OREGON STATE
UNIVERSITY
COMPUTER
SCIENCE
DEPARTMENT

# VNS - A Virtual Network Simulator

Walter F. Domka

Michael J. Freiling

Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

83-60-3

# VNS -- A Virtual Network Simulator

by

Walter F. Domka and Michael J. Freiling

Computer Science Department
Oregon State University

## ABSTRACT

A software system to study network algorithms was implemented on UNIX*. Each part of a network algorithm can be 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** 11/44. 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. Run-time support routines include communication facilities and a mechanism for recording communication histories.

---------------

* UNIX is a Trademark of Bell Laboratories.

** PDP is a Trademark of Digital Equipment Corporation.

# VNS -- A Virtual Network Simulator

by

Walter F. Domka and Michael J. Freiling

Computer Science Department
Oregon State University

## INTRODUCTION:

Efficient design and implementation of distributed computer systems require a means of ascertaining the correctness of software which governs interaction between computers in the network. Testing such software requires the use of a special purpose simulator program, an existing computer network or a testbed. The concept behind VNS is to allow network algorithms to be rapidly prototyped in a normal program development environment, without requiring special simulator languages or hardware support.

VNS provides a user with a core of support programs which perform the necessary pre-simulation network specification, post-simulation analysis of the inter-node message transactions, compilation of the C programs which make up the nodes of the network, and a means of checking the progress of a simulation while it is executing. Also included in VNS are two programs and several functions which are transparent to the user. The first transparent program, called the Simulation Driver Program, supervises execution of a simulation by creating node processes, initializing node processes, establishing the pipes used in a simulation, starting a simulation, and terminating a simulation. The second transparent program, called the Message Queuing Process, provides run-time support for communication between node processes by running as a separate process in the UNIX environment. During a simulation the Message Queuing Process queues inter-node messages and governs use of the inter-process pipes. The transparent functions, called utilities, interface the network nodes' programs with the two transparent programs.

Since the support programs, transparent programs and the utilities are all composed of executable code, VNS does not expend a great amount of "system overhead" in carrying out a simulation. The duration of a simulation is determined by network size, number of other users on the UNIX system, and computations unique to the simulation at hand.

## SYSTEM ARCHITECTURE:

VNS consists of seven programs and functions: Specifications Program, Utilities, Compilation Program, Simulation Driver Program, Message Queuing Process, Checkup Program, and a Report Program. Each of these components is described below.

## SPECIFICATIONS PROGRAM:

The Specifications Program is an interactive program with which the user selects specifications for the network. These specifications are then saved by the Specifications Program for use by other components of VNS. Specifications may be reused or altered for subsequent simulations. Specifications which are used in the present version of VNS are:

1. Number of nodes in the network,
2. Topology of the network,
3. Node code file names, and
4. Number of inter-node messages.

The number of nodes allowed in a network is limited only by the number of processes which may be present in the UNIX operating system. A tunable parameter is used to set the number of nodes allowed in the simulator. Forty-five nodes are allowed in the present version.

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 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 for node identification by VNS during a simulation.

A network's topology is set during entry of the specifications. Topologies are described in terms of the nodes which are connected by links. Since each link is bidirectional, a link from node i to node j implies that node j may transmit messages to node i. 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. A message from node i to node j is valid only if there exists a virtual link between the nodes as described during entry of the network specifications*. The Specifications Program allows selection of standard topologies such as ring, star or complete, and more general topologies may be established by specifying individual links or by altering one of the standard topologies.

Programs which run at the nodes are called node code. Node code file names are used by the Compilation Program to compile the node code programs and store the resulting core images on files which have predetermined names. Predetermined names are used to allow the Simulation Driver Program to locate the core images when starting a simulation.

The number of inter-node messages transmitted during a simulation is used to control the duration of the simulation. Each inter-node message delivered by the Message Queuing Process is counted until the number of messages delivered equals the limit specified at invocation.

## UTILITIES:

Utilities are functions which provide special services to a node during a simulation and are called from within the node's program. These functions interface node programs with the message queuing program and the simulation driver.

Utilities are included with node code by using the C compiler's preprocessor. This relieves the user of tedious details involving the utilities' source code which must be present with the node code at compilation time.

Currently there are two utilities available: a communication utility and a random number generator. The communication utility is used by every node as it provides the mechanism for inter-node communication. The random number utility generates ranged, random integer values in the range 0 - ((2^15)-1).

-----------

* Inter-node messages sent between nodes which are not connected by a link must be handled by routing algorithms implemented in the node code.

Additional utilities which perform other specialized tasks may be written by a user. These utilities are easily incorporated into node code and do not require any modification to other VNS components.

COMPILATION PROGRAM:

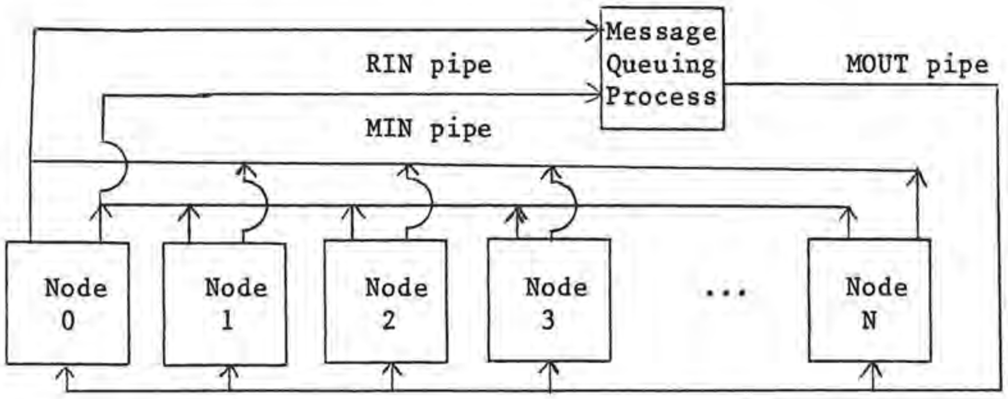
This program uses the node code file names which were entered by the user as part of the specifications to compile the node code. If an error occurs during compilation of a node's code, the user is notified and given an opportunity to obtain a listing of the node code with error messages.

SIMULATION DRIVER PROGRAM:

The Simulation Driver Program 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 Program broadcasts a signal to each node process and the Message Queuing Process to commence the simulation. Three inter-process pipes are also established by the Simulation Driver Program for use by the node processes and the Message Queuing Process. These pipes are used by the processes to transmit inter-node messages via the Message Queuing Process. Pipes are limited to 4096 bytes in size, which in turn limits the maximum message size. Pipe size is also tunable by changing a defined constant within the UNIX operating system source code.

Each of the three pipes is used for a separate purpose. 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. Organization of the pipes and processes is diagramed in Figure 1.

Figure 1.



After the Simulation Driver Program initiates a simulation it waits for the Message Queuing Process to terminate, signaling completion of the simulation. When the simulation is completed the Simulation Driver Program removes unneeded temporary files.

MESSAGE QUEUING PROCESS:

The Message Queuing Process performs four functions during a simulation: 1) queuing of messages which have been transmitted but have not been "delivered" to the receiver node, 2) acting as a monitor to synchronize use of the MIN and MOUT pipes by the node processes, 3) controlling the duration of a simulation, and 4) trapping each inter-node message and creating a file of the same for post-simulation analysis. This file of messages is called a Network History File.

An inter-node message transmission is performed by the transmitting node, the Message Queuing Process and the receiver node as follows.

1. When the transmitting node code calls its communication utility to transmit, the utility sends a request-to-transmit to the Message Queuing Process via the RIN pipe (Figure 2, step 1). The transmitting 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 to the Message Queuing Process.
2. The Message Queuing Process signals the transmitting node (Figure 2, step 2) when the request-to-transmit has been acknowledged.
3. The transmitting node locks the MIN pipe and writes the message into the MIN pipe. When the lock is removed by the transmitting node, the Message Queuing Process reads the message from the pipe
4. After reading the message, the Message Queuing Process writes it onto the Network History File which is a random access file. A pointer to the message's location in the file is saved by the Message Queuing Process and placed on a queue of messages destined for the receiver node (Figure 2, step 4).
5. When the receiver node code calls its communication utility to obtain the next message a request-to-receive is sent to the Message Queuing Process by the communication utility via the RIN pipe (Figure 3, step 1). The receiver node is then blocked.
6. After processing the request-to-receive, the Message Queuing Process refers to the receiver's message queue and uses the pointer to locate the message. The Message Queuing Process reads the message
7. The Message Queuing Process places the message into the MOUT pipe and then signals the receiver node to begin reading the message (Figure 3, step 4). The Message Queuing Process is then blocked.
8. When the receiver node completes reading the message, a signal is sent to the Message Queuing Process by the receiver node (Figure 3, step 5).

Figure 2.

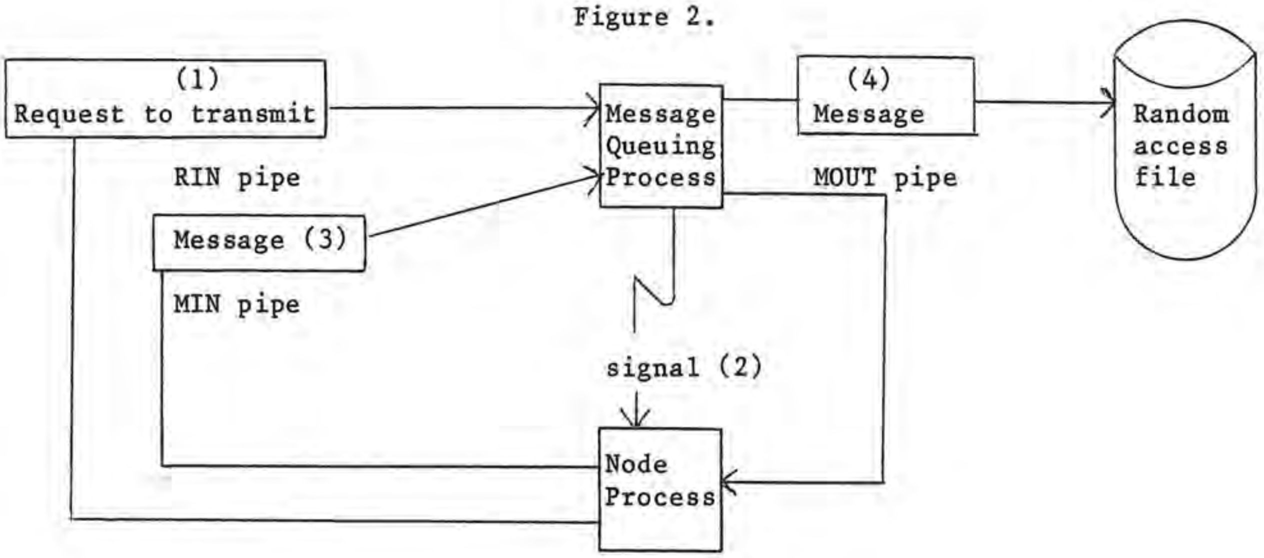
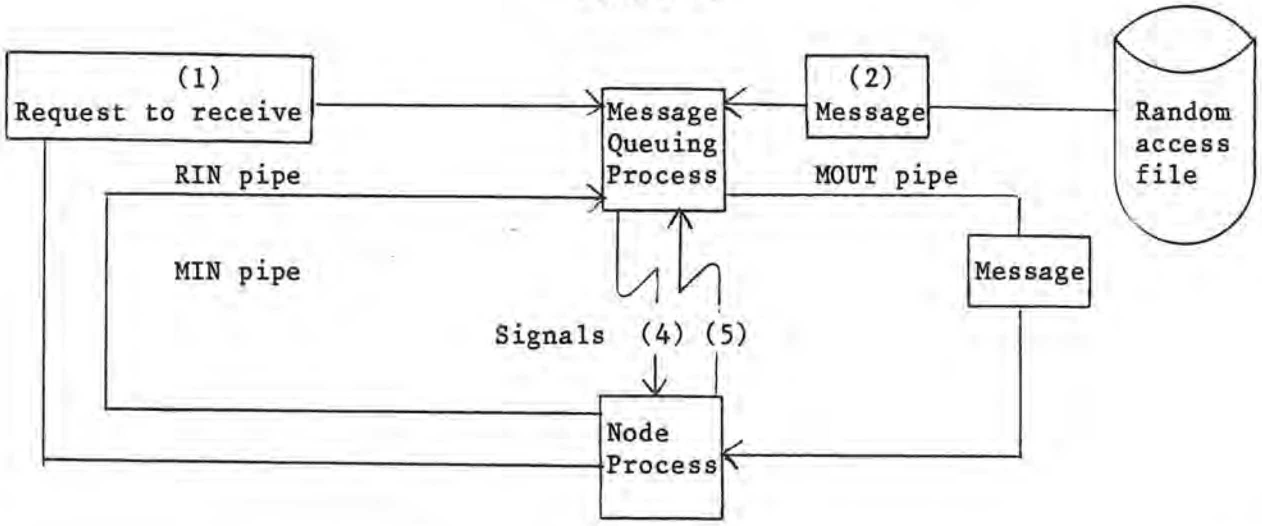


Figure 3.



Pipes are in-memory buffers which the Message Queuing Process and node process have read/write access to. 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 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. Node processes wait their turn to use a pipe until receiving a signal from the Message Queuing Process, after which the node process locks the pipe. The recipient process is awaiting the signal, so it becomes an "acknowledged/proceed" message.

Mutually exclusive locking of a pipe is accomplished with the UNIX system call, "link", by linking a temporary file to a unique file name. A signal is a standard UNIX inter-process communication feature which allows one process to interrupt another.

Requests for messages are noted by the message queuing process upon receipt of them. When more than one node has requested a message and messages are enqueued for the nodes, the Message Queuing Process must act as an arbiter in order to decide which node will be serviced. The algorithm used by the

Message Queuing Process is a round-robin in which each node is given its turn. The round-robin algorithm is designed to be fair and in the worst case a node would have to wait N-1 turns before receiving its message. Use of a round-robin also produces the most even distribution of processor time among message receiving nodes.

Ideally, the node processes and the Message Queuing Process would alternate in being scheduled to run by the operating system. Since there is contention between processes for central processor time, the order in which the simulation's processes are scheduled is non-deterministic. Use of signals and locks by the processes prevents scheduling from affecting the integrity of messages. Synchronization overhead does increase the elapsed time of a simulation, but does not necessitate any modification of the UNIX process scheduling algorithm. This permits VNS to run on a standard UNIX, in the presence of other users.

## CHECKUP PROGRAM:

A simulation may require considerable elapsed time, particularly if several users are on the UNIX system while a simulation is running. UNIX allows programs to be run in the background or completely detached from an active user. Background jobs permit the 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 while a simulation runs. Simulations which run as background or detached jobs do not give the user any means of determining how the simulation is progressing. 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 yet to be transmitted. Communication between the Message Queuing Process and the Checkup Program is effected by use of a temporary file which is created by the Message Queuing Process during initialization. The file contains the process identification number of the Message Queuing Process which is read by the Checkup Program. The Checkup Program writes its own process identification number onto the file before sending a signal to the Message Queuing Process using the Message Queuing Process's 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 Program's process identification number from the file and then rewrites the file with the Message Queuing Process's process identification number and the number of inter-node messages left. The Checkup Program is then signaled. Upon receipt of the signal, the Checkup Program reads the file and prints the number of messages left to be transmitted on the user's terminal.

## REPORT PROGRAM:

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 the Network History File. Each inter-node message transmitted from a node to the

Message Queuing Process is noted by the transmitting node´s number, the receiver node´s number and the number of bytes in the message. A request for a message is noted by the requesting node´s number. If an error message is present, it is listed with the node 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 in the Network History File.

Information which is pertinent to a simulation may also be output by a node process to a file. A user may exploit this technique to trap information which is not saved in the Network History File.

## LEVELS OF SIMULATION:

Topologies which require specialized components, for instance a complete topology implemented via a bus, may be simulated by including network nodes for the specialized components. A bus topology would be implemented with a star where the central node contained program code to simulate the arbitration and broadcast capabilities of the bus under consideration.

VNS permits simulation of networks at several levels of detail. At the highest level a bus architecture appears as a complete network (Figure 4.1). The existence of the bus itself can be modeled via a star as mentioned above, (Figure 4.2) and another level of detail can be reached where the bus interfaces themselves from part of the network (Figure 4.3).

Figure 4.1

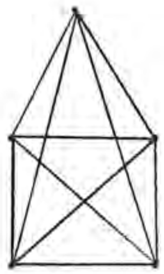


Figure 4.2

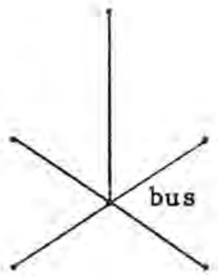
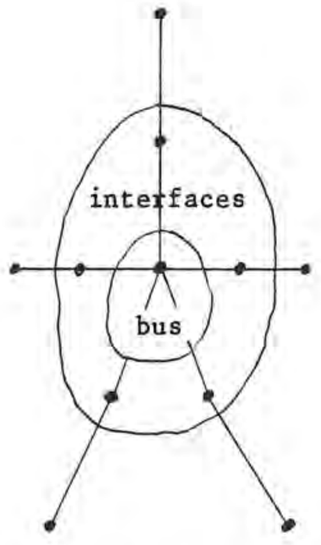


Figure 4.3



EXAMPLE SIMULATION:

A small simulation is given in this section to demonstrate how a simulation is carried out using VNS. The network to be simulated is a star network of seven nodes and node 6 is the center of the star. The task to be simulated is to send a message from each node on the periphery to every other peripheral node. When each peripheral node has received a message from all other peripheral nodes it will terminate processing of messages. The task of the center node is to forward each message received to its destination node.

The message transmitted by each node will consist of two fields, a sixteen bit integer value which identifies the destination node and a sixteen bit integer value which identifies the transmitting node (Figure 5). This small message format is used for rapid prototyping only, a more elaborate scheme could easily be implemented which would use more fields and have headers and trailers to form a packet.

Figure 5.

```
Inter-node Message Format:

 0           15 0           15
---------------------------------
| Destination  | Transmitter   |
| (16 bits)    | (16 bits)     |
---------------------------------
```

Synthesization of the node code programs involves writing the following two C programs and editing them into files which are named center.c and peripheral.c for the center and peripheral nodes respectively.

```
******************************************************************************
# define FOREVER 1
char buffer [4];           /* Message buffer */
int to;
# include "cm.gvar"        /* Include data structures used by communication */

main () {      /* Main program for center node */
 nodeinit();
 while (FOREVER) {         /* Loop continuously, */
   receive();              /* receive a message and */
   send();                 /* send it to its destination */
   }
 }

# include "cm.c"           /* Include the communication utility */

receive () {
 cm(1, 0, buffer, 0);     /* Request next message */
 to = _ctoi(buffer);      /* Unpack destination node id */
 }

send () {
 cm(2, to, buffer, 4);    /* Send message to destination */
 }
```

```
**************************************************************************
# define NODES 7
# define CENTER 6
# define TRUE 1
# define FALSE 0
char buffer [4];          /* Message buffer */
int LOCAL;                /* Local node id */
int received [NODES-1]; /* Boolean array to keep track of received messages */

# include "cm.gvar"       /* Include data structures for communication util. */

main () {    /* Main program for peripheral nodes */
 int i, j;
  nodeinit();             /* Initialize from simulation driver */
  initialize();           /* Initialize local data structures */
  for (i=0; i < NODES; ++i) {  /* For every other peripheral node, */
    if ((i != LOCAL) && (i != CENTER)) {  /* form and send message. */
    formmessage(i);
    send(i);
    }
  while (!(i = receive()))  /* Receive messages from other nodes */
    ;
  }

# include "cm.c"    /* Include communication utility */

initialize () {
 int i;
  LOCAL = _ln();    /* Obtain local node id */
  for (i=0; i < NODES; ++i)    /* Set received to false except */
    received [i] = FALSE;      /* for center and local node */
  received [LOCAL] = TRUE;
  received [CENTER] = TRUE;
  }

formmessage (i)
 int i;
 {
  _itoc(i, &buffer[0]);    /* Pack destination node id */
  _itoc(LOCAL, &buffer[2]); /* and local node id into message */
 }

send (i)
 int i;
 {
  cm(2, CENTER, buffer, 4);    /* Request to send the message */
 }
```

```
receive () {
 int from, finished, i;
  cm(1, 0, buffer, 0);          /* Request next message */
  from = _ctoi(&buffer[2]);     /* Unpack node id of sender */
  received [from] = TRUE;       /* Check off sender */
  finished = TRUE;              /* Find out if all nodes have */
  for (i=0; i < NODES; ++i) { /* been heard from */
    if (! received[i]) finished = FALSE;
    }
  return(finished);
  }
************************************************************************
```

After debugging the programs, the user is ready to select the network specifications. The following terminal session would establish the necessary specifications.

```
************************************************************************
% 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?

star

What is the center node of the star?

6

The topology will be a star network of 7 nodes
and node 6 will be the center.


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

      To
From  0123456
   0  0000001
   1  0000001
   2  0000001
   3  0000001
   4  0000001
   5  0000001
   6  1111110

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

  The links in your network are:

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

center.c 6
peripheral.c 0 1 2 3 4 5


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

```
At node 0, peripheral.c
At node 1, peripheral.c
At node 2, peripheral.c
At node 3, peripheral.c
At node 4, peripheral.c
At node 5, peripheral.c
At node 6, center.c


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

yes

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

60

There will be 60 messages sent.


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

yes
*************************************************************************
```

Compilation of the nodes´ programs is accomplished with the Compilation Program as shown below.

```
*************************************************************************
% 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.
*************************************************************************
```

At this point the user is ready to execute the simulation. The following example shows how the simulation is executed as a background job, and how the Checkup Program is utilized to determine how the simulation is progressing.

```
*************************************************************************
% sim > simout&
% checkup
58 messages to go.

Again?
yes
57 messages to go.

Again?
no
*************************************************************************
```

A portion of the report produced by the Report Program is given below.

```
*************************************************************************
% netprt


The network contains 7 nodes.
The nodes are numbered from 0 to 6.
The network traffic during the simulation
was 60 inter-node messages.

Transmit 8 bytes from node 0 to node 6
Transmit 8 bytes from node 0 to node 6
Transmit 8 bytes from node 0 to node 6
Node 6 requested a message
Transmit 8 bytes from node 1 to node 6
Transmit 8 bytes from node 1 to node 6

   .
   .
   .

Transmit 8 bytes from node 6 to node 0
Transmit 8 bytes from node 6 to node 1
Transmit 8 bytes from node 6 to node 2
Node 3 requested a message

   .
   .
   .

Network simulation errors = 0
Messages transmitted = 60
Messages requested = 61
Idle nodes = 0
*************************************************************************
```

SUMMARY:

A software system has been presented which promotes study of networks. The salient features of this system are:

1. Rapid prototyping of network algorithms is possible.
2. Network algorithms are developed and tested in a normal program development environment.
3. The system does not require the overhead effort of more traditional simulation languages.
4. A core of support utilities are available. Other, more specialized utilities may be added as needed.