A Software Implementation of a Virtual Network Simulator

by

Walter F. Domka

Submitted to
Oregon State University

in partial fulfillment of
the requirements for the
degree of

MASTER OF SCIENCE

June 1982

APPROVED:

Michael J. Freiling

Michael J. Freiling, Assistant Professor
Department of Computer Science
In Charge of Major

Date presented: August 1981

Abstract

A software system to study network algorithms was implemented on UNIX. Each part of a network algorithm is written as a single C program which becomes a virtual node in the network. During a simulation, all virtualized nodes run as separate processes on a single PDP 11/40. Inter-node communication is carried out with procedures local to each node, to send and receive inter-node messages to and from a message queuing process. Communication between nodes is effected by use of virtual links which are specified in the simulated network's topology. The links are implemented on inter-process pipes between the message queuing process and the node processes.

Table of Contents

I.	Background	1
II.	Overview	2
	A. Specifications	3
	B. Utilities	6
	C. Compilation	6
	D. Simulation Driver	7
	E. Message Queuing Process	9
	F. Checkup	19
	G. Report	20
III.	Example	22
IV.	Extensions	26
V.	Conclusions	30
VI.	References	32
	Appendices	
A.	Loop Node Program	A-1
B.	Control Node Program	B-1
C.	Specifications Example	C-1
D.	Compilation and Simulation Example	D-1
E.	Report Program Output	E-1
F.	User Manual	F-1

I. Background

Simulation tools which have been developed to study computer networks fall into two broad categories, software-based simulators and testbeds.

The use of software-based simulation tools is generally to obtain quantified measurement of network performance for response times, throughput rates and other queuing theory related analyses. These simulators are developed using general purpose programming languages and simulation languages such as GASP and SIMULA. Many of these simulation tools are somewhat limited in that they are designed to study a single aspect of computer networks. Examples of this type of simulation tool are the VANS system [1], which models value-added communication networks, and the Scientific Control Systems database control system [2] which models database performance predicated on the number of processors in a database machine.

The second broad category of simulation tools is the testbed. A testbed is a collection of processors and inter-processor communication paths designed for easy reconfiguration of the communication paths contained within the testbed. A control processor supervises the collection of information from processors involved in a simulation. Each processor in the testbed performs its assigned tasks within the network being simulated. Processors are halted and restarted by the control processor to allow for data collection. An example of a testbed is the CHIMPNET system which is described in [3].

The simulator described in this paper is a program which uses multiple, parallel processes in a single processor machine to simulate a network. Supporting programs which perform pre-simulation and post-simulation processing are also used. The simulation tool was implemented in the C programming language on a PDP 11/40 computer running the UNIX operating system.

II. Overview

A network consists of nodes and links. Nodes may be viewed on a macro level as separate computer installations, or on a micro level as tasks within a distributed system. A user may simulate a network at any virtual level. The user creates a node by writing a program in the C programming language. Each node program, called node code, is compiled and then runs as a separate process during a simulation.

Nodes in a network interact by communication of mutually agreed upon messages via the links which connect nodes. Message formats and contents are determined solely by the user. Links are bidirectional, virtual communication channels which are devoid of operating properties, other than maximum message size. The flexible nature of the links allows modeling of messages at any level from the physical level to levels governed by higher protocols such as the X.25 protocol.

The simulation tool consists of seven components which are described in the next sections.

Specifications:

The specifications program is used to collect the specifications for a network to be simulated. A user enters responses to queries from the program to select specifications for the network. These specifications are then saved for use by other components of the simulator and for reuse or alteration during subsequent simulations.

Certain specifications are required: number of nodes, network topology, node code file names, number of inter-node messages to be sent and number of data files in the network. Appendix C contains a demonstration of a typical terminal session in which specifications are entered.

The number of nodes allowed in a network is limited only by the number of processes that may be present in the UNIX operating system. A few processes are needed to maintain the operating system, ie., process scheduling, but all others may be used for node processes. A tunable parameter is used to set the number of nodes allowed in the simulator's current configuration. Forty five nodes are allowed in the present simulator.

Nodes in a network are identified by numbering them in the increasing sequence 0, 1, 2, ..., n-1 where

n is the total number of nodes. Assignment of numbers to nodes is arbitrary but after an assignment is made it may not be changed during preparation for a simulation. The node numbering is binding due to the use of the numbers by the simulator during a simulation.

A network's topology is set during entry of the specifications for use during a simulation. Topologies are described in terms of the nodes which are connected by static links. Since each link is bidirectional, a link from node x to node y also implies that node y may transmit messages to node x. Once a topology has been selected, it is saved as an adjacency matrix which becomes part of the network specifications. The adjacency matrix is used by the message queuing process to determine if transmission of an inter-node message is valid. Validity is determined by existence of a link from the transmitting node to the receiving node.

Standard topologies may be used during entry of the network specifications. These include ring, star and complete topologies. More general topologies may be established by specifying individual links or by altering one of the standard topologies. Altering a standard topology is accomplished by adding or deleting links.

The node code file names are used to compile the C programs which are loaded into node processes.

The number of inter-node messages sent during a simulation is used to control the duration of a

simulation. Each inter-node message delivered by the message queuing process is counted until the number of messages delivered equals the number to be delivered. This allows the user to choose the number of messages for the network under study.

If data files are to be loaded into any node's database then the number of data files is needed. Once a user indicates that data files are to be used, the names of all data files, which nodes each is to be used at, and the file formats must be entered. A file format is a description of the fields in a record from the file. File formats are used to load the databases. This additional information is used to load the data files into the proper nodes' databases.

The maximum number of data files permitted in a network is equal to the maximum number of nodes. This convention allows each node to have at least one file which is unique in the entire network. A capability such as this permits study of file transfer problems in database applications.

The specifications are summarized in Table 1.

Table 1 - Specifications

Number of nodes in the network
Topology of the network
Node code file names
Data file names (optional)
Data file usage (optional)
Data file formats (optional)
Number of inter-node messages

Utilities:

Utilities are procedures which provide special functions to be used at a node during a simulation. Utilities are called from within the node code program to perform the desired function.

The utilities are included with the node code by using the C compiler's preprocessor. This relieves the user of tedious details involving the utilities.

Currently there are three utilities available to be used: communications utility, random number utility and a database utility. The communications utility is always used by every node as it provides the means for inter-node communication. The random number utility generates random integer values in the range 0 - ((2^15) - 1). The database utility is a primitive database management system which allows loading and unloading of the data files into a node's database. After being loaded, a database may be used for traditional database operations; such as retrieves, stores and updates, by additional calls to the database utility.

Compilation:

During entry of the specifications, a user enters the name of a file which contains the node code for each node in the network. The compilation program uses these file names to compile each node's program into a core image file. After compilation the core image is

saved by copying it to a reserved file for use by the simulation driver program.

If an error occurs during compilation of a node's code, the user is notified and given an opportunity to obtain a listing of the node code with error messages.

Simulation Driver:

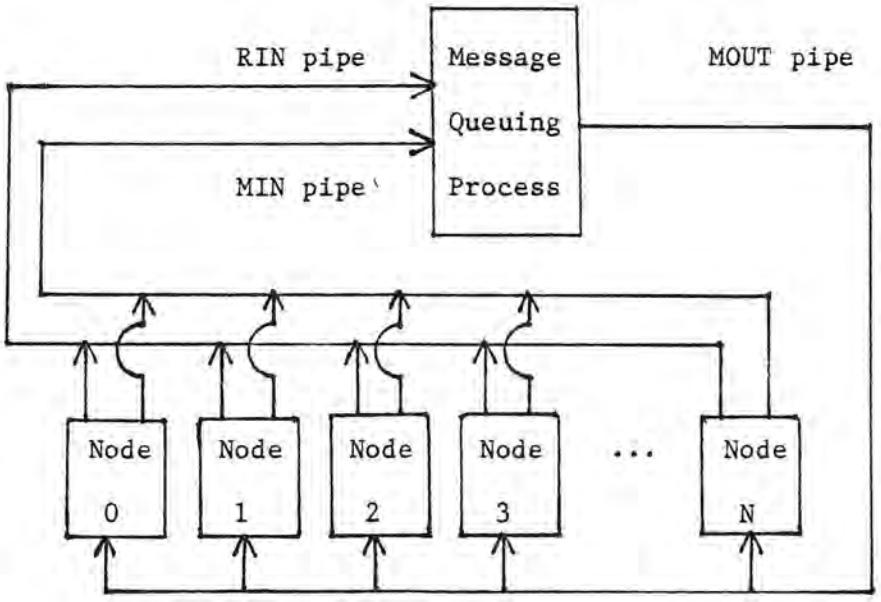
The simulation driver is responsible for creating and initializing a separate process within the UNIX operating system for each node and the message queuing process. After creation and initialization, the simulation driver initiates a simulation by broadcasting a signal to each node process and the message queuing process.

The simulation driver establishes three inter-process pipes for use by the node processes and the message queuing process. An inter-process pipe is an in-memory buffer which may be written to and read from by any process created by the simulation driver. These pipes are used by the nodes to communicate with the message queuing process, transmit inter-node messages to other nodes (via the message queuing process), and receive inter-node messages (via the message queuing process). Buffers are limited to 4096 bytes in size, which in turn limits the maximum message size. The buffer size can be adjusted by recompiling the UNIX kernel's source code with a larger buffer size.

Each of the three pipes is used for a separate function. The first pipe, RIN, is used to send requests from nodes to the message queuing process. The second pipe, MIN, is used to send inter-node messages from nodes to the message queuing process. The third pipe, MOUT, is used to send inter-node messages from the message queuing process to their destination nodes. The use of pipes maps all links in a simulated network into one resource.

The organization of the pipes and processes is diagramed in Figure 1.

Figure 1



Use of additional pipes in the simulator would have a negative impact on the simulator's performance due to the added contention in UNIX for use of buffers.

Only sixteen buffers are available for use by all processes in UNIX.

After the simulation driver initiates a simulation, it waits for the message queuing process to terminate, signaling completion of the simulation. When the simulation driver is notified that a simulation has completed, it cleans up unneeded temporary files.

Message Queuing Process:

The message queuing process is a separate program whose core image is loaded into a separate process by the simulation driver when a simulation is beginning. During a simulation the message queuing process performs four functions: 1) queuing of messages which have been transmitted but have not been "delivered" to the receiver node, 2) acting as a monitor to sychronize use of the three pipes amongst the node processes in order to carry out the simulation, 3) controlling the duration of a simulation, 4) trapping each inter-node message and creating a file of these messages for later analysis. This file of messages is called a network history file.

When a node is ready to transmit a message the communication utility at the node sends a request to transmit a message to the message queuing process via the RIN pipe. The node is then blocked until the request is processed by the message queuing process. Thus the RIN pipe serves as a queue of requests for

action by the message queuing process. When the transmit request is processed by the message queuing process it signals the node to begin writing the message into the MIN pipe. The message queuing process then reads the message from the MIN pipe and places the message on a random access file, keeping a pointer to the position of the message on the file. Lastly, the pointer to the message is placed on a queue of pointers to messages destined to the message's receiver node. Figures 2(a) - 2(d) demonstrate the sequence of events.

Figure 2(a)

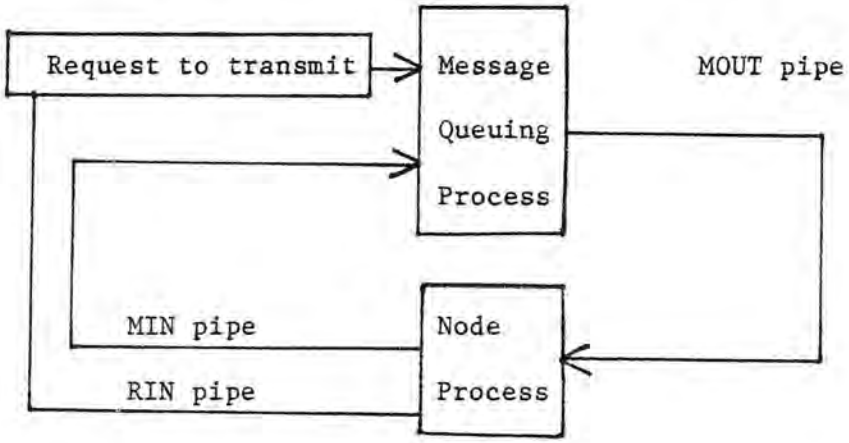


Figure 2(b)

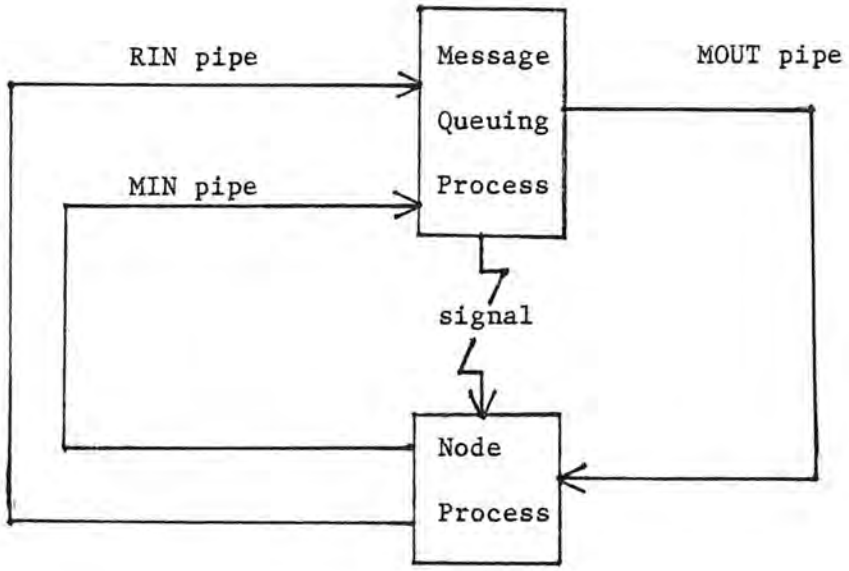


Figure 2(c)

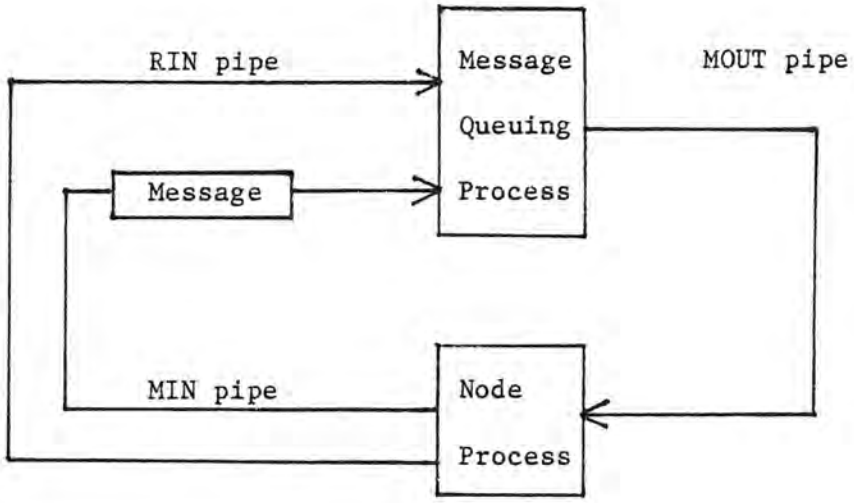


Figure 2(d)

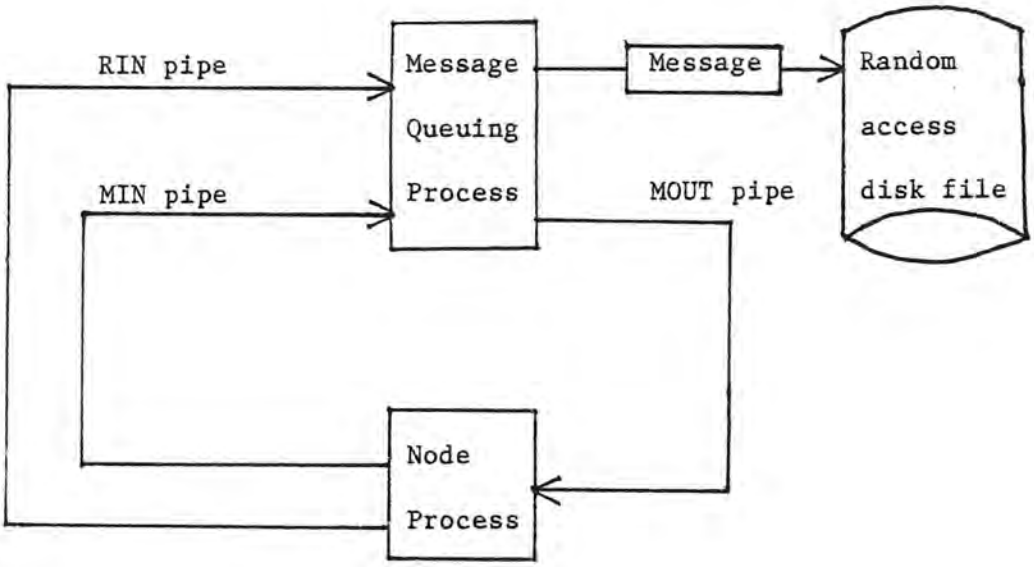


Figure 2(e)

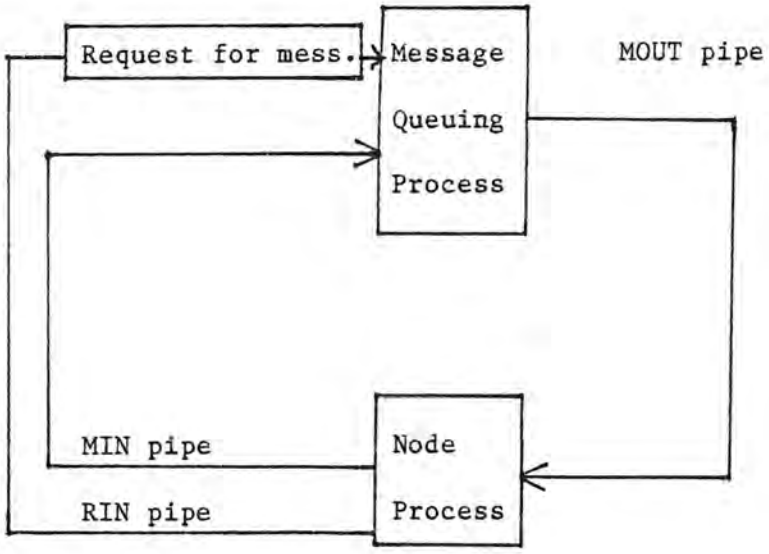


Figure 2(f)

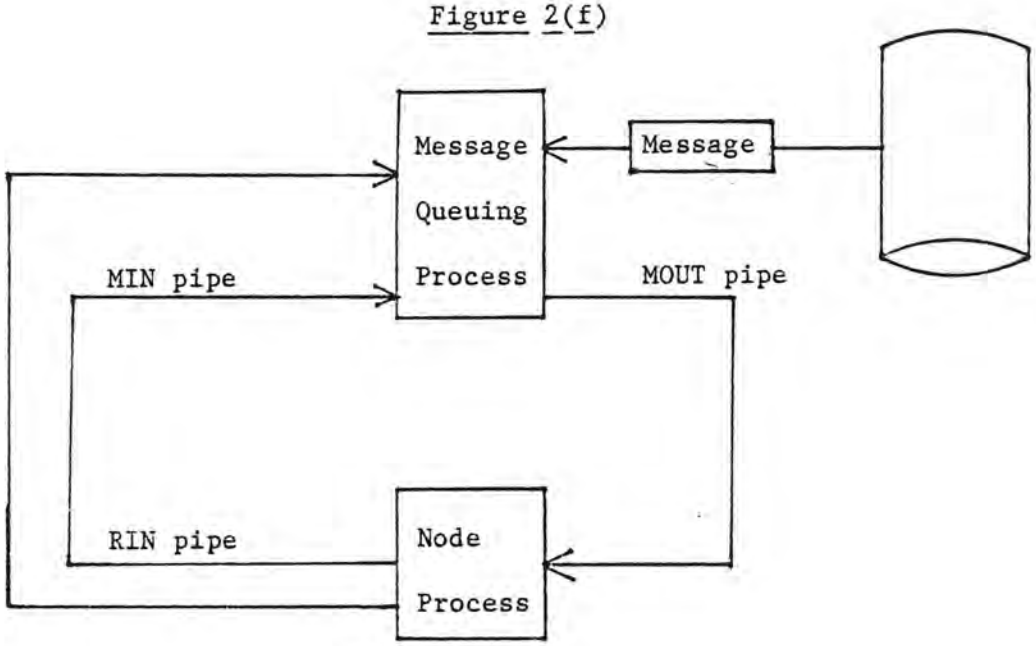


Figure 2(g)

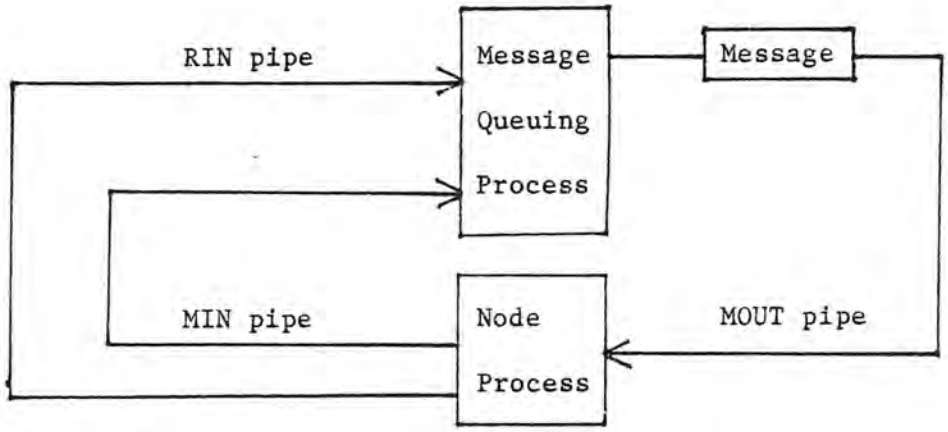
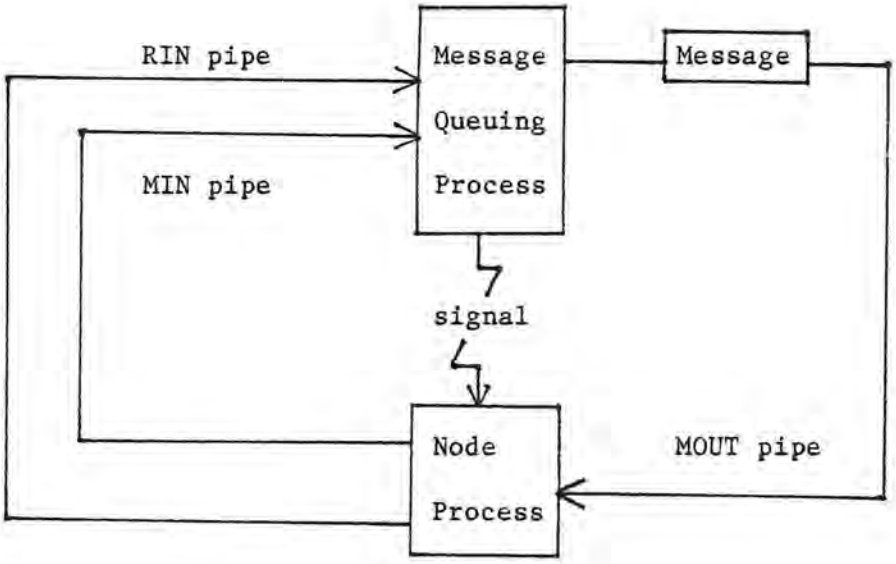


Figure 2(h)



Requests for a message are handled much in the same manner in that the message at the front of the requesting node's queue is retrieved from the random access file and placed in the MOUT pipe. The message queuing process then signals the requesting node to begin reading the message out of the MOUT pipe, as shown in Figures 2(e) - 2(h).

The message queuing process is continually responding to requests by nodes to use the MIN and MOUT pipes. In order to guarantee safe and fair use of the pipes, the message queuing process allows only one node to use either the MIN or the MOUT pipe at any given time. Unsynchronized access to the pipes is prevented by the message queuing process through use of mutually exclusive locks and inter-process signals. A lock allows a single process to read or write on a pipe.

Node processes wait their turn to use the pipes until receiving a signal from the message queuing process, after which the node gains exclusive access to the pipe by locking it.

The mutually exclusive locking of a pipe is accomplished with a non-standard UNIX system call which was implemented by Robert Eifrig of the Oregon State University Computer Science Department. A signal is a standard UNIX inter-process commmunication feature which allows a process to interrupt another. In the simulator, the receipient process is awaiting the signal, so the signal becomes an "acknowledged/proceed" message.

Scenarios for the sequence of operations performed by a node and the message queuing process during use of the MIN and MOUT pipes are given below.

MOUT Pipe Scenario

Message Queuing Process	Node Process
	Send request to MQP via the RIN pipe. Await signal from MQP before proceeding further.
Read request from RIN pipe. Write message into MOUT pipe. Signal node to begin reading the message.	
	Receive signal from MQP. Lock MOUT pipe for reading. Read message out of MOUT pipe. Unlock MOUT pipe.

MIN Pipe Scenario

Message Queuing Process	Node Process
	Send request to MQP via the RIN pipe. Await signal from MQP before proceeding further.
Read request from RIN pipe. Signal node to proceed.	
	Receive signal from MQP. Lock MIN pipe for writing. Write message into MIN pipe. Unlock MIN pipe.
Read message out of MIN pipe.	

Requests for messages are noted by the message queuing process upon receipt of them. The message queuing process maintains a logical array which is used to record the requests. An element of the array represents a request if it contains a TRUE value, which is set to FALSE when a message is sent to the node. When more than one node has requested a message and messages are enqueued for the nodes, the message queuing process must act as an arbitrator to decide which node will be serviced. The algorithm used by the message queuing process to decide which node to service is a round-robin in which each node is given its turn. A pseudo code version of the algorithm is given below.

The message queuing process main program:

```
begin
   initialize;
   while (messages_to_be sent > messages_sent)
      begin
      read an incoming request;
      if (incoming request is present) then
         begin
         process incoming request;
         service the queued messages;
         end
       else
          service the queued messages;
      end
end.
```

The initialize procedure:

```
begin
   for i = 1 to number_of_nodes
      begin
      messages_in_queue [i] = 0;
      waiting_for_message [i] = FALSE;
      idle [i] = FALSE;
      end
   token = 0;
end;
```

The process incoming request procedure;

```
begin
   if (message_type = transmit) then
      begin
      enqueue message on receiver queue;
      increment messages_in_queue [receiver];
      idle [receiver] = FALSE;
      end
   else if (message_type = request) then
      begin
      waiting_for_message [requestor] = TRUE;
      idle [requestor] = FALSE;
      end
   else if (message_type = idle) then
      begin
      idle [sender] = TRUE;
      end
end;
```

The service the queued messages procedure:

```
begin
   while (idle [token])
      begin
      advance token to next node;
      end
   if (messages_in_queue [token] <= 0) then
      begin
      advance the token to next node;
      end
   else
      begin
      dequeue a message for node [token];
      send message to node [token];
      decrement messages_in_queue;
      waiting_for_message [token] = FALSE;
      advance token to next node;
      end
end;
```

The message queuing process algorithm is designed to be fair. Examination of the algorithm shows that in the worst case a node would have to wait N - 1 turns before receiving a message. The token, initially given to node 0, must be at a node before the mode may receive a message. Since the token is advanced at least one node each time the service procedure is executed, all nodes will receive the token. It must be noted that possession of the token by a node does not guarantee that the node will be scheduled to run.

Ideally, the node processes and the mesage queuing process would alternate in being scheduled to run. Since there is contention between processes carrying out the simulation and other user's processes, the order in which the simulation processes are scheduled is non-deterministic. Use of signals by the message

queuing process to synchronize interactions between itself and node processes prevents the scheduling of processes from affecting the integrity of messages. The synchronization does increase the time required to carry out a simulation, but does not necessitate any modification to the UNIX process scheduling algorithm.

Checkup:

A simulation may require a considerable amount of elapsed time particularly if there are several users on the UNIX system while the simulation is running. UNIX allows programs to be run in the background or completely detached from an active user. Background jobs permit a user to do other useful work while a simulation is in progress, but the user may not log off of UNIX. Detached jobs allow the user to log off of UNIX while a simulation runs.

Simulations which are run as background or detached jobs do not give the user any means of determining the progress of a simulation. The checkup program allows a user to check on the progress of a simulation.

The checkup program communicates with the message queuing process to obtain the number of inter-node messages that are yet to be transmitted, which is then reported to the user. Communication between the message queuing process and the checkup program is effected by use of a temporary file which is initially

created by the message queuing process. The file contains the process identification number of the message queuing process which is read from the file by the checkup program. The process identification number of the checkup program is then written onto the file and then the message queuing process is signaled, using its process identification number.

The message queuing process contains a signal catching function which is executed upon receipt of the signal from the checkup program. This function reads the checkup process identification from the file and then rewrites the file with the message queuing process's identification number and the current number of inter-node messages left. The checkup program is signaled. Upon receipt of the signal, the checkup program reads the information from the file and reports it to the user.

Report:

At the conclusion of a simulation the network history file contains the inter-node messages which were transmitted during the simulation. Error messages generated in the message queuing process or in any of the node processes are also present in the network history file. Transactions between the node processes and the message queuing process are stored in the file in the sequence they occurred.

The report program produces a list of the transactions contained in network history file. Each inter-node message transmitted from a node to the message queuing process is noted by the transmitting node's identification number, the receiver node's identification number and the number of bytes in the message. A request for reception of an inter-node message is noted by the requesting node's identification number. If an error message is present in the network history file, it is listed with the node identification number where the error occurred along with an error number.

Totals of the number of messages transmitted, requests for messages and errors are accumulated by the report program. These totals are printed in a summary at the end of the listing.

Contents of the inter-node messages are not included in the report because of the large variety of message formats that might be used in different simulations. The UNIX operating system has a file dumping utility which may be used to view the contents of messages contained in the network history file.

Information which is pertinent to a simulation may also be output by any node onto files. A user may use this technique to trap information which is not saved in the network history file.

III. Example

An example is given in this section to demonstrate how a simulation is carried out. The problem to be studied is a communication loop which is described in [4]. The loop consists of loop interface processor nodes, called loop nodes, which transmit data amongst each other on a unidirectional data loop. Use of the data loop by the loop nodes is controlled by a control processor node which is connected to the loop nodes by a unidirectional control loop.

The control processor is responsible for maximizing use of the data loop. This is accomplished by transmission of a message from the control processor to the loop nodes instructing them when they may use the data loop. During the circuit of a control message around the control loop, each loop node uses a portion the control message to update its loop interface configuration. If the loop node wishes to send a message on the next cycle it then packs this request into the control message before forwarding it. Since a loop node must respond to the next control message, we will assume all data transfers occur within the amount of time required for one control message cycle.

Pseudo code for the loop node program is given below.

```
begin
  initialize the node;
  choose node to send first message to;
  while (forever)
    begin
    wait for control message to arrive;
    receive control message;
    unpack control message;
    case (control message)
      begin
      "receive data" : prepare loop inteface
```

```
                    to receive data;
                    /* Not implemented */
      "send data" : prepare loop interface
                    to send data;
                    /* Not implemented */
      "through data" : prepare loop interface
                    to allow data to pass
                    through;
                    /* Not implemented */
      end case;
    if (need to send a message)
       begin
       pack control message with destination's
         node number;
       send control message to next node on the loop;
       end;
    else
       begin
       pack control message with default (-1)
       send control message to next node on the loop;
       select next node to send a message to;
       end
  end while;
end.
```

When the control message has been processed by all loop nodes it returns to the control node. Upon its arrival the control message is decoded to determine which loop nodes need to use the data loop. These requests for the data loop are then examined to determine how to achieve maximum use of the data loop.

After analyzing the requests the control node forms the next control message and transmits it to the loop nodes. The process described above is repeated as long as the data loop is in operation.

Pseudo code for the control node program is given below.

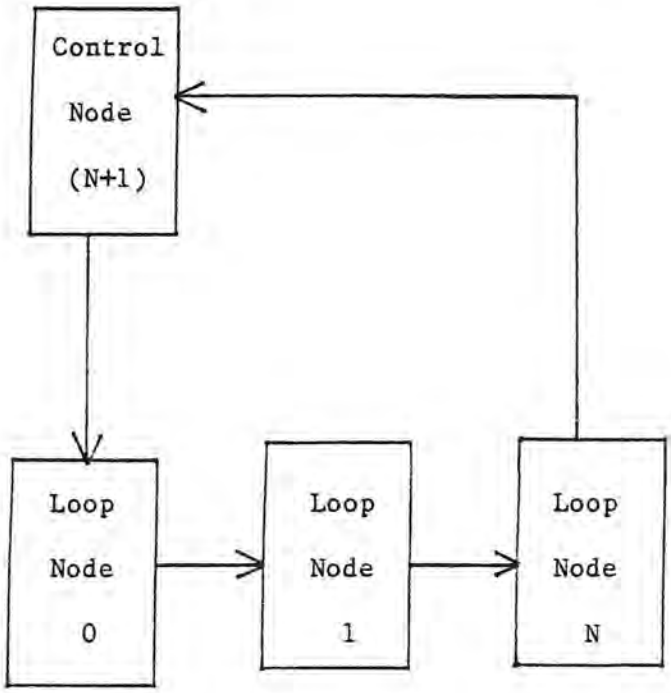
```
begin
  initialize the node;
  while (forever)
    begin
    send control message to the first node on the loop;
    wait for return of the control message;
    receive the control message;
    place requests to use the data loop in a queue;
```

```
    search the queue for requests which can proceed;
    form the control message;
    determine how many concurrent data transfers are
      occurring on the data loop and output the number to a
      file for later analysis.
    end while;
end.
```

The purpose of the example simulation is to determine the number of concurrent data transfers that may be in progress as a function of the number of loop nodes on the data loop. In order to obtain this performance data one may simulate the data loop and vary the number of loop nodes during successive simulation runs.

First one must determine the basic topology of the data loop to be used in the simulation. Since only the control loop is needed, the data loop will be excluded from the topology. The topology of the control loop is shown in Figure 3.

Figure 3



One must develop representative algorithms for a loop node and the control node. These algorithms are then coded in the C programming language and text edited into UNIX files. Node code for the loop node is given in Appendix A, and for the control node in Appendix B. Since the number of loop nodes will vary, the node code must contain a means for easily changing this number during successive runs. This is accomplished by using a defined constant whose value determines the number of loop nodes, data structure sizes and buffer sizes. The control node must save the number of concurrent data transfers as each control message is sent to the loop nodes. This is done by having the control node program write this information onto a file during a simulation.

The file created by the control node program would be analyzed by the user to determine the average number of concurrent data transfers. Each record in the file contains the number of concurrent transfers for the corresponding control message.

The specifications program is then used to set the specifications for the simulation. Each successive simulation will require the specifications to be changed to reflect the addition of loop nodes to the network. An example showing entry of the specifications is given in Appendix C.

The compilation program is then used to compile the node code programs. After completing the compilations, the simulation is ready to run. Appendix D contains an example of a terminal session in which the compilation program and the simulator are executed. Appendix E shows the output from the report program, for the example simulation.

IV. Extensions:

A simulation may be thought of as a sequence of events. Each event is caused by receipt and processing of an inter-node message or by some node's self-initiation. Effects of an event are then manifested in subsequent inter-node messages or some intra-node action by the receipient of a message. During the execution of the sequence of events the simulator is not capable of measuring elapsed time. Since the ability to measure time is crucial to obtain meaningful results about certain aspects of a

simulation, this section describes how the present network simulator might be enabled to measure time by modification of the UNIX kernel.

The modifications are explained by first describing how UNIX currently performs the tasks which are pertinent to measurement of elapsed time. The modifications are then outlined.

UNIX is an interrupt-driven operating system. This implies that all operations performed by the UNIX kernel are initiated by some form of interrupt. UNIX uses the entire hierarchy of interrupts which are supported by the PDP/11 computer. For a complete description of the interrupts the reader is referred to [5].

Interrupts are used in UNIX to implement system calls from a running process to the UNIX kernel. System calls are requests by a process to the kernel to perform some privileged operation eg., data transfers from an external device to main memory, process spawning, suspension of an executing process. At the time a running process executes a system call its current state is saved and control passes to an interrupt vector within the UNIX kernel. The interrupt vector contains the memory address within the kernel's address space where the system call handler function resides. Control passes to the address obtained from the interrupt vector and then the actual operation to be performed by the system call is performed by executing the system call handler code. After completion of the system call, the state of the process which initiated the system call is restored and the process is restarted.

During execution of a system call, the process which initiated the system call is suspended and so it is not "aware" of

being suspended from execution. At completion of the system call, control returns to the initiating process in the form of a return as from any subprogram. Time which elapses while a system call is executed by the kernel is overhead which cannot be separated from the total amount of time which a node process runs.

There are two possible modifications which would enable an accounting of each node process's execution time to be maintained. Both modifications require addition of an accumulator variable to the process entry for each process in the UNIX process table. This accumulator would be initialized at zero at the start of a simulation. As each node process is scheduled to execute, the current system time would be saved and when the process is suspended the accumulator for the node process would be updated to reflect the amount of time accrued.

The first possible modification to the UNIX kernel which would prevent system call overhead time from being accumulated would be to rewrite the system calls which are used by the network simulator. Each system call handling function would contain code to update the node process's accumulator. A modification as outlined above would be feasible because only nine system calls (read, write, seek, sleep, lock, open, close, signal, kill) are used in the simulator.

The second possible modification would conserve memory usage within the kernel. In this scenario a single, new system call is used within the simulator. When executed, the new system call handling function updates the accumulator and then calls the system call handling function which performs the desired operation.

A parameter passed to the new system call would be used to select the desired system call.

The accumulator maintained for each node may be regarded as a local clock. Timestamps may be implemented by obtaining the accumulator's value prior to and after some event that occurs in a simulation. The accumulator's value would be retrieved from the kernel by a system call.

The last major modification which must be made to the UNIX kernel involves the scheduler. Each node in a simulated network exists as a UNIX process during a simulation. As processes, the nodes are scheduled to run by the UNIX scheduler. The scheduler uses the priorities of all processes present in the system to decide which process to run next, whenever process swapping occurs. The priority of a process is initially set by UNIX and then increased until the process is selected by the scheduler to run. After running until some event; such as a system call, I/O request or a clock interrupt, the process's priority is adjusted to reflect the CPU usage it has accumulated, thus decreasing the process priority.

The UNIX operating system allows a process to adjust its priority via a system call. This adjustment causes the UNIX scheduler to run the process more often. Unfortunately, the system call does not provide a means for scheduling one process in relation to other processes involved in a simulation.

Since the order of execution of node processes during a simulation is nondeterministic, the amount of time which one node process executes may vary considerably from other node processes. In an ideal simulation environment there would be no variance in

the amount of time that each node process executes. The variance amongst the execution times of the node processes may be negated by modification of the UNIX scheduling algorithm.

The modified scheduling algorithm would use the execution time accumulators, which are maintained for each node process, to choose the node process which has the smallest accumulator value when scheduling the next process to execute. In a scheme such as this the largest variance between the execution times of all node processes would be one time slice. Duration of a simulation could then be controlled with the time accumulated by any node process. The time slice could be adjusted by changing a constant in the UNIX kernel source code. Any value used for the constant must be large enough to prevent UNIX from thrashing.

V. Conclusions

The simulation tool provides the following benefits to a user:

- Networks may be modeled at any level.
- Communication protocols need not be used, but may be implemented if needed.
- Utilities provide special functions which may be used easily.
- Network specifications are easily selected. They may be reused or changed for a later simulation.

- Preparation for a simulation is performed by the compilation program.
- Simulations may be executed in the background or as detached jobs, freeing the user for other tasks. Progress of a simulation maybe determined in this situation by using the checkup program.

The impact of a simulation upon the operating system is determined by the network size. Swapping of node processes out of main memory will occur more frequently as more nodes are present in a simulated network. The message queuing process is also susceptible to swapping, but to a lesser degree as its system priority is higher than any node process's priority.

Other than network size, the simulator does not have any properties which adversely affect the operating system. Extensive network topologies do not affect any operating system resource more than small topologies. This is prevented by using the same inter-process pipes for transfer of the inter-node messages.

VI. References

1. Schneider, G.M., "VANS--A Resource-Sharing Computer Network Design Tool", Computer Networks and Simulation, Schoemaker, S., (Ed.), North Holland, New York, 1978, pp 227-248.

2. Davison, B., "Using a Simulation Model in the Design of a Computer Network", Computer Networks and Simulation, Schoemaker, S., (Ed.), North Holland, New York, 1978, pp 169-185.

3. Kain, R. Y., Franta, W. R., Jelatis, G. D., "CHIMPNET: A Network Testbed", Computer Networks, Vol. 3, No. 6, North Holland, New York, 1979, pp 447-457.

4. Jafari, H., Lewis, T. G., Spragins, J. D., "Simulation of a Class of Ring-Structured Networks", IEEE Trans. on Computers, Vol. C-29, No. 5, May 1980, pp 385-392.

5. Digital Equipment Company, PDP-11 Processor Handbook, Digital Equipment Company, Maynard, Mass., 1975, pp 2-7 to 2-11.

The node code for the loop nodes is given in this section. Line numbers are only for reference purposes.

```
 1  # define NUMNODES 6
 2  # define TRUE 1
 3  # define FALSE 0
 4  # include "cm.gvar"
 5
 6  char buffer [NUMNODES * 2]; /* Buffer for control message */
 7  int nodeid; /* Node's id number */
 8  int t [2]; /* Current time - used to set rn utility */
 9  int needtosend; /* Is another data transfer needed */
10  int control; /* Control value from controller node:
11                  0 = data is to go through the node.
12                  1 = begin receiving data.
13                  2 = begin transmitting data. */
14  int sendto; /* Node to which the next data transfer
15                 is directed */
16  int forever; /* Used to loop on */
17
18  main () {
19
20  int i;
21
22   nodeinit();  /* Initialize node */
23   nodeid = _ln();  /* Get node id */
24   time(t);  /* Initialize rn utility with
25                the current time */
26   srand(t[2]);
27   needtosend = rn(0,1); /* Set to True or False */
28   sendto = rn(0,(NUMNODES-1)); /* Decide which node
29                                   to send data to */
30   forever = TRUE;
31   while (forever) {
32     if (i = receive()) {
33        /* Receive and process a control message */
34        control = _ctoi(&buffer[nodeid * 2]);
35        switch (control) {
36          case 0 : throughtraffic();
37                   break;
38          case 1 : indata();
39                   break;
40          case 2 : outdata();
41                   break;
42          default : printf("Bad message.\n");
43        }
44     }
45     if (needtosend) {
46        /* If a data transfer is needed, pack node
47           id of receiver into the control message
48           and send the control message to the next
49           node on the control loop. */
50        _itoc(sendto, &buffer[nodeid * 2]);
51        _send();
```

```
52          }
53       else {
54          /* No data transfer is needed, so pack
55             -1 into control message and send it. */
56          _itoc(-1, &buffer[nodeid * 2]);
57          send();
58          /* Reset sendto and needtosend for next
59             time. */
60          needtosend = rn(0,1);
61          sendto = rn(0,(NUMNODES - 1));
62          }
63       }
64    }
65
66 /* Include the communications and random number
67    utilities into the node code. */
68
69 # include "cm.c"
70 # include "rn.c"
71
72 /*
73  *  Function Receive
74  */
75
76 receive () {
77
78 int i;
79
80  if ((i = cm(1,0,buffer,0)) <= 0) {
81     printf("Node %d -- receive error.\n",
82            nodeid);
83     return(FALSE);
84     }
85  return(TRUE);
86  }
87
88 /*
89  *  Function Send
90  */
91
92 send () {
93
94 int nextnode;
95
96  nextnode = nodeid + 1;
97  cm(2, nextnode, buffer, (NUMNODES*2));
98  }
99
100 /* The throughtraffic, indata and outdata
101    functions are stubs in the loop node code
102    program because data transfers on the
103    data loop, per se, are not part of this
104    simulation. */
105
106 /*
107  *  Function Throughtraffic
```

```
108   */
109
110   throughtraffic () {
111
112   /* Throughtraffic would configure the data loop
113      interface for transfer of data through the
114      node's interface.  The node would neither
115      receive nor send data. */
116
117   }
118
119   /*
120    *  Function Indata
121    */
122
123   indata () {
124
125   /* Indata would configure the data loop
126      interface to receive data at the node. */
127
128   }
129
130   /*
131    *  Function Outdata
132    */
133
134   outdata () {
135
136   /* Outdata would configure the data loop interface
137      to transmit data from the node */
138
139   }
```

Appendix B -- Control Node Program

The node code for the controller node is given in this section. Line numbers are only for reference purposes.

```
 1  # define TRUE 1
 2  # define FALSE 0
 3  # define NUMNODES 6
 4
 5  # include "cm.gvar"
 6
 7  char buffer [NUMNODES * 2];  /* Control message buffer */
 8  int queue [NUMNODES * 2] [3];  /* Queue of requests to use
 9                                    the data loop.
10                                    queue[i][0]=sender
11                                    queue[i][1]=receiver
12                                    queue[i][2]=link to next
13                                                 queue node */
14  int fd;  /* File descriptor for output file */
15  int free;  /* Next free queue node */
16  int front, back;  /* Front and back of queue */
17  int inuse [NUMNODES] [NUMNODES];  /* Matrix of links in use.
18                                       rows=nodes
19                                       columns=links */
20  int busy [NUMNODES];  /* Busy nodes */
21  int links [NUMNODES];  /* Busy links */
22  int inqueue [NUMNODES];  /* Nodes which have requests in queue */
23
24  main () {
25
26   int forever, i;
27
28    /* Initialize control node for the simulation */
29    nodeinit();
30    initialize();
31    /* Loop forever, processing control messages */
32    forever = TRUE;
33    while (forever) {
34      /* Send a control message */
35      if (i = sendpacket()) {
36         /* Wait on return of control message */
37         if (i = receivepacket()) {
38            /* Place requests in queue */
39            enqueue();
40            /* Determine which nodes may use the
41               data loop */
42            maxuse();
43            }
44         /* Form next control message */
45         formessage();
46         }
47      }
48    }
49
50  # include "cm.c"
51
```

```
 52  /*
 53   *  Function Initialize
 54   */
 55
 56  initialize () {
 57
 58   int i, j;
 59
 60    /* Set up queue as a singly linked list */
 61    for (i = 0; i < ((NUMNODES * 2) - 2); ++i)
 62        queue [i] [2] = i + 1;
 63    queue [(NUMNODES * 2) - 1] [2] = -1;
 64    /* Set all data structures to FALSE because no
 65       data transfers are in progress */
 66    for (i = 0; i < NUMNODES; ++i) {
 67        for (j = 0; j < NUMNODES; ++j)
 68            inuse [i] [j] = FALSE;
 69        busy [i] = FALSE;
 70        links [i] = FALSE;
 71        inqueue [i] = FALSE;
 72        }
 73    /* Initialize queue pointers */
 74    free = 0;
 75    front = -1;
 76    back = -1;
 77    /* Open output file for number of busy nodes */
 78    fd = creat("control.out", 0600);
 79    /* Form first control message */
 80    formessage();
 81    }
 82
 83  /*
 84   *  Function Sendpacket
 85   */
 86
 87  sendpacket () {
 88
 89   int i;
 90
 91    /* Send control message to node 0 */
 92    if ((i = cm(2, 0, buffer, (NUMNODES * 2))) !=
 93        (NUMNODES * 2)) {
 94       printf("Transmit error at controller.\n");
 95       return(FALSE);
 96       }
 97    return(TRUE);
 98    }
 99
100  /*
101   *  Function Receivepacket
102   */
103
104  receivepacket () {
105
106   int i;
107
```

```
108    /* Receive control message from last node on loop */
109    if ((i = cm(1, 0, buffer, 0)) <= 0) {
110       printf("Receive error at controller.\n");
111       return(FALSE);
112       }
113    return(TRUE);
114    }
115
116  /*
117   *  Function Freelist
118   */
119
120  freelist () {
121
122   int i;
123
124    /* Return pointer to next free element in the queue */
125    i = free;
126    free = queue [free] [2];
127    return(i);
128    }
129
130  /*
131   *  Function Enqueue
132   */
133
134  enqueue () {
135
136   int i, j, k;
137
138    /* For each node on the loop */
139    for (i = 0; i < NUMNODES; ++i) {
140      /* Unpack node's request */
141      j = _ctoi(&buffer[i*2]);
142      /* If request is ok place it in the queue */
143      if ((j >= 0) && (j != i) && (! inqueue[i])) {
144         k = freelist();
145         queue [k] [0] = i;
146         queue [k] [1] = j;
147         queue [k] [2] = -1;
148         queue [back] [2] = k;
149         back = k;
150         /* Clear the node's data structures,
151            it is idle */
152         for (j = 0; j < NUMNODES; ++j)
153           inuse [i] [j] = FALSE;
154         busy [i] = FALSE;
155         inqueue [i] = TRUE;
156         }
157      }
158    /* Update the links array to show which links are in use */
159    for (j = 0; j < NUMNODES; ++j) {
160      links [j] = FALSE;
161      for (i = 0; i < NUMNODES; ++i)
162          links [j] = links [j] || inuse [i] [j];
163      }
```

```
164    }
165
166  /*
167   *  Function Maxuse
168   */
169
170  maxuse () {
171
172   int qp, i, ok, j;
173
174    qp = front;
175    while (qp > -1) {
176      ok = TRUE;
177      j = queue [qp] [0];
178      /* Check links to see if they are in use by other nodes */
179      while ((j != ((queue [qp] [1] + 1) % NUMNODES)) && ok) {
180         if (links [j])
181            ok = FALSE;
182         j = (j + 1) % NUMNODES;
183         }
184      if (ok) {
185         /* Links are free, reserve them */
186         j = queue [qp] [0];
187         while (j != ((queue [qp] [1] + 1) % NUMNODES)) {
188            links [j] = TRUE;
189            inuse [queue [qp] [0] ] [j] = TRUE;
190            }
191         /* Node is busy, take its request off
192            of the queue */
193         busy [queue [qp] [0] ] = TRUE;
194         inqueue [queue [qp] [0] ] = FALSE;
195         release();
196         qp = front;
197         }
198      else
199         /* No more requests can be honored, due to
200            the use of the data loop, wait until next
201            time */
202         qp = -1;
203      }
204    /* Count busy nodes and output the number to the file
205       for later analysis */
206    i = 0;
207    for (j = 0; j < NUMNODES; ++j)
208      i = i + busy [j];
209    write(fd, &i, 2);
210    }
211
212  /*
213   *  Function Release
214   */
215
216  release () {
217
218   int i;
219
```

```
220    /* Release queue element at the front of the queue and
221       place it back on the freelist */
222    i = front;
223    front = queue [i] [2];
224    queue [i] [2] = free;
225    free = i;
226    }
227
228  /*
229   *  Function Formessage
230   */
231
232  formessage () {
233
234   int i, j, end;
235
236    /* Initially set all nodes to "through" traffic */
237    for (i = 0; i < NUMNODES; ++i)
238        _itoc(0, &buffer[i*2]);
239    /* For each busy node, find the end of its
240       transmission path */
241    for (i = 0; i < NUMNODES; ++i) {
242        if (busy [i]) {
243           j = (i + 1) % NUMNODES;
244           end = (i - 1) % NUMNODES;
245           while (j != i) {
246              if (inuse [i] [j])
247                 j = (j + 1) % NUMNODES;
248              else {
249                 end = (j - 1) % NUMNODES;
250                 j = i;
251                 }
252              }
253           /* Set control message fields to notify
254              transmitting and receiving nodes */
255           _itoc(2, &buffer[i*2]);
256           _itoc(1, &buffer[end*2]);
257           }
258       }
259    }
```

Appendix C -- Specifications Example

This section shows how a user would use the specifications program to select the necessary specifications for a simulation. The specifications correspond to the simulation which would use the node code in Appendices A and B. Responses to queries from the program are underlined for clarity.

```
% specs
Welcome to the Network Simulator.

   This program is the initial step in using the
network simulation package on the PDP 11.


You may set up and/or use a network
simulation specification file by entering one
of these modes:

  new
  old
  change

new

A network may contain 2 - 45 nodes.
How many nodes are needed in your network?

7

There will be 7 nodes in the network.
They will be numbered 0 - 6.


Please enter the topology specifications.
The topologies which are available are:

   complete
   ring
   star
   general

What is the topology of your network?

ring

The topology will be a ring network of 7 nodes.


  This is the adjacency matrix which represents
the topology of your network.

      To
```

```
From  0123456
   0  0100001
   1  1010000
   2  0101000
   3  0010100
   4  0001010
   5  0000101
   6  1000010
```

The adjacency matrix has been translated into a more readable form.

The links in your network are:

```
From  To
   0  1, 6,
   1  0, 2,
   2  1, 3,
   3  2, 4,
   4  3, 5,
   5  4, 6,
   6  0, 5,
```

The topology specifications have been completed.
Are you satisfied with the current specifications? Enter "yes" or "no".

yes

Please enter the node code specifications.

Please use a carriage return on a new line to terminate input of the node code specifications.

loop.c 0 1 2 3 4 5
control.c 6

The node code file for each node is:

At node 0, loop.c
At node 1, loop.c
At node 2, loop.c
At node 3, loop.c
At node 4, loop.c
At node 5, loop.c
At node 6, control.c

The node code specifications have been completed.
Are you satisfied with the current specifications? Enter "yes" or "no".

yes

Please enter the data file names of all files needed in your network. You may enter

```
up to 45 data file names, if more are
entered they will be ignored.

Please use a carriage return on a new line to
terminate input of the data file name specifications.

(CR)

No data files will be used in your network.


The data file name specifications have been completed.
Are you satisfied with the current
specifications?  Enter "yes" or "no".

yes

How many inter-node messages
are to be sent during the simulation?

6000

There will be 6000 messages sent.


The number of inter-node messages specifications have been completed.
Are you satisfied with the current
specifications?  Enter "yes" or "no".

yes
```

Appendix D -- Compilation and Simulation Example

In the following example the compilation program, simulator and checkup program are executed. The simulator is ran as a background job.

```
% comp
The node code for node 0 has been compiled.
The node code for node 1 has been compiled.
The node code for node 2 has been compiled.
The node code for node 3 has been compiled.
The node code for node 4 has been compiled.
The node code for node 5 has been compiled.
The node code for node 6 has been compiled.
% sim > simout&
processid
% simcheck
5999 messages to go.

Again?
yes
5997 messages to go.

Again?
no
%
```

Appendix E -- Report Program Output

In the following example the report program is executed. Only a portion of the output is shown due to its length.

```
The network contains 7 nodes.
The nodes were numbered from 0 to 6.
The network traffic during the simulation
was 6000 inter-node messages.

Node 1 requested a message
Node 2 requested a message
Node 3 requested a message
Node 4 requested a message
Node 5 requested a message
Transmit 12 bytes from node 6 to node 0
Node 0 requested a message
Node 6 requested a message
Transmit 12 bytes from node 0 to node 1
Transmit 12 bytes from node 1 to node 2
Node 1 requested a message
Transmit 12 bytes from node 2 to node 3
Node 2 requested a message
Node 0 requested a message
Transmit 12 bytes from node 3 to node 4
Node 3 requested a message
Transmit 12 bytes from node 4 to node 5
Node 4 requested a message
Transmit 12 bytes from node 5 to node 6
Node 5 requested a message
Transmit 12 bytes from node 6 to node 0

 .
 .
 .

Network simulation errors = 0
Messages transmitted = 6000
Messages requested = 6006
Idle nodes = 0
```

Appendix F -- User Manual

This appendix contains the User Manual for the network simulator.

Oregon State University
Computer Science Department

WASP-2
August 1981

Network Simulator User Manual

Walter Domka
Pat Kalvin
Michael Freiling

A software system to study network algorithms was implemented on UNIX. Each part of a network algorithm is written as a single C program which becomes a virtual node in the network. During a simulation, all virtualized nodes run as separate processes on a single PDP 11/40. Inter-node communication is carried out with procedures local to each node, to send and receive inter-node messages to and from a message queuing process. Communication between nodes is effected by use of virtual links which are specified in the simulated network's topology. The links are implemented on inter-process pipes between the message queuing process and the node processes.

Working and Software Papers are intended primarily for internal circulation. They contain software documentation and rough drafts of ideas. These papers are kept as up-to-date as possible.

T A B L E O F C O N T E N T S

1. Introduction 1
2. Definitions 4
3. Capabilities 5
4. Reserved Files 7
5. Node Code 9
   5.1 Utilities 10
      5.1.1 Communication Utility 11
      5.1.2 Random Number Utility 14
      5.1.3 Database Utility 15
   5.2 An Example 15
6. Specifications 17
   6.1 Modes 17
      6.1.1 NEW Mode 18
         6.1.1.1 Topology 18
         6.1.1.2 Node Code 21
         6.1.1.3 Data File Names 22
         6.1.1.4 Data File Usage 24
         6.1.1.5 Data File Formats 24
         6.1.1.6 Number of Messages 25
         6.1.1.7 An Example 26
      6.1.2 CHANGE Mode 31
         6.1.2.1 An Example 32
      6.1.3 OLD Mode 34
7. Compilation 35
8. Simulation 36
9. Report 37
10. Checkup 40

Appendices

A. Query Response Syntax 41

1. Introduction

A computer network consists of two or more communicating virtual machines. The algorithm that controls the interaction between components of a network is distributed amongst these components. Until now, the developement of network algorithms has required that a hardware implementation of the network be in existence for testing of the algorithm. The network simulator which this manual describes, allows a network algorithm to be created and then tested under a simulated environment. This manual is intended to be the source of information needed by anyone to successfully use the network simulator.

The network simulator runs on the UNIX operating system of the PDP 11/40. In fact, the simulation of a network is made possible by the flexibility and accessibility of the UNIX operating system. To use the simulator, one must be fluent in the C programming language, the UNIX operating system and the use of the UNIX text editor.

A simulation run requires that the user create one or more component algorithms which make up the network algorithm. The components are then coded in the C programming language. After this, the C code and any needed data files are text edited into UNIX files.

The network to be simulated is then described to the specifications program. These specifications will be used by the simulator to carry out the simulation. The specifications program which collects this information concerning the network from the user is interactive. It queries the user to input his network

specifications by using full sentence queries. Any errors detected in these specifications are reported to the user who may then correct them before proceeding further. After each step of the specification input has been completed, the specifications are displayed back for the user's inspection. If any changes are needed, the user is given a chance to make them. After completing the input of a particular network's specifications, they are saved on a file that may be reused or altered.

Node code programs must be compiled prior to a simulation. The node code file names, which are stored in the specifications file, are used by the compilation program to compile the programs. After each simulation, the resulting core image file is copied onto a reserved file. Reserved files are used so that the files may be located by the simulation driver program.

The compilation program is also capable of detecting compilation errors which occur while compiling node code programs. If this occurs, the user is queried to determine if a listing file of the node code program and compilation error messages is needed for diagnostic purposes. An affirmative response from the user causes the compilation program to create the listing file.

When all node code programs have been successfully compiled, the simulation driver program is used to carry out the actual simulation. The driver program creates a separate UNIX process for each node and for the message queuing process, and establishes the UNIX pipes used to transmit inter-node messages. The core image files created during compilation of node code programs are then loaded into the UNIX processes. A core image of the message queuing program is also loaded into one process which

becomes the message queuing process. Initialization of each process is accomplished by having the process read the information needed to communicate with other processes from a temporary file. The temporary file was created by the simulation driver program prior to initialization of the process.

After all node processes and the message queuing process are initialized, the simulation driver broadcasts a startup signal to the processes informing them to begin the simulation. While the node processes are busy with the simulation, the driver waits on the message queuing process to terminate indicating completion of the simulation.

During a simulation, the message queuing process creates a file which contains a record of each inter-node transaction and each inter-node message. The contents of the file, called a network history file, may be examined by the user to determine success or failure of a particular simulation. A report program is available which prints a listing of the transactions that occurred during a simulation. The report program reads the history file in order to generate its listing.

Simulations may require a considerable amount of machine time, particularly if many nodes are in a network. Progress of a simulation may be determined by using the checkup program. The checkup program interrupts the message queuing process and obtains the current number of inter-node messages which must be transmitted before the simulation is completed. After the checkup program obtains the number of messages, it prints the number at the user's terminal.

2. Definitions

It is necessary to define the meaning of several words and phrases as used in this manual.

Node - The entity which represents a virtual machine in a network. The virtual machine level at which a network is simulated may be chosen by the user. For example, one user might regard a node as being a complete computer installation while another user might view a node as a low level communication loop interface processor.

Link - A communication channel that connects a pair of nodes. A link is best viewed as a virtual, bidirectional channel between the nodes in that the mechanics of the channel are completely transparent to the user and the nodes.

Topology - A description of how the nodes in a network are connected together by links.

Node code - The code which implements the part of a network algorithm which is local to a particular node. If all nodes of a network are duplicated, then a user would only have to create one version of the node code. If a network is to contain different node code at some nodes, the user would code each distinct algorithm separately. Each node code is a complete algorithm in itself, but also a component of the network algorithm.

Data file - A rectangular set of values which is local to a node in the network. The values are treated as attributes in the relational database sense. Each node can manipulate the contents of any of its data files during a simulation by performing updates, retrievals, deletions and stores.

Utility - A commonly used procedure which may be called by the user from his node code. The utilities are loaded with the node code as needed by using the C compiler's preprocessor. The utilities are useful for such tasks as manipulating the data files, and most importantly, performing inter-node communication.

History file - A file created during a simulation which contains information about the simulated network's performance. The inter-node communication utility traps each message and writes it and identifying information onto the history file.

3. Capabilities

The capabilities of the network simulator are determined by the values of defined constants in the simulator and the current configuration of the UNIX operating system on which the simulator is running.

These defined constants are listed below:

```
MAXDATATYPES 2
MAXFIELDS 10
MINNODES 2
MAXNODES 45
MAXLINE 100
MAXPIPE 4096
```

Each of the constants control the indicated capability of the simulator:

MAXDATATYPES - The number of different data types that may be used in any data file. The 2 types are integer (16 bit) and character strings.

MAXFIELDS - The number of fields or attributes that may occur in any data file tuple.

MAXLINE - The maximum number of characters in a line of input from the user's terminal to the simulation control software.

MAXNODES - The maximum number of nodes that may be in a simulated network. This constant is very dependent upon the local UNIX operating system. Each node is simulated by using a process in the operating system. Care should be taken not to set this constant too high and therefore conflict with the number of processes needed for the UNIX kernel.

This constant also sets the number of data files used at any node. This allows each node to have a local data file that is not used at any other node in the network.

MINNODES - The minimum number of nodes in a network.

MAXPIPE - The maximum size, in bytes, of a UNIX pipe buffer. Since messages are sent through pipes, the maximum pipe size forces the maximum message size to be 4092 bytes. Four bytes are used by the simulator for a message header.

4. Reserved Files

During a simulation several UNIX files are used. The files which contain programs used before, during and after a simulation, are present in the user's directory. In addition to these, temporary files are created and used by several of the programs.

The user must obtain the programs which are used for simulations from the backup device on which they are stored. These files are not normally kept on the system.

To prevent file name conflicts the user must not use any of the files in the list shown below.

File Name	Contents
cm.c	C source code for the communication utility.
cm.gvar	An "include" file of global variables needed by the communication utility.
comp	The compilation program.
dm.c	C source code for the database utility.
dm.gvar	An "include" file of global variables needed by the database utility.
mqp	The message queuing program.
netprt	The report program.

ntext	A procedure file which calls the editor to place line numbers on node code files for diagnostic purposes.
rn.c	C source code for the random number utility.
sim	The simulation driver program.
simcheck	The checkup program.
specs	The specifications program.

While completing a simulation temporary files are created, used and unlinked within the user's directory. The list of files shown below describes which files are used by the various programs and processes.

File	Creator	Used By	Unlinked By	Contents
net.spec	specs	specs, comp, sim	It is not unlinked	network specifications
net.history	message queuing process	netprt	It is not unlinked	inter-node messages, error messages from a simulation
node.init	sim	node processes	sim	initialization information needed at the node processes
cm.init	sim	message queuing process	message queuing process	initialization information needed at the message queuing process
net.check	message queuing process	message queuing process	message queuing process	process id number of the message queuing process and number of messages, or process id of the checkup process
t	comp	comp	comp	compilation error messages

a.out	comp	comp	comp	core image of a node code program
nout	comp	comp	comp	preprocessed node code program with line numbers
nodecode.i	comp	comp	comp	preprocessed node code program
nodecode.e	comp	comp	It is not unlinked	preprocessed node code program with line numbers and compilation error messages
ncXX	comp	comp, sim	sim	core image of a node code program for node XX

5. Node Code

Node code programs are simply C programs. As such, they must meet all requirements inposed by the Version 6 C compiler. Three additional requirements are imposed on node code programs, to allow them to be used in a simulation. The three requirements are:

1. The C compiler's preprocessor is used to include utility source code and global variable declarations into node code programs. The first line of the source code must have the "#" character at the front of the line. This alerts the C compiler to preprocess the source code.

2. Variable names and function names may not begin with the underscore character, "_". The underscore character is reserved for unique variable names and function names within the utilities' source code.

   Additional function names are reserved because they are used for a utility or the node initialization function. The reserved function names are: "nodeinit", "cm", "rn" and "dm".

3. The first executable statement within the "main" function of every node code program must call a function which initializes the node prior to a simulation. This function is part of the communication utility since it must always be used. The function is called with a function reference as shown below.

```
main() {
nodeinit();
 .
 .
 .
}
```

5.1 Utilities

Utilities are collections of one or more C function subprograms. Each utility performs a special function at a node. The

communication utility must be used at all nodes to allow inter-node communication. The other utilities may be used if they are useful.

Some utilities require global variables in order to work correctly. These global variables are also included in the node code programs by using the C preprocessor. The included global variables must be positioned in the node code program where global variable declarations normally appear.

Utilities and their global variables are included by placing a statement of the following form in a node code program.

```
# include filename
```

where filename is the UNIX file that contains the utility or global variables.

5.1.1 Communication Utility

This utility allows nodes to:

1. Send messages to other nodes.
2. Receive messages.
3. Set the node to an idle status. This informs the message queuing process that the node is not going to receive any messages. This allows the message queuing process to skip the node when deciding which node to deliver a message to, and decreases the amount of time required to carry out a simulation.

4. Poll the message queuing process to determine how many messages are queued up for the node.
5. Obtain the node number which identifies the node within the network. This allows the user to write node code which is independent of any particular node numbers.
6. Pack integers into character arrays when forming inter-node messages in a character array buffer.
7. Unpack integers from character arrays when messages are stored in character array buffers.

The communication utility is called from node code programs with the following function reference.

```
cm(code, to, madr, nbytes);
with:  int code, to, nbytes;
       *char madr;
```

The value of the parameter code determines what the utility does as explained below.

Code Value

1 In this case, the utility will receive the next message which is to be delivered to the node. The other parameters have the following interpretations.
to = Not used.
madr = The address of a buffer into which the message will be placed. The buffer must be large enough to accommodate the message.
nbytes = Not used.
If a message is received successfully, the utility returns the number of bytes transfered into the buffer, otherwise it returns a -1 value.

2 A message is to be sent to another node.
to = Node identification number of the node to which the message is to be sent.

madr = Address of a buffer in which the message is stored.
nbytes = Number of bytes in the message, must be less than 4093.
In this case the utility returns the number of bytes transmitted (nbytes) if no errors occurred, otherwise -1 is returned.

3 Set the node to idle status.
to, madr, nbytes = Not used.
In this case the utility returns a value of 0 if no errors occurred, otherwise -1 is returned.

4 Poll the message queuing process to find out how many messages are waiting to be delivered to the node.
to, madr, nbytes = Not used.
In this case, the utility returns a value that is greater than or equal to zero to inform the node code of the number of messages, otherwise -1 is returned if an error occurred.

The communication utility contains a function which returns the local node's identification number. The function reference has the following form.

```
_ln();
```

Packing of integers into character array buffers may be accomplished with the following function reference.

```
_itoc(n, c);

with:  int n;
       *char c;
```

where n is the integer value and c is the address of a 2 byte area into which the integer value is to be packed. No value is returned by the function.

Unpacking of integers from character arrays is accomplished with the following function reference.

```
_ctoi(c);
with: *char c;
```

where c is the address of a 2 byte area from which a 16 bit integer is unpacked. The function returns the integer's value.

The communication utility is stored on a file named "cm.c" and the global variables used by the utility are stored on "cm.gvar". Placement of the include statements in a node code program in order to use the utility is demonstrated in section 5.2.

5.1.2 Random Number Utility

Random numbers are often used in simulations. UNIX provides a system call, rand, which generates pseudo-random integer values in the range 0 to ((2^15)-1). The random number utility allows a user to specify a smaller range from which random values may be selected. This is done by calling the utility with two parameters which set the lower and upper bounds of a closed interval in which the number may occur.

The utility is called with the following function reference.

```
rn(lo, hi);
with:  int lo, hi;
```

where lo and hi are the lower and upper bounds of the range. The endpoints (lo and hi) are considered to be in the valid range.

The source code for the utility is stored on a file named "rn.c". No global variables are used by the utility.

5.1.3 Database Utility

This section will be completed by Pat Kalvin.

5.2 An Example

This section contains two node code programs. They demonstrate how utilities are included in node code, and where the node initialization function reference is placed. These programs make up a simple network in which node 0 continually sends messages to node 1. The message is the character "a".

Node Code for Node 0

```
# include "cm.gvar"

char buf [5];

main () {

 int i, j;

  nodeinit();
  cm(3,0,0,0);
  buf [0] = 'a';
  for (j = 0; j < 32000; ++j) {
      if ((i = cm(2,1,buf,1)) != 1)
         printf("Node 0 cant send\n");
      }
  }

# include "cm.c"
```

Node Code for Node 1

```
# include "cm.gvar"

char buf [5];

main () {

 int i, j, sum;

  nodeinit();
  j = 1;
  sum = 0;
  while (j) {
      if ((i = cm(1,0,buf,0)) != 1)
         printf("Node 1 cant receive\n");
      else
         printf("%d\n", ++sum);
      }
  }

# include "cm.c"
```

6. Specifications

This section describes entry of the specifications. The order in which the different types of specifications are presented in this section is the same as the program will use when querying the user.

6.1 Modes

The specifications may be completed by using one of three modes: NEW, CHANGE, or OLD. Each of the modes has the following effects:

NEW - An entirely new network specification file, net.spec, is to be created. The user will have to preserve any previous net.spec by copying it to a backup file with another name prior to entering the NEW mode.

CHANGE - An existing network specification file is to be altered. The specifications must be present in a file named net.spec in the current UNIX directory being used.

OLD - An existing network specification file, net.spec, is to be used without alteration. Again, the file must be present in the user's directory.

The specifications program will display an introductory message and a mode menu on the user's terminal, and then wait for entry of one of the modes. If the user enters an incorrect mode, an error message will be displayed and the user will be allowed to try again.

6.1.1 NEW Mode

If the user wishes to create a new specifications file the NEW mode is used. After entering the mode the user will be queried to begin the topology input.

6.1.1.1 Topology

The topology specifications contain two parts; the number of nodes and the network's topology. The number of nodes is input first. Any value entered here must be in the range displayed.

The number of nodes is displayed back to the user along with the numbers that will identify the nodes. These identification numbers will be used when entering other specifications. It is important to remember that the numbers range from 0 to (N-1) rather than from 1 to N, where N is the number of nodes in the network.

There are several error messages that may be displayed to the user if the number of nodes entered is out of the valid range.

After entering the correct number of nodes, a topology menu will be displayed and the user may then enter the topology which he needs. The topologies which appear in the menu are commonly used to describe networks. The topologies have the following interpretations.

COMPLETE - The nodes will be connected so that each node has a link to every other node.

RING - A ring is formed with links connecting node 0 to node 1, node 1 to node 0, node 1 to node 2, node 2 to node 1, ..., node N-2 to node N-1, node N-1 to node N-2, node N-1 to node 0 and node 0 to node N-1.

STAR - A center node is specified and that center node is connected to every other node. The other nodes are connected to the center.

GENERAL - A network which is not one of the above, rather the user adds links to a network as needed until the desired topology is created. When using this topology, the user specifies pairs of nodes which will be connected with links from one to the other. Links may also be dropped from a network while using this topology.

When specifying the network's topology the user should remember that after completing a topology, he may alter it. An example would be that the user wishes to set up a backbone topology of 30 nodes. To efficiently do this, the user would first set up a ring topology of 30 nodes and then break the ring to form the backbone topology. Other general topologies may be created in the same manner by altering one of the topologies rather than entering all the links in the GENERAL mode.

In every case, the program will display the topology type and number of nodes, an adjacency matrix that shows the links, a translated form of the adjacency matrix, and a query to the user asking if he is satisfied with the topology.

A star topology requires that the center of the star be entered. The center may be entered by either placing the center's identification number on the line, or waiting to be queried for it after entering "star".

A general topology allows the user to add or delete links in the network. The general mode differs in that after entering the links, the program checks the topology to ascertain that the network is connected. Connectedness is determined by performing a depth first search of the nodes beginning at node 0. If a node is found that is not connected to the remaining nodes, the program displays the disconnected nodes. This check for connectedness is also performed if the user alters the topology.

While entering the link specifications, a variety of errors may occur. Each error message explains the situation and instructs the user on how to rectify the erroneous response.

6.1.1.2 Node Code

The node code specifications allow the user to indicate which UNIX files contain the node code that will be used at the individual nodes. These specifications consist of the node code file name and the associated node numbers for each node in the network. The node code files for all nodes are displayed to the user who may change them if desired.

The keyword "all" may be used if one node code file is to be used at all the nodes.

Each node code file name is tested to ascertain that it is valid. A valid file name is 14 or less characters in length, ends with ".c" and is composed of only valid characters. Valid characters are either upper or lower case letters, the digits and the period. If these syntax rules are violated an error message will be displayed.

After a node code file name has been accepted as being syntactically correct, a check is made to determine if the file exists in the user's current directory. A file's existence is assumed if it can be opened. If the file cannot be closed after being opened, a fatal error message will be displayed and the specifications program will terminate.

If any line of the node code specifications contain a syntax error or an invalid node number, an appropriate error message will be displayed.

Since each node must have a node code file, the program checks to make sure that the user has in fact specified a file

for all nodes in the network. If any nodes were forgotten, a message will inform the user of the oversight.

Entry of the node code specifications for networks with many nodes which use the same node code may be simplified by entering the predominant node code file name with the keyword "all". Other file names are then entered for the few nodes at which they are needed.

6.1.1.3 Data File Names

The data file name specifications are the UNIX file names of all data files to be used in the network. These file names are needed only if data files are used in the network. Data file names and data file usage specifications are used by the simulator to place the data files at the nodes where they are to be used.

Each node reads the contents of its data files into data structures local to the node. The node's database utility may then manipulate the contents of the data structures when called by the user's node code.

All file names are displayed back to the user. As always, the user may change the data file name specifications by repeating the process.

Each data file name is checked for syntactic validity after it is entered. A valid data file name is 14 or less characters in length, and is made up of valid characters only. A valid character is either an upper or lower case letter, one of the

digits, or the period. If a file name violates one of the syntax rules, the user will be notified.

A data file must exist in the user's current directory before it can be used by the network simulator. The existence of a file is determined by the ability to open it. If a file name is entered that is syntactically correct, but the file cannot be opened, a message will be displayed.

Any data file which was opened but cannot be closed will cause termination of the specification program.

Since the data file names are entered for all data files in the network, the program checks for duplicates. If a duplicate file name is inadvertantly entered, a message is displayed.

The number of data file names that may be entered is limited (See Section 3). If the user enters more data file names than are allowed, a message will be displayed and the extra file names will be ignored by the program.

Some networks which will be studied do not require data files. In this situation the user simply enters a null line. After the specification program is informed that data files are not needed, it will not query the user for data file usage specifications (Section 6.1.1.4) or data file format specifications (Section 6.1.1.5).

6.1.1.4 Data File Usage

Network simulations which use data files require that the user declare which files are used at each individual node. The data file usage specifications allow the user to do this.

A user will be queried with a single data file name and asked to enter the node numbers of all nodes at which the data file is to be used. This process is repeated for all data files declared in the data file name specifications.

As many node numbers may be entered as needed when responding to the query. The keyword "all" may be used if a data file is needed at all the nodes. A null line is used to terminate entry of the node numbers.

After completing entry of the data file usage specifications, they are displayed to the user for inspection. The user is given an opportunity to change the specifications if they are not satisfactory.

6.1.1.5 Data File Formats

Data file formats are used in conjunction with the data file name and usage specifications to load data files into databases at nodes. A format is a description of a logical record in a data file. Two types of data may be used, integer and character strings.

Assume that a data file contains logical records with the following fields.

Field	Columns	Data Type
1	1-5	Integer
2	6-15	Character
3	16 20	Character
4	21-25	Character
5	26-28	Integer

The format specification for the data file would be,

i 5 c 10 c 5 c 5 i 3

Entry of the formats is done in the same manner as entry of the usage specifications. The user is queried for the format of each file. After entering a format for all files, they are displayed back to the user for inspection. If any changes are needed, the user is given an opportunity to do so.

6.1.1.6 Number of Messages

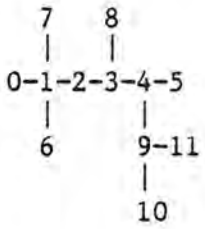
The duration of a simulation is controlled by the number of inter-node messages that are transmitted by all nodes. A user is queried to enter the number of messages to be used by the simulator. The maximum value that may be entered is 32767.

As always, the number of messages will be displayed back to the user for inspection.

6.1.1.7 An Example

This section demonstrates entry of the specifications for a network. The network topology is shown in Figure 6.1.

Figure 6.1



Nodes 0 through 5 use node code file nc1.c, nodes 6, 9 and 11 use nc2.c, and nodes 7, 8 and 10 use nc3.c.

There are 3 data files used. Data file df1 is used at all nodes, df2 is used at nodes 0 through 5 and df3 is used at node 10. Data file formats of the 3 files are shown below.

```
File
df1   i 2 c 5
df2   c 10 c 6 c 8
df3   i 1 i 3 c 5 i 2 i 1
```

The number of messages sent between nodes during the simulation will be set at 30000.

The terminal session in which these specifications are entered is shown below.

```
Welcome to the Network Simulator.

   This program is the initial step in using the
network simulation package on the PDP 11.


You may set up and/or use a network
simulation specification file by entering one
of these modes:

  new
  old
  change

new

A network may contain 2 - 45 nodes.
How many nodes are needed in your network?

12

There will be 12 nodes in the network.
They will be numbered 0 - 11.


Please enter the topology specifications.
The topologies which are available are:

   complete
   ring
   star
   general

What is the topology of your network?

general

The topology will be a general network of 12 nodes.
You may now specify the links in the network.

Please use a carriage return on a new line to
terminate input of the link specifications.

1 -> 0 2 6 7
2 -> 3
3 -> 4 8
4 -> 3 5 9
9 -> 10 11
```

```
  This is the adjacency matrix which
represents the topology of your network.

     To
               11
From  012345678901
   0  010000000000
   1  101000110000
   2  010100000000
   3  001010001000
   4  000101000100
   5  000010000000
   6  010000000000
   7  010000000000
   8  000100000000
   9  000010000011
  10  000000000100
  11  000000000100

  The adjacency matrix has been translated
into a more readable form.

  The links in your network are:

From  To
   0  1,
   1  0, 2, 6, 7,
   2  1, 3,
   3  2, 4, 8,
   4  3, 5, 9,
   5  4,
   6  1,
   7  1,
   8  3,
   9  4, 10, 11,
  10  9,
  11  9,

The topology specifications have been completed.
Are you satisfied with the current
specifications?  Enter "yes" or "no".

yes

Please enter the node code specifications.

Please use a carriage return on a new line to
terminate input of the node code specifications.

nc1.c all
nc2.c 6 9 11
nc3.c 7 8 10
```

```
The node code file for each node is:

At node 0, nc1.c
At node 1, nc1.c
At node 2, nc1.c
At node 3, nc1.c
At node 4, nc1.c
At node 5, nc1.c
At node 6, nc2.c
At node 7, nc3.c
At node 8, nc3.c
At node 9, nc2.c
At node 10, nc3.c
At node 11, nc2.c


The node code specifications have been completed.
Are you satisfied with the current
specifications?  Enter "yes" or "no".

yes

Please enter the data file names of all files
needed in your network.  You may enter
up to 45 data file names, if more are
entered they will be ignored.

Please use a carriage return on a new line to
terminate input of the data file name specifications.

df1
df2
df3

These are the data files which are needed in your network.

  df1
  df2
  df3


The data file name specifications have been completed.
Are you satisfied with the current
specifications?  Enter "yes" or "no".

yes

Please enter the data file usage specifications.

Please use a carriage return on a new line to
terminate input of the data file usage specifications.
```

```
At which nodes is data file df1 needed?

all

At which nodes is data file df2 needed?

0 1 2 3 4 5

At which nodes is data file df3 needed?

10

Data file df1 will be used at nodes:

  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,

Data file df2 will be used at nodes:

  0, 1, 2, 3, 4, 5,

Data file df3 will be used at nodes:

  10,

The data file usage specifications have been completed.
Are you satisfied with the current
specifications?  Enter "yes" or "no".

yes

Each data file may contain up to 10 fields.
If more than this are entered, they will be ignored.


Please enter the format of the df1 data file.

i 2 c 5

Please enter the format of the df2 data file.

c 10 c 6 c 8

Please enter the format of the df3 data file.

i 1 i 3 c 5 i 2 i 1

The formats of the data files are:


df1  i 2  c 5

df2  c 10  c 6  c 8

df3  i 1  i 3  c 5  i 2  i 1
```

```
The data file formats specifications have been completed.
Are you satisfied with the current
specifications?  Enter "yes" or "no".

yes

How many inter-node messages
are to be sent during the simulation?

30000

There will be 30000 messages sent.


The number of inter-node messages specifications have been completed.
Are you satisfied with the current
specifications?  Enter "yes" or "no".

yes
```

6.1.2 CHANGE Mode

When using the CHANGE mode, specifications which were entered during a previous terminal session may be altered. A user is given an opportunity to change any of the specifications. The program queries the user to determine if any change is needed in each type of specification. If the user responds with an affirmative, the user is allowed to enter new specifications. A negative response from the user leaves the specifications in their original form.

Entry of changes to a particular specification is done in the same manner as NEW specifications. As with NEW specifications, if no data files are used the program will not query the user for data file usage or format specification changes.

6.1.2.1 An Example

An example of using the CHANGE mode is given in this section. The network which was described in section 6.1.1.7 will be changed in the following ways: node 7 will now be linked to node 0 rather than node 1, and only 20000 messages will be sent during the simulation.

```
Welcome to the Network Simulator.

   This program is the initial step in using the
network simulation package on the PDP 11.


You may set up and/or use a network
simulation specification file by entering one
of these modes:

  new
  old
  change

change

Do you need to change the topology specifications?

yes

A network may contain 2 - 45 nodes.
How many nodes are needed in your network?

12

There will be 12 nodes in the network.
They will be numbered 0 - 11.


Please enter the topology specifications.
The topologies which are available are:

   complete
   ring
   star
   general

What is the topology of your network?

general
```

```
The topology will be a general network of 12 nodes.
You may now specify the links in the network.

Please use a carriage return on a new line to
terminate input of the link specifications.

1 -> 0 2 6
2 -> 3
3 -> 4 8
4 -> 3 5 9
9 -> 10 11
7 -> 0


  This is the adjacency matrix which
represents the topology of your network.

      To
                11
From  012345678901
   0  010000010000
   1  101000100000
   2  010100000000
   3  001010001000
   4  000101000100
   5  000010000000
   6  010000000000
   7  100000000000
   8  000100000000
   9  000010000011
  10  000000000100
  11  000000000100

  The adjacency matrix has been translated
into a more readable form.

  The links in your network are:

From  To
   0  1, 7,
   1  0, 2, 6,
   2  1, 3,
   3  2, 4, 8,
   4  3, 5, 9,
   5  4,
   6  1,
   7  0,
   8  3,
   9  4, 10, 11,
  10  9,
  11  9,

The topology specifications have been completed.
Are you satisfied with the current
specifications?  Enter "yes" or "no".
```

```
yes

Do you need to change the node code specifications?

no

Do you need to change the data file name specifications?

no

Do you need to change the data file usage specifications?

no

Do you need to change the data file formats specifications?

no

Do you need to change the number of inter-node messages specifications?

yes

How many inter-node messages
are to be sent during the simulation?

20000

There will be 20000 messages sent.

The number of inter-node messages specifications have been completed.
Are you satisfied with the current
specifications?  Enter "yes" or "no".

yes
```

6.1.3 OLD Mode

The OLD mode is used to check for existence of the specifications file prior to a simulation. The file's existence is determined by opening it, and if it cannot be opened by the program, the program displays an error message and terminates.

7. Compilation

The compilation program causes the node code programs to be compiled and saves the resulting core image files on reserved files.

The network specifications file, net.spec, is used by the compilation program to obtain the node code file names. Each node code file is compiled using the Version 6 C compiler. After each successful compilation, a message is printed to inform the user. If a compilation error occurs, the user is queried to determine if a listing of the node code program and compilation error messages is to be saved for diagnostic purposes. If the user responds affirmatively to the query, the compilation program creates a file named nodecode.e which contains the listing and error messages. This listing file will have the same name and the node code file except ".e" is appended to the file name rather than ".c".

An example terminal session in which the compilation program is executed is shown below.

```
% comp
The node code for node 0 has been compiled.
A compilation error occurred while compiling
the node code for node 1.

Do you want a source code listing with error messages?
Enter "yes" if you do.

yes
%
```

8. Simulation

The actual simulation is initiated by the simulation driver program. This program causes a separate UNIX process to be created for each node in the network and for the message queuing process. The simulation driver also establishes the pipes which are used by the message queuing process and the node processes to transmit inter-node messages.

Each node's core image file which was created by compiling the node code program is loaded into a UNIX process and allowed to initialize itself. The message queuing process is also loaded and initialized. The simulation driver then broadcasts a signal to all of these processes to begin the simulation. After starting the simulation, the driver waits for the message queuing process to terminate which means that the simulation is complete (for normal or abnormal reasons). Temporary files used during the simulation are then unlinked by the driver and the UNIX processes are killed.

The simulation driver program may be executed as a background job or as a detached job. Either of these techniques free the user to do other tasks at the terminal. Detached jobs allow the user to log off of UNIX while a simulation proceeds. In either background of detached jobs, the user should redirect any output from the simulation driver to a file. This allows any error messages to be saved on the file. The two commands shown below cause the simulation driver to execute while saving any output.

As a background job:

```
sim > temp&
```

As a detached job:

```
nohup sim > temp&
```

Any error which the simulation driver detects will cause a self-explanatory error message to be printed. Most of these errors are caused by lack of the proper files in the user's directory when attempting to run a simulation (See Section 4).

9. Report

During each simulation a history file will be created by the message queuing process. The transactions which took place between nodes during the simulation are recorded in the file. The report program prints a list of these transactions.

The report program listing does not print the actual messages which are also saved in the history file. Messages are not printed because of the wide variety of message formats that different users might use. These messages may be obtained by using the "od" utility which is supported by UNIX (see od(I) in the UNIX Programmer's Manual).

A user should always review the report listing because non-fatal errors which occurred at nodes during the simulation are included in the listing. The errors are identified by numbers which are explained below.

Error	Cause
1	Communication utility was called with an invalid value for code parameter. User Action: Correct the parameter value.
2	Communication utility could not place a request to receive a message in the RIN pipe. User Action: Refer the problem to the UNIX system maintenance person.
3	Message received at a node does not contain the correct number of bytes as determined by the message header. User Action: Refer the problem to the UNIX system maintenance person.
4	Message was sent to the wrong node. User Action: Refer the problem to the UNIX system maintenance person.
5	An attempt was made to send a message which was too large to fit into a UNIX pipe. User Action: If the larger message size is crucial to the simulation at hand, the UNIX kernel code can be recompiled with a larger pipe size. This is not recommended, however. A more reasonable approach would be to split messages into packets.
6	A request to send a message cannot be placed in the RIN pipe. User Action: Refer the problem to the UNIX system maintenance person.
7	A message cannot be placed in the MIN pipe. User Action: Refer the problem to the UNIX system maintenance person.
8	A idle message cannot be placed into the RIN pipe. User Action: Refer the problem to the UNIX system maintenance person.

9 A node attempted to send a message to another node via a non-existent link.

User Action: Correct the node code or change the network specifications.

The message queuing process can also generate error messages which will appear in the report listing. These errors indicate failure of the message queuing process to perform correctly, and should be reported to the UNIX system maintenance person. A list of the error numbers and their causes is shown below.

Error	Cause
1	A request read from the RIN pipe caused an I/O error.
2	An invalid request message identifier appeared on a request message.
3	The history file could not be closed.
4	A seek operation on the history file to position the file pointer at EOF failed.
5	An I/O error occurred while writing something onto the history file.
6	A seek operation on the history file to position the file pointer at a message failed.
7	An I/O error occurred while reading a message from the history file.
8	An I/O error occurred while reading a message from the MIN pipe.
9	Not used.
10	The free list of message queue elements was exhausted. This error may be corrected by increasing the LISTLENGTH defined constant, in net.const.c, and recompiling the message queuing program.

The following listing is an excerpt from a report listing.

```
The network contains 7 nodes.
The nodes were numbered from 0 to 6.
The network traffic during the simulation
was 6000 inter-node messages.

Node 1 requested a message
Node 2 requested a message
Node 3 requested a message
Node 4 requested a message
Node 5 requested a message
Transmit 12 bytes from node 6 to node 0
Node 0 requested a message
Node 6 requested a message
Transmit 12 bytes from node 0 to node 1
Transmit 12 bytes from node 1 to node 2
Node 1 requested a message
Transmit 12 bytes from node 2 to node 3
Node 2 requested a message
Node 0 requested a message
Transmit 12 bytes from node 3 to node 4
Node 3 requested a message
Transmit 12 bytes from node 4 to node 5
Node 4 requested a message
Transmit 12 bytes from node 5 to node 6
Node 5 requested a message
Transmit 12 bytes from node 6 to node 0

  .
  .
  .

Network simulation errors = 0
Messages transmitted = 6000
Messages requested = 6006
Idle nodes = 0
```

10. Checkup

A user may use the checkup program to determine how a simulation is progressing. This is particularly useful when a simulation is being run as a background or detached job.

The checkup program interrupts the message queuing process to obtain the number of messages left to be transmitted before

the simulation is completed. This number of messages is reported to the user's terminal. The checkup program repeatedly performs this process until the user indicates that no further checkups are needed.

The example below shows a typical terminal session in which the checkup program is used.

```
% simcheck
5999 messages to go.

Again?
yes
5997 messages to go.

Again?
no
%
```

A. Query Response Syntax

The user must use the proper syntax while responding to queries from the program that collects the network's specifications. The syntax rules that govern the responses are expressed in a modified Backus-Naur Form (MBNF) for the user's convenience.

MBNF Convention	Interpretation
|	Alternation
" "	Terminal symbol
{ }	Optional, unlimited repetition
{ }n	Optional repetition up to n times
n{ }	Repetition at least n times
n{ }m	Repetition at least n times

	but not more than m times
[ ]	Optional inclusion
A // B	Concatenation of A to B
( )	Grouping
A <> B	A may not be equal to B
...	Intervening characters or symbols are allowed

The syntax rules show that terminal symbols contain only lower case letters. This was done for brevity although either lower or upper case letters are permitted. Terminal symbols are separated by at least one space unless concatenation is indicated.

Mode query response:

```
modes := "new" | "old" | "change"
```

Topology query response:

```
numberofnodes := validnode
validnode := "2" | "3" | "4" | ... | "43" | "44" | "45"
topologies := "complete" | "ring" | star | "general"
star := ("star" validnode) | twolinestar
twolinestar := starlinel starline2
starlinel := "star"
starline2 := validnode
linkspecs := 1{ addlink | droplink } nulline
addlink := ["+"] originatingnode ["->"]
           ( 1{ terminalnode } | "all" )
```

```
        droplink := "-" originatingnode ["->"]
                    ( 1{ terminalnode } | "all" )
        originatingnode := validnode
        terminalnode := validnode <> originatingnode
        nulline := "carriage return CR on a new line"
```

Completion or change query response:

```
        completeorchange := "yes" | "no"
```

Node code query response:

```
        nodecodespecs := 1{ nodeline } nulline
        nodeline := nodefile ( 1{ validnode } | "all" )
        nodefile := character  { character }11 // ".c"
        character := "a" ... "z" | "A" ... "Z" |
                     "0" ... "9" | "."
```

Data file names query response:

```
        datafilenamespecs := { datafilename }45 nulline
        datafilename := character  { character }13
```

Data file usage query response:

```
        usagespecs := ( "all" | 1{ validnode } ) nulline
```

Data file format query response:

```
        datafileformat := 1{ fieldformat }10
        fieldformat := ( "i" "1" ... "5" ) |
                       ( "c" "1" ... "51" )
```

Number of messages response:

```
messages := { "1" | "2" | ... | "9" | "0"}
```