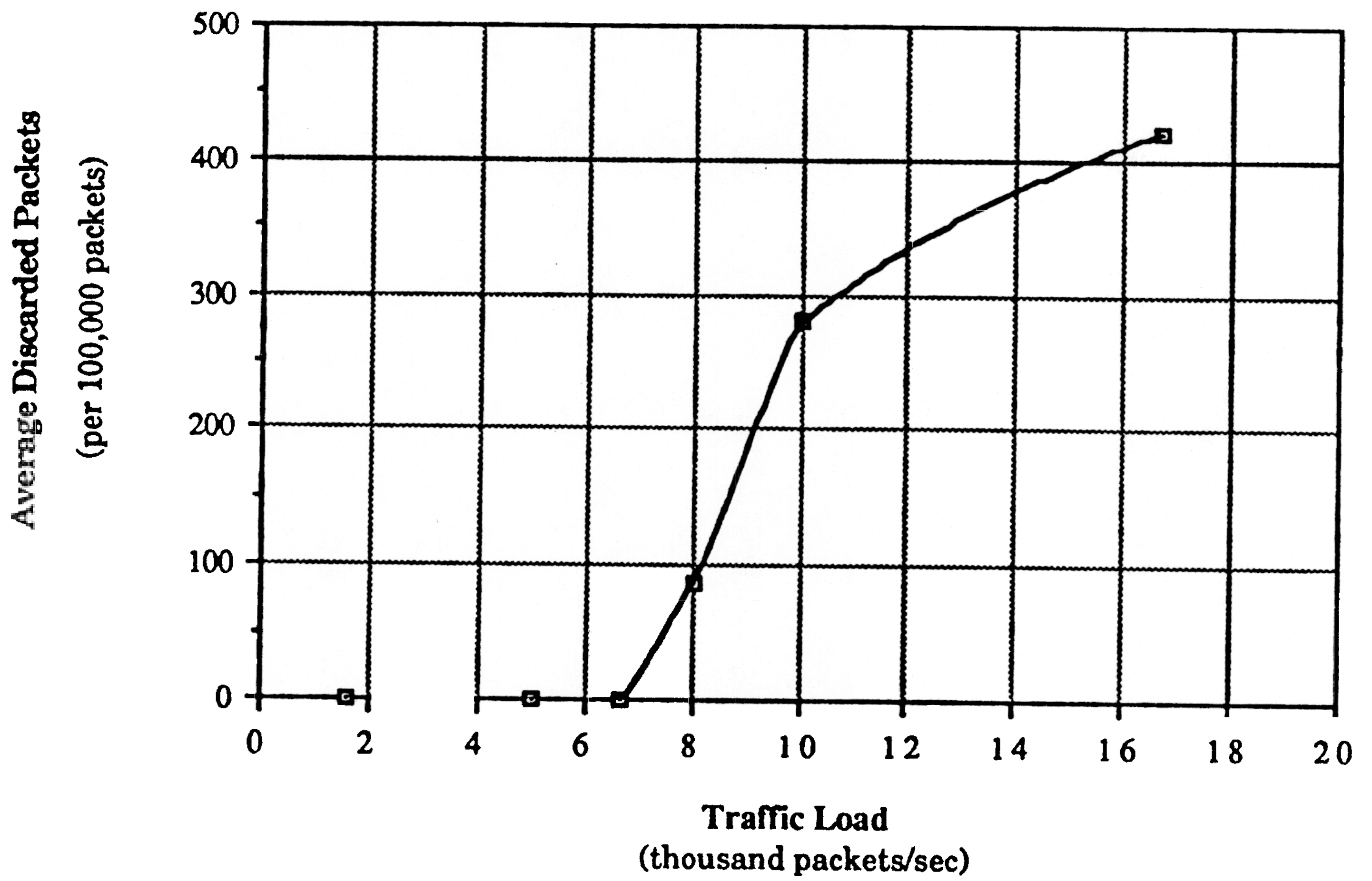
Measure	Value
Average LAN load	9000 frames/sec
Packets discarded	0.534%
Max. Attempt	16
Avg. Attempt	3.36
Throughput	7119 frames/sec
Utilization	80.5%
Avg. Packet Delay	33206.2 microsec

One of the configurati that considered in ETHERNET tests is unique to the applicati of local networks to the interconnection of simulation training devices. In this configuration, optimizati made to reduce the load the access medium. An explanation of this optimization for the ETHERNET case is given below Upon state hange, simulator (node the twork sends the val of its state to other nodes the LAN. Each new sta results in the generation of new packet at the application lay (i.e at the node level). The packet is then bmitted to the data link layer in order to start the process of its transmission. In ETHERNET only packet per node is delivered for transmission at time. Other packets are normally queued up at the application level (i.e., the node level) waitin for the end of the ongoing attempt of transmission. In this context, the arrival of packet (carrying the most recent state of the node) be simply used replace the previous packet stored at the application layer) which holds less recent state The discarding of th old (obsolete) packet helps speed up the transmission of the latest state of node Notice that the packet already submitted to the ETHERNET data link layer is not affected by arri val; (this packet is under the control of ETHERNET protocol boards and is accessible from the application layer).

The performance of this ETHERNET configuration is give in Graphs and 2. and in several tables givin detailed results of some individual runs. Graph gives the relationship between the throughput of the LAN and the total initial traffic load from all simulators (i.e., before any discardin at the applicati level) Graph gives the relationship between the total initial load and the packets discarded by ETHERNET as result of exceeding the maximum count for transmission attempts (due exces- collisions). Notice that Graph gives the packets discarded by the ETHERNET protocol and not the obsolete packets discarded at the application level.



Graph 1. Throughput



Graph 2. Average Discarded Packets vs Traffic Load (ETHERNET)

ETHERNET STATISTICS

80 Simulators

Measure	Value
Average LAN load	5000 frames/sec
Packets discarded	0.0%
Max. Attempt	8
Avg. Attempt	1.38
Throughput	5000 frames/sec
Utilization	52.7%
Avg. Packet Delay	232.4 microsec

ETHERNET STATISTICS

80 Simulators

Measure	Value
Average LAN load	6667 frames/sec
Packets discarded	0.0%
Max. Attempt	14
Avg. Attempt	2.25
Throughput	6667 frames/sec
Utilization	72%
Avg. Packet Delay	1029.7 microsec

ETHERNET STATISTIC

80 ımulators

Measure	Value
Average LAN load	7200 frames/sec
Packets discarded	0.0%
Max. Attempt	14
Avg. Attempt	2.50
Throughput	6842 frames/sec
Utilization	74.6%
Avg. Packet Delay	1594.0 microsec

ETHERNET STATISTICS

80 Simulators

Measure	Value
Average LAN load	8000 frames/sec
Packets discarded	0.086%
Max. Attempt	16
Avg. Attempt	2.88
Throughput	6986 frames/sec
Utilization	77.3%
Avg. Packet Delay	2682.9 microsec

ETHERNET STATISTICS

80 Simulators

Measure	Value
Average LAN load	9000 frames/sec
Packets discarded	0.159%
Max. Attempt	16
Avg. Attempt	3.04
Throughput	7045 frames/sec
Utilization	78.4%
Avg. Packet Delay	3678.9 microsec

ETHERNET TATISTIC

80 Simulators

Measure	Value
Average LAN load	10000 frames/sec
Packets discarded	0.280%
Max. Attempt	16
Avg. Attempt	3.15
Throughput	7118 frames/sec
Utilization	79.7%
Avg. Packet Delay	4183.7 microsec

ETHERNET STATISTICS

80 Simulators

Measure	Value
Average LAN load	13333 frames/sec
Packets discarded	0.421%
Max. Attempt	16
Avg. Attempt	3.46
Throughput	7093 frames/sec
Utilization	80.6%
Avg. Packet Delay	5571.4 microsec

5.2. Performance of Token-Ring

Numerous tests have been performed using the token-ring simulation program given in Appendix B. In this section, we present the results of token-ring LANs under the following assumptions:

* The recreation of the free token (F.T.) onto the ring is assumed to follow the "early token release protocol." According to this protocol, the transmitting station (the one which removed the free token from the ring) recreates the free token and puts it onto the ring immediately after it finishes transmitting its packet. This protocol results in better LAN throughput and smaller packet delays than protocols that require the header (or the tail) of the transmitted packet to complete one cycle around the ring before the free token is recreated.
* The length of the free token has been used as a variable whose value ranged from 1 bit to 24 bits (the case of 1 bit is theoretical in nature and is given to help evaluate the overhead of token management).
* To make this model correspond closely to the ETHERNET configuration, LAN concentrators were used. Each concentrator was assumed to connect 8 SIMNET nodes to the token-ring (similar to the TCL box used in the ETHERNET simulation). Delays in these concentrators were assumed to be negligible. We have also conducted tests without the use of concentrators (but the results are not discussed in this report).

The tables given on the following pages present the numerical results of throughput, average packet delay, and the F.T. cycle time for different values of F.T. length. Although the results show that at high loads token-ring LANs have packet delays that are order of magnitude better than those of ETHERNET, the above assumptions have undoubtedly contributed to widening the performance gap between the two models. Using "late free token release" policy and increasing the F.T. length would certainly result in an increase in the value of packet delays and would therefore reduce the performance gap between the two configurations.

TOKEN-RING STATISTICS

Length of Free Token= 8 bits

Load (packets/sec)	Throughput (packets/sec)	Packet Delay (microsecond)	Free-Token Cycle Time (microsecond)
235	235	2.95	1.65
1,025	1,025	2.44	1.79
2,075	2,075	19.36	2.03
2,733	2,733	19.45	2.22
6,100	6,100	42.30	4.24
6,400	6,400	47.36	4.65
8,600	8,600	178.90	13.96
10,000	9,500	660.09	52.77

TOKEN-RING STATISTICS

Length of Free Token= 24 bits

Load (packets/sec)	Throughput (packets/sec)	Packet Delay (microsecond)	Free-Token Cycle Time (microsecond)
140	140	1.03	2.43
1,075	1,075	6.51	2.70
2,400	2,400	14.35	3.20
3,100	3,100	26.42	3.54
5,150	5,150	54.78	5.16
6,083	6,083	102.75	6.50
8,000	8,000	166.21	14.13
8,800	8,300	450.28	16.98

TOKEN-RING STATISTICS

Length of Free Token= 1 bit

Load (packets/sec)	Throughput (packets/sec)	Packet Delay (microsecond)	Free-Token Cycle Time (microsecond)
210	210	0.74	1.64
2,100	2,100	5.95	2.04
3,400	3,400	31.67	2.45
5,100	5,100	64.09	3.29
9,300	9,300	310.65	29.50

6. Conclusions

In this section, we present the conclusions of this project concerning the evaluation of ETHERNET and token-ring protocols and their suitability for real-time simulation networks.

6.1. Comparative Results of the Simulation Models

* The numerous simulation tests that we have conducted show that the throughput of SIMNET with the ETHERNET protocol reaches a maximum of approximately 60-70% of the transmission medium bandwidth. As explained below, the saturation in throughput is primarily due to the excessive collision rate that characterize the behavior of ETHERNET at high loads. The token-ring configuration, however, has consistently yielded maximum throughput in the range 90-95% of the transmission medium bandwidth. The token-ring configuration uses a collision-free protocol and does not therefore suffer from the problem of declining throughput at high loads.

* Our simulation results show that ETHERNET is an excellent choice for lightly loaded networks. For example, loads of 10% to 20% of ETHERNET bandwidth represent ideal condition for the ETHERNET medium access protocol (very small collision rate, negligible packet delays, and zero packet loss rate due to excessive collision count). As the traffic load on the ETHERNET bus increases, the performance of the network deteriorates quickly resulting in significant increases in packet delays. With further higher loads, some packets are lost due to exceeding the limit of retransmission attempts, and the performance of ETHERNET rapidly

collapses causing packet delays to become too large to be acceptable for real-time simulation.

* Token ring LANs, on the other hand, are much less sensitive to increased transmission rates compared to ETHERNET. Unlike collisions in ETHERNET, the overhead of token management in ring LANs does not result in throughput decline as the traffic load on the LAN increases. Therefore at very small loads, the time overhead of token management in ring LANs would be more costly (in terms of packet delays) than the overhead of collision handling in ETHERNET LANs. As the traffic increases, collision rates in ETHERNET become significant while the token-passing algorithm shows a more stable behavior. Since token rings are collision free, a maximum packet delay can be guaranteed for a given number of stations. Thus the real-time requirements of applications having high traffic loads (e.g., networks with large number of simulation training devices) can be handled more gracefully using the contention-free token ring scheme.

6.2. Other Considerations

In addition to the comparative results obtained by the simulation models, many other factors need to be considered when choosing and/or designing a medium access protocol for simulation networks. Below, we discuss some important considerations relevant to the comparison of ETHERNET and token-ring LANs.

* Unlike ETHERNET, token rings provide a priority-based scheme for packet transmission across the network. In the ANSI/IEEE 802.5 ring implementation, the passing token has three bits indicating the current priority level of the ring (this gives 8 priority levels: 0 is the lowest priority and 7 is the highest priority). A station that captures the token, can only transmit packets whose priority is equal to or higher than the priority of the passing token. The 802.5 protocol also provides mechanisms that enable stations to request/change the priority of the passing token. In simulation networks, this means that it will be possible to assign priorities to the different types of messages in order to optimize real-time performance and visual display at peak load conditions.

* Because of its point-to-point connection property, rings readily accommodate the use of optical fiber as a transmission medium. In addition to offering, reduced size/weight and enhanced safety features, optical fiber also offers very high signal bandwidth. One very promising implementation of ring networks using optical fiber is the Fiber Distributed Data Interface (FDDI). FDDI is a 100 Mbits/sec token ring LAN protocol that is rapidly becoming accepted as the premier high speed LAN standard. With its embedded extensibility to support even higher speeds (500 to 1000 Mbits/sec), FDDI is poised to become the dominant high-end LAN of the 1990's. The paradigm for FDDI topology is known as a "dual counter-rotating ring of trees". The physical layer topology consists of independent, full-duplex, point-to-point physical connections, while the logical layer consists of one or two rings. The FDDI MAC (medium access control) protocol provides data services similar to those of tl.e IEEE 802.5 token rings. An extension

to FDDI (called FDDI II) is currently being investigated to add an isochronous data transmission capabilities to the network, thus enabling it to handle both voice and data. FDDI technology will eventually provide the simulation/training industry with powerful real-time LANs capable of interconnecting an unprecedented number of stations.

* One main advantage of the bus structure over ring LANs is the reliability of network operation following a node failure. Bus-based LANs are persistent to node failures since the propagation of messages on the bus does not require the participation of any given node. A failure of a station on the ring structure, however, can bring the entire LAN down. This problem has been considerably reduced by the increased reliability of today's ring chips and off-the-shelf ring attachments. Furthermore, FDDI rings use optical bypass switches in order to allow inactive (off-line) stations to pass the traveling data-carrying light waves directly from one neighbor to the next without active power.

* Bus-based ETHERNET LANs have enjoyed economic advantages because of their widespread use in the past two decades. These advantages, however, are about to disappear since VLSI technology, fiber optics, and other near-term advances will soon be supplying the market with ring chips at the same low cost as bus chips. Also, hardware support for FDDI is rapidly growing and the projected increased development/installation investments in FDDI are expected to drive the cost of FDDI hardware by as much as 30% per year.

Acknowledgements

I would like to thank the three graduate assistants: Samir Chatterjee, Ming Chiu and Hon Chu for their help in coding/debugging the Concurrent-C simulation programs and for performing the numerous evaluation tests of the ETHERNET and token-ring LANs. Special thanks go to Mr. Jack Thompson, Dr. Michael Georgiopoulos, and Mr. Jorge Cadiz for their technical support during the course of this project. I would also like to thank Mr. Ernie Smart for his guidance and administrative assistance. Finally, I am indebted to the Department of Computer Science and the Institute for Simulation and Training for providing the facilities needed to carry out this project.

Appendix A

Source Code Listing of the ETHERNET Simulation Model

```
/*****************************************************************/
/*****************************************************************/
/* This is a header file. It contains declarations & specification  */
/* part of each of the concurrent C processes.It also contains some */
/* structures and functions that are used in the simulation.        */
/*                                                                  */
/*****************************************************************/
/*****************************************************************/

#define FRTL      0    /* FRTL stands for "from right to left "        */
#define FLTR      1    /* FLTR stands for "from left to right "        */
#define T_NORMAL  1       /* mode=1, transmission terminated normally */
#define T_COLLIDE 2       /* mode=2, terminatrion after collision     */
#define T_START   3       /* mode=3, start transmission               */
#define T_NOWAIT  4       /* server takes message without waiting     */
#define BUSY      1       /* the medium is busy */
#define IDLE      0       /* the medium is idle */


/* The type name_t is a structure containing a character string     */
/* and is passed by value to any process that calls it.             */

typedef struct {char str[20];} name_t;

/* stats is a structure for obtaining statistics.                   */
typedef struct {        /* Statistics for items */
    long        nv;             /* number of values */
    float       maxv;           /* max value */
    double      sumv;           /* sum of all values */
    double      sumsq;          /* sum of squares */
    } stats;

/* sItem is a structure for keeping the overall transmission stats.
   at a simnode and the whole system as well. */
typedef struct {
    int         nSend;      /* number of packets sent */
    int         success;    /* number of successful transmission */
    int         discard;    /* number of packets failed to transmit */
    int         maxAt;  /* maximum attempts required to transmit */
    float       sumAt;  /* sum of attempts made by each packet */
    } sItem;

/* This is the specification part of the time scheduler            */

process spec sched(float simt) {/* Virtual time scheduler process */
    trans float  now();          /* return current simulated time */
    trans long  reqDelay(float, process simnode); /* request a delay */
    trans float  wait(long);     /* wait for delay request */
    trans void  addUser();       /* add a client process */
    trans void  dropUser();      /* delete a client process */
    trans void  passive();       /* active client became passive */
    trans void  active();        /* passive client became active */
    trans void  jam(process simnode); /*  transmit jam signal after collision */
    };

/* This is the specification part of statistical process which keeps track */
/* of system parameters such as average delay,utilization,throughput and   */
/* mean interarrival times.                                                 */

process spec statProc(float simt) {
    trans void  addCus(), dropCus(); /* add, drop a client process */
    trans void  doSystime(stats);    /* get delay stats. of whole system. */
    trans void  doIit(stats);   /* get interarrival time stats. */
    trans void  doSst(sItem);   /* get transmission stats. */
    trans void  doUtiliz(float); /* get network utilization stats. */
    };
```

```
/* specification for busnode process    */

process spec busnode(process sched s, process statProc stp,
                        float simt, name_t name) {
    trans void  addProd(), dropProd(); /* add/delete a producer */
    trans void  addCons(), dropCons(); /* add/delete a consumer */
    trans void  addLoProd();           /* add local producer (simnode) */
    trans void  dropLoProd(sItem, stats, stats); /* drop local producer */
    trans int   putReq(int, int);      /* initiate put request */
    trans void  insure(int, int);
    trans int   takeReq(int);          /* initiate take request */
    trans int   takeWait(int);         /* complete take request */
    trans float tranReq(process simnode);/* request by simnode to transmit */
    trans void  stopTran(int);/* stop transmission due to collision */
    };
/* specification for simnode process.  */

process spec simnode( sched s, process busnode bid,
                float meanIit, name_t name);

/* function call by server to request a packet from busnode */
int     qTake(process busnode, int, int*);/* function to take an item */

/* specification for right server ,it carries the messages from left */
/* busnode to its right neighbor .                                    */

process spec Rserver(process sched s, process busnode inbus,
                process busnode outbus, name_t name);

/* specification for left server,it carries the messages from right */
/* busnode to its left neighbor .                                    */

process spec Lserver(process sched s, process busnode inbus,
                process busnode outbus, name_t name);

void    stInit(), stVal(); /* functions to initialize & add a value */
void    stMerge(); /* merge statistics contained in two stats structure */
void    sstInit(); /* initiate sItem structure */
void    sstMerge(); /* merge statistics contained in two sItem structure */
double  stMean(), stSdev();/* to get mean & std. deviation */
double  drand01();              /* return random number uniform in [0,1] */
double  erand();                /* return exponential random number */
```

```
/*****************************************************************************/
/* The main process creates all the processes that simulate the entities    */
/* of ETHERNET model for SIMNET.   It assigns unique names to each process  */
/* and also supplies appropriate values for system parameters either by     */
/* default or command line arguments. The input parameters are:              */
/*                                                                           */
/*          simt :simulation time desired to run the program;               */
/*          iit  :inter arrival time ;                                       */
/*         seed  :for random number generation;                              */
/*                                                                           */
/*****************************************************************************/

#include "dcls.h"
#include "stdio.h"
int c_nprocs = 500;

/* makeName function returns names for each of the processes. */
/* Return the char* argument as a name_t structure.             */
name_t makeName(narg, name, ch1, ch2)
        int     narg;
        char    *name;
        char    ch1, ch2;
{       name_t  ret;
        int     i;

        i = 0;
        while ((ret.str[i] = name[i]) != '\0')
                i++;
        ret.str[i++] = ch1;
        if (narg == 3)
                ret.str[i++] = ch2;
        ret.str[i] = '\0';
        return ret;
}

        /* Main process -- create all simulation processes. */
main(ac, av)
        int     ac;
        char    *av[];
{       process sched   s; /* process variable for sched */
        process statProc stp;  /* process variable for stats process */
        process busnode b[21]; /* array of busnodes */
        float   simt = 20000000., iit = 1200000.0;
        int     seed = 12345;
        int     i, j;
        int     nBusnode = 10;

        if (ac > 1) sscanf(av[1], "%f", &simt);
        if (ac > 2) sscanf(av[2], "%f", &iit);
        if (ac > 3) sscanf(av[3], "%d", &seed);
        srandom(seed);
        printf("main: simt %10.1f iit %7.1f seed %d nBus %d\n",
                simt, iit, seed, nBusnode);

    /* create virtual time scheduler */
        printf("Creating sched!\n");
        s = create sched(simt);

    /* create process that gathers statistics */
        stp = create statProc(simt);
        s.addUser();            /* for main */

    /* create busnodes */
        printf("Creating busnode!\n");
        for (i=1; i<=nBusnode; i++)
            b[i] = create busnode(s,stp,simt,
```

```
                               makeName(2,"Busnode",'A'-1+i,'0'));

    /* create left and right servers */
    /* first, create the leftmost Lserver */
        printf("Creating servers!\n");
        create Lserver(s, b[1], c_nullpid, makeName(3,"Lserv",'A','0'));
        for (i=1; i<=nBusnode-1; i++) {
            create Rserver(s,b[i],b[i+1],makeName(3,"Rserv",'A'+i-1,'A'+i));
            create Lserver(s,b[i+1],b[i],makeName(3,"Lserv",'A'+i,'A'+i-1));
        }
    /* last, create the rightmost Rserver */
        create Rserver(s, b[nBusnode], c_nullpid,
                makeName(3,"Rserv",'A'+nBusnode-1,'0'));

    /* create simnodes, each busnode connects up to eight simnodes (similar
       to a TCL box) */
        printf("Creating simnodes!\n");
        for (i=1; i<=nBusnode; i++)
            for (j=1; j<=8; j++)
                create simnode(s, b[i], iit,
                                makeName(3,"Sim",'A'+i-1,'0'+j));

    /* wait for all processes to tell sched they are clients */
        delay 2.0;
    /* all started -- can delete main as scheduler client. */
        s.dropUser();
        printf( "Main.cc terminate!\n");
```

```
/*********************************************************************/
/* SIMNODE is a process type that simulates the transmitting stations */
/* in the SIMNET configuration. When a node has nothing to transmit, it*/
/* monitors the transmission  medium by sensing a carrier signal.      */
/* Whenever the medium is busy, the node defers to the passing frame by*/
/* delaying any pending transmission of its own. After the last bit of */
/* the passing frame, the simnode continues to defer for a proper inter*/
/* frameSpacing. Now at the end of this interval, if a frame is waiting*/
/* to be transmited, transmission is initiated. When transmission      */
/* has completed, simnode resumes its original monitoring of carrier   */
/* sense.                                                              */
/* Collision detection and handling is also done by simnode.           */
/*                                                                     */
/*                      VARIABLE and FUNCTION  DICTIONARY              */
/*                      ---------------------------------              */
/* SLOT_TIME          :unit of time for collision handling;            */
/* packetTime         :time to transmit a packet of 1024 bits;         */
/* s                  :process scheduler ;                             */
/* bid                :process busnode to which simnode is connected;  */
/* simid              :identification of the simnode process type;     */
/* waitcarrier        :a procedure which defers for the proper         */
/*                     interFrameSpacing before transmission;          */
/* backOff            :a procedure which computes the retransmission   */
/*                    schedule due to collision backOff;               */
/* count              :integer which keeps track of the number of      */
/*                    retransmission attempts;                         */
/* range              :integer between 0 and 2^count;                  */
/* timeout            :integer multiple of SLOT_TIME to delay;         */
/* float t1,t2        :floating point variables to hold exponential    */
/*                     times;                                          */
/* float arrive       :time that the locally generated packet arrives  */
/* int tranCount      :integer to count the number of transmission     */
/*                     attempts due to collision;                      */
/* stats iit,sysTime :two variables of stats structure, one to keep    */
/*                     statistics for inter arrival time and the othe  */
/*                     to gather statistics for average delay;         */
/*********************************************************************/

#include "dcls.h"
#include "stdio.h"
#define SLOT_TIME 512.0
#define packetTime 1024.0

/************************************************************************/
/* waitCarrier is a procedure which the simnode calls to defer itself     */
/* before transmitting its packets. This procedure calls a transmission   */
/* request transaction of its corresponding busnode, and thereby delays   */
/* itself for the appropriate interFrameSpacing .If the channel is busy   */
/* then the packet is kept in the pending list of the busnode.            */
/************************************************************************/

static float waitCarrier(s, bid, simid)
        process sched s;
        process busnode bid;
        process simnode simid;
{       float dt;
        float curT = -1.;
        /* makes a transmission request to the busnode */
        while (curT != 0. && (dt = bid.tranReq(simid)) != 0.)
            curT = s.wait(s.reqDelay(dt, c_nullpid));
        return curT;

/************************************************************************/
/* When a transmission attempt has been aborted due to collision, it is   */
/* retried by the transmitting simnode, until either it is succesful or a */
```

```
/* maximum number of attempts have been made without success.              */
/* Truncated binary exponential backoff process is used. At the end of     */
/* a collision it delays an integer multiple of SLOT_TIME. The number      */
/* of SLOT_TIMES to delay before the nth. transmission attempt is chosen   */
/* as a uniformly distributed random integer r in the range                */
/* 0 <= r <= 2^k where k = min(n.10);                                      */
/**************************************************************************/

static float backOff(count)
        int count;
{       float timeout;
        int range;
        int i, k;

        if (count > 10)
                k = 10;
        else
                k = count;
        range = 1;
        for (i=0; i<k; i++)
                range = 2 * range;
        timeout = SLOT_TIME * range * drand01();
        return timeout;
}

/* Main body of the simnode process     */

process body simnode(s, bid, meanIit, name)
{       float t1,t2;
        float timeOut, timeNow;
        float arrive, curTime;
        int tranCount;
        int waitFlag = 1;
        process simnode simid;
        stats iit, sysTime;
        sItem sst;

    /* Initialization phase */
        simid = c_mypid();
        c_setname(simid, name.str);
        /* calling the sched and busnode to tell that its a producer */
        s.addUser(); bid.addLoProd();
        stInit(&iit);            /* initialize sturctures for inter-  */
        stInit(&sysTime);        /* arrival, system delay and         */
        sstInit(&sst);           /* transmission statistics.          */

    /* Main processing phase */
    /* generating an exponentially distributed interarrival time */
        t1 = erand(meanIit);
        stVal(&iit, t1);
        t2 = erand(meanIit); /* t2 is the expected inter-arrival */
        stVal(&iit, t2);         /* time for the next packet.  */
        while (1) { /* run until similation time expires */
            if (waitFlag) /* packet hasn't arrived yet! wait t1 time */
                if ((timeNow = s.wait(s.reqDelay(t1, c_nullpid))) != 0.)
                        arrive = timeNow; /* arrival timestamp of packet */
            if (timeNow != 0.)
                if ((timeNow = waitCarrier(s, bid, simid)) != 0.) {
                      /* delay for transmitting packet */
                        timeNow = s.wait(s.reqDelay(packetTime, simid));
                /* printf("%s tran:1 return curTime = %10.2f id = %ld\n",
                        name.str,timeNow,simid); */
                }
            tranCount = 1;
            /* start counting the number of attemps for each packet */
            /* timeNoe < 0 indicates a collision. */
```

```
            while (tranCount < 16 && timeNow < 0.) {
                bid.stopTran(T_COLLIDE); /* due to collision */
                timeOut = backOff(tranCount);
                timeNow = s.wait(s.reqDelay(timeOut, c_nullpid));
                if (timeNow != 0.)
                    if ((timeNow = waitCarrier(s, bid, simid)) != 0) {
                        timeNow = s.wait(s.reqDelay(packetTime, simid));
                /* printf("%s tran2: return curTime = %10.2f id = %ld\n"
                        name.str,timeNow,simid); */
                    }
                tranCount += 1;
            }
            curTime = s.now();
            if (timeNow == 0.) {
                break;

            if (timeNow > 0.) { /* normal termination of transm.  */
                bid.stopTran(T_NORMAL);
            /* append systime with the delay of the packet */
                stVal(&sysTime, curTime-arrive);
                /* printf(" delay time = %10.2f curTime = %10.2f\n",
                        s.now()-arrive, s.now()); */
                /* update transmission statistics(a success) */
                sst.success += 1;
                sst.nSend += 1;
                sst.sumAt += tranCount;
                if (tranCount > sst.maxAt)
                        sst.maxAt = tranCount;
            }
            else {
                /* update transmission statistics(a discard) */
                bid.stopTran(T_COLLIDE); /* due to collision */
                stVal(&sysTime, curTime-arrive);
                printf("%s a packet discarded ! curTime = %10.2f\n"
                        name.str, s.now());
                sst.discard += 1;
                sst.nSend += 1;
                sst.sumAt += tranCount;
                if (tranCount > sst.maxAt)
                        sst.maxAt = tranCount;
            }
            /* next packet hasn't arrived yet, wait for t1 time */
            if (curTime <= t2 + arrive) {
                waitFlag = 1;
                t1 = t2 + arrive - curTime;
                t2 = erand(meanIit); /* t2 is the expected inter-arrival */
                stVal(&iit, t2);         /* time for the next packet.  */
            }
            else  { /* There is a packet waiting to be transmitted. There is
                   only one packet in the queue which is the most recent one.
                   The older one is discarded whenever a new one comes. */
                while(curTime > t2 + arrive) {
                    arrive = t2 + arrive; /* update the arrival time of the
                                                     newest packet */
                    t2 = erand(meanIit);
                    stVal(&iit, t2);

                waitFlag = 0;
            } /* else */
        } /* while */
        /* tell busnode to drop simnode as a producer */
        bid.dropLoProd(sst, iit, sysTime);
printf("%s  success = %d maxAt = %d sysTime = %10.2f curTime = %12.2f\n",
        name.str,sst.success,sst.maxAt, stMean(&sysTime),s.now());
        s.dropUser(); /* tell sched that it is terminating */
}
```

```
/*****************************************************************************/
/* Rserver is a process type whose main function is to carry messages from   */
/* its left busnode to the neighboring right busnode. Whenever a packet is  */
/* delivered by  simnode, the right server delays itself by the propagation  */
/* time needed to transmit the packet from one busnode to the other.It then  */
/* delivers the packet to the right busnode by calling a putrequest trans-   */
/* action.                                                                   */
/*               CONSTANT AND VARIABLE DICTIONARY                            */
/*               --------------------------------                            */
/* prop_delay         :time taken by a packet to propagate a distance        */
/*                     of 50m at the speed of light;                         */
/* s                  :scheduler;                                            */
/* inbus,outbus       :identification of input & output bus;                 */
/* float ts           :floating point time;                                  */
/* int qt,mode        :qt is an integer to hold the value returned by qTake  */
/*                     function  ;                                           */
/* qTake()            : a function to take an item from the busnode;         */
/*                                                                           */
/*****************************************************************************/

#include "dcls.h"
#include "stdio.h"
#define prop_delay 1.6

process body Rserver(s, inbus, outbus, name)

        float   ts;
        int     mode;

        /* initialization phase */
        c_setname(c_mypid(), name.str);
        s.addUser();
        if (outbus != c_nullpid)
                outbus.addProd();
        inbus.addCons();
        /* main processing phase */
        while (qTake(inbus, FLTR, &mode)) {
                /* delay for prop_time */
                ts = s.wait(s.reqDelay(prop_delay, c_nullpid));
                if (outbus != c_nullpid)
                    if (outbus.putReq(FLTR, mode)) {
                        ts = s.wait(s.reqDelay(0.0001, c_nullpid));
                        outbus.insure(FLTR, mode);
                    }

        /* termination phase */
        if (outbus != c_nullpid)
                outbus.dropProd(); /* notify output busnode of termination */
        inbus.dropCons(); /* notify input busnode of termination */
        printf("%s terminate!  curTime = %12.2f\n", name.str, s.now());
        s.dropUser(); /* notify sched of its termination */
}
```

```
/*********************************************************************/
/* Busnode is a process type whose main function is to coordinate   */
/* the transmission of packets by simnodes and to handle collision  */
/* as specified in the CSMA/CD standards.                            */
/* There are three types of traffic that passes each busnode: left  */
/* traffic, right traffic and local traffic. Locally, each busnode  */
/* handles up to eight simnodes as does a TCL box.                   */
/*                                                                   */
/*              TRANSACTIONS AND THEIR MEANING                       */
/*              ------------------------------                       */
/* putReq()          :this transaction is called by both L & R       */
/*                    servers to put a packet;                       */
/* insure()          :busnode detects a collision and the trans-     */
/*                    mitting simnode gets a delay request slot      */
/*                    in the scheduler ;                             */
/* takeReq()         :servers call busnode to take any available     */
/*                    packet that is either loacl or moving traffic; */
/* tranReq()         :request by simnode to transmit local packet.   */
/* stopTran()        :terminate local transmission due te either     */
/*                    normal transmission or collision;              */
/* addProd(),addCons() :change producer & consumer count ;           */
/* dropCons(),dropProd() :delete producer & consumer count ;         */
/* addLoProd()       : increment simnode count (eight simnodes       */
/*                     are connected to each busnode );              */
/* dropLoProd()      : simnode calling to terminate & thereby        */
/*                     decrementing count ;                          */
/*********************************************************************/
/*
*/
/* Each data transmission from simnode is simulated by 2 requests:  */
/*  the first is START TRANSMISSION (mode = T_START), the other is  */
/* TERMINATE TRANSMISSION which can be either normal termination    */
/* (mode = T_NORMAL) or collision (mode = T_COLLIDE).               */
/*********************************************************************/

#include "dcls.h"
#include "stdio.h"
#define IFG 96.0 /* Inter frame gap between frame transmissions
                    equal to 96 bit_times. */

/* qTake is an interface function between the busnode and the servers .   */
/* A server wants to take a packet from the busnode and it supplies the   */
/* correct direction. The qTake function in turn makes a takeReq trans-   */
/* action where pending packets in the appropriate direction are checked. */
/* If there is a pending packet, it returns a mode else -1. If there is   */
/* no packet but the server has requested, then it does a takeWait trans- */
/* action which returns a different mode to the function.                 */

int qTake(bid, dir, modep)
        process busnode bid;
        int dir;
        int *modep;
{
        if((*modep = bid.takeReq(dir)) > 0)
                return T_NOWAIT;
        else
                return (*modep = bid.takeWait(dir));
}

typedef struct { /* structure for maintaining busnode information */
        process sched s;
        process simnode curSimid; /* the id of simnode currently
                                     transmitting data locally */
        int carrier; /* indicates whether the medium is currently busy */
        int local; /* indicates whether there is local transmission,
                      either normal or jam signal */
```

```
        int uno; /* indicates  normal transmission is going on */
        int nProd, nCons; /* number of server processes to busnode */
        int nLoProd; /* number of simnodes connected to local busnode */
        /* for the following arrays, the indices correspond to FRTL(=0)
           and FLTR(=1).  */
        int Traffic[2]; /* # of traffics(with start transmission request)
                           existing in the medium from either direction */
        int Pending[2]; /* # of pending requests(can be any mode) waiting
                           to be taken by server */
        int head[2], tail[2]; /* pointers to pending list */
        int Mode[2][10]; /* pending list(a circular queue) contains mode
                            of each pending request */
        float timeOn; /* starting time of busy period of the medium */
        float timeOff; /* ending time of busy period */
        float busyTime; /* duration of the busy period */
        float util; /* utilization of the medium observed from each busnode */
        name_t name;
} bInfo;

/* retrieve the mode of the oldest pending request and update the pending
   list */
static int takeMode(bp, dir)
        bInfo   *bp;
        int     dir;
{       int mode;

        mode = bp->Mode[dir][bp->head[dir]];
        bp->head[dir] = (bp->head[dir] + 1) % 10;
        bp->Pending[dir] -= 1;
        return mode;
}

/* insert new pending request to the pending list */
static void addPending(bp, dir, mode)
        bInfo *bp;
        int dir, mode;
{
        bp->Pending[dir] += 1;
        if(bp->Pending[dir] > 2) { /* usually trere should be at most 2 */
          if(dir == FRTL)
            printf("%s has %d pending in FRTL diretion\n",
            bp->name.str, bp->Pending[dir]);
          else
            printf("%s has %d pending in FLTR diretion\n",
            bp->name.str, bp->Pending[dir]);
        }
        bp->Mode[dir][bp->tail[dir]] = mode;
        bp->tail[dir] = (bp->tail[dir] + 1) % 10;


/* initialize busnode information */
static void initBInfo(bp, name)
        bInfo   *bp;
        name_t  name;
{       int i, j;

        bp->name = name;
        bp->curSimid = c_nullpid;
        bp->carrier = bp->local = bp->uno = 0;
        bp->nProd = bp->nCons = bp->nLoProd = 0;
        bp->timeOn = bp->timeOff = bp->busyTime = 0.;
        for (i=0; i<2; i++) {
                bp->Traffic[i] = bp->Pending[i] = 0;
                bp->head[i] = bp->tail[i] = 0;
                for (j=0; j<10; j++)
                    bp->Mode[i][j] = 0;
```

```
        }


process body busnode(s, stp, simt, name)
{       bInfo b;
        int sound;
        float dt;
        stats iit, sysTime;
        sItem sst;

    /* Initialization phase */
        c_setname( c_mypid(), name.str);
        s.addUser(); s.passive();
        initBInfo(&b, name);
        stInit(&iit); stInit(&sysTime);
        sstInit(&sst);
        accept addProd() {b.nProd += 1;}
        stp.addCus();

    /* Main processing phase */
        while (b.nProd + b.nCons > 0) {
            select {
    /* Put request from left or right server */
            accept putReq(dir, mode)
                {if (mode == T_START) { /* a start transmitting request */
                        /* data transmission is taking place locally,
                           therefore it is a collision. */
                        if (b.local && b.uno) { /* check uno flag as well
                        to make sure that we don't do this twice, while
                        jam signal is being generated locally */
                            b.uno = 0;
                            treturn 1;
                        }
                        /* no local transmission or transmitting
                           jam signal */
                        b.Traffic[dir] += 1; /* increment count */
                        /* the medium was previously idle */
                        if ( !b.carrier) {
                            b.timeOn = s.now(); /* record the start time
                                                   of busy period */
                            b.carrier = BUSY; /* set carrier to busy */
                        }
                 }
                 else { /* This is a request of second type indicating
                 a remote simnode has just finished transmitting. */
                        sound = --b.Traffic[dir] +
                            b.Traffic[(dir+1) % 2] + b.local;
                        if ( !sound ) { /* the medium is silent again! */
                            b.carrier = IDLE;
                            b.timeOff = s.now(); /* record the ending time
                            ond get the duration of busy period */
                            b.busyTime += b.timeOff - b.timeOn;
                        }
                 }
                 addPending(&b, dir, mode); /* insert pending request */
                 treturn 0;

    /* To insure that simnode gets a slot in scheduler waiting queue first.
       Note that there is possibility that the moment local transmission
       gets admitted to the medium, a T_START request arrives (a putReq
       transaction call) from some remote simnode. The collision is
       detected (since b.local bit is set) but the sched process hasn't started
       processing reqDelay call from the local simnode yet, therefore its
       id is not in the wait queue.  An s.jam call issued subsequently founds
       no match of the id in the waiting queue. An error! So, let server
       delay for a very small amount of time and then make insure call. */
```

```
        or accept insure(dir, mode)
            { b.Traffic[dir] += 1; b.carrier = 1;
              s.jam(b.curSimid);
              addPending(&b, dir, mode);
              /* printf("%s: a collision! curTime = %10.2f\n",
                                  name.str,s.now()); */
            }
/* Take request from left or right server */
        or accept takeReq(dir)
            /* there are pending requests, retrieve the oldest one */
            {if (b.Pending[dir] > 0)
                    treturn takeMode(&b, dir);

             else { /* no pending request, go to passive state */
                    s.passive();
                    treturn  -1;}
            }
        or accept takeWait(dir) suchthat (b.Pending[dir] > 0)
            {s.active(); /* a new request is here, wake up the
                            server which is waiting */
             treturn takeMode(&b, dir);}

/* local transmission request from simnode */
        or accept tranReq(simid)
            {if (b.carrier == BUSY)
            /* The medium is busy now, wait for another IFG time, since
               it is the least time you have to wait */
                    treturn IFG;
             else if ((dt = s.now()-b.timeOff) < IFG)
             /* The medium has just become idle, you need to wait until
                it is idle for IFG time */
                    treturn (IFG - dt);

             else { /* The medium is available for transmitting locally */
                    b.local = 1; /* set local transmission flag */
                    b.carrier = BUSY;
                    b.curSimid = simid; /* get id of the sending simnode */
                    b.timeOn = s.now(); /* record start time */
                    addPending(&b, FRTL, T_START); /* insert request to */
                    addPending(&b, FLTR, T_START); /* pending list */
                    b.uno = 1; /* set uno flag whenever a local
                                  transmission is started afresh */
                    treturn 0.;
             }
            }
/* terminate local transmission */
        or accept stopTran(mode)
            /* The medium is back to idle again under the following
               condition:
               1. the mode is T_NORMAL indicating a successful local
                  transmission. or,
               2. the mode is T_collide but there is no other traffic
                  in the medium.
            */
            {if (mode == T_NORMAL || b.Traffic[FLTR] +
                            b.Traffic[FRTL] == 0) {
                    b.carrier = 0;
                    b.timeOff = s.now();
                    b.busyTime += b.timeOff - b.timeOn;
             }
             b.local = 0;
             addPending(&b, FRTL, mode); /* add pending request */
             addPending(&b, FLTR, mode);
             b.curSimid = c_nullpid; /* no local transmission now */
```

```
    /* When all local simnodes terminated reactivate the server blocked
       by the busnode */
            or (b.nLoProd == 0):
                accept takeWait(dir)
                        {s.active();
                         treturn 0;}
    /* other transactions */
            or accept addCons() {b.nCons += 1;}
            or accept addProd() {b.nProd += 1;}
            or accept dropCons() {b.nCons -= 1;}
            or accept dropProd() {b.nProd -= 1;}
            or accept addLoProd() /* process local simnode on add request */
                {b.nProd += 1;
                 b.nLoProd += 1;}
            or accept dropLoProd(sstItem, iItem, sysItem)
                {b.nProd -= 1;
                 b.nLoProd -= 1;
            /* merge the statistics gathered on the simnode just dropped out
               one by one */
                 sstMerge(&sst, &sstItem);
                 stMerge(&iit, &iItem);
                 stMerge(&sysTime, &sysItem);}
            } /* select */
        } /* while */
        /* get utilization of the medium as viewed from the busnode */
        b.util = b.busyTime/simt;
        /* merge statistics to the whole system */
        stp.doSystime(sysTime);   /* delay stats. */
        stp.doIit(iit); /* Inter-arrival time stats. */
        stp.doSst(sst); /* transmission stats. */
        stp.doUtiliz(b.util); /* utilization */
        stp.dropCus();
printf("%s  success = %d meanIit = %10.2f sysTime = %10.2f curTime = %12.2f\n",
        printf("   Utilization = %8.5f timeOn = %10.2f timeOff = %10.2f\n",
                b.util, b.timeOn, b.timeOff);
        s.active();
        s.dropUser();
} /* process */
```

```
#include "dcls.h"
#include "stdio.h"
#define MAXREQ 300    /*max # of request clients */
typedef struct{
            float ts;
            float start_time;
            long ticket;
            process simnode id;
            char type;
            }request;


static request Rtab[MAXREQ];
static int qSize=0;
static long ticket = 0;

/* other inserting routine */
static void siftUp (m)
        int m;
{       int i ,j;
        request x;

        i = m;
        while ((j = i/2) > 0) {
            if (Rtab[j].ts <= Rtab[i].ts)
                break;
            else {
                x = Rtab[i];
                Rtab[i] = Rtab[j];
                Rtab[j] = x;

            i = j;
        }
}

static int enqueue(ct,ts,id)        /* add a request to priority queue */
       float ts,ct;
       process simnode  id;                /* and return a index */
{ if (qSize < MAXREQ -1) {
    qSize=qSize+1;
    Rtab[qSize].ts = ct + ts;
    Rtab[qSize].id = id;
    Rtab[qSize].start_time = ct;
    Rtab[qSize].type = 'N';
    ticket += 1;
    Rtab[qSize].ticket = ticket;
    siftUp(qSize);
    /* if (id != c_nullpid )
    printf("enqueue: Rtab[qSize].ts = %10.1f id = %ld ticket = %ld curT = %f\n",
                        Rtab[qSize].ts,id,Rtab[qSize].ticket,ct); */
   }
return ticket;
}

/* This procedure is for inserting into heap */
static void Insertheap (x,a,b)
       request x;
       int a,b;
{ int m;
  m = 2*a;
  while(m <= b)
    {if(m < b)
       if (Rtab[m].ts > Rtab[m+1].ts)
               m= m+1;
     if (x.ts <= Rtab[m].ts)
        m = b+1;
```

```
      else { Rtab[a] = Rtab[m];
                   a = m;
                   m = 2*a;
           }
      }
      Rtab[a] = x;
}
/* This is for building heap */
static void buildheap(n)
    int n;
{ int p;
  request x;
  for (p = n/2;p >= 1;p--)
     {x = Rtab[p];
      Insertheap(x,p,n);
      }
}

process body sched(simt)
{ int nClients,nActive,i;
  float curTime = 0.0;
  float t,coll_time;
  float temp;
 /* Initialization phase */
    c_setname(c_mypid(),"sched");
    accept addUser(){nClients = nActive=1;}
/* Main processing phase:accept requests while clients exists */
 while(nClients > 0) {
         select {
         accept addUser(){nClients +=1;nActive +=1;}
 or      accept dropUser(){nClients -=1;nActive -=1;}
 or      accept passive(){nActive -=1;}
 or      accept active(){nActive +=1;}
 or      accept now()
                { treturn curTime;}
                /* { if (curTime > simt)
                        treturn simt;
                  else
                        treturn curTime;} */
 or      accept reqDelay(x,id)
                { nActive -= 1;
                  /* if (ticket == ticket/100*100)
                        printf("sched: curTime = %13.2f qSize = %d\n",
                               curTime,qSize); */
                  treturn enqueue(curTime,x,id); }
 or       accept jam(id)
                 { coll_time = curTime;
                   for(i = qSize;i >= 1; i--)
                        if (Rtab[i].id == id )
                               break;
                   if (i == 0) {
                printf("No? there should be a collision with id %ld\n",id);
                        treturn;
                   }
                   temp = Rtab[i].ts;
                   t = Rtab[i].start_time + 64.0;
                   if(coll_time <t)
                     {Rtab[i].ts = t + 32.0;
                      Rtab[i].type = 'J';
                     }
                    else {
                         Rtab[i].ts = coll_time+ 32.0;
                         Rtab[i].type = 'J';

                    if (temp < Rtab[i].ts)
                        printf("HOW STRANGE!\n");
```

```
                    /* printf("Rtab.ts = %10.1f i = %d ticket = %ld jamming!\n",
                        Rtab[i].ts,i,Rtab[i].ticket); */
                    siftUp(i);
                }
            }
/*If all clients are waiting ,find the first event and allow all clients
  waiting for it to proceed */
if (nActive == 0 && qSize != 0){
    /* buildheap(qSize); */
    curTime = Rtab[1].ts;
   while(Rtab[1].ts == curTime && qSize != 0)
   {
    accept wait(key) suchthat (key == Rtab[1].ticket)
           {nActive += 1;
            if (curTime > simt) {
               treturn 0.;
            }
            else {
                if (Rtab[1].type == 'N') {
                   /* if (Rtab[1].id != c_nullpid )
                   printf("Sched: type N curTime = %10.1f ticket = %ld\n",
                        curTime, Rtab[1].ticket); */
                   treturn curTime;
                }
                else {
                   /* printf("Sched: type J curTime = %10.1f ticket = %ld\n",
                   curTime,Rtab[1].ticket); */
                   treturn -curTime;
                }
             }
           } /* accept */
      if (qSize>1)
         Insertheap(Rtab[qSize],1,qSize-1);
      qSize = qSize-1;
     } /* while */
                                }
 }
         printf("Sched terminate! curTime = %10.2f ticket = %ld\n",
                curTime, ticket);
 } /* terminate */
```

```
/****************************************************************/
/* This process takes various statistics structures passed from */
/* busnodes upon their completion and combines them into system */
/* overall statistics.                                          */
/****************************************************************/

#include "dcls.h"
#include "stdio.h"

process body statProc(simt)
{       stats iit, sysTime;
        sItem sst;
        int nCus = 0;
        int num = 0;
        float sumUt = 0.;

    /* Initializing */
        c_setname( c_mypid(), "statProc");
        sstInit(&sst);
        stInit(&iit); stInit(&sysTime);
        accept addCus() {nCus += 1;}

    /* Main processing phase */
        while (nCus > 0) {
            select {
            accept doSystime(sysItem) {stMerge(&sysTime, &sysItem);}
            or accept doIit(iItem) {stMerge(&iit, &iItem);}
            or accept doSst(sstItem) {sstMerge(&sst, &sstItem);}
            or accept doUtiliz(fator) {sumUt += fator; num += 1;}
            or accept addCus() {nCus += 1;}
            or accept dropCus() {nCus -= 1;}
            }/* select */
        }/* while */

    /* print statistics */
        printf("stat:\n   nSend = %d\n   success = %d\n   discard = %d\n",
                sst.nSend, sst.success, sst.discard);
        printf("   maxAt = %d\n   avgAt = %5.5f\n",
                sst.maxAt,sst.sumAt/sst.nSend);
        printf("   utilization = %5.5f\n",sumUt/num);
        printf("   throughput = %10.2f\n",sst.success*10000000./simt);
        printf("   meanIit = %10.2f\n   sysTime = %10.2f\n",
                stMean(&iit), stMean(&sysTime));
}
```

```c
#include "dcls.h"
#include "math.h"

double drand01()  /* Return random number uniform in range [0,1]. */
{       double  p;
        extern long random();

        p = random() & 0xffffff;
        return p / 0xffffff;
}

double erand(mean)  /* Return exponentially distributed random number. */
        float   mean;
{       double  r;

        r = -(mean) * log(1.0 - drand01()) + 0.5;
        /* if (r == 0)
                r = 1; */
        return r;
}

void stInit(p)          /* Initialize a "stats" structure. */
        stats   *p;
{
        p->nv = 0;  p->maxv = -1;  p->sumv = 0;  p->sumsq = 0;
}

void stVal(p, v)        /* Add a value to a "stats" structure. */
        stats   *p;
        float   v;
{
        if (p->nv == 0 || v > p->maxv)
                p->maxv = v;
        p->nv++;  p->sumv += v;
        /* p->sumsq += v*v; */
}

void stMerge(p1, p2)
        stats   *p1;
        stats   *p2;
{
        p1->nv = p1->nv + p2->nv;
        if (p2->maxv > p1->maxv)
                p1->maxv = p2->maxv;
        p1->sumv = p1->sumv + p2->sumv;
        p1->sumsq = p1->sumsq + p2->sumsq;


void sstInit(p)
        sItem   *p;
{
        p->success = p->discard = p->nSend = 0;
        p->sumAt = 0;
        p->maxAt = 0;
}

void sstMerge(p1, p2)
        sItem   *p1;
        sItem   *p2;
{
        p1->success += p2->success;
        p1->discard += p2->discard;
        p1->nSend += p2->nSend;
        p1->sumAt += p2->sumAt;
        if (p2->maxAt > p1->maxAt)
                p1->maxAt = p2->maxAt;
```

```
}

double stMean(p)   /* Return the mean value for a "stats" structure. */
        stats   *p;
{
        return p->nv != 0 ? p->sumv/p->nv : 0;
}

double stSdev(p)   /* Return the std deviation for a "stats" structure. */
        stats   *p;
{       double avg;

        if (p->nv == 0)
                return 0;
        else {
                avg = p->sumv/p->nv;
                return sqrt(p->sumsq/p->nv - avg*avg);
        }
}
```

Appendix B

Source Code Listing of the Token-Ring Simulation Model

```
/**********************************************************************/
/*                                                                    */
/*  This is a header file containing  the declarations of variables and    */
/*  functions that are global to all processes in the simulation. Moreover, */
/*  it contains the specification part of each of the Concurrent C        */
/*  processes.                                                         */
/*                                                                    */
/**********************************************************************/

/*  The type name_t is a structure containing a character string and is  */
/*  passed by value to any process that calls it.                      */
typedef struct {char str[20];} name_t;

/*  Specification of the virtual time scheduler  */
process spec sched(float simt){  /* simt is the total simulation time */
        trans float now();       /* return current simulated time */
        trans long reqDelay(float); /* request a delay */
        trans float wait(long);  /* wait for a request delay */
        trans void addUser(int id),dropUser(); /* add/delete a client process */
        trans void passive(),active();  /* client changed to new state */
};

/*  nItem is a structure type for packets/tokens circling the network  */
typedef struct {
 int code;     /* code distinguish if the packet is a free or connector token  */
               /* code=0 means a free token, code=1 means a connector token &  */
               /* packet                                                        */
 long length;  /* length of the packet */
 int id;       /* the id of the node sending the packet */
}nItem;

/*  Specification of the process that collects statistics from individual  */
/*  process and performs statistics on the network as a whole.            */
process spec statProc(float simt) {
        trans void  addCus(), dropCus(); /* add/delete a client process */
        trans void  doThruput(int);   /* compute throughput for network */
        trans void  doUtil(float);    /* calculate network utilization */
        trans void  doDelay(float);   /* calculate packet transmit delay */
        trans void  doFT(float);      /* calculate free token cycle time */
        };

/*  Specification of the node process  */
process spec node (process sched s, process statProc stp, float simt,
                  float transRate, name_t name, long FTlength, long CTlength) {
   trans void addProd(), dropProd(); /* add/delete a producer */
   trans void addCons(), dropCons(); /* add/delete a consumer */
   trans int transReq(long,float); /* initiate to transfer packet from source */
   trans void transWait(long,float); /* complete transfer of packet */
   trans void put(nItem); /* accept a packet/token from server */
   trans nItem takeReq(); /* initiate to transmit packet by server */
   trans nItem takeWait(); /* complete transmission of packet by server */
   };

/*  Specification of the source process which produces packets and  */
/*  transfers them to the node process                              */
process spec source(process sched s,process node nid,float iit,
                   float meanlength,name_t name);

/*  Specification of the server process, which takes packet/token     */
/*  from the input node and transmits the packet/token to output node  */
process spec server(process sched s,process node inN,process node outN,
                    float propTime, float transTime, name_t name);

/*  A function called by source to initiate a transfer of packet to node  */
float nTransfer(/* process node, process sched, long length float start_t */);
```

```
/*  A function called by server to initiate getting a packet from its  */
/*  input node                                                          */
void nTake (/* process node, nItem */);

/*  Structure used to obtain statistics  */
typedef struct {
        long  nv;   /* number of values */
        float maxv;  /* maximum value */
        double  sumv, sumsq; /* sum and sum of squares of all values */
        } stats;

double drand01();    /* return a uniform random from 0 to 1 */
double erand(float); /* return exponential random number */
void  stInit(), stVal(); /* Initialize and add a value to the statistics */
double  stMean(), stSdev(); /* obtain the mean and standard deviation */
                            /* of the statistics                       */
```

```
/************************************************************************/
/*                                                                      */
/*   The main process creates all the processes - scheduler process     */
/*   statistical process, node processes, source processes, and server */
/*   processes, that simulate the entities in the token ring model.     */
/*   During initialization, it assigns a unique name to each of the new */
/*   processes and supplies appropriate values for system parameters    */
/*   by either default or command line arguments.  It also initiates the */
/*   simulation by putting a free token in the first node.              */
/*                                                                      */
/*       VARIABLE DICTIONARY :-                                         */
/*       simt : simulation time desired to run;                         */
/*       iit : inter-arrival time;                                      */
/*       seed : for random number generation;                           */
/*       packlength : length of the packet generated;                   */
/*       apart2Nodes : distance between two nodes;                      */
/*       transRate : transmission rate expressed as #bits per bit time  */
/*       propTime : propagation delay between 2 nodes in bit time       */
/*       FTlength : length of a free token in terms of #bits            */
/*       CTlength : length of a connector token in terms of #bits       */
/*       FT : a free token used to initialize the simulation            */
/*                                                                      */
/************************************************************************/

#include "dcls.h"
#include "stdio.h"
int  c_nprocs = 500;

#define  NUMNODE 80    /*  number of nodes in the tokenring network  */


/*  makeName function returns a char* argument to the calling function  *
/*  It returns a string to the calling function based on the arguments  *
/*  supplied by the caller.                                              *
name_t makeName(narg,name,ch1,ch2)
        int     narg;           /*  number of arguments supplied  */
        char    *name;          /*  the initial characters of the string  */
        char    ch1,ch2;        /*  the other two characters of the string  */
{       name_t  ret;            /*  the resulting string  */
        int     i;

        i = 0;
 while ((ret.str[i] = name[i]) != '\0')
        i++;
 ret.str[i++] = ch1;
 if (narg == 3)
             ret.str[i++] = ch2;
 ret.str[i] = '\0';
 return ret;
}

main(ac,av)
        int ac;
        char *av[];
{
process sched s;               /* the virtual time scheduler */
process statProc stp;           /* the process to calculate statistics */
process node n[NUMNODE+1];      /* an array of node processes */
float simt=15000,iit=500,packlength = 1024;
float apart2Nodes=500/NUMNODE, transRate=1;
float propTime;
long FTlength = 1, CTlength = 4;
int seed = 12345;
int i;
nItem  FT;          /* FreeToken used to initiate the simulation */
```

```
if (ac > 1) sscanf(av[1],"%f",&simt);
if (ac > 2) sscanf(av[2],"%f",&iit);
if (ac > 3) sscanf(av[3],"%f",&seed);
srandom(seed);

printf ("simt= %f, iit=%f \n", simt, iit);
printf (" Main process is created \n");
printf ("simt = %f, iit = %f, no of nodes = %d \n", simt, iit, NUMNODE);

/* determine the propagation delay between any two adjacent nodes */
propTime = 1.6*apart2Nodes/50;

/* create virtual time scheduler */
s = create sched(simt);

/* create statistics process */
stp = create statProc(simt);
s.addUser(1);              /* for main */

/* create nodes in ring */
for (i=1;i <= NUMNODE; i++)
    if (i<10)
       n[i] = create node(s,stp,simt,transRate,makeName(2,"node",'0'+i ,'0'),
                FTlength, CTlength);
    else
       n[i] = create node(s,stp,simt,transRate,
                makeName(3,"node",'0'+i/10,'0'+i%10),FTlength, CTlength);

/* create source for each of the nodes */
for (i=1;i <= NUMNODE;i++)
    if (i<10)
       create source(s,n[i],iit,packlength,makeName(2,"source",'0'+i,'0'));
    else
       create source(s,n[i],iit,packlength,makeName(3,"source",'0'+i/10,
                                                     '0'+i%10));

/* create servers to transmit packets/tokens between nodes */
for (i=1;i < NUMNODE;i++)
   if (i<10)
      create server(s,n[i],n[i+1],propTime,transRate,
                    makeName(2,"server",'0'+i,'0'));
   else
      create server(s,n[i],n[(i+1)],propTime,transRate,
                    makeName(3,"server",'0'+i/10,'0'+i%10));

/* create the last server in the network */
if (NUMNODE < 10)
   create server (s,n[NUMNODE],n[1],propTime,transRate,
                  makeName(2,"server",'0'+NUMNODE,'0'));
else create server (s,n[NUMNODE],n[1],propTime,transRate,
                    makeName(3,"server",'0'+NUMNODE/10,'0'+NUMNODE%10));

/* To put a free token in the first node to initiate the flow of tokens
   in the ring  */
FT.code = 0;
n[1].put(FT);

/* wait for all processes to tell the sched that they are clients */
delay 2.0;

/* all started :can delete main as sched client */
printf("Main.cc terminate!\n ");
s.dropUser();
}
```

```
/****************************************************************************/
/*                                                                          */
/*   The sched process is a virtual time scheduler.  It keeps a list of     */
/*   pending delay requests, ordered by the time at which the client is     */
/*   to be reactivated.  List entries are pairs (t,n) where t is a          */
/*   simulated time and n is the number of processes to be awakened at     */
/*   that time.                                                             */
/*                                                                          */
/*   After initialization, the scheduler process repeatedly accepts         */
/*   requests until no clients are active.  For each delay request, the     */
/*   scheduler calculates the absolute time at which that process           */
/*   should be re-activated, and adds an entry to the list of pending       */
/*   delay requests.  List entries are allocated from an array; the         */
/*   reqDelay transaction uses the array index as a "ticket" value.         */
/*   When no client processes are active, the scheduler takes the next      */
/*   request from the list, advances the simulated time, and accepts the    */
/*   wait requests from all clients waiting for that time.                  */
/*                                                                          */
/*      VARIABLE AND FUNCTION DICTIONARY                                    */
/*                                                                          */
/*   addReq()   :  Add a request for delay until time ts, and return        */
/*                 Rtab index.                                              */
/*   curTime    :  current time.                                            */
/*   Ifree      :  index of first free entry.                               */
/*   Ihead      :  index of entry with lowest timestamp.                    */
/*   MAXREQ     :  maximum no of requests in the table.                     */
/*   nActive    :  no of active processes.                                  */
/*   nClients   :  no of client processes.                                  */
/*   rqInit()   :  function to initialize a request table.                  */
/*   Rtab       :  a request table.                                         */
/*                                                                          */
/****************************************************************************/


#include "dcls.h"
#include <stdio.h>
#define MAXREQ 300

typedef struct {  /* reqent:structure describing a delay req*/
        float ts;
        int  next;
        int  nwait;
      }reqent;

static reqent Rtab[MAXREQ];
static int    Ifree;              /* index of first free entry */
static int   Ihead = -1;          /* index of entry with lowest timestamp */

static void rqInit()              /* Initialise request table */
{  int i;
   Ifree = 0;Ihead = -1;
   for (i = 0;i< MAXREQ;i++)
       Rtab[i].next = i+1;
       Rtab[MAXREQ-1].next = -1;
}
static int addReq(ts)           /* add a req for delay until time ts */
       float ts;
{      int i,iprev;
/* printf("Sched : Start to add a request for delay \n"); */
/*
printf("Sched :inside addReq: timestamp to be awakened after delay=%f \n",ts);
*/
  for(iprev = -1,i=Ihead;i != -1;iprev =i,i=Rtab[i].next)
      if(Rtab[i].ts >= ts)
            break;
```

```
 if (i== -1 || Rtab[i].ts > ts){       /* add new entry */
        i = Ifree;
     Ifree = Rtab[i].next;
  Rtab[i].ts = ts;Rtab[i].nwait =0;
      if (iprev != -1){
           Rtab[i].next = Rtab[iprev].next;
           Rtab[iprev].next = i;
           } else {
             Rtab[i].next = Ihead;Ihead = i;
                  }


 Rtab[i].nwait += 1;
/* printf ("Sched : After adding a request for delay \n"); */
 return i;
}
process body sched(simt)
{ int nClients,nActive,i;
  float curTime = 0;
/* Initialization phase */
 c_setname(c_mypid(),"sched");
rqInit();

printf ("Scheduler is created \n");

accept addUser(id){nClients = nActive = 1;
  /*           printf("Sched: no of clients = %d\n", nClients);
            printf("ID of process requesting addUser is %d \n",id); */


/* Main processing phase:accept requests while clients exists */
while (nClients > 0) {
    select {
           accept addUser(id) {nClients += 1;nActive  += 1;
         /*   printf("Sched :  in adduser, nclients = %d, nActive \n",
nClients,nActive);
            printf("ID of process requesting addUser is %d \n",id); */ }
or         accept dropUser() {nClients -= 1;nActive -= 1;
     /*  printf("Sched: inside dropUser, nActive = %d \n",nActive);*/ }
or         accept passive()  {nActive -= 1;
   /*    printf("Sched : inside passive, nActive = %d \n", nActive);*/ }
or         accept active() {nActive += 1;
  /*     printf("Sched : inside active, nActive = %d \n",nActive); */ }
or         accept now() {treturn curTime;}
or accept reqDelay(x) {nActive -= 1;
            /*   printf("Sched: in reqDelay, curtime = %f\n",curTime);
               printf("Sched : in reqDelay, nActive = %d \n",nActive);
            */
                treturn addReq(curTime + x);}
}/* end select */
/* If all clients are waiting,find first event,
   and allow all clients waiting for it to proceed */
/*
printf("Sched: check if increment time, nActive =%d, Ihead=%d \n", nActive,
       Ihead);
*/
if(nActive == 0 && Ihead != -1){
/*  printf("Sched : none of the processes are active \n"); */

curTime = Rtab[Ihead].ts;
nActive = Rtab[Ihead].nwait;
/*
 printf("Sched : nActive = %d \n",nActive);
*/
while (--Rtab[Ihead].nwait >= 0)
 accept wait(key) suchthat (key == Ihead)
```

```
         {if (curTime > simt)
            curTime = -999999999;
          treturn curTime;
         }
   i = Ihead;
   Ihead = Rtab[i].next;
   Rtab[i].next = Ifree;
   Ifree = i;
   }
   }
   /* Termination phase */
   /* printf("\nSched done;final time %10.2f\n",curTime); */
   }
```

```
/***************************************************************************/
/*                                                                         */
/*  The node process is a passive process   which  waits to  accept        */
/*  requests  from  other processes - the  source  and  the server.        */
/*  After a source has generated a packet, it requests to transfer the     */
/*  packet to the node.  If there is no packet in the node, the transfer   */
/*  is successful, else the source is blocked until the packet             */
/*  originally present is transmitted from the node to the network.        */
/*  Then, at that time, the transfer of the packet from source to node     */
/*  is made.                                                               */
/*                                                                         */
/*  From the viewpoint of the node, there are two types of server          */
/*  processes - producing server and consuming server.  The producing      */
/*  server puts a packet or token to the node.  The node identifies if     */
/*  the packet is a free token (FT) or a connector token (CT).  If it      */
/*  is a FT, it checks if an available pakcet is ready for transmission    */
/*  to the next node.  If yes, a CT and a new packet is transmitted to     */
/*  the next node.  If not, the FT is just passed to the next node. The    */
/*  node also  checks  if the  received  packet  is initiated by the       */
/*  node itself.  If yes, it is withdrawn from the network. Otherwise,     */
/*  it is passed to the next node.                                         */
/*                                                                         */
/*  The consuming server requests to take a packet from the node.  If a    */
/*  packet is available, the transmission is made.  If not, the            */
/*  requesting server is blocked until a packet is ready for               */
/*  transmission.                                                          */
/*                                                                         */
/*  The node process also calculates its local statistics and prints       */
/*  them at the end of the simulation.  The statistics includes :          */
/*       (1).  Utilization (Traffic intensity) of the node.                */
/*       (2).  Transmission delay for a packet.                            */
/*       (3).  The time period between  two successive  free  token        */
/*             passing through the node.                                   */
/*                                                                         */
/*      VARIABLE AND FUNCTION DICTIONARY                                   */
/*                                                                         */
/*  begin_t   :  time when a packet is generated from source.              */
/*  datalen   :  length of packet transferred from source to node.         */
/*  FTcur     :  time when a current FT is encountered.                    */
/*  FTtime    :  time in between 2 FTs pass through the node.              */
/*  FTprev    :  time when a previous FT is encountered.                   */
/*  item      :  packet transferred from node to source.                   */
/*  nCons     :  number of consumer clients to the node.                   */
/*  nid       :  the identity number of the node.                          */
/*  noItem    :  a packet used to notify server that no packet is          */
/*               available for transmission.                               */
/*  noPackets  :  total no of packets that have completed transmission     */
/*                through this node.                                       */
/*  nProd     :  no of producer clients to the node.                       */
/*  nTake()   :  a function called by server to request a new  packet      */
/*               from the node.                                            */
/*  nTransfer()  :  a function called by source to request to transfer     */
/*                  a packet from source to node.                          */
/*  packPresent  :  a flag to indicate if a packet is available in the     */
/*                  node.                                                  */
/*  sendFT    :  a variable to determine if a free token needs to be       */
/*               transmitted next.  If it is 0, FT not to be sent.  If     */
/*               it is 1, a CT & packet has been transmitted after the     */
/*               CT.  If it is 2, a FT has to be transmitted now.          */
/*  takeI assive  :  a flag to show if a server is blocked when trying     */
/*                   to take a packet from node.                           */
/*  takeReady  :  a flag to indicate if a packet is ready for trans-       */
/*                mission to server.                                       */
/*  totalLength  :  total length of all packets that were transmitted      */
/*                  through the node.                                      */
/*  trans_delay  :  transmission delay for a packet.                       */
```

```
/*  transmit_t   :  time when a packet is transmitted from node to network*/
/*  transPassive   :   a flag to show if a source is blocked.              */
/*                                                                         */
/**************************************************************************/



#include "dcls.h"
#include "stdio.h"

/*  This function is called by the source process to request to transfer  */
/*  a packet to its corresponding node.                                    */
float nTransfer (nid,s,length,start_t)
      process node nid;
      process sched s;
      long         length;
      float        start_t;
{
      int success = nid.transReq(length,start_t);
      if (success == 0) nid.transWait(length,start_t);
/*
      printf("Sched : inside nTransfer, time = %f \n",s.now()
*/
      return s.now();
}

/*  This function is called by the server process to request to take  */
/*  a packet from its corresponding node.                              */
void nTake (nid,itemp)
     process node  nid;
     nItem         *itemp;
{
     *itemp = nid.takeReq();
     if (itemp->code == -1) {/*  printf("takereq is blocked \n"); */
                                *itemp = nid.takeWait(); }
/*
     printf("takereq is completed \n");
*/

}

/*  The main body of the process, node.  */
process body node (s, stp, simt, transRate, name, FTlength, CTlength)
{
     nItem     item, noItem;
     int       nProd=0, nCons=0;
     int       packPresent = 0,
               takeReady = 0,
               transPassive = 0,
               takePassive = 0,
               sendFT = 0;
     int       nid;
     long      datalen=0;
     int       noPackets=0;
     float     totalLength=0.0;
     float     begin_t=0.0, transmit_t;
     float     FTprev= -1.0, FTcur;
     float     utilization;
     stats     trans_delay, FTtime;

   /* Initialization */
   nid = c_mypid();
   c_setname (nid, name.str);
/* printf("%s is created \n",name.str); */
   s.addUser(nid);
   s.passive();
```

```
    stp.addCus();

    noItem.code = -1;

    stInit(&trans_delay);
    stInit(&FTtime);

    /* Accept a producer client */
    accept addProd()  { nProd += 1;


    while (nProd+nCons > 0) {

          select {

    /* Transfer data(packet) from source to node */

    /* No packet is in the node, the transfer of packet from source to */
    /* node is successful.                                             */
     (packPresent == 0) :
       accept transReq(length,start_t)
         {  datalen = length;
            begin_t = start_t;
            packPresent = 1;
/*
  printf ("%s : packet successfully transferred from source \n",name.str);
*/
            treturn 1;
         }

   /* Packet is already present in the node, transfer is unsuccessful, */
   /* Source is being blocked.                                          */
  or (packPresent == 1) :
        accept transReq(length,start_t)
          { s.passive();
            transPassive = 1;
/*
  printf ("%s : packet not transferred from source, source is blocked \n",
            name.str);
*/
            treturn 0;
          }

   /* Packet originally in the node has been transmitted to server, now */
   /* a previously blocked source can transfer packet to the node.      */
  or (packPresent == 0 ) :
        accept transWait (length,start_t)
          { datalen = length;
            begin_t = start_t;
            packPresent = 1;
/*
    printf ("%s : packet transferred to node after source has waited \n",
             name.str);
*/
          }

  /* Simulation time is up, any source that is blocked will be  unblocked. */
  or (s.now()<0.0)  :
        accept transWait(length,start_t)

            if (transPassive == 1) {
/*
        printf("%s : forced unblocking in transWait \n",name.str);
*/
                s.active();
                transPassive = 0;
```

```
            }
        }

     /* Accept put requests of packets/token from adjacent producing-server */
 or     accept put (nodeitem)
        {

          if (nodeitem.code == 0)          /* this is a FT */
             {
              /* Calculate the time for the FT to make a complete cycle */
              FTcur = s.now();
              if (FTprev >= 0 && FTcur >= 0) {
/*
                 printf("%s : FTcur=%f, FTprev=%f \n",name.str, FTcur, FTprev);
*/
                 stVal(&FTtime,(FTcur - FTprev));
                 stp.doFT(FTcur - FTprev);
                 FTprev = FTcur;
                 }
              else if (FTprev < 0 && FTcur >= 0)
                     FTprev = s.now();
              if (packPresent == 1) {  /* A packet is ready for transmission */
                 /* Get ready for the transmission of the packet */
                 item.code = 1;
                 item.length = CTlength + datalen;
                 item.id = nid;
                 packPresent = 0;
                 /* Calculate transmission delay for the packet */
                 transmit_t = s.now();
                 if (transmit_t >= 0.0 && begin_t >= 0.0) {
                    stVal(&trans_delay,(transmit_t - begin_t));
                    stp.doDelay(transmit_t - begin_t);
/*
                printf("%s: begin_t=%f, transmit_t=%f, transmit delay=%f\n",
                       name.str,begin_t,transmit_t,transmit_t-begin_t);
                printf("%s: no=%d,max=%f,sumv=%f \n",name.str,
                  trans_delay.nv, trans_delay.maxv,trans_delay.sumv);
*/

                 if (transPassive == 1)    /* Unblock a blocked source */
                    { transPassive = 0;
/*
  printf("%s : unblock a blocked source to transmit packet\n",name.str);
*/
                      s.active();
                    }
                 /* a FT has to be sent after the packet is transmitted */
                 sendFT = 1;
/*
     printf ("%s : a CT & data ready for transmission \n", name.str);
*/
                 }
              else {
                    /* No packet is available for transmission, just */
                    /* pass the FT to the server.                    */
                    item.code = 0;
                    item.length = FTlength;
                    item.id = 0;
/*
    printf ("%s : a FT is received, no data ready for transmission \n",
            name.str);
*/
                   }
              /* Set the flag to notify that a packet/token is ready */
              /* for transmission to network.                        */
```

Appendix E

Literature Search Summary

Appendix E

Summary of Literature Search

A bibliographic list of the technical reports, research articles, and books examined during the one-year period of this project is given at the end of this appendix. Below, we give a brief discussion summarizing our literature survey.

Local Area Networks- General

Various choices exist for the implementation of the medium access protocol of local area networks. Three of these protocols are known as the Carrier Sense Multiple Access with Collision Detection (CSMA/CD), Token-Passing Bus, and Token-Ring access methods. Standards of these protocols are given in [ANSI85a], [ANSI85b], and [ANSI85c], respectively. In this project, we focussed our efforts on examining and evaluating the CSMA/CD and the token-ring protocols.

Several books and articles contain overviews and general treatment of the different aspects of local area networks. A survey and a comparison study of real-time transport protocols are given in [STRA88]. General discussions on the concept of LANs and their technology can be found in [TROO81], [STAL84], and [TANE81]. Articles and books on the performance evaluation and simulation of computer systems/networks [e.g., LAVE83, YEH79] also contain information and concepts relevant to the analysis and evaluation of local area networks.

Simulation Networks

This project has focussed on SIMNET as a model for interconnecting a large number of combat vehicle simulators. Description of SIMNET protocols and the developmental progress of the SIMNET project in the last two years are given in [POPE89], [POPE88], and [POPE87]. A technical report published by BBN [FRIE88] presents the results of a high-level simulation for computing estimates of the performance of the SIMNET protocol running under ETHERNET. The implementation of a wide-area network for SIMNET is discussed in [MILL88]. This latter article discusses techniques to successfully link ground and aircraft simulators (at widely dispersed LAN sites) into a long-haul communication network.

CSMA/CD and ETHERNET Protocols

The literature on CSMA/CD and ETHERNET protocols is rich and growing. The ANSI/IEEE standards for the CSMA/CD protocol are given in [ANSI85a]. A classical description of the technical aspects of ETHERNET-type protocols is given in [METC76]. Methods to determine the maximum and mean data rates in LANs is proposed in [STUC83]. Theoretical aspects of the ETHERNET protocols are covered in several papers and journal articles which present approximate analytical models for the evaluation of ETHERNET performance. A simple model for computing an estimate of the maximum throughput of ETHERNET is presented in [METC76]. Two more sophisticated models for computing estimates of the maximum throughput and the delay-throughput characteristics of ETHERNET are presented in [LAM80] and

[TOBA80]. All these models assume that the channel is slotted with slot-time related to the end-to-end propagation delay. According to Gonsalves' work [GONV85], the analytical predictions of these models are estimated to vary from approximately correct to quite optimistic.

Token-Ring Networks

Token ring LANs have received considerable attention in the past decade. A tutorial and discussion of the important aspects of the IBM token-ring architecture can be found in [STRO83] and [DIXO83]. A real-time messaging system for token-ring networks is described in [SIMO88a]. This system is operational in shipboard environment and its design conforms to the IEEE 802.5 standards and is consistent with the Navey's SAFENET specifications. More aspects of this real-time token-ring LAN is given in [WEAV88]. A heuristic algorithm for computing estimate values of mean packet delays in a token-ring LAN is presented in [BERR83].

Recently, a token-ring protocol based on fiber-optics technology has emerged. The protocol, called Fiber Distributed Data Interface (FDDI), is poised to become the dominant high-end LAN of the 1990's. A discussion of the emerging FDDI standards along with some technical and commercial considerations are given in [MARR89]. A preliminary performance evaluation study on FDDI token-ring networks is presented in [SIMO88b]. In [KOLN87], a fiber optic LAN called FINEX is proposed to meet the specifications of the FDDI standards. The FINEX LAN implements the lower four layers of the OSI protocol stack; the medium access controller and physical layer are implemented in discrete logic to the FDDI specification.

Programming Languages for LAN Simulation

Several programming languages were considered and evaluated for the purpose of simulating ETHERNET and token-ring local area networks. Based on both suitability for simulation and availability, the final set of candidate languages was narrowed down to six members: the concurrent language Concurrent-C [GEHA86], the simulation language SIMSCRIPT [CACI87], the general-purpose language C [KERN78], the concurrent language CSP/K [HOLT78], the concurrent language Concurrent-Euclid [HOLT83], and the simulation language SLAM [PRIT84]. Because of its powerful synchronization and concurrency aspects, the Concurrent-C language was selected for the implementation of the simulation models of ETHERNET and token-ring LANs.

Network Analyzers

LAN protocol analyzers are useful in analyzing network traffic problems and in evaluating existing protocols. The network analyzer HP 4972A has been used in this project to monitor and intercept packets transmitted on the SIMNET ETHERNET network at the Simulation Networking Laboratory at IST. Documentation of the HP 4972A analyzer is given in [HEWL87]. The paper by Haugdahl [HAUG88] examines LAN analyzers from a benchmarking perspective and discusses some of the basic data capture, filtering and traffic analysis features of LAN analyzers.

References

[ANSI85a] ANSI/IEEE- International Standard 8802/3 "Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specification" IEEE Computer Society Press, 1985.

[ANSI85b] ANSI/IEEE- Draft International Standard ISO/DIS 8802/4 "Token-passing bus access method" IEEE Computer Society Press, 1985.

[ANSI85c] ANSI/IEEE- International Standard 8802/5 "Token ring access" IEEE Computer Society Press, 1985.

[BERR83] Berry, R. and Chandy, K. "Performance models of token ring local area networks" Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, 1983, pp. 29-31.

[CACI87] CACI, Inc. *SIMSCRIPT II.5 Programming Language.* CACI, Inc.-Federal Los Angeles, CA, 1987.

[DIXO83] Dixon, R. ; Strole, N. and Markov, J. "A token ring network for local data communication" IBM system Journal, Vol. 22, 1983, pp. 74-62.

[FRIE88] Friedman, D. and Haimo, V. "SIMNET ETHERNET performance" BBN Technical Report No. 6711, BBN Communications Corporation, MA 1988.

[GARV88] Garvey, R. and Radgowsjki, T. "Data collection and analysis: the keys for interactive training for combat readiness" Proceedings of I/ITSC Conference, 1988, pp. 572-576.

[GEHA86] N. Gehani and Roome, W. "Concurrent C" Technical Report, AT & T Bell Laboratories, 1986.

[HAUG88] Haugdahl, J. "Benchmarking LAN protocol analyzers" Proceedings of 13th IEEE Conference on Local Computer Networks, 1988, pp. 375-384.

[HEWL87] Hewlett-Packard, *HP 4792A LAN Protocol Analyzer: Vol. I: Getting Started, Vol. II: Operating Manual.* HP Telecommunication Division, Colorado, 1987.

[HOLT83] Holt, R. *Concurrent Euclid, The Unix System and Tunis.* Addison-Wesley, Reading MA, 1983.

[HOLT78] Holt, R. et al *Structured Concurrent Programming with Operating System Applications.* Addison-Wesley, Reading MA, 1978.

[KERN78] Kernighan, B. and Ritchie, D. *The C programming Language.* Prentice-Hall, 1978.

[KOLN87] Kolnik, I and Garodnick, J. "First FDDI local area network" Proceedings of 12th IEEE Conference on Local Computer Networks, 1987, pp. 7-11.

[LAM80] Lam, S. "A Carrier sense multiple access protocol for local networks", Computer Networks, Volume 4, Number 1, February 1980, pp. 21-32.

[LAVE83] Lavenberg, S. *Computer Performance Modeling Handbook.* Academic Press, 1983.

[MARR89] Marrin, K. "Emerging standards, hardware, and software light the way to FDDI" Computer Design, Vol. 28, No. 7, April 1989, pp. 51-57.

[METC76] Metcalfe, R. and D.R. Boggs "Ethernet : Distributed Packet Switching for Local Computer Networks", Communications of ACM, Vol. 19, No. 7, 1976, pp. 395-403.

[MILL88] Miller, D.; Pope, A.; Waters, R. "Long-haul networking of simulators" Proceedings of 10th I/ITSC Conference, 1988, pp. 577-582.

[POPE89] Pope, Arthur "The SIMNET network and protocols" BBN Report No. 7102, BBN Communications Corporation, MA, July 1989.

[POPE88] Pope, Arthur "The SIMNET network and protocols" BBN Report No. 6787, BBN Communications Corporation, MA, May 1988.

[POPE87] Pope, Arthur "The SIMNET network and protocols" BBN Report No. 6369, BBN Communications Corporation, MA, February 1987.

[PRIT84] Pritsker, A. *Introduction to Simulation and Slam II* (2nd Edition) Systems Publishing Corp., West Lafayette, Indiana, 1984.

[SIMO88a] Simonson, R. et al "SHIPNET: a real-time local area network for ships" Proceedings of 13th IEEE Conference on Local Computer Networks, 1988, pp. 424-432.

[SIMO88b] Simonson, R. "Performance analysis of the FDDI token-ring" Technical Report TR-88-02, Department of Computer Science, University of Virginia, 1988.

[STAL84] Stallings, W "Local networks ACM Computing Surveys, Vol. 6, No. March 984 pp. 3-42.

[STRA88] Strayer W and Weaver, A. "Evaluation of real-time transport protocols" Technical Report TR-88-21 Department of Computer Science. University of Virginia, 988.

[STRO83] Strole, N. A local communication network based interconnected token-access rings: tutorial IBM Journal Research and Development, Vol. 27 N 5, 1983, pp. 48 -496.

[STUC83] Stuck, B "Calculating the maximum mean data rate in local area networks IEEE Computer, May 1983 pp. 72-76.

[TANE81] Tanenbaum, A. "Network protocols" ACM Computing Surveys. Vol. 3. No 4, Dec. 198 pp. 453-490.

[TOBA80] Tobagi, F and V.B Hunt "Performance analysis of carrier sense multiple access with collision detection Computer Networks. Volume 4, Number 5, 980. pp. 245-259

[TROO8 Trooper C. *Local Computer Network Technologies* Academic Press, 98

[WEAV88] Weaver, A. and Colvin, M. A real-time messaging system for token-ring networks Technical Report TR-88-08, Department of Computer Science, University of Virginia, 988

[YEH79] Yeh, Simulation of local computer networks' Proceedings of 4th IEEE Conference on Local Computer Networks, October 979, pp. 1-4.

```
    /* Now, a FT should be sent following this.                          */
    or (takeReady == 0 && sendFT == 2) :
        accept takeReq()
         {
          item.code = 0;
          item.length = FTlength;
          item.id = 0;
          sendFT = 0;
          if (s.now() + (float)item.length <= simt)
             totalLength += (float)item.length;
          else totalLength += (simt - s.now());
/*
    printf("%s: A FT has been transmitted to  R server \n", name.str);
*/
          treturn item;
         }

    /* No packet/token is available for transmission, and a FT should */
    /* not be sent.  The consuming-server is being blocked.           */
    or (takeReady == 0 && sendFT != 2) :
        accept takeReq()
         { s.passive();
           takePassive = 1;
/*
        printf("%s : inside takeReq, the R server is blocked \n",name.str);
*/
           treturn noItem;
         }

    /* A token/packet is now available, and can be transmitted to a */
    /* previously blocked server.                                   */
    or (takeReady == 1)   :
        accept takeWait()
         { if (sendFT == 1)
              sendFT = 2;
           takeReady = 0;
           if (s.now() + (float)item.length <= simt)
              totalLength += (float)item.length;
           else totalLength += (simt - s.now());
/*
    printf("%s : data transmitted to R server after being blocked \n",name.str);
*/
           treturn item;


    /* Simulation time is up, unblock any blocked server. */
    or (s.now()<0.0) :
       accept takeWait()
       {
         if (takePassive == 1) {
/*
    printf("%s : forced unblock a server in takeWait\n",name.str);
*/
            s.active();
            takePassive = 0;
            }
          treturn item;
       }

    /* Accept add and drop requests from producer and consumer clients */
    or accept addCons()  { nCons += 1; }
    or accept addProd()  { nProd += 1; }
    or accept dropCons() { nCons -= 1; }
    or accept dropProd() { nProd -= 1; }

   }
```

```
              takeReady = 1;
              if (takePassive == 1)   /* Unblock any blocked server */
                 { takePassive = 0;
/*
   printf("%s : unblock a server as a FT permits transmission\n",name.str);
*/
                 s.active();


          else                    /* This is a CT */

               /* The CT/packet is not initiated by this node, therefore */
               /* just pass the CT/packet to the next node.             */
               if (nodeitem.id !=  nid) {
                  item.code = 1;
                  item.length = nodeitem.length;
                  item.id = nodeitem.id;
                  takeReady = 1;
/*
    printf("%s : a CT is ready to be passed to the next node\n",name.str);
*/
                  if (takePassive == 1) {
                     takePassive = 0;
/*
   printf("%s : unblock a blocked server as a CT avail for transmit\n",name.str);
*/
                     s.active();
                     }
               }
               else   {
                  /* The CT/packet just received is initiated by the node, */
                  /* it has to be withdrawn from the network.              */

/*
                  printf("%s: CT&data withdrawn from network \n",name.str);
*/
                  /* Increment the number of packets that have completed */
                  /* transmission through the network.                   */
                  noPackets++;
                     }



   /* take Requests by the consuming-server. */

   /* A packet/token is ready to be passed to the server */
   or (takeReady == 1) :
       accept takeReq()
        {
         takeReady = 0;
         /* If the CT/packet is sent by this node, then a FT has to be */
         /* sent after the packet has been transmitted.                */
         if (sendFT == 1)
             sendFT = 2;
         /* Keep statistics of the total length of packet that has been */
         /* transmitted.                                                */
         if (s.now() + (float)item.length <= simt)
            totalLength += (float)item.length;
         else totalLength += (simt - s.now());
/*
   printf ("%s : A CT or FT has been transmitted to R server \n", name.str);
*/
         treturn item;


   /* A CT/packet has just been initiated and transmitted by this node. */
```

```
}

/* Print out the statistics for the node */
printf("%s : total no of packets transmitted through this node = %d \n",
       name.str, noPackets);
utilization = totalLength/(transRate * simt);
printf("%s : total length = %f, utilization in the node = %f \n",
        name.str, totalLength, utilization);
printf("%s : Transmit_delay : Avg = %f, Sdev=%f, Max=%f \n",
        name.str, stMean(&trans_delay), stSdev(&trans_delay),
        trans_delay.maxv);
printf("%s : FTtime : Avg=%f, Sdev=%f, Max=%f \n", name.str,
        stMean(&FTtime), stSdev(&FTtime), FTtime.maxv);
stp.doThruput(noPackets);
stp.doUtil(utilization);
stp.dropCus();

 printf ("Node process : %s terminates \n", name.str);

 s.active();
 s.dropUser();

}
```

```
/****************************************************************************/
/*                                                                          */
/*   The source  process produces  packets according  to an  exponential   */
/*   distribution.  Once a packet is generated, the sourcee attempts to     */
/*   transfer it to the node process.  If there is already a packet waiting */
/*   to be transmitted in the node, the source and the transfer is blocked  */
/*   until the packet originally in the node has been transmitted to the    */
/*   network.  At that time, the source is reactivated to complete the      */
/*   transfer.  After transfer is made, the source will determine if a      */
/*   delay request (and if yes, how long) is needed to produce the next     */
/*   packet based on the curreent time and the time that the next packet    */
/*   should be generated according to the exponential distribution.         */
/*                                                                          */
/*         VARIABLE DICTIONARY                                              */
/*                                                                          */
/*   done      :  a flag to determine if the loop for generating packets    */
/*                should be exited.                                         */
/*   length    :  length of the packet generated.                           */
/*   sid       :  id of the source process.                                 */
/*   start_t   :  time when the packet is generated.                        */
/*   t, ts     :  variables to keep track of time after a delay request.    */
/*   t1        :  a variable to hold the random number generated from the   */
/*                exponential distribution.                                 */
/*   time      :  time after a packet has been successfully transferred to  */
/*                the node.                                                 */
/*                                                                          */
/****************************************************************************/


#include "dcls.h"
#include "stdio.h"
     /* Poisson source - make transmission request to node */
process body source(s,nid,meanIit,packlength,name)
{
float t,t1,time,ts,start_t;
int done=0;
int sid;

/* Initialization phase */
sid = c_mypid();
c_setname(sid,name.str);

/*
printf ("Source : %s is created, id = %d \n", name.str,sid);
*/

/* Add to the scheduler and to the associated node */
s.addUser(sid); nid.addProd();

/* Generate a random inter-arrival time */
t = (float)erand(meanIit);

/* length = (long)(drand01()*2500); */

/* Request to delay for the inter-arrival time */
ts = s.wait(s.reqDelay(t));

/* Keep track of the time when the packet is generated */
start_t = s.now();

time = nTransfer(nid,s,packlength,start_t);
while(!done){
     t1 = (float)erand(meanIit);
/*     length = (long)(drand01()*2500);    */
/*
   printf ("%s : random time, t1= %f, length of packet = %d \n", name.str
```

```
            t1, length);
*/
     /*  Request for additional delay is made if the time after a transfer
         is made, is less than the time that the next packet should be
         generated according to an exponential distribution.          */
     if (time <= t+t1)
     ts = s.wait(s.reqDelay(t+t1-time));
     t += t1;
/*
printf("%s : before transfer A, want to transmit packet to node \n",name.str);
*/
     start_t = s.now();
     time = nTransfer(nid,s,packlength,start_t);
/*
printf("%s : after transfer A, time = %f \n", name.str, time);
*/
     if(time < 0)
     done = 1;
} /* end while */

/* Termination phase */
nid.dropProd();

printf ("Source : %s terminates \n", name.str);
s.dropUser();
}
```

```
/*****************************************************************************/
/*                                                                           */
/*   The server process acts as a transmission medium between two nodes:    */
/*   input node and output node.  The server requests to take a packet/      */
/*   token from the input node.  If the request is unsuccessful, the server  */
/*   is blocked until a packet/token is available.  Upon receiving a packet/ */
/*   token, the server will start to transmit it to the output node.  Each   */
/*   server maintains a linked list to keep track of the id of the originat- */
/*   ing node, remaining transmission and propagation time, etc. of the      */
/*   packets/tokens that are in the transmission process to the output node. */
/*                                                                           */
/*   Depending on the type of packet received (a FT or a CT with message),   */
/*   the remaining propagation time of the packet in the front, and  the     */
/*   remaining transmission time for the packet just received, the server    */
/*   makes the appropriate request for delay.                                */
/*                                                                           */
/*   When the packet/token in the front arrives at the output node, the      */
/*   server calls the output node to receive it.                             */
/*                                                                           */
/*       VARIABLE AND FUNCTION DICTIONARY                                    */
/*                                                                           */
/*   freeTokenReceived  :  a flag to indicate if a free token has been       */
/*                         received or not.                                  */
/*   *front      :  pointer to the front of the linked list.                 */
/*   nid         :  id of the server process.                                */
/*   nodeitem    :  the packet received from the input node.                 */
/*   *p          :  pointer to a structure of the linked list.               */
/*   prtlist()   :  function used to print the contents of the linked list.  */
/*   *rear       :  pointer to the rear of the linked list.                  */
/*   t           :  time after a request delay.                              */
/*   testandCall()   :  a function used to check if the packet in the front  */
/*                      has arrived the output node.  If so, the server      */
/*                      will call the output node to receive the packet.     */
/*   timeUpdate()    :  a functionn to update the remaining transmission     */
/*                      time and propagation time for all the packets in the */
/*                      linked list after a time delay.                      */
/*   transTime   :  the remaining transmission time for the packet.          */
/*                                                                           */
/*****************************************************************************/

#include "dcls.h"
#include "stdio.h"

/*  a structure to contain information about a packet during transmission  */
struct sItem {
      int    code;     /*  to indicate if it is a FT(0) or a CT(1)  */
      long   length;   /*  the length of the packet  */
      int    id;       /*  the id of the node initiating the packet  */
      float  remainProp;  /*  the remaining propagation time for packet  */
      float  remainTrans; /*  the remaining transmission time for packet  */
      int    arrived; /* flag to check if the 1st bit of packet has arrived */
      struct sItem   *next;  /* pointer to the next structure  */
     };


/*  This function is used to update the remaining transmission time and   */
/*  propagation time for all the packets in the linked list after a time  */
/*  delay.                                                                */
void timeUpdate (front,time)
   struct sItem   *front;
   float          time;
{
struct sItem   *s;

 /*
printf("Server : beg of timeUpdate \n");
```

```
printf("%s is created \n",name.str);
*/
s.addUser(nid);
inN.addCons();
outN.addProd();

front=rear=NULL;

while (s.now() >=0) {      /*  simulation has not ended  */

/* printf("%s : 1st line after while loop \n",name.str);  */
   nTake (inN, &nodeItem);
/* printf("%s : 2nd print line after while loop \n",name.str); */
   p = (struct sItem *)malloc(sizeof(struct sItem));
   p->code = nodeItem.code;
   p->length = nodeItem.length;
   p->id = nodeItem.id;
   p->arrived = 0;
   transTime = nodeItem.length/transRate - 1/transRate;
   t=s.wait(s.reqDelay(1/transRate));
/*
printf("%s : p=%d, packet code = %d, packet length = %d ,p->arrived=%d \n",
        name.str,p, p->code, p->length,p->arrived);
*/

   if (p->code == 0) {    /*  This is a free token, nothing will follow it  */
      if (front == NULL) {  /* There is no C.T. & packets in front of it  */
        if (transTime <= propTime) {
         t = s.wait(s.reqDelay(propTime));
/*
   printf ("%s : 1 FT propagated to next node, curtime = %f \n", name.str,
            t);
*/
         outN.put(nodeItem);
         free(p);
        }
      else  {   /* transTime greater than propTime */
/*
    printf("%s : 2 FT, request to delay for propTime \n",name.str);
*/
         t = s.wait(s.reqDelay(propTime));
/*
    printf("%s : 3 FT, curTime = %f \n", name.str,t);
*/
         outN.put(nodeItem);
/*
    printf("%s : 4 FT, request to delay for the remaining transTime \n",
          name.str);
*/
         t = s.wait(s.reqDelay(transTime - propTime));
/*
   printf("%s : 5 FT, curTime = %f \n", name.str,t);
*/
         free(p);
        }
      }
     else  {   /* still a FT but there are still packets in front of it */
      p->remainProp = propTime;
      p->remainTrans = transTime;
      p->arrived = 0;
      p->next = NULL;
      rear->next = p;
      rear = p;
      prtlist(front,name);
   /*  Remark:       There should be no packet immediately following
                     a FT so that all the packets and the FT can be
```

```
                        transmitted to the next node, without any incoming
                        tokens or packets   */
     while (front != NULL)  {
           if (rear->remainTrans > 0) {
              if (front->remainProp <= rear->remainTrans) {
/*
      printf("%s : 6 FT, request to delay for the front remainProp \n",
              name.str);
*/
                 t = s.wait(s.reqDelay(front->remainProp));
/*
     printf("%s : 7 FT, curTime = %f \n",name.str,t);
*/
                 timeUpdate (front,front->remainProp);
                 }
              else   {
/*
     printf("%s : 8 FT, request to delay for rear remainTrans \n",
             name.str);
*/
                 t = s.wait (s.reqDelay(rear->remainTrans));
/*
     printf("%s : 9 FT, curTime = %f \n",name.str,t);
*/
                 timeUpdate (front, rear->remainTrans);
                 }
           }
           else
/*
     printf("%s : 10 FT, request to delay for front remainProp \n",
             name.str);
     printf("%s : 10.5 FT, front->remainProp = %f \n",name.str,
             front->remainProp);
*/
              t = s.wait (s.reqDelay(front->remainProp));
/*
    printf("%s : 11 Ft, curTime = %f \n",name.str, t);
*/
              timeUpdate (front,front->remainProp);
              }
/*
    printf("%s : before testandCall, front=%d, rear=%d\n",name.str,front,rear);
    prtlist(front,name);
*/
           testandCall (front,rear,outN);
if ((front->remainProp <= 0.0001) && (front->remainTrans <= 0.0001)) {
   p = front;
   front = front->next;
   if (front == NULL)
      rear = NULL;
   free(p);
   }
/*
   prtlist(front,name);
*/
/*
printf("%s : after, front=%d,rear=%d \n",name.str,front,rear);
*/
          }
      }

   else  {  /* This is a CT, note that a CT is immediately followed by another
              CT or a FT */
      if (front == NULL) {
         if (transTime <= propTime) {
            p->remainProp = propTime - transTime;
```

```
                p->remainTrans = 0;
                p->arrived = 0;
                p->next = NULL;
/*
  printf("%s: before, front=%d, rear=%d \n",name.str, front,rear);
*/
                front=rear=p;
/*
  printf("%s: after, front=%d, rear=%d \n",name.str,front,rear);
          prtlist(front,name);
*/
/*
   printf("%s : 12 CT, request to delay for transTime \n",name.str);
*/
                t=s.wait(s.reqDelay(transTime));
/*
   printf("%s : 13 CT, curTime = %f \n",name.str, t);
*/
                }
              else  {
                p->remainProp = 0;
                p->remainTrans = transTime - propTime;
                p->next = NULL;
                front=rear=p;
/*
    prtlist(front,name);
   printf("%s : 14 CT, request to delay propTime \n", name.str);
*/
                t = s.wait(s.reqDelay(propTime));
/*
   printf("%s : 15 CT, curTime = %f \n", name.str,t);
   prtlist(front,name);
*/
                testandCall(front,rear,outN);
if ((front->remainProp <= 0.0001) && (front->remainTrans <= 0.0001))
   p = front;
   front = front->next;
   if (front == NULL)
      rear = NULL;
   free(p);
   }
/*
prtlist(front,name);
printf("%s: after, front=%d,rear=%d \n",name.str,front,rear);
   printf("%s : 16 CT, request to dealy for remain transTime \n",name.str);
*/
                t = s.wait (s.reqDelay(p->remainTrans));
/*
   printf("%s : 17 CT, curTime = %f \n",name.str, t);
*/
                timeUpdate (front,p->remainTrans);
/*
prtlist(front,name);
*/
                testandCall(front,rear,outN);
if ((front->remainProp <= 0.0001) && (front->remainTrans <= 0.0001)) {
   p = front;
   front = front->next;
   if (front == NULL)
      rear = NULL;
   free(p);
   }
/*
prtlist(front,name);
printf("%s: after, front=%d,rear=%d \n",name.str,front,rear);
*/
```

```
              }
           }
       else {  /* a CT but there are packets in front of it */
          p->remainProp = propTime;
          p->remainTrans = transTime;
          p->arrived = 0;
          p->next = NULL;
          rear->next=p;
          rear=p;
/*
         prtlist(front,name);
*/
         while (rear != NULL && rear->remainTrans > 0.0)   {
          if (front->remainProp <= rear->remainTrans && front->remainProp > 0.0) {
/*
   printf(" %s : 18 CT, request to delay for front remainProp \n",name.str);
*/
               t = s.wait(s.reqDelay(front->remainProp));
/*
   printf("%s : 19 CT, curTime = %f \n", name.str, t);
*/
               timeUpdate (front,front->remainProp);

            else {
/*
   printf("%s : 20 CT, request to delay for rear remainTrans \n", name.str);
*/
               t = s.wait (s.reqDelay(rear->remainTrans));
/*
   printf("%s : 21 CT, curTime= %f \n", name.str, t);
*/
               timeUpdate (front,rear->remainTrans);
               }
/*
prtlist(front,name);
*/
            testandCall(front,rear,outN);
if ((front->remainProp <= 0.0001) && (front->remainTrans <= 0.0001)) {
   p = front;
   front = front->next;
   if (front == NULL)
      rear = NULL;
   free(p);
   }
/*
prtlist(front,name);
printf("%s: after, front=%d,rear=%d \n",name.str,front,rear);
*/
            }

        }
    }

    outN.dropProd();
    inN.dropCons();
/*
printf ("%s terminates \n", name.str);
*/
    s.dropUser();

}
```

```
/*************************************************************************/
/*                                                                       */
/*   statProc collects statistics from all the node processes, calculates */
/*   the overall network statistics and prints them at the end of simul-  */
/*   ation.  The statistics include  network throughput, network utili-   */
/*   zation, time between a  packet is generated  and  transmitted,       */
/*   average time delay for a free token to make a complete cycle.       */
/*                                                                       */
/*            VARIABLE DICTIONARY                                        */
/*                                                                       */
/*   FTtime    :  a structure to collect statistics for a free token cycle */
/*                time.                                                  */
/*   nCus      :  number of client processes.                            */
/*   totalPackets  :  total number of packets that have finished trans-  */
/*                    mission and propagation through the network.       */
/*   trasmit   :  a structure used to collect statistics for a packet    */
/*                transmission delay.                                    */
/*   util      :  a structure used to collect statistics on utilization. */
/*                                                                       */
/*************************************************************************/


#include "dcls.h"
#include "stdio.h"

process body statProc(simt)
{
        int  nCus=0;
        int  totalPackets=0;
        stats  util, transmit, FTtime;

c_setname(c_mypid(),"statProc");

stInit(&util); stInit(&transmit); stInit(&FTtime);

accept addCus() {nCus++;}

while (nCus > 0) {
  /* Accept calls from other processes and collect the statistics */
      select {
        accept doThruput(noPackets) { totalPackets += noPackets; }
        or
        accept doUtil(utilization) { stVal(&util,utilization); }
        or
        accept doDelay(trans_delay) { stVal(&transmit, trans_delay); }
        or
        accept doFT(FTdelay) { stVal(&FTtime, FTdelay); }
        or
        accept addCus() {nCus++;}
        or
        accept dropCus() {nCus--;}
        }  /* select */
      } /* while */

/* Print the statistics at the end */
printf ("\n**************   STATISTICS   **************\n");
printf ("      total no of completed packets = %d \n",totalPackets);
printf ("      network throughput = %f \n",totalPackets*10000000/simt);
printf ("      network utilization = %f \n", stMean(&util));
printf ("      delay before packet is transmitted = %f \n", stMean(&transmit));
printf ("      cycle time for a free token = %f \n", stMean(&FTtime));
printf ("**************************************************\n\n");

}
```

```
/*************************************************************************/
/*                                                                       */
/*   This is a file containing a number of functions called by processes */
/*   within the program, mainly for sending information about statistics. */
/*   The functions used are :                                            */
/*     drand01()     :  return a random number uniformly distributed in  */
/*                      the range 0 to 1.                                */
/*     uniform()     :  returns a uniformly distributed number.          */
/*     erand()       :  returns exponentially distributed random number. */
/*     stInit()      :  initializes a stats structure by setting its     */
/*                      components to zero.                              */
/*     stVal()       :  adding a value to a stats structure pointed to by */
/*                      p, the value added is v.                         */
/*     stMean()      :  computes the mean of a stats structure.          */
/*     stSdev()      :  computes standard deviation of a stats structure. */
/*                                                                       */
/*************************************************************************/


#include "dcls.h"
#include "math.h"

double drand01()        /* Return random number uniform in range [0,1]. */
{       double  p;
        extern long random();

        p = random() & 0xffffff;
        return p / 0xffffff;
}

int uniform()
{
        static int cc = 0

        long random(), g;

        if ( cc <= 0 ){
        srandom(getpid ()   ;
        cc++ ;

        g =  (random() & 0x7f);
        while (g <= 0)
        {g =  (random() & 0x7f);}
        return(g);
}


/*  returns an exponentially distributed random number  */
double erand(meantime)
    float  meantime;
{
double mul;
double num, num1;

mul =   1 / meantime ;
num =  uniform()/127.0 ;
num1 = - log(num) / mul ;

return(num1);
}


/*
long erand (mean)
     long   mean;
```

```
        long  r;
        r=-((double)mean) * log(1.0 - drand01()  + 0.5;
        if (r == 0) r = 1;
        return r;
}
*/


/*  Initializes a 'stats' structure   */
void stInit(p)
        stats   *p;
{
        p->nv = 0;  p->maxv = -1;  p->sumv = 0;  p->sumsq = 0;
}


/*  Add a value to a 'stats' structure.  */
void stVal(p, v)
        stats   *p;
        float   v;
{
/*
printf("Inside stVal : beginning, v=%f\n",v);
*/
        if (p->nv == 0 || v > p->maxv)
                p->maxv = v;
        p->nv++;  p->sumv += v;  p->sumsq += v*v;
/*
printf("End of stVal \n");
*/



/*  Returns the mean for a "stats" structure  */
double stMean(p)
        stats   *p;

        return p->nv != 0 ? p->sumv/p->nv : 0.0;
}


/*  Returns the standard deviation for a "stats" structure  */
double stSdev(p)
        stats   *p;
{       double avg;

        if (p->nv == 0)
                return 0.0;
        else {
                avg = p->sumv/p->nv;
                return sqrt(p->sumsq/p->nv - avg*avg);
        }
}
```

Appendix C

Copy of Paper Published in Proceedings of the First IST Networking Conference April, 1989

SIMULATION NETWORKING AND PROTOCOL ALTERNATIVES

M. Bassiouni, Department of Computer Science
M. Georgiopoulos, Department of Electrical Engineering
J. Thompson, Institute for Simulation and Training

Graduate Student Assistants: S. Chaterjee, M. Chiu and N. Christou

University of Central Florida
Orlando, FL 32816

ABSTRACT

In this paper, we focus on the implementation of an efficient local area network (LAN) which will be used to interconnect simulation training devices. In particular, we present preliminary efforts in modeling and analyzing the performance of three different network protocol access methods: CSMA/CD (Carrier Sense Multiple Access with Collision Detection), Virtual Token-Passing Bus Access Protocols and Token-Ring Access. A detailed discussion of the advantages and disadvantages of the above access protocols and anticipated results are also presented.

INTRODUCTION

The networking of simulation training devices departs from the traditional use of computer networks whose purpose is to allow for the sharing of computing resources among multiple computers. In the application of networking simulators, the network is used almost exclusively for communication of process state information between training devices engaged in the training exercise.

There are many inherent limitations to using a network in this application. For example, as the number of simulators on the network and workload per simulator increases, there will be a deterioration in throughput and degradation of other performance measures. If throughput delays become significant, the effectiveness of a real-time training simulation may be overly compromised due to the time-critical response requirements in the simulation of true-to-life, action-requiring training scenarios. Depending upon communication protocols, there may also be an increase in the frequency of retransmissions and lost or distorted messages. The magnitude of this problem is functionally related to how data is distributed throughout the system, and the soundness of the network access and internal network protocols.

Various choices exist for the implementation of a local area network (LAN), (e.g. transmission medium, topology, access protocols, etc.) to interconnect simulation devices. In this paper, we present efforts in modeling and analyzing the performance of three different network protocol access methods. In particular, the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) such as ETHERNET (ANSI/IEEE 802.3 Standards [1,2]), the Virtual Token-Passing bus protocols such as the Generalized Broadcast Recognizing Access Methods (GBRAM) [3], and Token-Ring Access protocols (ANSI/IEEE 802.5 Standards [4,5]) are examined.

SYSTEM MODEL

Our system consists of a complex web of armor, fixed and rotary wing aircraft, and air-defense simulated vehicles linked together via a Local Area Network (LAN) to create a simulated world in which war-gaming can be conducted. In our system, combat forces and their commanders must move, shoot, communicate and navigate just as they do in a real battle. Hence, a tremendous amount of information must be exchanged among the simulators in real-time if a realistic battle scenario is to be created.

Local Area Networks can be characterized by the following factors:

- transmission medium (coaxial cable, twisted pair, optical fiber)
- modulation scheme (baseband, broadband)
- wiring scheme (bus or ring)
- medium-access control schemes (random-access or controlled-access).

We intend to investigate the capability of three LAN's to interconnect the simulators. Two of these LAN's are **bus networks**, which utilize baseband transmission to send messages over a coaxial cable. The medium-access control schemes for one is the ETHERNET protocol [2] and for the other is Generalized Broadcast Recognizing Access Method (GBRAM) protocol [3]. The third LAN is a **ring network**, which utilizes baseband transmission to send messages over a fiber optic cable. Its medium-access control scheme is a token passing protocol.

In the ETHERNET protocol, if a simulator, or other node, has a packet ready to transmit onto the network, it monitors the network to determine whether any transmissions are in progress. If a transmission is in progress, the network is said to be "busy", otherwise, it is "idle". If the node finds the network busy, transmission of the data packet is deferred. When it finds the network idle, packet transmission is initiated. If multiple nodes attempt to transmit at the same time, their transmissions interfere, or collide. The collision is acknowledged by each transmitting node sending out a bit sequence onto the network referred to as a "jam-signal". After the jam-signal has been transmitted, the nodes involved in the collision schedule a retransmission attempt at a randomly selected time in the future.

In the GBRAM protocol, the nodes employ a "virtual-token" scheme in which each node gains network access (the virtual token) at a unique time which is determined by a decentralized scheduling function, hence avoiding collisions completely.

The Token-ring access protocol is even more straight-forward. A node gains the right to transmit onto the network when it detects and captures a free token passing on the network medium. The token is a control signal that circulates on the medium following each information transfer. Any node, upon detection of a free token, may capture the token, set it to busy, and then send its packet. Upon completion of transmitting its data, and after appropriate checking for proper operation, the node generates and transmits a "free token" which begins circulating around the network and provides other nodes the opportunity to gain network access.

Bus Network Topology System Configuration

The system configuration corresponding to the bus network topology is shown in Figure 1. In this CSMA/CD (ETHERNET) implementation, up to eight simulators or other types of nodes can be connected through an ETHERNET multi-port transceiver to a single point on the ETHERNET coaxial cable, via a media-access unit (vampire tap). A single coaxial cable is available to link all simulators together.

Some important parameters pertaining to the implementation of the ETHERNET and the GBRAM protocols are as follows:

- the time that it takes for a message to traverse the medium
- the time elapsed from the instant the coaxial cable becomes idle or becomes busy until the node "realizes" that the cable is idle or busy
- the time elapsed from the moment that a transmitting node realizes that it is involved in a collision until it generates the first bit of the jam-signal

Ring Network Topology System Configuration

The system configuration corresponding to the ring network topology is shown in Figure 2. A ring network consists of a closed sequence of individual point-to-point (node-to-node) links. For efficient operation, the token protocol dictates a minimal delay per station, and the ability to change a single bit in the data stream (e.g. the token) "on-the-fly". An important parameter pertaining to the implementation of a token ring protocol is the time it takes for the data to propagate through a node on the network.

Node Traffic Generation

Each node generates a certain amount of traffic into the network. In the simulation of the network traffic, some of the options for the packet inter-arrival time at a node site are:

- Exponential - the traffic generated by the simulator is a Poisson process

  Fixed with a specified percentage of "jitter" - a fixed time, plus or minus a random time within the specified percentage of the fixed time

  Uniformly distributed in a specified interval

  Trace-driven - the traffic used to drive the network is a trace of real network traffic data.

One of the nodes in our network operates differently from ordinary simulator units. It produces network packets for a large quantity of different types of simulated vehicles. It transmits the data packets for a portion of its simulated vehicles at regular intervals. Hence, its traffic can be characterized as periodic.

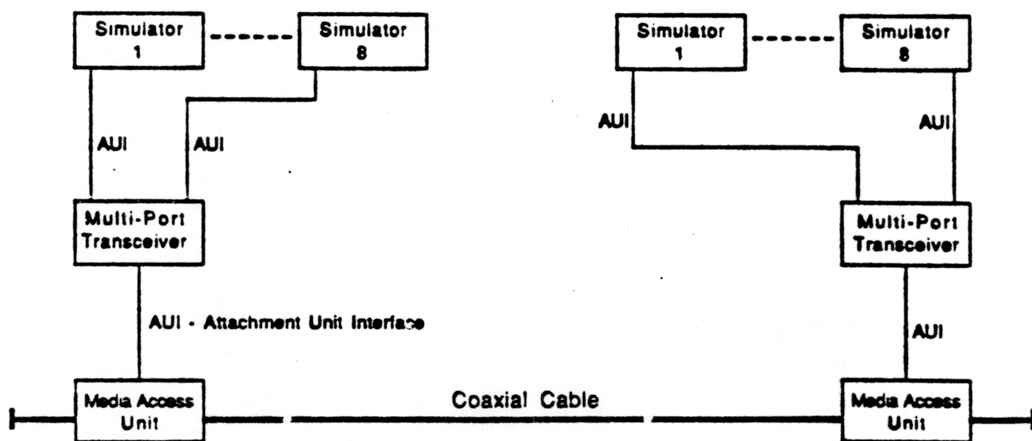


Figure 1. Bus Network Topology System Configuration

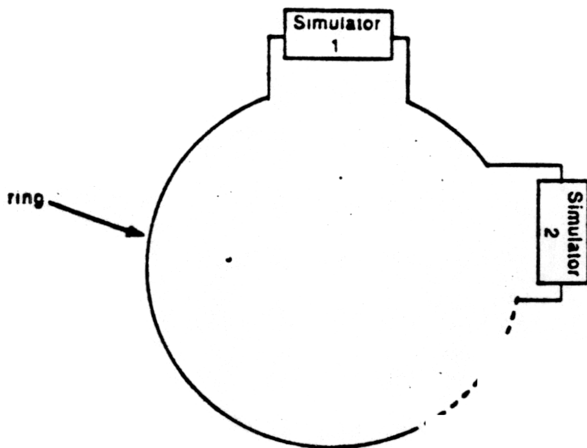


Figure 2. Ring Network Topology System Configuration

THE ETHERNET SIMULATION MODEL

In this section, we give a high-level description of the simulation model used in evaluating and predicting the performance of the CSMA/CD implementation of ETHERNET. The simulation model is written in Concurrent-C (an extension of the C programming language with concurrent programming facilities based on the "rendezvous" concept). The powerful synchronization and concurrency aspects of Concurrent-C [6] have provided us with a notationally convenient and conceptually elegant tool for modeling the parallel activities of the simulation network nodes and the underlying networking layer.

The process interaction model of Concurrent-C has been used in our simulation to map the different entities and activities of the simulated network to corresponding Concurrent-C processes. The following process types are the major generic entities used in our simulation. Figure 3 gives a block diagram showing the interactions among these different processes.

- Process **Simnode** is used to represent a vehicle simulator on the network. A process of this type is created for each such simulator.
- Process **Busnode** is used to represent the point of contact of each network node with the ETHERNET bus (coaxial cable). A process of this type is created for each such point of contact on the bus.
- Process **Lserver** is used to implement and control the flow of data (packets and jam signals) in the direction from right to left for each network node. A process of this type is created for each network node.
- Process **Rserver** is analogously defined for traffic flowing in the direction from left to right.
- Process **Scheduler** is used to order time events and control the sequencing of activities of the entire simulation.

Typically, eight simulators connect to the coaxial transmission cable at a single point via a multi-port transceiver. Each of the simulators is modeled as a **Simnode** process. A **Busnode** process for each point of contact is created to receive and transmit local traffic from any one of the eight network nodes, as well as retransmit any external messages arriving at the node. For this purpose, we use two separate processes called **Rserver** and **Lserver**. The **Rserver** process implements the transfer of data from its left **Busnode** process to its right **Busnode** process. This transmission is actually simulated by calling the **Scheduler** process to wait for the propagation delay (the time needed for the message to travel from one network node to the next). The **Lserver** similarly carries data signals from the right **Busnode** to its left neighbor. The **Busnode** process detects collisions of transmitted data by checking for the existence of local traffic, left traffic or right traffic.

The Simnode Process

This process is the source of local traffic. It generates packets according to a specified input method (e.g. using traces of real data or random stochastically generated inter-arrival times such as exponential, uniform, fixed with jitter, etc.). Upon arrival of a local packet, the **Simnode** process makes a request to the corresponding **Busnode** process in order to transmit the new packet. This is done by calling a specific transaction in the **Busnode** process as illustrated by the code presented later. At this point, the **Busnode** process checks for a carrier flag. If the flag has been off for at least the inter-frame gap, the **Simnode** process can proceed with its transmission. If the carrier flag is on, the **Simnode** process must wait for the inter-frame gap

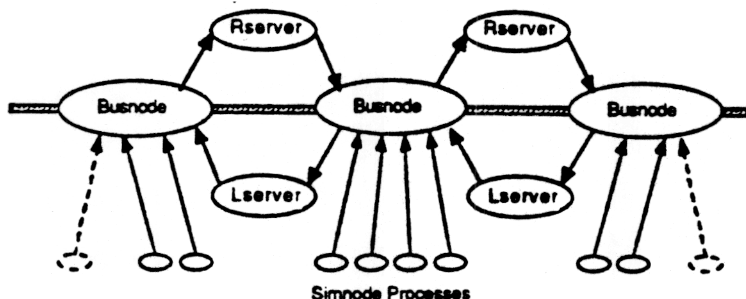


Figure 3. ETHERNET Simulation Model Process Interactions

and then retry its transmission. When a collision is detected during transmission, the **Simnode** process generates and transmits a jam signal and increments the collision counter. This is followed by invoking a back-off algorithm for retransmission. A packet is discarded after 16 unsuccessful transmission attempts. The specification and major activities of the **Simnode** process are described by the following code:

```
Process spec Simnode (process sched s,
                  process bus bid,
        long meanlit, name_t name)

Process body Simnode (s, bid, meanlit, name)
  /* Initialization phase */
    c_setname(c_mypid(), name.str);
    s.adduser(); bid.addProd();

  /* Main processing phase */
    while (not done) do
    /* get arrival time */
    t=erand(meanlit)
    /* call Scheduler to wait for arrival */
    loc_traffic.arrive = s.wait(s.reqDelay(t));
    /* attempt transmission */
    while ((dt = bid.transReq(c_mypid())!= 0)
        s.transDelay(dt);
    /* code for collision check and
    subsequent backoff algorithm */
    collision_handler (collis_counter);
    /* Termination phase */
    statistic_fun();
```

The Busnode Process

The **Busnode** process acts like a server process ready to accept transaction calls from the local **Simnode** processes, the **Lserver** processes or the **Rserver** processes. The **Busnode** is responsible for detecting collisions and it continuously monitors the carrier flag to see if it is busy. In the case of a collision, the **Busnode** process calls the **Scheduler** to awaken the transmitting **Simnode** process which then stops transmission and sends the jam signal. The following code gives the Concurrent C specification of the **Busnode** process.

```
Process spec Busnode (process sched s,
                  process Rserver idr,
                  process Lserver idl,
                  process Simnode name)

    /*transactions to change producer count */
    trans void addProd(), dropProd();
    /* transactions to change consumer count */
    trans void addCons(), dropCons();
    /* transaction to handle right to left traffic */
    trans put_FRTL(type); /* type can be start,
                      completion or jam */
    /* transaction to handle left to right traffic */
    trans put_FLTR(type);
    /* transaction to transmit local traffic */
    trans done(type);
    /* transaction to accept requests */
    trans trans_req(send_id); /* for Simnode */
    trans takereq(); /* for Rserver & Lserver */
```

The Rserver and Lserver Processes

These processes transmit the traffic delivered to the **Busnode** process by any transmitting Simnode process to the left and/or right. The specification and body of the **Rserver** process are given below.

```
Process spec Rserver(process sched s,
        process Busnode inbus,
        process Busnode outbus, Process Simnode
name)

Process body Rserver(s, inbus, outbus, name)
typedef struct /* data submitted by Simnode */
  { /* time of arrival */
  long arrive;
  /* Packet length */
  int packet_length;
  /* No. of update messages per sec */
  int update_num;
  /* No. of attempts to transmit */
  int attempt_index;
     } local_traffic

/* Initialization phase */
  c_setname(c_mypid(), name.str);
  s.adduser(); inbus.addCons();
  outbus.addProd();
/* Main processing phase */
  while (takereq(1)) {
  /* wait for propagation delay */
  t = arrivaltime + propagation delay;
  ts = s.wait(s.reqDelay(t));
  /* deliver message */
  put_FLTR(type);
  }
```

The Scheduler Process

Delays in the simulated network (such as transmission delays) are handled by the **Scheduler** process. This process maintains the simulated clock and advances it appropriately. For each delay request from a process, the Scheduler determines the time when the process needs to be reactivated and saves this time in an "activation request" list. When all processes are waiting, the scheduler picks the next process to run, advances the simulated clock and reactivates the process. The simulated clock advances only when all processes are waiting; thus any (non-delay) computation done by a process takes place in zero simulated time. At any given moment, each client process is in one of the following three states:

- Waiting: for an explicit delay request from the Scheduler, .
- Active: computing in zero simulated time;
- Passive: waiting for an event other than a delay request from the Scheduler.

The specification and the body of the **Scheduler** are given below.

```
process spec sched()
/* return current simulated time */
trans long now();
/* request a delay */
trans long reqDelay(long);
/* wait for a reqDelay */
trans long wait(long);
/* add or delete client process */
trans void adduser(), dropuser();
/*change client to new state */
trans void passive(), active()

/* handle collision */
trans void collision(id);
typedef struct {
/* structure describing a delay request */
long ts; /* time stamp */
int next; /* index of next entry or -1 */
/* number of clients waiting for this time */
int nwait;
} reqent;
static reqent Rtab MAXREQ ;
static int Ifree; /* first free entry */
/* Ihead is entry with lowest timestamp */
static int Ihead = -1;

process body sched()
  int nclients, nactive, i;
long curtive = 0;
/* initialization phase */
c_setname(c_mypid(),"sched");
rqInit();
accept adduser() nclients = nactive = 1
/* main processing phase: accept requests while
clients exist */
while (nclients >0)
{
select
accept adduser() nclients += 1; nactive += 1;
or
accept dropuser() nclients -= 1; nactive -= 1;
or
accept passive()  nactive -=1;
or
accept active()  nactive +=1
or
accept now()  treturn curtime;
or
accept reqDelay(x)
nactive -= 1; treturn (addreq(curtime+x));
or
accept jam(id)
change timestamp of record with this id
}

/* If all clients are waiting, find the first event */
/* and allow all clients waiting for it to proceed */
if (nactive == 0 && Ihead != -1)
curtime = RtabIhead.ts;
nactive = RtabIhead.nwait;
While (--RtabIhead.nwait >=0)
accept wait(key) such that (key == Ihead)
treturn curtime;
```

In addition to the above entities, several other auxiliary processes/routines are used to collect/print statistics and appropriate performance measures, perform consistency checks, print error messages, create and initialize all required processes, and start/terminate the concurrent simulation. The software system is written in a modular fashion with emphasis on ease-of-modification and the use of parameterized values that facilitate the testing of a wide range of network characteristics and the simulation of different load conditions and different network parameters.

PERFORMANCE CHARACTERISTICS OF MEDIUM-ACCESS PROTOCOLS

In the real-time networking of simulators, two performance aspects are of particular interest: the delay-throughput characteristic of the medium-access control schemes, and network system behavior under heavy traffic loads. These characteristics will be considered for the general classes of contention and non-contention (token passing) protocols.

Contention Protocols

Contention protocols such as ETHERNET perform well in environments with a large number of bursty (ratio of average to high traffic is small) users. For its reliable operation, however, the ETHERNET bus protocol requires that a transceiver must be capable of detecting the weakest other transmitter on the network during its own transmissions, and of distinguishing the signals from other transmitters from the echoes of its own transmitter. Because of this, the use of high-quality coaxial cable is required to cover longer distances and a limitation on the maximum distance which can be covered by a single segment network cable is imposed.

An advantage of the bus structure (where ETHERNET and GBRAM operate) over the ring structure is that users attached to the bus are passive units, while users connected to the ring are active units. An immediate consequence of this observation is that if a node on the ring breaks down it can bring the entire network system down. This is highly unlikely to happen in the bus configuration.

A disadvantage of contention protocols is there is no guarantee of packet delivery time due to the undeterministic nature of contention and collision/back-off.

Token Passing Protocols

An advantage of token passing protocols is that they are much less sensitive to increased transmission rates and smaller packet lengths compared to contention protocols . and they operate more efficiently with longer length cables than the contention protocols [5]. Furthermore, since token passing protocols are conflict free, a maximum packet delivery can be guaranteed for a given number of users, making them desirable protocols for real-time applications.

Token-Ring LAN's [5] offer other advantages including the following:

- Because of its point-to-point connection property, rings readily accommodate the use of optical fiber as a transmission medium. In addition to offering reduced size and weight, and enhanced safety features, optical fiber also offers very high signal bandwidth (100 Mbps for fiber token-rings).

- Token-rings easily provide a priority-based scheme for packet transmission across the network. This is because the token has bits indicating the priority assigned to it, thereby providing multiple levels of access to the ring. In simulator networking this means that it will be possible to assign priorities to the different types of messages in order to optimize real-time performance and visual display at peak load conditions.

- The technological advantages enjoyed by bus topologies to date are about to disappear. Inevitably, VLSI technology and other near-term advances will soon be supplying the industry with ring chips and off-the-shelf ring attachments at the same low cost as bus chips. This low cost, combined with their reliability and ease of configuration and implementation, will make token-ring LAN's a very promising tool for simulator networking.

CONCLUSIONS

In this paper we have described an ongoing effort to model and evaluate the performance of three different network protocol access methods suitable for networking of simulation training devices: a contention access method based on the CSMA/CD (ETHERNET) protocol, and two contention-free methods based on Virtual Token Bus Access such as GBRAM and Token-Ring Access protocols. The system models pertaining to the above three access methods were addressed and a high-level description of a detailed simulation software system implemented for evaluating the performance of an ETHERNET scheme was given.

The models developed for the three access methods will enable us to perform a comparison study and evaluate different design decisions. Some of the numerical performance measures that will be gathered by the models are:

- The overall throughput of the network.
- The utilization of the transmission medium.
- The collision ratio for contention access.
- The average delay time per packet.
- The average ratio of lost packets (data loss rate).
- The relationship of the number of nodes on the network and the above parameters.
- The effect of packet arrival rates on network performance.

The models developed under this effort offer a very flexible tool for the evaluation and analysis of important classes of networking schemes that can be used to interconnect large numbers of real-time simulation training devices. Further investigations will be carried out to perform a comparison study of the three access methods and to evaluate different design decisions aimed at improving the overall throughput and enhancing the capability of simulation networks.

ACKNOWLEDGEMENTS

This work is supported by the U.S. Army Program Manager for Training Devices (PM TRADE) under Broad Agency Announcement # 88-01. The opinions expressed herein are those of the authors' and not necessarily those of the U.S. Government. The authors would like to thank J. Cadiz and E. Stadler for their help in obtaining and analyzing network traffic data and preparing this report.

REFERENCES

[1] ANSI/IEEE - International Standard 8802/3 "Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specification", IEEE Computer Society Press, 1985.

[2] Metcalfe, R. M. and Boggs, D. R., "Ethernet Distributed Packet Switching for Local Computer Networks", Communication Ass. Comput. Mach., Vol. 19, no. 7, pp. 395-403, 1976.

[3] Liu, T. T., Li, L. and Franta, W. R. "A Decentralized Conflict-Free Protocol, GBRAM for Large Scale Local Networks", Computer Network Symposium Proceedings, pp. 39-54, Dec. 1981.

[4] ANSI/IEEE - International Standard 8802/5 "Token Ring Access", IEEE Computer Society Press, 1985.

[5] Dixon, R., Strole, N. and Markov, J. "A token ring network for local data communication", IBM System Journal, Vol. 22, 1983, pp.62-74.

[6] Gehani, N. and Roome, W. "Concurrent C" Technical Report, AT&T Bell Laboratories, 1986.

ABOUT THE AUTHORS

M. A. Bassiouni received his Ph.D. degree in Computer Science from Pennsylvania State University in 1982. He is currently an Associate Professor of Computer Science at the University of Central Florida, Orlando. His current research interests include computer networks, distributed systems, databases, and performance evaluations. He has authored several papers and has been actively involved in research on local area networks, concurrency control, data encoding, I/O measurements and modeling, schemes of file allocation and user interfaces to relational database systems. Dr. Bassiouni is a member of the IEEE Computer Society, the Association for Computing Machinery, and the American Society for Information Science.

M. Georgiopoulos received his Ph.D degree in Electrical Engineering from the University of Connecticut in 1986. He is currently an Assistant Professor in the Department of Electrical Engineering and Communication Sciences at the University of Central Florida, Orlando. His current research interests include multi-user communication theory, communication networks, computer networks, and spread spectrum communications. Dr. Georgiopoulos is a member of IEEE and the Technical Chamber of Greece.

J. Thompson received his BS degree in Electrical Engineering from the University of Central Florida in 1978. He is currently a Research Associate for the Institute for Simulation and Training (IST) at the University of Central Florida, Orlando. Mr. Thompson has technical responsibility for all IST research activities involving computer networking.

Appendix D

Copy of Paper Accepted for Publication in Proceedings of the 11th I/ITSC Conference November, 1989

Real Time Simulation Networking: Network Modeling and Protocol Alternatives

M. Bassiouni, Department of Computer Science
M. Georgiopoulos, Department of Electrical Engineering
J. Thompson, Institute for Simulation and Training

Graduate Student Assistants: S. Chaterjee, M. Chiu and N. Christou

University of Central Florida
Orlando, FL 32816

ABSTRACT

In this paper, we present the findings of a comparison study using predictive detailed simulation models for three different network protocol access methods: Carrier Sense Multiple Access with Collision Detection (ANSI/IEEE 802.3 STD), Token-Passing Bus Access (ANSI/IEEE 802.4 STD) and Token-Ring Access (ANSI/IEEE 802.5 STD). Discussions of network performance, the implications of the results of the comparison study, and the insight gained from this project for improving real-time simulation networking are presented.

INTRODUCTION

A local area network (LAN) is a geographically confined communication system that uses a shared transmission medium. Various choices usually exist for the main ingredients of LAN (i.e., transmission medium, topology, access protocols, etc.), with each exhibiting advantages and providing benefits that depend on the objectives of the LAN. The ability to model, analyze and evaluate the impact of these choices on network performance is essential to ensuring maximum utilization of the LAN.

One of the pioneering LAN's for connecting computers was a bus-based ETHERNET developed by Xerox Corp. in the early 1970's. The contention access method used by each node in ETHERNET is based on a pre-emptive protocol of first listening for network activity and then broadcasting the message onto the network. If a collision with another message occurs, each sender (node) backs-off from transmitting their message for a random period of time and then attempts the transmission again. This access technique is known as Carrier Sense Multiple Access with Collision Detection (CSMA/CD) [1, 2]. Standards for CSMA/CD protocols such as ETHERNET are known as IEEE 802.3 standards approved by the American National Standards Institute (ANSI).

The networking of real-time, interactive simulation training systems departs from the traditional use of a computer network whose function is to provide sharing of computing resources among multiple users (nodes) on the network. When used to interconnect real-time training simulators, the network is used almost exclusively for communication of process state information between the simulators engaged in the training exercise.

There are many inherent limitations to using a network in this application. For example, as the number of simulators on the

network and the workload per simulator increases, there may be a deterioration in throughput and a degradation of other network performance parameters. If throughput delays become too large, for example, the effectiveness of a real-time training simulation may be overly compromised due to the time-critical response requirements in the simulation of true-to-life, action-requiring training scenarios. Depending upon the network communication protocol being used, there may also be an increase in the frequency of retransmitted and lost or distorted messages.

Recently, there has been a tremendous interest in LAN's implemented using the non-contention class of network protocols known as Token-Passing protocols. Two schemes falling under this class are Token-Ring and Virtual Token-Bus protocols . In a Token Ring LAN, a distinctive bit sequence, called a token, is passed from one node to another in order to signify the availability of the network medium for the transmission of data for that node. Possession of the token by the node gives it,and only it, permission to transmit across the network, as opposed to having all nodes contend for this privilege. In a Virtual Token-Bus LAN, a virtual, or imaginary token, is passed from user to user thus providing access to the network. This virtual token is actually a predetermined instant in time when each user knows it is its turn to access the network. Each of these three protocols will be discussed in detail in later sections.

The primary goal of this research effort has been to develop predictive and analytical models for network performance of three LAN configurations operating under real-time, interactive simulation and training constraints. Two of these LAN's are **bus networks** which utilize baseband transmission to send messages over a coaxial cable which is common to all users. The medium-access control schemes for the first is ETHERNET which is a member of the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) protocol family and for the second is the Generalized Broadcast Recognizing Access Method (GBRAM) [3] which is a member of the Virtual Token-Bus protocol family. The third LAN is a **ring network**, which sends its messages over either coaxial or fiber optic cable. Its medium-access control scheme is the Token-Ring [4] protocol.

NETWORK MEDIUM-ACCESS PROTOCOLS

ETHERNET

CSMA/CD protocols, including ETHERNET, are characterized by their distributed network control whereby each node on the network determines its own channel access time based only on information available from the common network channel (bus). When a node is ready to transmit a message onto the network, it first monitors the network bus to determine whether any other transmissions from other nodes are in progress. If the node senses the network channel to be busy, it simply waits for the channel to become idle before attempting to transmit its message. Once the channel is sensed to be idle, the node waits a pre-specified amount of time to assure the channel is clear and then begins transmitting its message. During its own transmission, the node also monitors the channel in order to detect whether its message is interfering (colliding) with messages from other nodes. If a collision is detected, each node involved in the collision transmits a bit sequence onto the network known as a **jam signal**, after which each node involved in the collision waits (backs-off) for a randomly generated amount of time before reattempting its transmission.

The performance of contention protocols is directly related to how efficiently nodes avoid collisions and handle retransmissions.

The problem of data collisions is directly related to the network traffic load.

Token-Ring

In Token Passing protocols [4, 5], which the Token-Ring LAN is a member, there is no contention for the network channel because only one node at a time is allowed to access the channel. In Token-Ring LAN's this is accomplished by arranging the nodes in a serial ring configuration such that the network channel actually passes through each node. The **token** is a control signal that circulates around the channel. An individual node gains the right to transmit onto the network when it first detects, and then captures a **free token** passing on the channel. Once a node captures the **free token**, it changes it to a **busy token** and begins transmitting its message onto the network. Upon completing the transmission of its message, the node generates and transmits a **free token** which begins circulating around the network channel, thus providing other nodes the opportunity to gain access to the network.

Generalized Broadcast Recognizing Access Method (GBRAM)

The GBRAM protocol is also a member of the Token Passing protocol family. It differs significantly,however, from the Token-Ring protocol. In the GBRAM, rather than each node having to capture the **free token** from the network to gain transmission access, an imaginary **(virtual) token** is passed from node to node achieving the same result. The **virtual token** scheme provides each node access to the network at a unique time instant which is determined by a decentralized scheduling function.

NETWORK SYSTEM CONFIGURATION MODELS

Bus Network

The bus ne'work configuration (applicable to both ETHERNET & GBRAM) is shown in Figure 1. In this implementation, up to eight nodes can be connected through a multi-port transceiver to a single point on the coaxial cable, via a media-access unit. A single coaxial cable links all nodes together.

Ring Network

The ring network configuration (Token-Ring) is shown in Figure 2. The ring network consists of a closed sequence of individual point-to-point (node-to-node) connections.

NETWORK PROTOCOL COMPUTER SIMULATIONS

Simulation Models

The simulation models for both the bus and ring LAN topologies are written in **Concurrent C** (an extension of the C language with concurrent programming facilities based on the "rendezvous" concept). The powerful synchronization and concurrency aspects of **Concurrent C** [6] have provided us with a notationally convenient and conceptually elegant tool for modeling the parallel activities of LAN nodes and the underlying networking layer.

A functional diagram of the simulation model for the bus topology is shown in Figure 3. The following process types are the major generic entities used in our simulation for the bus structure.

Process **Simnode** is used to represent a vehicle simulator on the network. A process of this type is created for each such simulator. This process is the source of local traffic and is capable of generating packets according to a specified input method (e.g., using traces of real data or random stochastically generated inter-arrival times such as exponential, uniform, fixed with jitters, etc.)

Process **Busnode** is used to represent the point of contact of each network node with the bus (coaxial cable). A process of type **Busnode** is created for each such

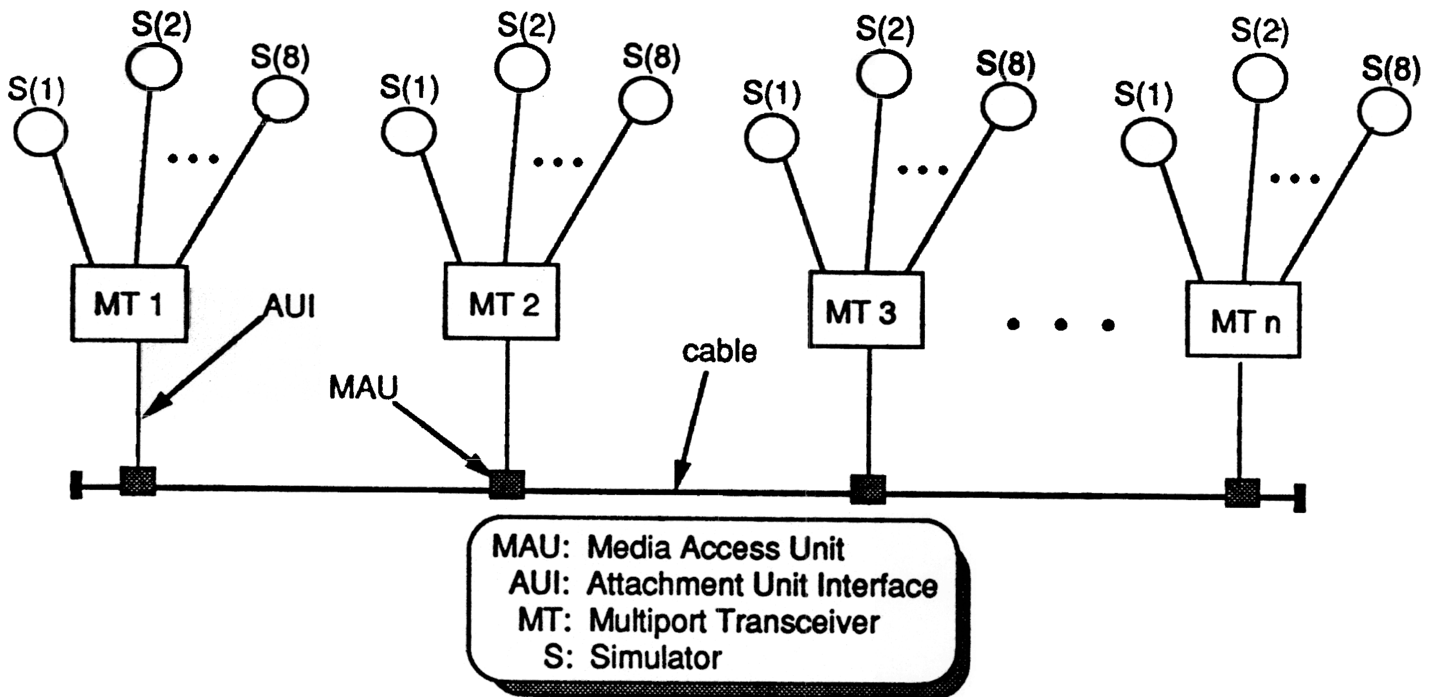


Figure 1. Bus Network Topology System Configuration

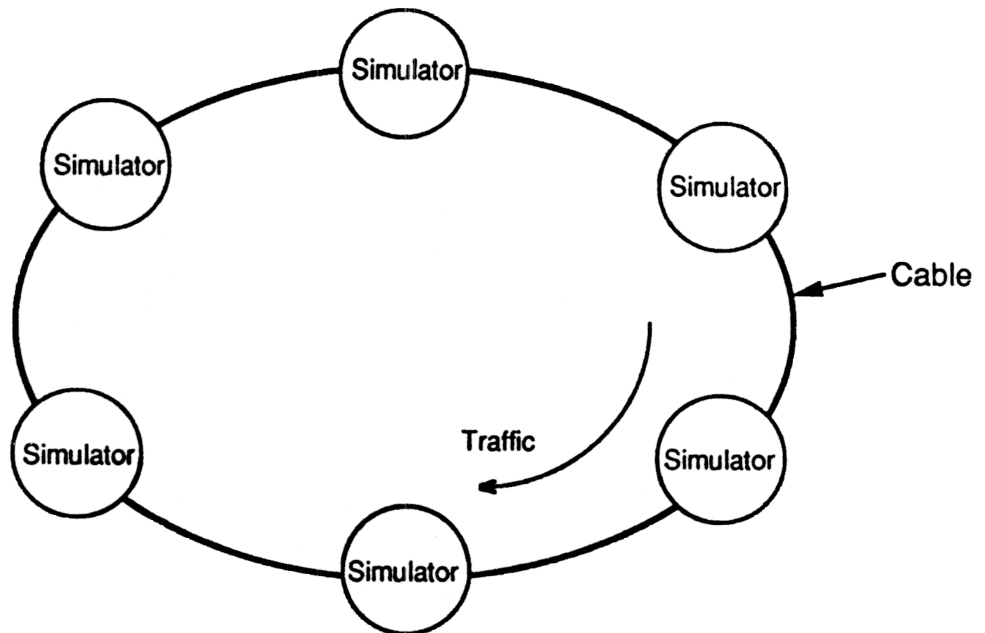


Figure 2. Ring Network Topology System Configuration

point of contact on the bus. Upon receiving a transmission request from a **Simnode** process, the **Busnode** attempts to fulfill the request based on the medium access protocol of the LAN. For example, in the CSMA/CD case, the **Busnode** process checks for a carrier flag and will allow transmission only if the flag has been off for at least the interframe gap. If a collision is detected during transmission, the **Simnode** process sends a jam signal and increments the collision counter. This is followed by invoking a back-off algorithm for retransmission.

Processes **Lserver** and **Rserver** are used to simulate the propagation delay and control the flow of data packets and jam signals in the direction form right to left and from left to right, respectively, for each network node. A pair of these processes is created for each network node.

Process **Scheduler** is used to order time events and control the sequencing of activities of the entire simulation.

Figure 4 shows a block diagram showing the simulation model for the ring topology. Process **Simnode** is responsible for generating the load (data packets) on the ring. Process **Ringnode** monitors the ring traffic and implements the token-based medium access protocol. Process **Server** is used to simulate the propagational delay between each pair of LAN nodes. As in the bus model, the simulation of the ring structure uses a **Scheduler** process to control the sequencing of activities.

DISCUSSION

Token Ring vs Contention Access

A token passing ring is a LAN with a loop topology in which a token is passed around the network in a round-robin fashion from one node to the next. Contention for transmission is resolved by stipulating that only the node currently in possession of the token is allowed to transmit a frame or a sequence of frames onto the ring. When the transmission is finished, the token is passed to the node downstream which then gets a chance to transmit. Since there is a single token on the ring, only one node can be transmitting at a time. Other (non-transmitting) nodes, however, continuously receive the bit stream, examine it and repeat it (i.e., place it on the medium to the next station). A station repeating the bit stream may copy it into local buffers or modify some control bits as appropriate.

In general, Token-Ring LAN's are much less sensitive to increased transmission rates and smaller packet sizes compared to contention protocols (e.g., ETHERNET with CSMA/CD). Since token-rings are collision free, a maximum packet delay can be guaranteed for a given number of stations. Thus, the real-time requirements of applications having high traffic loads (e.g., networks with large number of simulation training devices) can be handled more gracefully by using a contention-free ring scheme.

Because of its point-to-point connection property, rings readily accommodate the use of optical fiber as a transmission medium. In addition to offering reduced size, weight and enhanced safety features, optical fiber also offers very high signal bandwidth. One very promising implementation of ring networks using optical fiber is the Fiber Distributed Data Interface (FDDI). FDDI is a 100 Mbits/sec token-ring LAN protocol that is rapidly becoming accepted as the premier high speed LAN standard [7]. With its embedded extensibility to support even higher speeds (500 to 1,000 Mbits/sec), FDDI is poised to become the dominant high-end LAN of the 1990's. The paradigm for FDDI topology is known as a "dual counter-rotating ring of trees". The physical layer topology consists of independent, full-duplex, point-to-point physical connections, while the logical layer consists of one or two

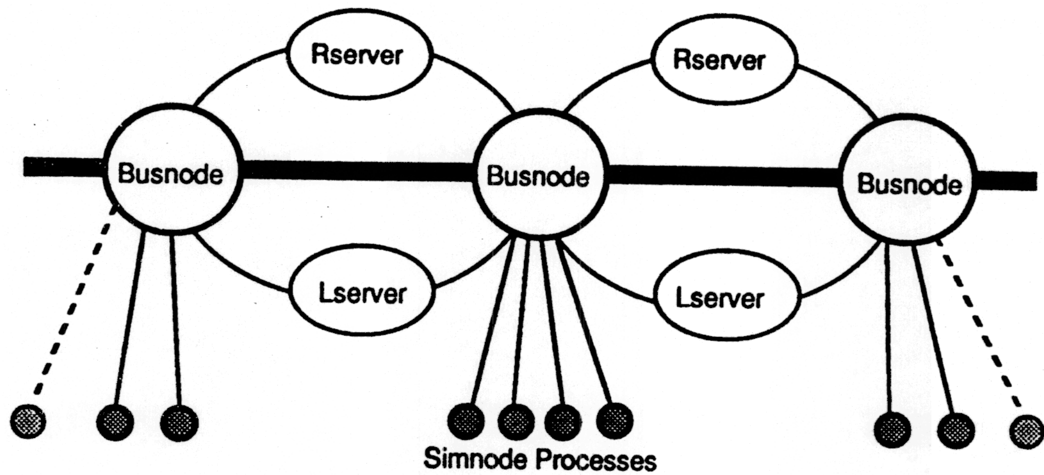


Figure 3. Simulation Model for Bus Topology Networks

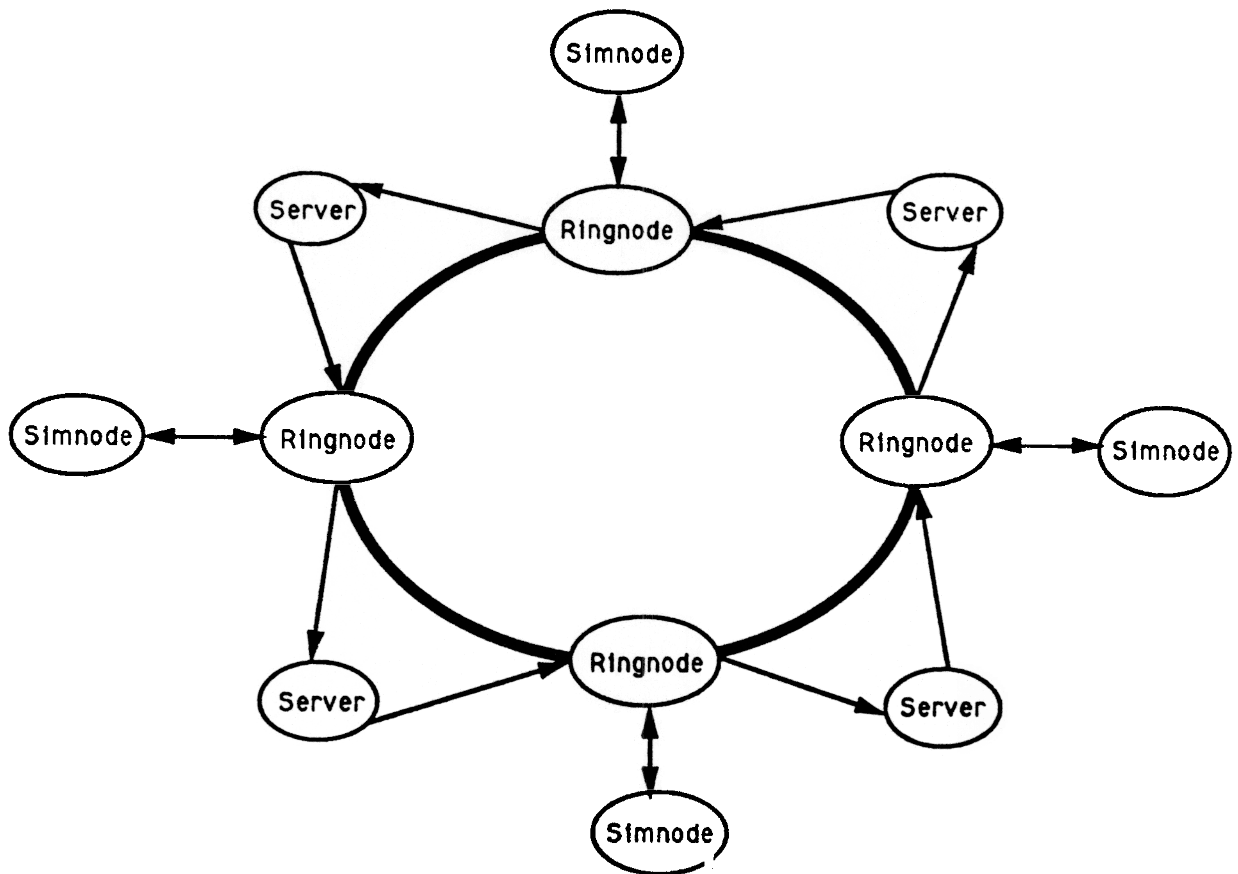


Figure 4. Simulation Model for Ring Topology Networks

rings. The FDDI Medium Access Control (MAC) protocol provides data services similar to those of the IEEE 802.5 token-rings. An extension to FDDI (known as FDDI II) is currently being investigated to add isochronous data transmission capabilities to the network, thus enabling it to handle both voice and data. FDDI technology will eventually provide the simulation and training industry with powerful real-time LAN's capable of interconnecting an unprecedented number of stations.

Another promising feature of Token-Rings is that they provide a priority-based scheme for packet transmission across the network. In the ANSI/IEEE 802.5 ring implementation, the passing token has three bits indicating the current priority level of the ring (this gives a total of 8 priority levels). A station that captures the token can only transmit packets whose priority is equal to or higher than the priority of the passing token. The ANSI/IEEE 802.5 protocol also provides mechanisms that enable stations to request and change the priority of the passing token. In simulation networks, this means that it will be possible to assign levels of priority to different types of messages which may be beneficial in attempting to optimize real-time system performance, especially under peak load conditions.

On the other hand, token rings are outperformed by bus topology LAN's in certain areas. One main advantage of the bus structure over ring LAN's is the reliability of network operation following a node failure. In general, bus-based LAN's are more resistant to network crashes due to node failures since the propagation of messages on the bus does not require the participation of any given node. Failure of a station on the ring structure, however, can bring the entire LAN down. This problem has been considerably reduced by the increased reliability of today's ring chips and off-the-shelf ring attachments. Furthermore, new fiber optic ring devices use optical bypass switches in order to allow inactive (off-line) stations to pass the traveling data-carrying light waves directly from one neighboring node to the next without active power and with little or no degradation of the optical signal.

Bus-based ETHERNET LAN's have enjoyed economic advantages because of their widespread use in the past two decades. These advantages, however, are about to disappear since VLSI technology, fiber optics, and other near-term advances will soon be supplying the market with ring chips and devices at the same low cost as comparable bus products. Also, hardware support for FDDI is rapidly growing and the projected increase in development and installation investments in FDDI are expected to drive down the cost of FDDI hardware considerably.

GBRAM vs Contention Access

The GBRAM LAN protocol implementation shares the same bus topology as the ETHERNET implementation (see Fig. 1). The nodes connected onto the network via the same multi-port transceiver belong to the same group and each node within the group has a unique identity. This node identity scheme plays an important part in the assignment of channel access time slots for each node. Every node on the network perceives the channel state under the GBRAM as consisting of cycles of **scheduling** and **transmission** periods. Roughly speaking, the end of a transmission period designates the beginning of a scheduling period and the end of a scheduling period signals the beginning of the next scheduling period. During a scheduling period, every node gains the right to access the network channel starting with the node whose identity sequentially follows the node which transmitted last.

GBRAM avoids collisions by scheduling different users at unique time instances. The time interval between two successive

scheduling instances depends on the physical location of the users who are allowed access to the network channel at these instances. In fact, this time interval is equal to the propagation delay between the two users who are scheduled to transmit at these two unique instances. In the GBRAM, therefore, the physical location of each user on the network is extremely important in calculating the network's scheduling algorithm.

It has been observed that in large scale simulation networks not all users are active at all times. Consider, for example, a vehicle simulator which is active at the beginning of a battle, but is destroyed by enemy fire during the simulation. These inactive users must be taken out of the token passing sequence list in order to reduce the number of wasted idle slots which will be scheduled for the network. Hence, there must be a procedure to sign-off users from the network. This procedure might be implemented as follows: an active user signs off by broadcasting, at its scheduled transmission time instant, a sign-off packet which would be read by all other active users who would, in turn, update their scheduling sequence accordingly.

The successful operation of the version of the GBRAM presented in this paper depend on the fact that all the users know a common time epoch. This common time epoch corresponds to the beginning of a scheduling period. In our version of GBRAM, the beginning of a scheduling period corresponds to either the end of a transmission period (as perceived by the transmitting user) plus the propagation delay along the cable or a complete scheduling cycle after the beginning of the previous scheduling period. It is obvious that the common time epoches can be determined by any user who observes the channel state at all times and knows its propagation delay from any other user in the network. Note that contention protocols require only that users observe the state of the channel at all times. There are other versions of GBRAM [3], however, that do not require the users to have a complete knowledge of the network topology. These versions of GBRAM will not perform as well as the GBRAM version considered in this paper.

NETWORK SIMULATION RESULTS

ETHERNET Simulated Performance

The network protocol simulation models described earlier have been used to gather information about the performance of local area networks used for real-time training, under various conditions of node (simulator) placement on the network, traffic load levels and packet scheduling policies. One configuration considered in our analysis is unique to the application of local area networks to the interconnection of simulation and training devices. In this configuration, an optimization is made to reduce the load on the network. An explanation of this optimization for the ETHERNET case is given below.

Upon a state change, a simulator (node) on the network sends the information concerning its new state to other nodes on the LAN. Each new state results in the generation of a new data packet at the **application layer** (i.e.,.at the node level). The packet is then submitted to the **data link layer** in order to start the process of its transmission. In ETHERNET, only one packet per node is delivered for transmission at a time. Other packets are normally queued up at the application level waiting for the end of the ongoing transmission attempt. In this context, the arrival of a new packet (carrying the most current state of the node) simply replaces the previous packet (stored at the application layer) which represents a now outdated state condition. The discarding of the outdated packet helps speed up the transmission of the most current state of the node. Notice that the

packet already submitted to the ETHERNET data link layer is under the control of ETHERNET protocol (board) which is not accessible from the application layer and, therefore, is not affected by new packet arrivals.

The performance of this specific ETHERNET configuration is given in Graphs 1 & 2, and Table 1. Graph 1 gives the relationship between the throughput of the LAN and the total initial traffic load from all simulators (i.e., before any discarding at the application level). Graph 2 gives the relationship between the total initial load and the packets discarded by ETHERNET as a result of exceeding the maximum count for transmission attempts (16) due to excessive collisions. Notice that Graph 2 gives the packets discarded by the ETHERNET protocol and not the obsolete packets discarded at the application level. Statistics about the average transmission delay and average and maximum number of transmission attempts are given in Table 1.

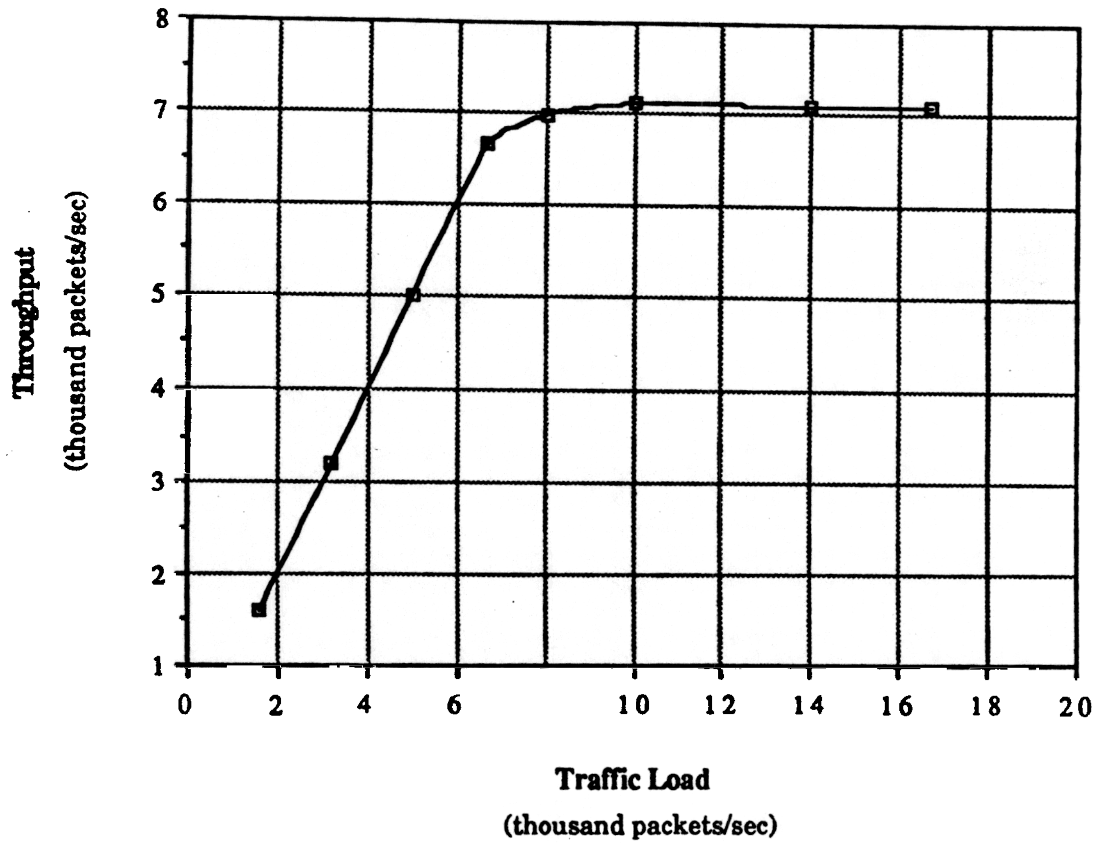
At low traffic load levels, the effect of collisions is small and no packets are discarded due to excessive collision counts. All packets submitted to the data link layer at such low loads eventually get transmitted successfully and the throughput of the network is equal to the total traffic load minus the obsolete packets discarded at the application level. As the traffic load increases, more collisions occur and the average number of transmission attempts per packet (and consequently the average packet delay) increases (see table 1). Since a packet is thrown away (by ETHERNET) if its transmission fails 16 consecutive times, the growing collision rate eventually results in the loss of some packets. At some point, the network becomes overwhelmed by the collision overhead and less LAN bandwidth becomes available for actual packet transmission. This is the reason that the throughput of this ETHERNET LAN starts to reach a saturation level even though the traffic load continues to increase. Since the actual throughput of this configuration is dependent on both the number of obsolete packets discarded by the application layer and the colliding packets discarded by ETHERNET boards, the performance of the network may show some slight perturbations in actual throughput; but otherwise will stay in the saturation throughput level it has attained.

Token Ring Simulated Performance

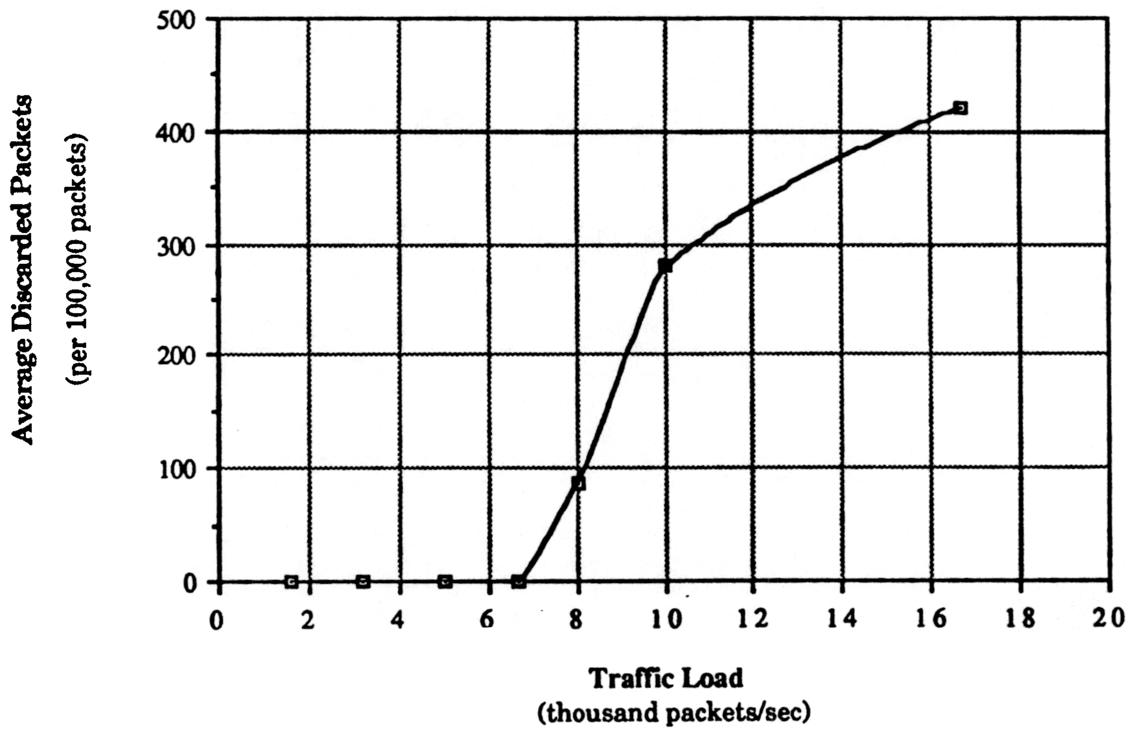
Token ring LAN's exhibit quite different behavior as compared to the ETHERNET. In some versions of token ring protocols, a transmitting station recreates the free token and puts it onto the ring as soon as it finishes packet transmission. In IEEE 802.5 rings, however, a transmitting station checks to see if its address (affixed at the header of the transmitted packet) has returned to it (indicating a complete cycle around the ring). Only after receiving this address is the station allowed to transmit the free token onto the ring, thus giving other stations an opportunity to transmit. This latter protocol is more conservative (from a reliability point of view) but imposes extra time overhead for token management. At low network traffic loads, the IEEE 802.5 token ring protocol causes more transmission delays for packets than the CSMA/CD counterpart. Unlike collision handling, the overhead of token management is largely independent of the LAN load. Therefore, the throughput of token-ring LAN's continues to increase as the traffic load increases. No degradation in performance at high loads is exhibited by token rings in contrast to the ETHERNET LAN.

GBRAM Simulated Performance

Preliminary GBRAM simulation results indicate that GBRAM should perform well for medium to high input traffic loads, but may be inferior to contention protocols for light to medium input traffic loads. Let us try to justify this observation based on the



Graph 1. Throughput vs Traffic Load (ETHERNET)



Graph 2. vs Load (ETHERNET)

Traffic Load (thousand packets/sec)	Average Transmission Delay (millisec)	Average Number of Transmission Attempts/Packet	Maximum Number of Attempts
1.60	.111	1.01	2
3.20	.152	1.16	5
5.00	.232	1.38	8
6.67	1.029	2.25	14
8.00	2.682	2.88	16
10.00	4.183	3.15	16
16.67	5.571	3.46	16

Table 1. ETHERNET Performance Statistics

description of the GBRAM protocol presented in the previous section.

Consider the case where there is only one out of a total of 100 users that generates traffic onto the network and that, for the majority of its transmissions, this user has one packet in its buffer every time a scheduling period of the GBRAM protocol starts. Let us assume that the propagation delay between two users in the same group is 30 bits, the propagation delay along the cable is 20 bits, and the packet length is 1000 bits. This is a case of light input traffic. It is easy to see that GBRAM induces an average and maximum packet delay of (30x100+20)/2 + 1,000 = 2,510 bits and (30x100+20) + 1,000 = 4,020 bits, respectively. Every contention protocol under the aforementioned light input traffic conditions, induces an average and a maximum packet delay of 1000 bits (the packet length).The performance difference widens as the number of the users in the network increases.

Suppose now that all 100 users in the network are active. Each one of them has exactly one packet to transmit at the beginning of a GBRAM scheduling period. This corresponds to a case of high input traffic load. Now GBRAM induces an average and a maximum packet delay of approximately (1000x100)/2 + 1000 = 51,000 bits and (1000x100) + 1000 = 101,000 bits, respectively. The length of the packet was once again taken to be equal to 1000 bits. The aforementioned input traffic load is approximately equal to 100% of ETHERNET capacity. Contention protocols attain a throughput smaller than 100% even under ideal conditions (i.e. small end-to-end propagation delay/packet length ratio). As a result, contention protocols are unstable (experience unbounded packet delays) for the above high input traffic scenario.

The above discussion, although simplified, verifies our point that there will be a region of input traffic loads (light to medium) where contention protocols outperform GBRAM and a region of input traffic loads (medium to high) where GBRAM outperforms contention protocols The cutoff point depends on the total number of users in the network and increases as the number of users in the network increases. The exact cutoff point will be determined by simulation.

CONCLUSIONS

In this paper we have described an ongoing effort to model and evaluate the performance of three different network protocol access methods suitable for networking of simulation training devices: a contention access method based on the CSMA/CD (ETHERNET) protocol, and two contention-free methods based on Virtual Token Bus Access such as GBRAM and Token-Ring Access protocols. The system models pertaining to the above three access methods were addressed and a high-level description of a detailed simulation software system implemented for evaluating the performance of these protocols was given.

The models developed for the three access methods will enable us to perform a comparison study and evaluate different design decisions. Some of the numerical performance measures that are being gathered by the models are:

- The impact of traffic loading on network throughput
- The utilization of the transmission medium
- The distribution of delay times of transmitted packets.

The models developed under this effort offer a very flexible tool for the evaluation and analysis of important classes of networking schemes that can be used to

interconnect large numbers of real-time simulation training devices. Further research is being conducted which is focusing on implementing these two alternate protocols in a hardware and software test bed with the ultimate goal of enhancing the capability of simulation networks.

ACKNOWLEDGEMENTS

This work is supported by the U.S. Army Program Manager Training Devices (PM TRADE) and the Defense Advanced Research Projects Agency (DARPA) under Broad Agency Announcement # 88-01. The authors would like to thank J. Cadiz and E. Stadler for their help in obtaining and analyzing network traffic data and preparing this report.

REFERENCES

[1] ANSI/IEEE - International Standard 8802/3 "Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specification", IEEE Computer Society Press, 1985.

[2] Metcalfe, R. M. and Boggs, D. R., "Ethernet Distributed Packet Switching for Local Computer Networks", Communication Ass. Comput. Mach., Vol. 19, no. 7, pp. 395-403, 1976.

[3] Liu, T. T., Li, L. and Franta, W. R. "A Decentralized Conflict-Free Protocol, GBRAM for Large Scale Local Networks", Computer Network Symposium Proceedings, pp. 39-54, Dec. 1981.

[4] ANSI/IEEE - International Standard 8802/5 "Token Ring Access", IEEE Computer Society Press, 1985.

[5] Dixon, R., Strole, N. and Markov, J. "A token ring network for local data communication", IBM System Journal, Vol. 22, 1983, pp.62-74.

[6] Gehani, N. and Roome, W. "Concurrent C" Technical Report, AT&T Bell Laboratories, 1986.

[7] Marrin, K. "Emerging standards, hardware and software light the way to FDDI", Computer Design, Vol 28, No. 7, April 1989, pp. 51-57.

ABOUT THE AUTHORS

M. A. Bassiouni received his Ph.D. degree in Computer Science from Pennsylvania State University in 1982. He is currently an Associate Professor of Computer Science at the University of Central Florida, Orlando. His current research interests include computer networks, distributed systems, databases, and performance evaluations. He has authored several papers and has been actively involved in research on local area networks, concurrency control, data encoding, I/O measurements and modeling, schemes of file allocation and user interfaces to relational database systems. Dr. Bassiouni is a member of the IEEE Computer Society, the Association for Computing Machinery, and the American Society for Information Science.

M. Georgiopoulos received his Ph.D degree in Electrical Engineering from the University of Connecticut in 1986. He is currently an Assistant Professor in the Department of Electrical Engineering and Communication Sciences at the University of Central Florida, Orlando. His current research interests include multi-user communication theory, communication networks, computer networks, and spread spectrum communications. Dr. Georgiopoulos is a member of IEEE and the Technical Chamber of Greece.

J. Thompson received his BS degree in Electrical Engineering from the University of Central Florida in 1978. He is currently a Research Associate for the Institute for Simulation and Training (IST) at the University of Central Florida, Orlando. Mr. Thompson has technical responsibility for all IST research activities involving computer and simulator networking.