9-1988

A simulation of a message passing protocol for a network of transputers

Janice R. Glowacki
Florida International University

ABSTRACT

A SIMULATION OF A MESSAGE PASSING PROTOCOL

FOR A NETWORK OF TRANSPUTERS

by

Janice R. Glowacki

With decreasing cost and size of processors and more sophisticated demands of computer users, it is becoming popular to execute programs in parallel on a distributed network. Processors communicate through shared memory or hard-wired links depending on the hardware and topology of the system. Simulation is an appropriate tool for the investigation of system throughput, and the projection of system behavior under various workloads.

In this paper is described the configuration and communication protocol of an INMOS Transputer network, and the construction, verification, and validation of a detailed simulation model for the network. Results obtained from the execution of the model, projecting system behavior under both heavy and moderate workloads, are presented. The most significant results obtained indicate that system throughput is severely degraded when increases are made to either message traffic distance or network buffer size. Several areas for further research are suggested, including an alternative topology for large networks.

A SIMULATION OF A MESSAGE PASSING PROTOCOL FOR A NETWORK OF TRANSPUTERS

by

Janice R. Glowacki

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

at

FLORIDA INTERNATIONAL UNIVERSITY

Committee in charge:

Professor John Craig Comfort — Chairperson

Professor David Barton

Professor Doron Tal

September 1988

To Professors John Comfort,
David Barton,
Doron Tal,

This thesis, having been approved in respect to form and mechanical execution, is referred to you for judgment upon its substantial merit.

Dean James Mau
College of Arts and Sciences

The thesis of Janice R. Glowacki is approved.

Professor David Barton

Professor Doron Tal

Major Professor John Comfort

Date of Examination: September 16, 1988

A SIMULATION OF A MESSAGE PASSING PROTOCOL FOR A NETWORK OF TRANSPUTERS

by

Janice R. Glowacki

A Thesis submitted in Partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

at

FLORIDA INTERNATIONAL UNIVERSITY

1988

ACKNOWLEDGEMENTS

I wish to thank my husband, Paul, and Professor John Comfort, whose support and encouragement helped make this project a success. In addition, I give special thanks to Li Qiang and Raja Gopal for sharing their friendship and Transputer expertise.

TABLE OF CONTENTS

LIST OF FIGURES

Figure | Page

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 DISTRIBUTED NETWORKS

Large computer networks, local area networks, and multiple processor systems are considered to be distributed networks. With these systems, processes of a single program can be distributed over several processors such that each processor on the network performs a subtask of the main program. Network processors need to share mutual information and are classified as tightly or loosely coupled [7]. Because tightly coupled systems have shared memory, an algorithm must exist to insure mutually exclusive access to it. Loosely coupled systems have local memory for each processor and communicate by using a message passing scheme.

Processors (nodes) in a ring network are loosely coupled and physically connected in a circle, usually with one-way communication links. Generally, a token or store-and-forward message passing scheme is used to support communication between nodes.

In a token passing scheme, a specific message, the token, continuously circulates through the network [7]. If a node wants to send a message, it must first acquire access to the network by removing the token when it arrives. This sending node forwards a message header followed by the message. When the message has traveled completely around the network, the sending node removes it (guaranteed the destination node received it) and forwards the token. Thus, only one message may travel through the system at one time.

With a store-and-forward message passing scheme, each node has designated storage (buffer) for incoming messages. As messages are received, they are placed in this buffer. When messages can be forwarded, they are removed from it. Because the buffer is a shared resource, the communication scheme is not trivial. The sending and receiving processes form a producer/consumer relationship and special techniques must be employed to prevent deadlock.

With advanced system architecture it is not uncommon to find systems with a large number of processors. The Ethernet[1] local area network, for instance, can support up to 1024 processors [5].

1.2 SIMULATION

In order to analyze a network and evaluate system throughput or determine the number of processors needed for efficient communication, a simulation model can be designed. The behavior of a simulation system, according to Banks and Carson [1], "can be used to experiment with new designs or policies prior to implementation". Shannon [6] explains:

---

[1]Ethernet is a registered trademark of the Xerox Corporation.

> Simulation is the process of designing a model of a real system and conducting experiments with this model for the purpose either of understanding the behavior of the system or of evaluating various strategies (within the limits imposed by the criterion or set of criteria) for the operation of the system.

Simulation models are classified as continuous or discrete-event. The difference is based on the way the state of the system changes over time. Continuous simulation is used to model a system that changes continuously over time. Discrete-event simulation is used to model a system which changes state at discrete intervals of time.

Banks and Carson explain a discrete-event simulation "proceeds by producing a sequence of system snapshots (or system images) which represent the evolution of the system through time" [1]. A snapshot for time (CLOCK = t) includes:

* the system state at time t--the variables that describe the system and are needed for the study
* the Future Events Queue (FEQ)--the list containing all activities in progress and the time they will terminate
* the status of all entities--the objects of interest
* current accumulators and counters used for statistic summaries

In discrete-event simulation models, events are classified as bound or contingent. Bound events mark the ending of an activity of specified length. Contingent events are determined by certain conditions of the system and are triggered by the occurrence of a bound event.

1.3 SUMMARY OF PREVIOUS WORK

Several distributed systems have been simulated in order to evaluate their performance. The maximum mean data rates for several local area networks are presented by Stuck [8]. He explained that transmission medium has a dual purpose: to control access to the network and to transmit the data. Traffic on the network may be of low or high delay. When the network has high delay traffic, it is a bottleneck, and more time may be spent controlling access to the network than actually transmitting data.

Stuck included an evaluation of two ring networks and two bus networks. The ring networks consisted of 100 stations using a token passing scheme. The first had a single station sending to any of the 99 other stations, while the second had all 100 stations sending messages to each other. The bus networks consisted of a token passing scheme and carrier sense multiple access with collision detection. Stuck concluded by stating "Token passing via a ring is the least sensitive to workload, offers short delay under light load, and offers controlled delay under heavy load".

Garcia and Shaw [3] studied transient behavior of a five-node network using a store-and-forward message passing scheme. Assuming message traffic would be changing in the future, they were interested in analyzing current communication channels to determine if they were adequate for future loads. In addition they were concerned with how performance might be improved.

Both a sudden burst of messages and a sudden reduction in interarrival time for given periods were modeled. They found network performance severely degraded by these transient message loads.

CHAPTER 2

THE REAL NETWORK

The INMOS Corporation manufactures microprocessors specifically designed for parallel processing. These processors are called Transputers[2] and can be put together as a distributed network connected by their fast, hard-wired communication links. Currently, the School of Computer Science at Florida International University has a four-processor distributed network of T414 Transputers.

2.1 TRANSPUTER HARDWARE AND SOFTWARE

According to the INMOS Transputer Reference Manual, these T414 Transputers context switch in a microsecond and perform approximately seven million integer/data move instructions per second [4]. The communication links between processors transmit data at a rate of 10 or 20 MHz (individually switch selectable) with effective rates of .8 and 1.6 million bytes per second, respectively.

INMOS markets several different configurations of its Transputers. The University owns several INMOS B004 and INMOS B003 boards. The B004 board is an IBM PC/XT or PC/AT

---

[2]Transputer is a registered trademark of the INMOS Group of Companies.

add-in board containing one T414 Transputer with two megabytes of memory. In addition, it contains an IMS C002 link adaptor which connects one of the T414 communication links with the Input/Output channel of the PC/XT or PC/AT. The PC can then be used as an Input/Output device and file server for the Transputer. For this reason, the T414 Transputer on the B004 board is referred to as the "host" Transputer.

The network of four T414 Transputers, each with 256 kilobytes of memory, resides on an INMOS B003 board. Each Transputer has four bidirectional communication links which can be connected to other Transputers or local memory. Therefore, several topologies are available for a network of Transputers. The current topology of the network is shown in Figure 1.

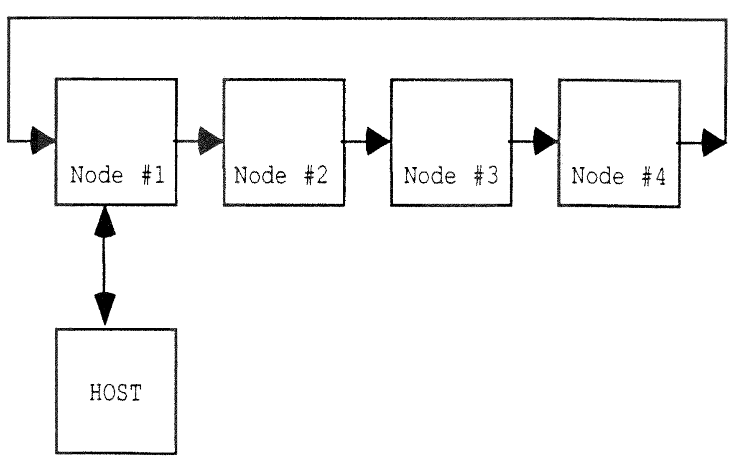

Figure 1: Transputer network topology

Occam[3] is the native language of the Transputer system. The basic elements of an Occam program are processes that can run sequentially or in parallel. Occam processes communicate over user-specified logical channels. These channels can be links connecting Transputers or local soft channels connecting processes running on the same Transputer. In addition, Occam supports most of the constructs available in modern high-level languages.

One advantage of the Occam view of processes is they are assigned to processors at compile time. Thus, a program developed as a set of parallel processes on a single Transputer system may be recompiled for any valid Transputer/process mapping [2].

2.2 THE COMMUNICATION PROTOCOL

A store-and-forward message passing scheme for the network of four Transputers was written by Li Qiang of Florida International University [9]. The system is comprised of five processes running on each node.

There exist two types of processes: network and local user. Network processes are those that have access to the physical links of the network. Local user processes do not have access to the physical network and are thereby "local" to a given node.

---

[3] Occam is a registered trademark of the INMOS Group of Companies.

There are three local user processes. The main one, performs the application program and generates messages for the node. The second receives all messages for the node. The third acts as an intermediate process supporting communication between the network and the receiving local process.

Figure 2 displays the five processes of a single node and shows the flow of message traffic through the network.

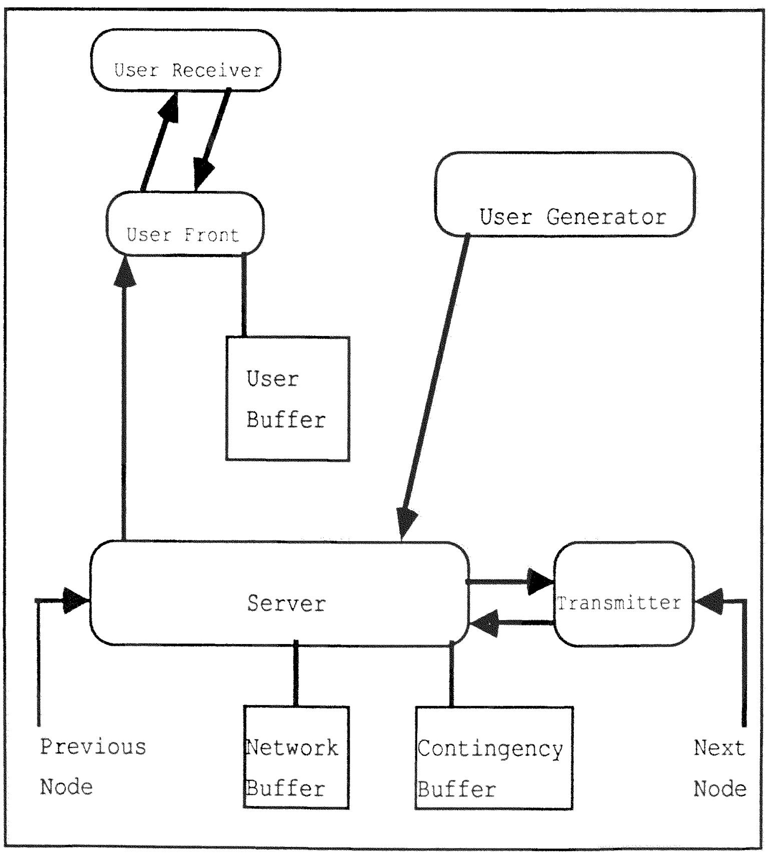

Figure 2: A single node in the network.

In order to accommodate incoming messages, there exist three buffers: the user, the network, and the contingency. The user buffer contains those messages received for the local node. The network buffer holds those messages to be transmitted to the next node. The contingency buffer is a protective buffer holding a message that would otherwise overflow the network buffer. This buffer is necessary to avoid deadlock as explained by Qiang [9] and later in this chapter.

Each message contains a message header that indicates its source, destination, and length. The header itself is exactly one word regardless of the length of the message. It is important to note that messages are handled at the "word level". Each word of a message is sent individually although it is part of an entire message.

2.2.1 The Five Communication Processes

The primary responsibilities of the five processes shown in Figure 2 are explained below. To clearly identify each individual process, they have been named and underlined.

The User Generator. This process is responsible for creating messages and passing them over a soft channel to the server. The channel acts as a blocking channel. Therefore, the user generator is blocked between passing each word of a message.

The User Receiver. This process is responsible for reading the messages sent to the current node. It sends a request over a soft channel to the user front to read each word. It is therefore blocked from the time it sends a request until a word is actually forwarded.

The User Front. This process is responsible for the user buffer. It handles the producer/consumer relationship of the server and user receiver. The server passes words to the user buffer via the user front, while the user receiver gets words from the user buffer via the user front.

Occam channels are blocking channels. That is, if process P1 sends a word to process P2, P1 cannot continue until P2 receives the word. If P2 is busy and not ready to receive, then P1 remains blocked. In order to create a non-blocking channel, an intermediate process,P3, must be created [10].

Accordingly, in order to have the server (P1) pass messages to the local user receiver (P2) without blocking, there must exist the user front (P3) as an intermediate process. The user front takes messages from the server and, transparent to the server, places them in the user buffer. Upon request, it removes them from the buffer and forwards them to the user receiver. Because messages are handled at the word level, a separate request must be issued for each word of the message.

The Server. This process takes words from the incoming link and places them in the appropriate buffer. Messages for the current node are sent to the user front and placed in the user buffer, while all other messages are placed in the network buffer for retransmission. It also receives messages from the <u>user generator</u> and places them in the network buffer for retransmission. Lastly, it answers the <u>transmitter's</u> requests by removing and forwarding messages from the network buffer (one word at a time).

<u>The Transmitter</u>. This process monitors the outgoing link. Whenever the link is available, it requests and receives a word from the <u>server</u> to be placed on the outgoing link.

2.2.2 Avoiding Deadlock

Deadlock can easily occur in this network if each <u>user generator</u> saturates the network to the point where every node is blocked from servicing incoming messages. In order to prevent this situation, there exists a protocol for filling the network buffer [9].

In short, the <u>server</u> receives messages from the <u>user generator</u> and the incoming link. Messages from the incoming link are categorized as "local" or "non-local". The <u>server</u> forwards local ones to the <u>user front</u> and fills the network buffer with non-local ones. The <u>server</u> places a message from the <u>user generator</u> into the network buffer if, and only if, the entire message can fit. Whenever the network buffer is full, however, the <u>server</u> blocks the <u>user generator</u> and processes messages from the incoming link by filling the contingency buffer. This buffer must be large enough to hold one complete message.

This protocol enables the server to push messages through the system even when the local user process has saturated the system. In other words, if the network buffer fills, the contingency buffer is still available to buffer network traffic.

The Transputer link, like a soft channel, behaves as a blocking link. Therefore, any word sent down a link remains on it until removed by the next node. For deadlock to occur, each link must be transmitting data, and each buffer (network and contingency) must be full such that every node is blocked and will remain blocked indefinitely. To avoid this situation, it is necessary to have the priority scheme for filling the network and contingency buffers as described.

2.2.3 Proof The Algorithm Is Deadlock-Free

The store-and-forward message passing algorithm by Qiang is deadlock-free [9].

Proof by contradiction. Assume the algorithm is not deadlock-free and the network is in the state of deadlock. In other words, each network and contingency buffer is full, each link has data on it, and each user generator is blocked from submitting a message into the network. Then, there is a situation just before deadlock similar to that shown in Figure 3.

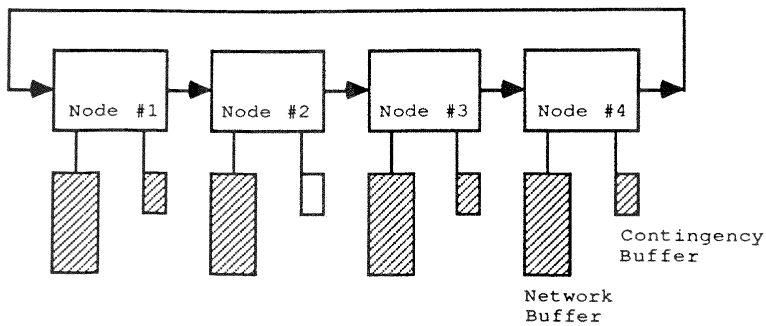

Figure 3: Pre-deadlock situation.

Suppose the last node to fill its contingency buffer was node #2. Then, when node #2 removed data from the incoming link it would enable node #1 to move data from its network buffer to its outgoing link, transfer data from its contingency buffer to its network buffer, and receive the data on its incoming link to be placed in its contingency buffer. But then the network is not in a state of deadlock.

Contradiction of assumption. Hence, the algorithm is deadlock free.

When the network buffer is full, the algorithm's protocol requires the data from the incoming link be received before submitting to the network messages generated by the local node. This way, it guarantees flow of traffic even when the network is saturated with messages.

When the pre-deadlock situation occurs, filling node #n's contingency buffer enables node #n-1 to unload data from its network buffer and transfer contents from its contingency buffer. Thus, node #n-1 now has an empty contingency buffer to place data from the incoming link. This will continuously propagate such that there is never an instance where each contingency buffer is full. Thus, when traffic is intense, the network can become blocked. However, because of this protocol for filling the network and contingency buffers, the network <u>cannot</u> deadlock.

CHAPTER 3

THE SIMULATION MODEL

Simulating a network communication protocol requires complete understanding of both the real system and of simulation techniques. The simulation is not a duplication of the system with added statistical computations. Instead, it models the real system by recording and gathering statistical information based on the events and actions that would be occurring in the system. The computer programs for both the real and simulated systems are given in the Appendices in order to exemplify the significant difference between them.

3.1 SIMULATION METHODOLOGY

It is not uncommon for a simulation to use an enormous amount of computing time due to the number of calculations used for generating random numbers, accumulating statistics, and managing the future events queue. One attractive solution to shortening the run-time of a simulation is to incorporate a network of computing power. Comfort has investigated the idea of distributed simulation whereby related processes of the simulation can be placed on separate processors of a network [2].

Comfort has written a distributed simulation package to run on the INMOS Transputer system [2]. The program identifies objects such as a statistics module, random number generator, and a priority queue handler. Each object is a unique process. The program can be run on a single Transputer system; however, when running the simulation on a network of Transputers, it is possible to distribute each object onto separate processors of the network and enjoy the benefit of decreased run-time.

A simulation program using this package must first instantiate specific instances of these objects. The future events queue is an instance of a priority queue. The objects are then accessed by standard calls. Statistics are updated for an entity in the simulation by sending messages to the statistics package whenever the entity changes its state.

A comprehensive simulation model, using Comfort's package, was designed to investigate system throughput of the four-node ring network on the INMOS B003 board. The topology is shown in Figure 4.

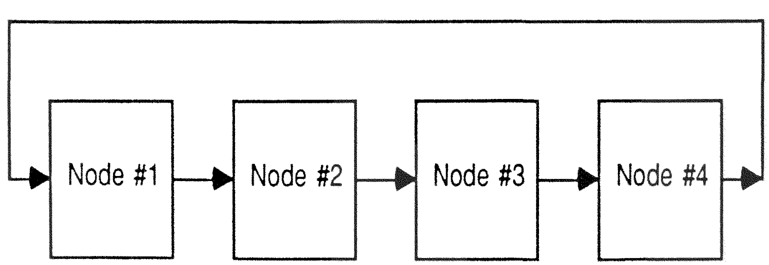

Figure 4: Simulated network topology.

Qiang's message passing protocol, as described in the previous chapter, is modeled. Also of interest were the effects of message length variation, message traffic destination (distance a message travels), and system workloads.

3.2 SYSTEM REPRESENTATION

This section explains how the processes, links, buffers, and messages were represented in the simulation model. In addition, timing of the network and parameters of the simulation are discussed.

The servers and entities. In order to simulate the real network it was necessary to determine how processes and messages should be represented. As processes service messages in the real network, servers process entities in the simulation model. Each server required a set of states and well-defined actions to be performed.

Although processes on the same processor are conceptualized as running in parallel, only one process can actually be running at a time. Thus, for every node in the model, only one server (process) could be servicing (running) at a time. Each type of server had a designated set of states and actions describing the process being modeled and could therefore be in only one state and perform only one action at a time.

Messages in the system. Messages in the real network consisted of two parts: the message header and message body. The header contained the source, destination, and length of the message. In the simulation model, each message header was an entity.

Simulating the buffers. Physically, the network and contingency buffers comprise one buffer and are logically separated in software. Because the contingency part was required to accommodate the largest message size, the total buffer space needed had to be at least as large as two maximum size messages (one for each part of the buffer). Let the term network buffer now refer to the combination of the contingency and network buffer.

In order to model the user and network buffers that held messages, it was necessary to create one FIFO queue for each buffer of every node. These queues held the message header entities while local counters were updated to track the total words in a given buffer.

Simulating the links. A Transputer link could only hold one word at a time (message headers were single words). Because actions performed depended on the type of data sent, links were simulated using two variables. The first variable indicated the type of data on the link: a message header, a word of the message body, or indication the link was free. If a message header was on the link, then it was necessary to identify the actual entity number. This was held in the second variable.

The Future Events Queue. A single future events queue (FEQ) held the bound event notices for the entire simulation. These notices included scheduling processes to time-out while waiting for a channel or because their run-time expired. Also included were notices from a node to another indicating data was sent down or removed from the link. In addition, there were batch run termination notices, as well as several others.

System timing. The time needed to perform each action was not easy to determine. Each Transputer cycle took about 67 nanoseconds which evaluates to 15 million cycles per second. In order to acquire accurate results, it was necessary to determine the time needed for each server to perform its various actions. The level of detail was so crucial that code for each process in the real network communication program was thoroughly evaluated to the point where instructions were literally counted [9]. In addition, the INMOS Reference manual was consulted for system timing statistics [4].

System clock. The simulation clock time referenced Transputer cycles rather than seconds. This was because each activity was evaluated in terms of the number of cycles necessary. If activity times were measured in nanoseconds, the clock time would become too large for some simulation runs. If activity times were measured in microseconds, then each activity would be rounded individually. Because each activity is performed a significant number of times each second, over or under estimating a time value would become significant. In order to minimize losing integrity in the times estimated, it was decided to keep all times in reference to Transputer cycles. As a result, a single simulation clock tick evaluated to 5 Transputer cycles. Thus, to simulate one second of real time, the simulation would have to run for time = 3,000,000.

Random number generators. There were five random number streams used for the model. Each stream required the mean, seed, and distribution type. There were three possible distributions: constant, negative exponential, or uniform. The streams were used to generate numbers for:

* Average links a message travels (distance)
* Number of messages to send at once
* Length of the current message
* Time to run the local user application
* Operating system delay to schedule a process

Parameters to the system. The system required 23 parameters. They were:

* The number of nodes in the network (2 to 32)
* The speed of the links (10 or 20 MHz)
* The number of batches to run
* The length of each batch
* The maximum length of a message
* The number of messages to send at once
* The size of the network buffer
* The size of the user buffer
* The distributions, means, and seeds, for each of the five random number streams

3.3 REFINEMENT OF THE SIMULATION MODEL

To simulate a computer system it is necessary to decide the level of detail which will be modeled. Specifically, "the circuit level, gate level, register-transfer level, and system level" [3]. The initial simulation model was revised several times. Each revision increased the level of detail modeled. The state diagrams and a description of the bound event actions for the final version are given in the Appendix.

3.2.1 The Original Version

In the original model there were three servers. One for each network process and one to represent all local user processes. The model itself would deadlock even though the real network did not.

The reason the simulation would deadlock is relatively simple and can be seen in the following scenario. Suppose each link contained a word being sent to the next node, and each contingency buffer was full. Furthermore, suppose node #n was the last node to fill it's contingency buffer. Then, the last bound event was for the server of node #n to place a word from the incoming link into the contingency buffer. The key here is the link between node #n-1 and node #n. Because the last bound event was for node #n, node #n-1 was not aware of the change in status of its outgoing link. It is possible for all servers on node #n-1 to be blocked. In such a case there would be no bound events for that node on the FEQ. Contingent events for node #n are only checked after a bound event has been processed for node #n. Therefore, if no bound events are scheduled for a node, then it can never reevaluate the status of its outgoing link. Hence the simulation could deadlock.

3.2.2 The Second Version

The second version eliminated the possibility of deadlock in the simulation. The "fix" was quite simple although not elegant. After a bound event was processed for node #n, the conditions for contingent events were checked for both node #n and node #n-1. Thus, the sending node would be able to update the status of the link when the receiving node made the link available. As expected, run-time of the simulation program was effected.

This model did not reflect the real network statistics as the simulated results were off by at least a factor of 5. All local user processes were handled as one server in the simulation and could not accurately reflect the real network. This was because the simulation did not account for the time needed for a context switch. In other words, the simulation modeled three separate processes running each for time t as one process running for time 3t. In reality, it requires time 3t + 2c where c is the time for a context switch to occur between running processes. Clearly, 3t + 2c is strictly greater than 3t.

3.2.3 The Third Version

In the third model, two servers were added, separating the three local user processes and clearly defining the duties of the user receiver, user generator, and user front. This version attempted to adjust the timing problem in the previous version. Although the simulation results were significantly closer to the real network statistics, it was clearly evident another level of detail needed to be modeled.

3.2.4 The Final Version

Unless a priority scheme for scheduling servers was represented, an unrealistic ordering occurred in the simulation. Therefore, it was necessary not only to keep track of the servers that could process an entity (message), but also the order in which they became available to do so.

For this reason, two queues were added in the final model: Block and Ready. The Block queue held those servers waiting for some event or condition to occur before they could run, while the Ready queue held those servers which could be run. The servers in the simulation were placed on the block queue after serving an entity (message) and moved to the ready queue according to pre-defined conditions for the process being modeled. Essentially, this modeled the operating system's scheduler.

After a bound event was processed, the status of each server on the Block queue (for that specific node) had to be evaluated in order to determine which servers, if any, needed to be moved to the Ready queue. Then, if no servers were currently running, one from the Ready queue was scheduled.

Although this approach modeled the network more realistically, it did add several drawbacks. First, significantly more computations were being performed and as a result, program run-time was severely degraded. Second, as contingent events were not tested in the "traditional" scheme, the simulation would deadlock in the same manner as the original model. Therefore, it was again necessary to design a technique to avoid deadlock in the simulation.

There were two solutions investigated. The first one would require moving node #n-1's <u>transmitter</u> from the Block queue to the Ready queue whenever node #n removed a word from the link. However, there did not seem reasonable justification to manipulate a node's data structures while processing events of another node.

The second solution required an additional bound event notice to be scheduled. Although sending node #n could compute the time a word would arrive at node #n+1, it could not determine when the word would actually be removed. Therefore, whenever node #n+1 removed data from the link, it was required to create and schedule a bound event notice for node #n indicating the link became available.

CHAPTER 4

VERIFICATION AND VALIDATION

The simulation model must be verified and validated. Model verification deals with verifying the code performs accurately and is implemented correctly. Model validation deals with showing the code accurately models the real system. The previous chapter discussed the several versions of the simulation model. Each version was carefully evaluated in an attempt to verify and validate it. However, the earlier versions did not accurately model the real network and the revisions became evident during the evaluation process. This chapter discusses the verification and validation of the final version.

4.1 MODEL VERIFICATION

Verifying the simulation model, like verifying any computer program, can be done using very common sense techniques [1]. Banks and Carson suggest:

* make the code "self-documenting"
* make a flow diagram indicating the possibilities encountered when an action for an event occurs
* verify the input parameters are not modified
* use a program trace while testing the code
* closely examine the output for "reasonableness"

Each of these techniques were incorporated in order to verify the simulation code. An explanation of the use of each techniques as it was applied to this project is given here.

Self-documenting code. An Occam program is viewed as a single fold comprised of other folds. A fold is simply the concept of grouping information or code together as a separate unit. Each fold can be identified with a name (generally used to explain the fold's contents) and may contain other folds, comments, and code. In general, folds are kept small and concise. Therefore, Occam programs are "self-documenting" by nature.

The code for the simulation program is given in the Appendix. Along with explanatory fold names, documentation for all variables, states, and actions were included in the source code.

Flow diagram. A flow diagram is suggested in order to evaluate each possible action the system can perform after each event. The flow diagram for the simulation model consists of the state diagrams for each of the servers. These can be found in the Appendix.

Verify input parameters. The 23 input parameters for the system were printed after several tests to verify they were not modified during the execution of the simulation.

Trace the execution. The trace was used to get output while the simulation was running to determine if the code was performing accurately. The trace was very useful and helped determine the reason the simulation would deadlock. In addition, it helped identify the unfair scheduling of processes in the earlier versions.

The trace included information about each queue (what was being added or removed from it), each random number stream (what stream was generating numbers and what the numbers were), the statistics package (what entity was entering and leaving what state), and each bound event action (what and when it was pulled from the FEQ).

Examine the output. The output for each version was evaluated. It was not until the final version that "reasonable" results were found. These results are explained and shown in the validation part of this chapter.

4.2 MODEL VALIDATION

Validation is an approach used to determine if the model accurately represents the real system. According to Banks and Carson [1]:

> Validation is usually achieved through the calibration of the model, an iterative process of comparing the model to actual system behavior and using the discrepancies between the two, and the insights gained, to improve the model. This process is repeated until model accuracy is judged to be acceptable.

The rest of this chapter presents the results obtained from both the real and simulated networks. The results are compared and the simulation is "judged to be acceptable".

The real four-node network was run until each node sent/received 30,000 messages of 15 words to/from the node three links away. This test was run several times with different network buffer sizes but with the user buffer and link speed set constant at 2000 words and 10 MHz respectively. A few timers were added and the system appeared to reach stability almost immediately. The average time in the system is displayed in Table 1.

TABLE 1

Simulated Versus Real: Average Message Time in System (Seconds)

Buffer Size	Real	Simulated	Difference	Relative Error
36	.00767	.00492	.00275	.3585
54	.00748	.00981	-.00233	.3115
150	.03380	.03900	-.0052	-.1538
300	.08300	.08300	.0000	.0000
500	.14616	.14633	-.00017	-.0012
2000	.60320	.60330	.00010	-.0002

Intuitively, we could visualize the local <u>user generator</u> flooding the <u>server</u> with messages so the network buffer would be filled to capacity. Then, the <u>user generator</u> would be blocked and the <u>server</u> would be able to handle incoming messages by placing them in the contingency buffer. At some point, the server could reach a steady state of handling both incoming and local messages.

The simulation was then tested where each of the four nodes were sending/receiving continuously to the node three links away. The user buffer size and link speed were set to constants of 2000 words and 10 MHz respectively. The test was run several times varying the network buffer size.

Each test was run for eight blocks, each representing one second of real time. The network was presumed to have been saturated with messages and reached steady state as the results for blocks three to eight were the same (as expected for constant input parameters). A comparison of the average message time in the system for both the real and simulated networks are shown in Table 1 and Figure 5.

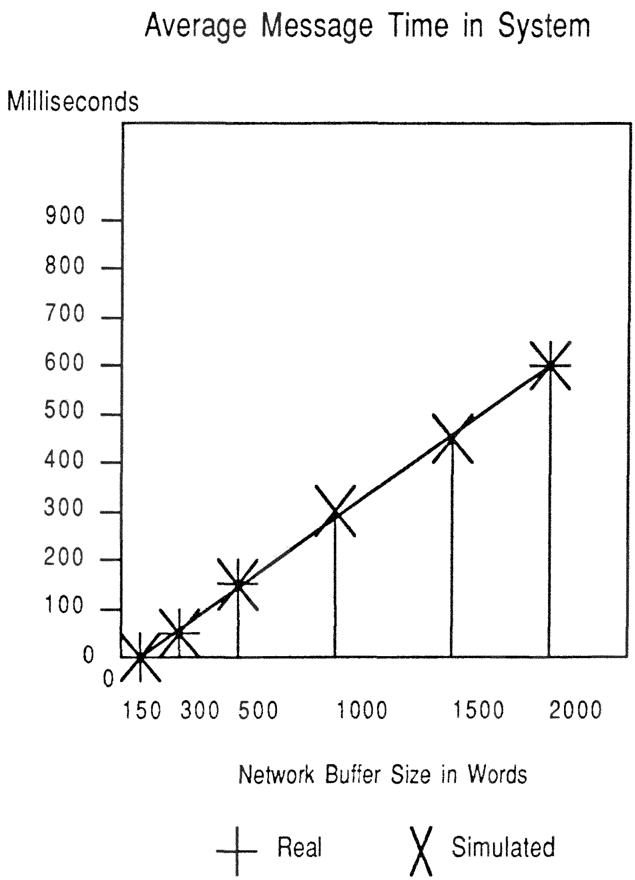

Figure 5: Simulated Versus Real: Message Time in System.

The simulation was then run with uniformly distributed random message lengths between 1 and 31 words. Again, each node was sending messages across 3 links at 10 MHz. The user buffer was set to 2000 words. The simulation was set to run for 25 intervals each representing one-half second of real time.

The results are shown in Table 2 along with the 90% confidence interval which encapsulates the real network's average message time in the system (see Table 1). Note that network buffer sizes of 32 and 54 could not be tested because the maximum size of a message was 31 words and the network buffer was required to accommodate two maximum size messages (one for the contingency buffer, one for the network buffer).

TABLE 2

Average Time a Message Remains in a Four-Node Network With Random Message Length (Seconds)

Network Buffer	Average Time in System	Standard Deviation	90% Confidence Interval
150	.03185	.00701	.02033 TO .04337
300	.09867	.01364	.07623 TO .12111
2000	.79333	.12100	.59426 TO .99235

With several test runs and the results listed here, it was decided the model was valid.

CHAPTER 5

RESULTS

In order to evaluate system performance, a well-defined, organized, and statistically sound testing method was required. Each test was run at least twice with different random number generator seeds in order to insure that no bias was added by the choice of seed. This chapter presents the major test results and findings of this research.

5.1 SYSTEM PERFORMANCE UNDER DIFFERENT WORKLOADS

When validating the model, it was noted that, message time in the system usually decreased as the buffer size decreased. However, real system performance was better at buffer size 54 than 36. This indicated that smaller buffers increased system performance, but that at some point there was a cut-off, at which time performance slightly decreased. However, as determining the cut-off point was not part of this evaluation, tests in this section incorporated the fact that smaller buffers increased system performance, but did not seek to determine an "optimal" buffer size.

Testing was extremely time consuming (12 minutes to simulate one second of real time). Therefore, not all configurations could be thoroughly studied. Although the system had 23 parameters and could model numerous configurations, certain consistent parameters were used for all the tests described here. The network size was fixed with four processors. Because message lengths may vary, the tests used message lengths uniformly distributed between one and eleven words. The network and user buffers were kept relatively small (33 words--chosen to hold three maximum size messages). Lastly, as preliminary tests from the real and simulated networks indicated only slight improvement in system performance when the links were set at 20 MHz, it was decided to test with links set at 10 MHz.

Two workloads describing the message traffic were defined: heavy and moderate. The heavy load assumed the user application program continuously generated messages. The application program would spend only a few microseconds processing before generating its next message. The moderate workload had the application program run for a short while, thereby generating only a moderate number of messages.

There are four cases discussed in this section. Two for heavy workload and two for moderate workload. The heavy workloads used a constant of five microseconds for processing time between generating messages, while the moderate loads used a uniformly distributed processing time between zero and two milliseconds. Therefore, the heavy loads had one random number stream (message length), and the moderate loads had two (message length and processing time).

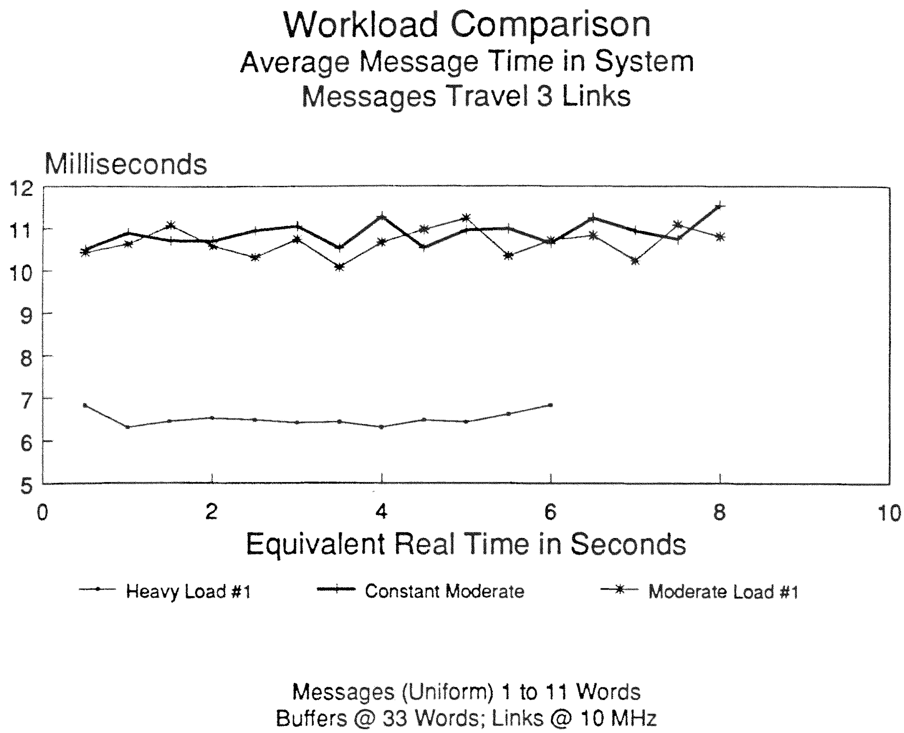

Figure 6: Workload Comparison.

Each load had a designated seed or seed pair used for each test. In order to compare workloads and to evaluate the effect of introducing the second random stream, the first heavy and moderate workloads used the same seed for message length. There was an additional run which used the same seed for message length but had a constant workload of one millisecond.

The simulation was run to model the network where each message destination was the previous node (message distance was three links/worst case analysis). Figure 6 displays the average message time in the system for the heavy, moderate, and moderate constant loads with the same message length seed. The randomness introduced by the process time can be seen along with the difference between workloads.

For each test case, several preliminary tests were run in order to determine when steady state was achieved. The simulation was run such that each node sent messages to the previous node. These preliminary tests were run for approximately 25 seconds of real time in block lengths equivalent to 1/4, 1/2, and one second. The "deleted moving average" for block lengths of 1/4 and 1/2 was computed and compared to the results of the one second block length. These data were examined to determine when steady state occurred and which block size was most appropriate.

It was found, that block length of 1/2 second was less sensitive to random variation as the 1/4 second block, and captured more information than the 1 second block. Thus, it was used for the block length of the following cases.

Each test workload was run for all possible message distances, for several seconds past the time determined as "steady state". The averages for message time in the system, following the decided steady state time, were then aggregated. Table 3 displays these aggregated averages and standard deviations.

TABLE 3

Aggregate Average Time in System: All Loads
(Milliseconds)

Message Distance	Heavy Load #1	Heavy Load #2	Moderate Load #1	Moderate Load #2
1 Link	1.755	1.766	4.031	4.003
2 Links	5.145	5.143	8.949	9.013
3 Links	6.413	6.554	10.777	11.048
4 Links	15.103	15.263	19.476	26.586

Figures 7 through 10 display each 1/2 block value for the different case workloads--from start-up through a couple of seconds at steady state.

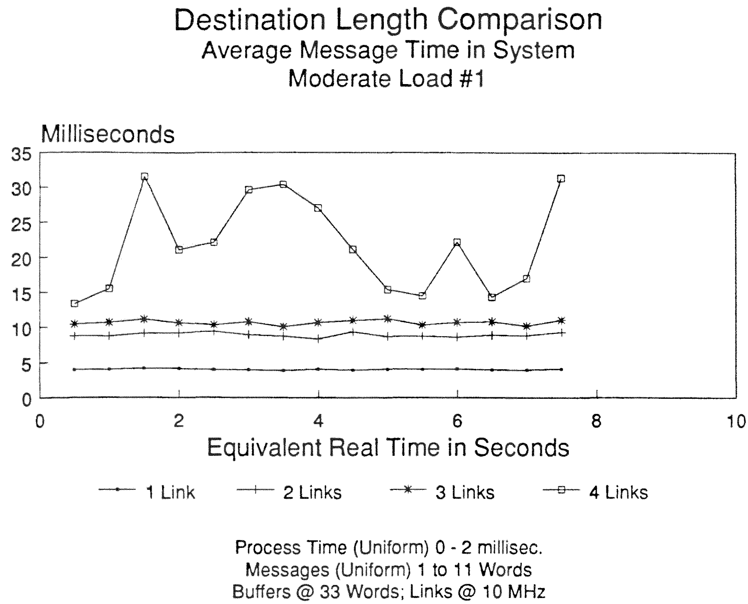

Figure 7: Average Time in System: Moderate Load #1.

For all workloads, when message distance was one link (best case scenario), the time in the system was minimal. Clearly, no message had to compete with network messages to get into the network buffer. Each message was immediately placed in its network buffer, sent across the link, and was placed in the user buffer of the successor node, never really competing for space in any buffer.

Significant difference was found as soon as the messages had to travel more than one link. The competition for the network buffers can be seen in Figures 7 to 10.

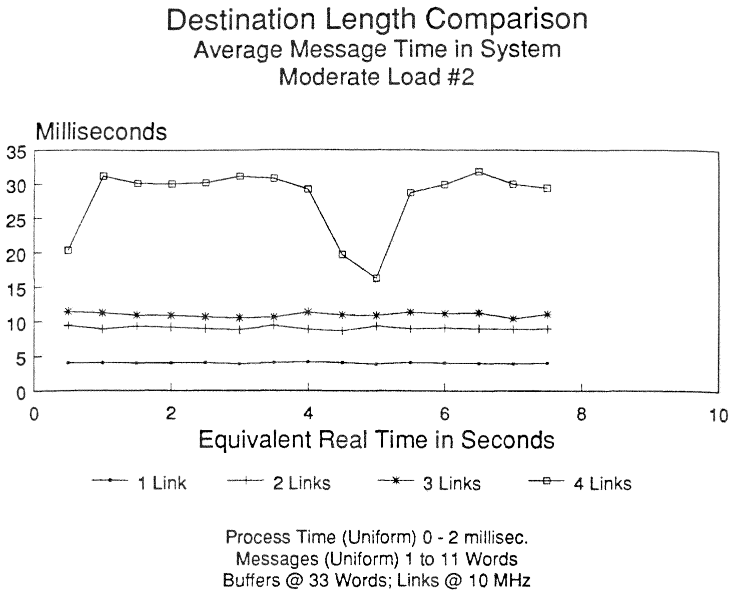

Figure 8: Average Time in System: Moderate Load #2.

Results of the two moderate workloads are displayed in Figures 7 and 8. Comparable results were found.

There was a dramatic degradation in system performance when messages had to travel across four links. Messages were in circulation longer, competed for even more buffers, and were affected more by the randomness of the test than any other message distance. If a network were to be increased, and message distance were significant to the size of the network, projected system performance would degrade radically.

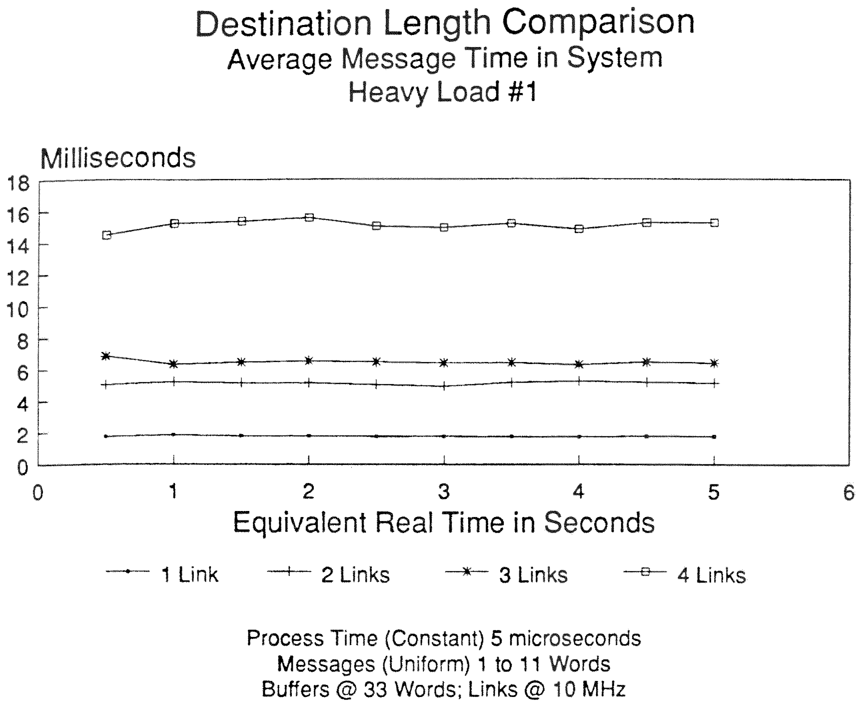

Figure 9: Average Time in System: Heavy Load #1.

Results of the two heavy workload systems are displayed in Figures 9 and 10. The results were consistent indicating the random seeds did not introduce a new bias. Because the application program was not really executing for any significant time, there was less time between the network processes running. As a result, message time in the system was decreased consistently across all message distances as compared with the moderate workloads. In fact, there was a minimum three millisecond increase for all message distances.

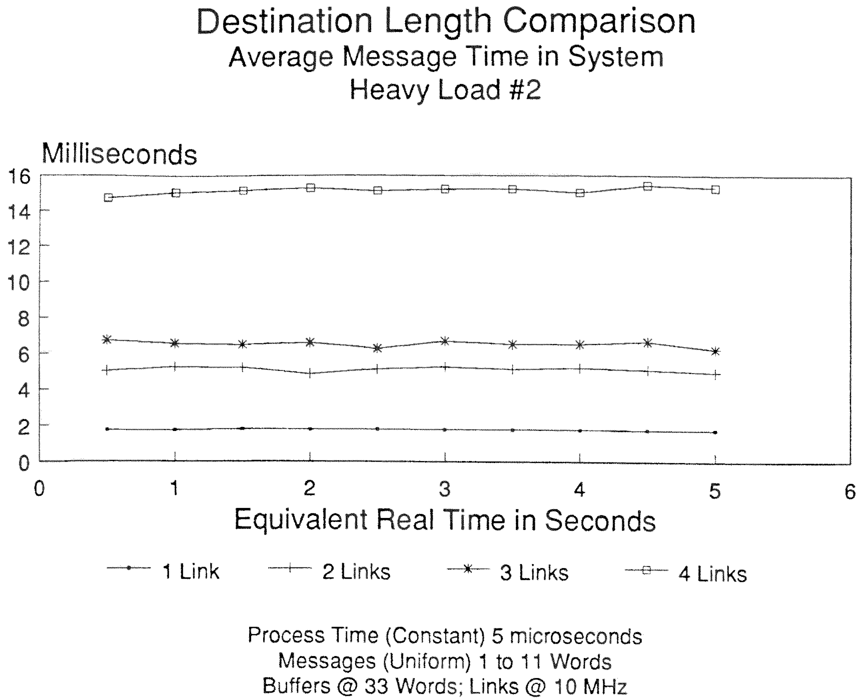

Figure 10: Average Time in System: Heavy Load #2.

A five-node network was run with the message distance held constant at four links. The average message time in the system was found to be greater than with the four-node network with message distance of four links. Although all test cases were not run yet for the five-node network, the evidence indicated considerable degradation of system performance as the network size increased along with message distance.

5.2 EFFECT OF BUFFER SIZES

Several tests were run in order to determine the effect of changes made to the user and network buffer sizes. The random number generator used for message lengths (uniformly distributed between one and eleven words) was run with several different test seeds. Message distance was held constant to three links. Once the system reached steady state, the averages were aggregated and some are shown in Table 4. For these tests, the link speed was set at 10 MHz and the network was run at heavy load.

TABLE 4

Effect of Buffer Size for Worst Case Scenario (Milliseconds)

Test Seed #37

Network Buffer	User Buffer	Aggregated Average	Standard Deviation
33	99	6.322	0.2045
33	33	6.322	0.2045
99	33	33.986	1.2344

Test Seed #83

Network Buffer	User Buffer	Aggregated Average	Standard Deviation
33	11	6.554	0.1716
33	22	6.554	0.1716
33	33	6.554	0.1716
333	33	131.022	1.5609

* Messages (Uniform) 1 to 11 Words
* Message Distance (Constant) 3 Links

These results indicated that the user buffer was not a bottleneck. Thus, for the system at heavy load, the user buffer could be small. This would be useful for applications programs with large memory requirements. However, further research is needed in order to determine if this conclusion remains valid when the system is running at other workloads.

If the application program were required to communicate with only its successor node (best case), would it be more efficient to have larger buffers? Table 5 shows the results of the simulation program running at heavy load with message distance constant at one link. These results indicate, again, that smaller buffers improve system performance.

TABLE 5

Effect of Buffer Size for Best Case Scenario (Milliseconds)

Network Buffer	User Buffer	Aggregated Average	Standard Deviation
33	11	1.766	0.007
33	33	1.766	0.007
330	33	30.928	2.009
330	330	30.928	2.009

* Message Distance (Constant) 1 Link
* System Running at Heavy Load #2

Consideration should be given to the type of application program being run. For instance, if a program required significant computing time, larger buffers would minimize time spent waiting to send a message. The application program could generate a message, deposit it in the buffer, and continue processing. Although the message itself would remain in the system longer, the application program would not be blocked for a significant time.

CHAPTER 6

CONCLUSIONS

Both system throughput and average message time in system were strongly influenced by the size of the network buffer. When the buffer was large, the system could accommodate more messages. However, each message would have to remain in the system longer because it had to trickle through larger buffers.

The ring network studied was quite sensitive to message distance. As message destination length increased, system performance was radically degraded. Message time in the system increased because messages, not only had to travel further, but also had to also compete for space in each network buffer with the local messages being generated. Therefore, system performance is projected to decrease as both the size of the network and the message distance increase.

Lastly, special attention should be given to the type of application program to be executed on the system. If it is more important for an application to be able to execute than to minimize message time in the system, larger buffers should be considered. The network processes would be delayed because of the longer application run time.

In order to evaluate system performance of the

Transputer network, a simulation model was designed. The model allowed investigation of workloads and conditions that would otherwise be at best difficult to monitor and analyze. With five processes running in parallel on each Transputer, the simulation attempted to model "chaos" in an organized and elegant fashion.

CHAPTER 7

FURTHER RESEARCH

When message distance is increased the network performance is severely degraded. Thus, poor performance can be projected for large ring networks demanding intensive communication between processors. Therefore, if this project were extended, it is suggested to investigate throughput of other network topologies. Specifically, topologies which reduce the number of links a message must travel. One such topology is shown in Figure 11.

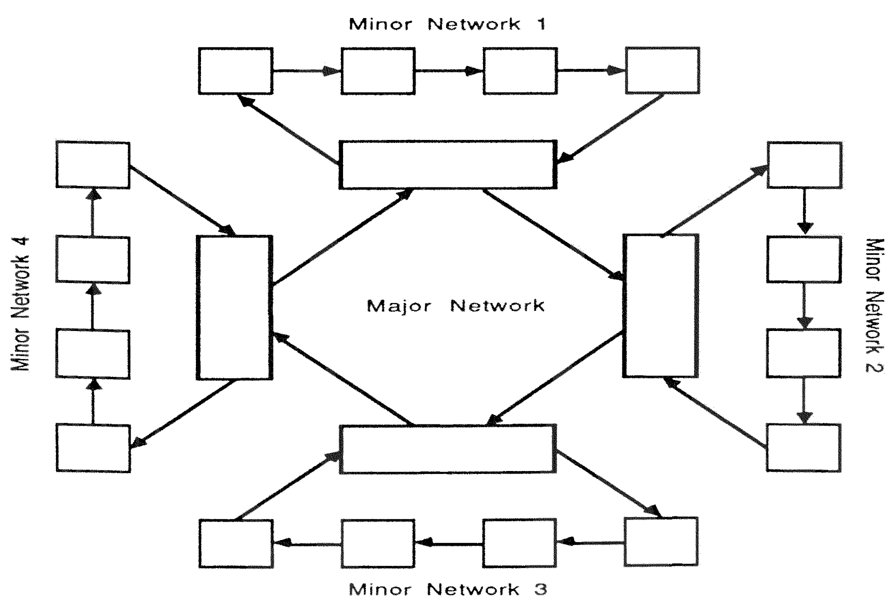

Figure 11: Alternate topology for large networks.

Each "host" Transputer for a minor network would be responsible for sending its minor network messages onto the major network. Likewise, it would be responsible for receiving messages for its minor network from the major network. This particular "network of networks" could be simulated in a two-step process. First, statistics about the minor networks would be gathered. Second, the major network would be simulated by incorporating the minor network statistics.

It is clearly evident from the results obtained that the network buffer size effects message time in the system. System performance degrades when this buffer is increased slightly. Further research may find an "optimal" message to buffer size ratio for either a given number of processors, a given workload, or both.

APPENDICES

APPENDIX A. THE STATE DIAGRAMS

A description of each state and bound event for every server in the simulation is described in this Appendix. The symbols used are described in Figure 12:

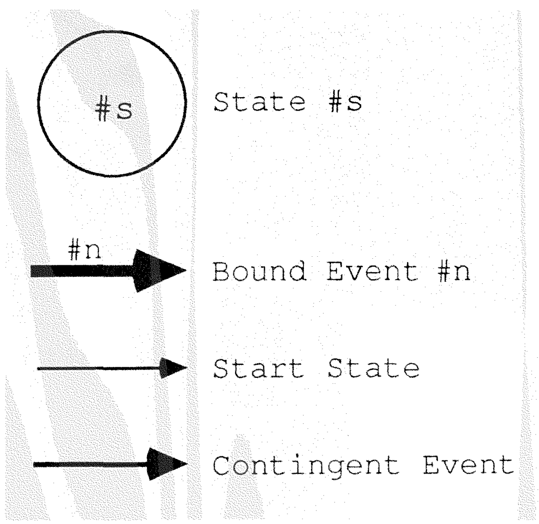

Figure 12: Summary of State Diagram Symbols.

The User Generator

The States:

0. UG.Think ---------> running, thinking up messages
1. UG.Block ---------> blocked waiting to send a word
2. UG.Fill.Nbuff ----> filling network buffer (the server process is not running)

Bound Event Actions:

2. UG. Time.Out -----> time out for running
3. UG. Xfer ---------> time required to transfer a word of a message to the nbuff

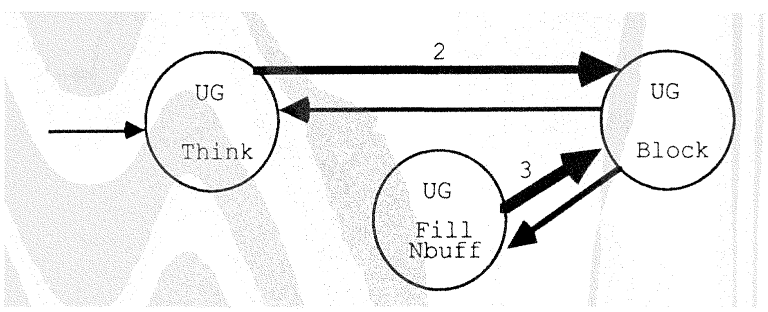

Figure 13: The User Generator State Diagram.

The User Receiver

<u>The States</u>:

3. UR.Block.UF ------> waiting for UF to pass a word
4. UR.Block ---------> blocked waiting to read one word
5. UR.Read.Mail -----> reading one word of a message

<u>Bound Event Actions</u>:

4. UR.Close.Mail ----> read one word of a message

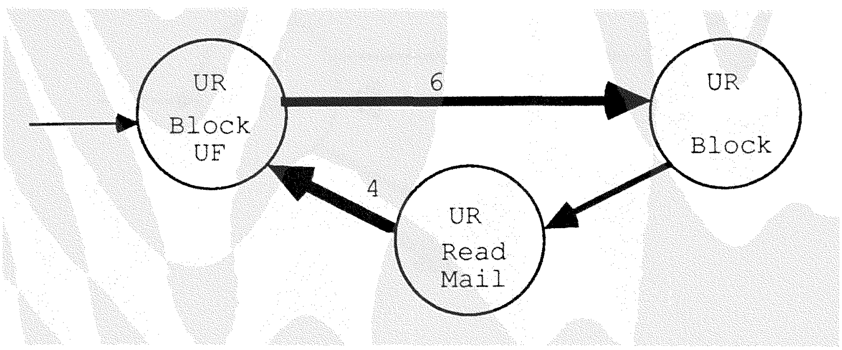

Figure 14: The User Receiver State Diagram.

The User Front

The States:

```
6. UF.Block ---------> blocked, waiting to run
7. UF.Fill.Ubuff ----> placing word in user buffer
8. UF.Remove.Ubuff --> removing word from user buffer
```

Bound Event Actions:

```
5. UF. Produce ------> place word in user buffer
6. UF. Consume ------> remove word from user buffer
```

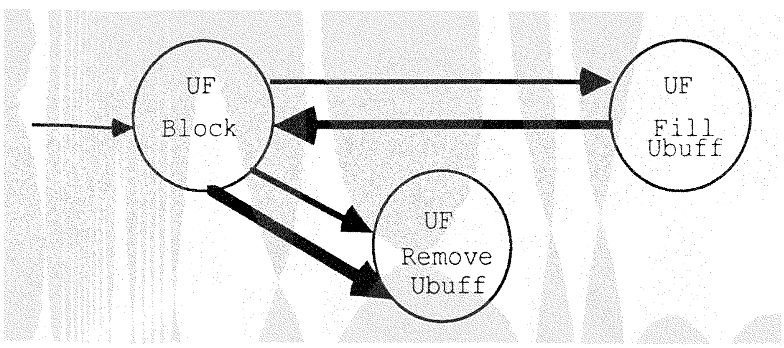

Figure 15: The User Front State Diagram.

The Network-In (Server)

The States:

```
 9. NI.Sleep ----------> nothing on link to get
10. NI.Block.Nbuff ---> waiting for room (net buffer)
11. NI.Block.Ubuff ---> waiting for room (user buffer)
12. NI.Block.UF -------> waiting for UF to run
13. NI.Wait.On.Link --> waiting to get word on link
14. NI.Fill.Nbuff ----> moving word (link to net
                        buffer)
15. NI.Fill.Ubuff ----> put word in user buffer via UF
```

Bound Event Actions:

```
7. NI.Get.Link ------> a word arrived on the link
8. NI.Xfer ----------> word was moved (link-buffer)
```

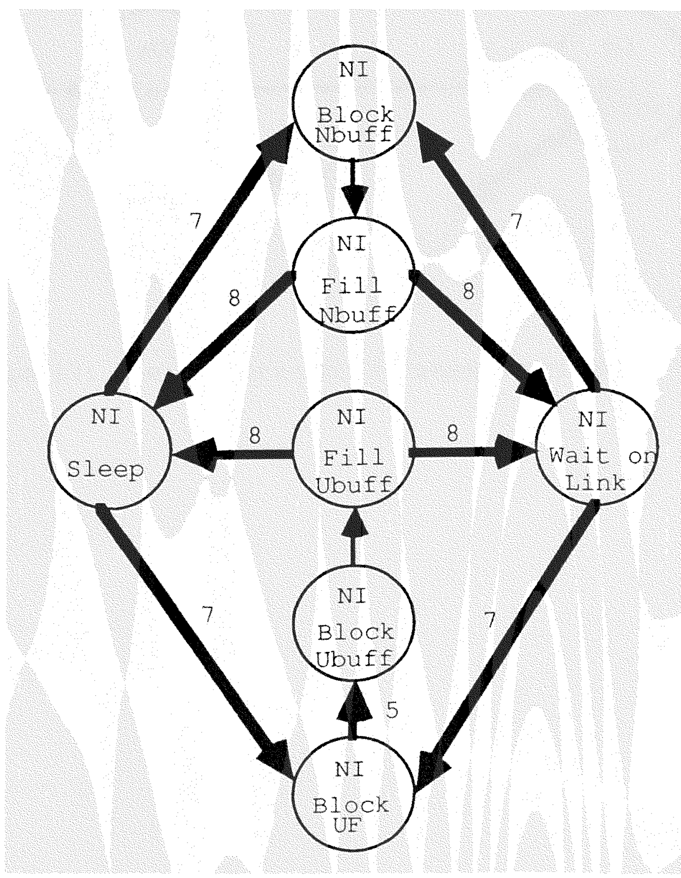

Figure 16: The Network-In (Server) State Diagram.

The Network-Out (Transmitter)

The States:

16. NO.Sleep ---------> link to next node is empty
17. NO.Busy ----------> link to next node is full
18. NO.Fill.Nlink ----> a word is being put on link

Bound Event Actions:

9. NO.Xfer ----------> a word arrived on link
10. NO.Received ------> the word on link was removed

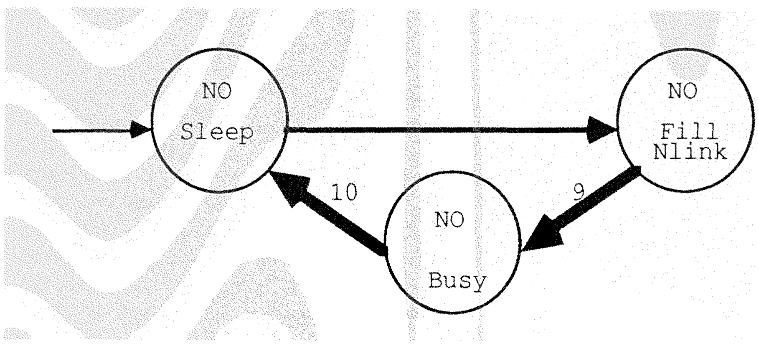

Figure 17: The Network-Out (Transmitter) State Diagram.

APPENDIX B. THE NETWORK COMMUNICATION CODE

```
PROC net.server(CHAN from.host, to.host, from.prev.node, to.next.node)
  VAL number.of.processors IS 4:
  VAL  max.msg.size IS 18:
  {{{ dcls
  {{{ channels connected to users
  CHAN OF INT user.to.front:
  CHAN OF INT user.to.server:
  CHAN OF INT front.to.user:
  }}}

  }}}

  PAR
    {{{ node.server processes (3 processes)
    {{{ channel dcls
    CHAN OF INT server.kill.user.front:
    CHAN OF INT server.kill.sender:
    CHAN OF INT server.to.user.front:
    CHAN OF INT msg.request:
    CHAN OF INT from.overflow:
    CHAN OF INT server.to.sender:
    }}}

    {{{ network msg header opcode definition
    VAL xfer        IS 0:
    VAL config      IS 1:
    VAL term        IS 2:
    VAL config.done IS 3:
    VAL term.done   IS 4:
    VAL ring.token  IS 5:
    VAL broadcast   IS 6:
    }}}

    {{{ misc definitions
    VAL  prog.start IS "program started*n*c":
    BYTE testch:
    }}}


    {{{ debug dcl
    }}}


    PAR
      {{{ user.front
```

```
                -- process to maintain buffer
{{{ dcl
VAL buff.size IS 5000:
[buff.size]INT buff:
INT next.slot, count:
INT next.data:
BOOL done:
BOOL consumer.waiting:
INT msg:
INT req.token:
INT quit.token:

to.consumer       IS front.to.user:
from.producer     IS server.to.user.front:
consumer.request  IS user.to.front:
quit              IS server.kill.user.front:
BOOL msg.hanging:
}}}


SEQ
  done:=FALSE
  consumer.waiting:=FALSE
  msg.hanging:=FALSE
  count:=0
  next.slot:=0

  WHILE NOT done
    PRI ALT
      (NOT msg.hanging) & from.producer ? msg
        {{{ get a msg and pass along if consumer is waiting
        SEQ
          {{{ COMMENT trace Fr
          }}}


          IF
            consumer.waiting
              SEQ
                to.consumer ! msg
                consumer.waiting:=FALSE
                -- done:=msg=stop.flag
            TRUE
              IF
                count < buff.size
                  {{{ insert into buff
                  SEQ
                    buff[next.slot]:=msg
                    next.slot:=next.slot+1
                    IF
                      next.slot=buff.size
                        next.slot:=0
                      TRUE
                        SKIP
```

```
                    count:=count+1
                  }}}

                TRUE
                  SEQ
                    msg.hanging:=TRUE
              -- ENDIF
        }}}

        consumer.request ? req.token
          {{{ pass a msg to consumer if one is available
          SEQ
            IF
              count=0    -- There are no msgs available
                consumer.waiting:=TRUE
              TRUE
                SEQ
                  next.data:=next.slot-count
                  IF
                    next.data<0
                      next.data:=next.data+buff.size
                    TRUE
                      SKIP
                  -- ENDIF
                  to.consumer ! buff[next.data]
                  -- done:=buff[next.data]=stop.flag
                  count:=count-1
                  IF
                    msg.hanging
                      {{{ insert the hanging msg into buff
                      SEQ
                        buff[next.slot]:=msg
                        count:=count+1
                        next.slot:=next.slot+1
                        IF
                          next.slot=buff.size
                            next.slot:=0
                          TRUE
                            SKIP
                        msg.hanging:=FALSE
                      }}}

                    TRUE
                      SKIP
            -- ENDIF
          }}}

        quit ? quit.token
          done:=TRUE
    {{{ COMMENT trace
    }}}
```

```
}}}

{{{ server
{{{ dcl
to.user          IS server.to.user.front:
from.user        IS user.to.server:
overflow         IS from.overflow:
to.sender        IS server.to.sender:
kill.user.front  IS server.kill.user.front:
kill.sender      IS server.kill.sender:

INT  my.addr:
BOOL configured:
INT  opcode:
INT  dest:
INT  msg.header:
INT  msg:
INT  msg.size:
BOOL run:
BOOL out.channel.avail:
BOOL terminating:
INT  kill.token:
INT  req.token:

INT buff.count, next.slot, next.data:
VAL buff.size       IS 2000:
VAL limit             IS ((buff.size+1) - (2*max.msg.size)):
[buff.size]INT buff:
-- [1970]INT dummybuff:

PROC decode(INT msg, opcode, dest, size)
  SEQ
     opcode:=(msg BITAND #F0000000) >> 28
     dest   :=(msg BITAND #0FF00000) >> 20
     size   :=(msg BITAND #000FFFFF)
:
PROC make.net.header(VAL INT opcode,dest,size, INT header)
  header:=(( opcode << 28 ) BITOR ( dest << 20 )) BITOR size
:

PROC wait.for.out.channel()
  IF
     NOT out.channel.avail
       msg.request ? req.token
    TRUE
       out.channel.avail:=FALSE
:
}}}

{{{ buff routine dcls
PROC insert.buff(INT msg)
  SEQ
    IF
       buff.count<buff.size  -- put the msg  into buff
```

```
{{{ put into buff
SEQ
   buff[next.slot]:=msg
   --next.slot:=((next.slot + 1) BITAND indxmask )
   next.slot:=next.slot+1
  IF
     next.slot=buff.size
       next.slot:=0
     TRUE
       SKIP
   buff.count:=buff.count+1
}}}

TRUE    -- buffer is full , wait for room
{{{
SEQ
  IF
     NOT out.channel.avail
       msg.request ? req.token
     TRUE
       out.channel.avail:=FALSE

   next.data:=next.slot - buff.count
  IF
     next.data<0
        next.data:=next.data + buff.size
     TRUE
       SKIP
   to.sender ! buff[next.data]
   buff[next.slot]:=msg
   next.slot:=next.slot+1
  IF
     next.slot=buff.size
       next.slot:=0
     TRUE
       SKIP
   --next.slot:=((next.slot + 1) BITAND indxmask )
   -- buff.count is not changed
}}}

:

PROC insert.a.msg(INT msg)
 SEQ
    buff[next.slot]:=msg
    next.slot:=next.slot+1
   IF
      next.slot=buff.size
        next.slot:=0
     TRUE
       SKIP
    --next.slot:=((next.slot + 1) BITAND indxmask )
    -- indxmask= buff.size-1
    buff.count:=buff.count+1
```

```
:

 PROC send.a.msg()
  SEQ
    next.data:=next.slot-buff.count
    IF
      next.data<0
        next.data:=next.data+buff.size
      TRUE
        SKIP
    to.sender ! buff[next.data]
    buff.count:=buff.count-1
    out.channel.avail:=FALSE
:

 PROC try.to.send.msg()
  IF
    buff.count>0
      send.a.msg()
    TRUE
      out.channel.avail:=TRUE
:
}}}


SEQ
  {{{ ini
  configured:=FALSE
  run:=TRUE
  out.channel.avail:=FALSE
  terminating:=FALSE
  my.addr:=1

  buff.count:=0
  next.slot:=0
  }}}


  WHILE run
    ALT
      msg.request    ? req.token
        {{{
        SEQ
          IF
            buff.count>0
              send.a.msg()
            TRUE
              out.channel.avail:=TRUE
        }}}

        -- try.to.send.msg()
      from.prev.node ? msg
        {{{
        SEQ
```

```
 decode(msg,opcode,dest,msg.size)
IF
   opcode=xfer
    {{{
    --INT d:
    SEQ
     --  d:=dummybuff[3]
      IF
         dest=my.addr
           {{{ transfer the whole msg to local user
          SEQ
             WHILE msg.size>0
              ALT
                  from.prev.node ? msg
                   SEQ
                      to.user ! msg
                       msg.size:=msg.size-1
                  msg.request ? req.token
                    try.to.send.msg()

          }}}

         (dest<>0) OR (my.addr<>1)
          {{{  put msg into buff and
                -- try to empty the buff at same time
          SEQ
             {{{ put msg into buff
            IF
               out.channel.avail
                SEQ
                   to.sender ! msg
                    out.channel.avail:=FALSE
              TRUE
                 insert.buff(msg)
            }}}

             WHILE (msg.size>0)
              ALT

                  from.prev.node ? msg
                   SEQ
                      {{{ put msg into buff
                     IF
                        out.channel.avail
                         SEQ
                            to.sender ! msg
                             out.channel.avail:=FALSE
                       TRUE
                          insert.buff(msg)
                     }}}

                      msg.size:=msg.size-1

                  msg.request ? req.token
```

```
                {{{
                SEQ
                  IF
                     buff.count>0
                       send.a.msg()
                     TRUE
                        out.channel.avail:=TRUE
                }}}

                 -- try.to.send.msg()
        }}}

      TRUE
         {{{ pass the whole msg to host
        SEQ
           to.host ! msg
           WHILE msg.size>0
             ALT
                from.prev.node ? msg
                 SEQ
                    to.host ! msg
                     msg.size:=msg.size-1
                msg.request ? req.token
                  try.to.send.msg()
        }}}

  }}}

opcode=broadcast
 {{{
 SEQ
   IF
       my.addr<number.of.processors
        {{{ put msg into buff
       IF
          out.channel.avail
           SEQ
              to.sender ! msg
               out.channel.avail:=FALSE
         TRUE
            insert.buff(msg)
       }}}

     TRUE
       SKIP

    WHILE (msg.size>0)
     ALT

        from.prev.node ? msg
         SEQ
            to.user ! msg
            IF
               my.addr<number.of.processors
```

```
                 {{{ put msg into buff
                 IF
                   out.channel.avail
                     SEQ
                       to.sender ! msg
                       out.channel.avail:=FALSE
                   TRUE
                     insert.buff(msg)
                 }}}

               TRUE
                 SKIP
             msg.size:=msg.size-1

         msg.request ? req.token
           {{{
           SEQ
             IF
               buff.count>0
                 send.a.msg()
               TRUE
                 out.channel.avail:=TRUE
           }}}

           -- try.to.send.msg()
  }}}

opcode=config
  {{{
  SEQ
    IF
      NOT configured
        SEQ
          configured:=TRUE
          my.addr:=dest
          dest:=dest+1
          make.net.header(config,dest,0,msg.header)
          wait.for.out.channel()
          to.sender ! msg.header
      TRUE
        SEQ
          make.net.header(config.done,dest,0,msg.header)
          to.host ! msg.header
  }}}

opcode=term
  {{{
  SEQ
    IF
      NOT terminating
        SEQ
          wait.for.out.channel()
          to.sender ! msg
      TRUE
```

```
              SEQ
                  make.net.header(term.done,0,0,msg.header)
                 to.host ! msg.header
            kill.user.front ! kill.token
            kill.sender ! kill.token
            run:=FALSE
          }}}

        TRUE
          SKIP


  }}}

  (buff.count < limit) & from.user ? dest
    {{{ take the user msg into the network
  SEQ
     from.user ? msg.size
       make.net.header(xfer,dest,msg.size,msg.header)
    IF
       (dest=0) AND (my.addr=1)
         {{{ pass the msg to host
        SEQ
           to.host ! msg.header
           SEQ i=0 FOR msg.size
             SEQ
               from.user ? msg
               to.host ! msg
        }}}

      TRUE
        {{{ get msg into buff
        SEQ
           insert.a.msg(msg.header)
           SEQ i=0 FOR msg.size
             SEQ
               from.user ? msg
               insert.a.msg(msg)
          IF
             out.channel.avail
               send.a.msg()
             TRUE
               SKIP
        }}}

  }}}

  (buff.count < limit) & from.host ? msg
    {{{ take the host msg into the network
  INT temp:
  SEQ
     decode(msg,opcode,dest,msg.size)
    IF
      opcode=xfer
```

```
SEQ
  {{{
  IF
      dest<>my.addr   -- my.addr is 1 in this case
        {{{  put the whole msg into buffer
       SEQ
          insert.a.msg(msg)       -- msg header
          SEQ i=0 FOR msg.size
           SEQ
             from.host ? msg
             insert.a.msg(msg)
         IF
           out.channel.avail
             send.a.msg()
           TRUE
             SKIP
       }}}

    TRUE
        {{{ transfer the whole msg to local user
       SEQ
          SEQ i=0 FOR msg.size
           SEQ
             from.host ? msg
             to.user ! msg
       }}}

  }}}

opcode=broadcast
  {{{  put the whole msg into buffer
 SEQ
    insert.a.msg(msg)       -- msg header
    SEQ i=0 FOR msg.size
     SEQ
       from.host ? msg
       insert.a.msg(msg)
       to.user ! msg
   IF
     out.channel.avail
       send.a.msg()
     TRUE
       SKIP
 }}}


opcode=config
 {{{
 SEQ
   my.addr:=dest
   dest:=dest+1
     make.net.header(config,dest,0,msg.header)
    wait.for.out.channel()
    to.sender ! msg.header
```

```
                    configured:=TRUE
                  }}}

                opcode=term
                  {{{
                  SEQ
                    terminating:=TRUE
                    wait.for.out.channel()
                    to.sender ! msg
                  }}}

               TRUE
                 SKIP


            }}}


   }}}

   {{{ sender / transmitter
   {{{ dcls
   from.server IS server.to.sender:
   quit         IS server.kill.sender:

   BOOL run:
   INT req.token, quit.token:
   INT msg:
   }}}


   SEQ
     run:=TRUE
     WHILE TRUE
       SEQ
         msg.request ! req.token
         from.server ? msg
         to.next.node ! msg
         {{{ COMMENT
         }}}

   }}}

{{{ channel dcl for user
get.msg             IS front.to.user:
request.msg         IS user.to.front:
send.msg            IS user.to.server:
}}}
}}}

{{{F usernode.tsr   (2 processes) *usernode.tsr
{{{ user msg header function code definitions
VAL data            IS 0:
VAL config          IS 1:
```

```
VAL config.done     IS 2:
VAL term            IS 3:
VAL term.done       IS 4:
VAL go              IS 5:
VAL test.done       IS 6:
}}}

{{{ user msg en/decoding procedures

PROC decode(INT msg, opcode, originator, size)
  SEQ
     opcode:=(msg BITAND #F0000000) >> 28
     originator:=(msg BITAND #0FF00000) >> 20
     size   :=(msg BITAND #000FFFFF)
:
PROC make.msg.header(VAL INT opcode,originator,size, INT header)
   header:=(( opcode << 28 ) BITOR ( originator << 20 )) BITOR size
:
PROC send.msg.header(VAL INT opcode,dest,size,originator)
  INT header:
  SEQ
     header:=(( opcode << 28 ) BITOR ( originator << 20 )) BITOR size
     send.msg ! dest
     send.msg ! (size+1)
     send.msg ! header
:
}}}
{{{ dcls
BOOL done:
INT msg, msg.header:
INT msg.size, opcode, orig:
BOOL done:
BYTE ch:
INT my.addr:
INT dest:

INT interval:
{{{
PROC delay( VAL INT interval )
  TIMER clock:
  INT timenow:
  SEQ
     clock ? timenow
     clock ? AFTER timenow PLUS interval
:
}}}

}}}
{{{ random number geneator abbreviations
VAL unif  IS 1: --uniform distribution.
VAL nexp  IS 2: --negative exponential distribution.
VAL const IS 3: --constant distribution.
VAL unifb IS 4: -- uniform with bound
```

```
VAL rn.init IS 1:
VAL rn.get  IS 2:
VAL rn.quit IS 3:
}}}
{{{ chan to rnd
 CHAN to.len.rand, from.len.rand:
 CHAN to.dest.rand, from.dest.rand:
}}}
 {{{ channels between user processes
CHAN control:
}}}


PAR
   {{{ User receiving messages
  SEQ
     interval:=5000
     done:=FALSE
     WHILE NOT done
       SEQ
          request.msg ! 0     -- ready to accept new msg
          get.msg ? msg.header
           decode(msg.header,opcode,orig,msg.size)
         IF
           opcode=data
              {{{ process data  (user read/eat mail & get fat!)
             SEQ
                SEQ i=0 FOR msg.size
                 SEQ
                    request.msg ! 0
                    get.msg ? msg

                {{{ COMMENT
                }}}

             }}}

           opcode=config
             {{{
             SEQ
                request.msg ! 0
                get.msg ? my.addr
                  -- send.msg.header(config.done,0,0,my.addr)
                control ! my.addr
             }}}

           opcode=go
             control ! 0
           opcode=term
             {{{ ternminate
             SEQ
                  -- send.msg.header(term.done,0,0,my.addr)
                done:=FALSE
                control ! 0
```

```
          }}}

}}}

{{{ User sending messages
{{{ dcl delay
PROC delay( VAL INT interval )
  TIMER clock:
  INT timenow:
  SEQ
    clock ? timenow
    clock ? AFTER timenow PLUS interval
:
}}}


TIMER clock:
INT start.time, finish.time:
INT my.addr, msg:
INT msg.size,dest:
SEQ

  control ? my.addr
  {{{ config
  SEQ
     send.msg.header(config.done,0,0,my.addr)
  {{{ COMMENT
  }}}

  }}}

  control ? msg -- go

  SEQ i=1 FOR 30000
    {{{ place messages into the network
    SEQ
      {{{ COMMENT
      }}}


      -- dest:= (( ( dest-1 ) + 3) \ 4)+1
      -- dest:=( my.addr REM 4 ) + 1
      SEQ
        dest := my.addr
        IF
          dest>4
            dest:=dest-4
          TRUE
            SKIP
      {{{ COMMENT
      }}}
      msg.size:=14
        send.msg.header(data,dest,msg.size,my.addr)
      SEQ j=0 FOR msg.size
```

```
            send.msg ! my.addr


          IF
            (i \ 1000)=0
              SEQ
                send.msg.header(data,0,1,my.addr)
                send.msg ! my.addr
                IF
                  i=20000
                    clock ? start.time
                  i=22000
                    clock ? finish.time
                  TRUE
                    SKIP

            TRUE
              SKIP
        }}}

        send.msg.header(data,0,8,my.addr)
        send.msg ! my.addr
        send.msg ! (finish.time-start.time)
        SEQ i=1 FOR 6
          send.msg ! ( (INT '=') - (INT '0'))

        send.msg.header(test.done,0,0,my.addr)
        control ? msg
        send.msg.header(term.done,0,0,my.addr)
    }}}
  }}}
:
```

APPENDIX C. THE SIMULATION CODE

```
PROC xnet (CHAN keyboard, screen)

  {{{ headers and declarations
  {{{F c:\janny\tdslibjr\header09.tsr *c:\janny\tdslibjr\header09.tsr
  ATTACHED
  }}}
  {{{F c:\janny\tdslibjr\ioproc06.tsr *c:\janny\tdslibjr\ioproc06.tsr
  ATTACHED
  }}}
  {{{F c:\janny\tdslibjr\ioint004.tsr *c:\janny\tdslibjr\ioint004.tsr
  ATTACHED
  }}}
  {{{F c:\janny\tdslibjr\ioreal39.tsr *c:\janny\tdslibjr\ioreal39.tsr
  ATTACHED
  }}}

  {{{ channels
  VAL max.sys.queues IS 129:    -- max # of queues needed
  VAL max.nodes IS 32:          -- max nodes in network
  VAL evs IS 0:                 -- the event set queue
  [max.nodes]INT nbuff, ubuff: -- the buffer queues
  [max.nodes]INT blockq,readyq:-- the operating system queues

  [5]CHAN to.rand,from.rand:
  [max.sys.queues]CHAN to.prq,from.prq:
  CHAN to.stats,from.stats:
  }}}

  {{{ random number stream names
  VAL proc.time IS 0:           -- user process time needed to do useful work
  VAL nbr.msgs IS 1:            -- number of msgs to create at once
  VAL msg.len IS 2:             -- length of a msg
  VAL os.time IS 3:             -- time for the operating system to run a proc
  VAL msg.dist IS 4:            -- the distance a msg should travel (in links)
  }}}

  {{{ action codes for the simulation

  -- BOUND event actions:
  VAL s.term IS 1:             -- terminate the simulation
  VAL ug.time.out IS 2:        -- user proc times out, context switch req.
  VAL ur.close.mail IS 3:      -- user proc finsihes reading a message
  VAL ug.xfer IS 4:            -- user proc moves a msg to the net buffer
  VAL uf.produce IS 5:         -- user front fills ubuff with word
  VAL uf.consume IS 6:         -- user front removes word from ubuff
  VAL ni.get.link IS 7:        -- net-in gets the msg just sent down the link
```

```
VAL ni.xfer IS 8:           -- net-in xferred word to the appropriate buffer
VAL no.xfer IS 9:           -- net-out is putting word on the link
VAL no.word.received IS 10: -- link is now free, word was removed

-- CONTINGENT event actions:
VAL ug.do.work IS 12:       -- user proc runs its application program
VAL ug.send.mail IS 13:     -- user proc places message in nbuff for xmit
VAL ur.get.mail IS 14:      -- user proc reads mail message
VAL uf.put.ubuff IS 15:     -- user front fill ubuff
VAL uf.get.ubuff IS 16:     -- user front removes word from ubuff
VAL ni.put.nbuff IS 17:     -- net-in proc places the word on link in nbuff
VAL ni.put.ubuff IS 18:     -- net-in proc places the word on link in ubuff
VAL no.send.word IS 19:     -- net-out proc places word of msg on link  link
}}}
{{{ function and distribution codes
{{{ distribution codes for the RNG
VAL invalid.distr IS 0:     -- invalid distribution type
VAL unif IS 1:              -- uniform distribution.
VAL nexp IS 2:              -- negative exponential distribution.
VAL const IS 3:             -- constant distribution.
}}}
{{{ common function codes
VAL error IS -1:
VAL init IS 0:
VAL quit IS 1:
}}}
{{{ PRQ function codes
VAL sched IS 2:            -- put an entity id on the queue
VAL next IS 3:             -- get the next entity id from the queue
VAL dump IS 4:             -- print contents of queue
VAL length IS 5:           -- return lenght of queue
VAL view IS 6:             -- return next item without removing it from queue
}}}
{{{ RNG function codes
VAL rn.init IS 1:          -- initialize the random number generator
VAL rn.get  IS 2:          -- get the next random number
VAL rn.quit IS 3:          -- destroy random number generator
}}}
{{{ entity and stat function codes
VAL get IS 2:             -- get an entity id number for new entity
VAL put IS 3:             -- return the entity id number for reuse later
VAL enter IS 4:           -- enter a new state
VAL leave IS 5:           -- leave a current state
VAL reset IS 6:           -- reset the statistics
VAL cpu IS 8:             -- cpu statistics
VAL dmp IS 9:             -- dump statistics
}}}
{{{ function codes for the simulation
VAL sim.init IS 0:        -- start the simulation
VAL sim.sim IS 1:         -- run a block of the simulation
VAL sim.quit IS 2:        -- end the simulation
}}}
}}}
}}}
```

```
{{{ SC c:\janny\tdslibjr\sim\random
{{{F c:\janny\tdslibjr\sim\random06.tsr *c:\janny\tdslibjr\sim\random06.tsr
ATTACHED
}}}
}}}
{{{F c:\janny\tdslibjr\sim\random07.tsr *c:\janny\tdslibjr\sim\random07.tsr
ATTACHED
}}}
{{{ SC c:\janny\tdslibjr\sim\prq
{{{F c:\janny\tdslibjr\sim\prq00001.tsr *c:\janny\tdslibjr\sim\prq00001.tsr
ATTACHED
}}}
}}}
{{{F c:\janny\tdslibjr\sim\prqif002.tsr *c:\janny\tdslibjr\sim\prqif002.tsr
ATTACHED
}}}
{{{ SC c:\janny\tdslibjr\sim\stats
{{{F c:\janny\tdslibjr\sim\stats002.tsr *c:\janny\tdslibjr\sim\stats002.tsr
ATTACHED
}}}
}}}
{{{F c:\janny\tdslibjr\sim\statsi03.tsr *c:\janny\tdslibjr\sim\statsi03.tsr
ATTACHED
}}}

[23] INT params:
{{{ parameter map
max.msg.len IS params[0]:        -- max length of a message
max.nbuff IS params[2]:          -- max number of words nbuff can hold
max.ubuff IS params[3]:          -- max number of words ubuff can hold

n.blocks IS params[4]:           -- number of blocks to run
block.len IS params[5]:          -- length of each block
trace IS params[6]:              -- values of the trace

seed.msg.dist IS params[1]:      -- seed for msg dist -- # links a msg travels
seed.proc.time IS params[7]:     -- seed for the process time between gen msgs
seed.gen.msgs IS params[8]:      -- seed for the number of msgs being created
seed.msg.len IS params[9]:       -- seed for the length of msg being created
seed.ostime IS params[10]:       -- seed for the ostime (op sys delay)

mean.proc.time IS params[11]:    -- mean process time between generating msgs
mean.ostime IS params[12]:       -- mean sleep time for the receiver
mean.gen.msgs IS params[13]:     -- mean number of messages created at once
mean.msg.len IS params[14]:      -- mean length of a generated message
mean.msg.dist IS params[15]:     -- the mean number of links a msg travels

cwxmit IS params[16]:            -- the speed of the link
n.nodes IS params[17]:           -- the number of nodes in the system

distr.proc.time IS params[18]:   -- distr type for local user process
distr.gen.msgs IS params[19]:    -- distr type for # msgs to generate at once
distr.msg.len IS params[20]:     -- distr type for message length
```

```
distr.ostime IS params[21]:     -- distr type for operating system delay
distr.msg.dist IS params[22]:   -- distr type for # links a msg should travel
}}}

PROC xnetsim(VAL INT opus, INT clock)
  {{{ run the simulation
  {{{ states of the system
  VAL ug.think IS 0:          -- user proc is thinking/processing
  VAL ug.block IS 1:          -- user proc blocked waiting to read/send mail
  VAL ug.fill.nbuff IS 2:     -- user proc is filling nbuff with a msg

  VAL ur.block.uf IS 3:       -- user proc is waiting for user front to run
  VAL ur.block IS 4:          -- user proc is blocked to read mail
  VAL ur.read.mail IS 5:      -- user proc is reading a mail msg

  VAL uf.block IS 6:          -- user front process is not doing anything
  VAL uf.fill.ubuff IS 7:     -- user front process filling ubuff
  VAL uf.remove.ubuff IS 8:   -- user front process is removing from ubuff

  VAL ni.sleep IS 9:          -- net-in is sleeping, nothing on link to get
  VAL ni.block.nbuff IS 10:   -- net-in is blocked waiting for the nbuff
  VAL ni.block.ubuff IS 11:   -- net-in is blocked waiting for the ubuff
  VAL ni.block.uf IS 12:      -- net-in is waiting for user front to run
  VAL ni.wait.on.link IS 13:  -- net-in is waiting to receive a word on link
  VAL ni.fill.nbuff IS 14:    -- net-in moves the msg to nbuff from the link
  VAL ni.fill.ubuff IS 15:    -- net-in moves the msg to local ubuff

  VAL no.sleep IS 16:         -- net-out in idle state (link is not busy)
  VAL no.busy IS 17:          -- net-out in xmit state (link is busy)
  VAL no.fill.nlink IS 18:    -- net-out is filling the link with a word

  VAL msg.traffic IS 19:      -- msg header is in the system
  }}}
  {{{ states of the link
  VAL link.no.msg IS 0:       -- link is free
  VAL link.head.msg IS 1:     -- link holds the header of the msg
  VAL link.word.msg IS 2:     -- link holds a word of the msg
  }}}
  {{{ constants for testing
  VAL max.proc.time IS 3000:  -- max proc time before time slice

  VAL u.read.header IS 45:    -- time to read header (+ 10 for clock)
  VAL u.read.word IS 16:      -- time to read one word of a msg

  VAL u.word.gen IS 10:       -- time to generate one word of msg
  VAL u.header.gen IS 62:     -- time to generate the header,  (+10 clock)

  VAL u.put.h.nbuff IS 61:    -- time to put header in nbuff
  VAL u.put.w.nbuff IS 19:    -- time to put word in nbuff

  VAL uf.get IS 40:           -- time for user front to get next word
  VAL uf.put IS 28:           -- time for user front to put next word

  VAL ni.put.h.nbuff IS 47:   -- time to place header in nbuff
```

```
VAL ni.put.h.ubuff IS 42:     -- time to place header in ubuff

VAL ni.insert.msg.wait IS 32:-- time to place word in nbuff if full
VAL ni.insert.msg.no.wait IS 13: -- time to place word in nbuff if not full

VAL ni.put.w.nbuff IS 21:     -- time to place word in nbuff
VAL ni.put.w.ubuff IS 26:     -- time to place word in ubuff

VAL no.put.word IS 60:        -- time to put word on link
}}}

{{{ declarations

{{{ array declarations, vars for each node
[max.nodes]INT succ, prev:      -- holds successor previous node numbers
[max.nodes]INT ug,ur,uf,ni,no: -- the 5 processes for each node
[max.nodes]INT ug.state:        -- holds current state for ug process
[max.nodes]INT ur.state:        -- holds current state for ur process
[max.nodes]INT ni.state:        -- holds current state for ni process
[max.nodes]INT no.state:        -- holds current state for no process
[max.nodes]INT uf.state:        -- holds current state for the user front

[max.nodes]INT ni.rest.msg:     -- holds # wrds left to send/receive
[max.nodes]INT ni.block:        -- holds the buffer ni is currently blocking
[max.nodes]INT ni.decode:       -- holds the time to decode a msg header

[max.nodes]INT nlink:           -- holds the entity on the link
[max.nodes]INT nlink.online:    -- holds type of contents in nlink

[max.nodes]INT msg.header:      -- holds current header of msg being read
[max.nodes]INT no.sending.words: -- holds # words no is currently sending

[max.nodes]INT u.think.time:    -- holds the time for user proc to think
[max.nodes]INT u.send.nbr.msgs:-- holds # of msgs to send before thinking
[max.nodes]INT u.sending.words:-- holds the # words currently being sent
[max.nodes]INT u.reading.words:-- holds the # words currently being read

[max.nodes]INT ubuff.nwords:    -- holds the nbr of words in the ubuff
[max.nodes]INT nbuff.nwords:    -- holds the nbr of words in the nbuff
[max.nodes]INT ubuff.nheaders: -- holds the nbr of headers in the ubuff
[max.nodes]INT nbuff.nheaders: -- holds the nbr of headers in the nbuff


[max.nodes]BOOL proc.running:  -- hold true when process is running on node
[max.nodes]BOOL u.filling:     -- holds true when u is filling nbuff
[max.nodes]BOOL ni.filling:    -- holds true when ni is filling nbuff
}}}

INT term, sys, sid:            -- index to entity objects
INT word, header:              -- index to entity objects
INT dummy,prior:               -- prq params prior
INT len, blockq.len:           -- temp var for length of queue
INT dist,dest:                 -- distance and destination of a msg
INT os:                        -- holds random operating system time delay
```

```
INT gen.msg.can.fit:          -- max length of nbuff so that g can add msg
INT send.nbr.msgs:            -- temp var to get random number
INT think:                    -- temp var & used to get random think time
INT read.time, send.time:     -- time to read or send a word of a msg
INT i, j:                     -- loop control vars
INT act, node, nbr.words:     -- holds values for an entity
INT ch:
INT stime, newtime:           -- used to hold clock time
TIMER realclock:              -- used for timing
INT etimer,stimer,ftimer:     -- more timers
REAL32 durance:
INT clock:
BOOL run:
}}}


{{{ entity control
{{{ entity object parameters

VAL maxent IS 20000:          -- the max entities in the system at once
VAL num.of.fields IS 5:       -- there are five fields in an entity
VAL maxstate IS 14:           -- the number of states in the system
VAL maxatr IS 4:              -- attributes: node.id, n.words, fdest

-- THE FIELDS OF THE STRUCTURE ENTITY:
VAL action IS 0:              -- the bound event action id
VAL link IS 1:                -- used to link entitites
VAL node.id IS 2:             -- the node associated with the entity
VAL n.words IS 3:             -- number of words in the msg / with header
VAL fdest IS 4:               -- holds the node to receive the msg

-- THE STRUCTURE ENTITY:
[maxent][num.of.fields]INT entity:  -- the storage for the entities

}}}
{{{F c:\janny\tdslibjr\sim\entitys.tsr *c:\janny\tdslibjr\sim\entitys.tsr
ATTACHED
}}}
}}}

SEQ
  IF
    opus = sim.init
      {{{ initialize the model
      SEQ
        {{{ initialize the entity object
        ent (init, sys)
        }}}

        {{{ determine the size of the network buffer less contingency part
            -- at least 2 maximum size msgs must be able to fit in the
            -- to insure that when a msg is placed in the network buffer
            -- there is still room for 1 max size msg.
        gen.msg.can.fit := (max.nbuff + 1) - (2 * params[0])
```

```
}}}

{{{ set the order of nodes in the system (successor, previous)
SEQ
                     -- compute the SUCCESSOR of every node
  node := 0
  succ[n.nodes - 1] := 0
  WHILE node < (n.nodes - 1)
    SEQ
      succ[node] := node + 1
      node := node + 1

                     -- compute the PREVIOUS node for every node
  prev[0] := n.nodes - 1
  node := 1
  WHILE node < (n.nodes)
    SEQ
      prev[node] := node - 1
      node := node + 1
}}}
{{{ create control entity term
SEQ
  ent (get, term)
  entity[term][action] := s.term      -- mark the termination point
}}}
{{{ schedule first user proc time out, init buffers and counters
SEQ
  think := 0          -- think for time 0 in order to get on evs
  node := 0           -- all contingent tests will fail/proc will run
  WHILE (node < n.nodes) -- for all nodes
    SEQ
      {{{ Schedule the first time out for the user procs
      SEQ
        ent(get,sys)                          -- get an entity id
        entity[sys][node.id] := node          -- set its node id
        entity[sys][action] := ug.time.out    -- set act to gen mail
        prq(sched,evs,sys,think,(trace/\2))  -- schedule time out
        proc.running[node] := TRUE            -- note proc is running
      }}}
      {{{ Initialize buffer and link counters
      SEQ
        -- initialize the buffer counters to zero
        nbuff.nwords[node] := 0
        ubuff.nwords[node] := 0
        nbuff.nheaders[node] := 0
        ubuff.nheaders[node] := 0

        -- Initialize link marker to zero (nothin on link)
        nlink.online[node] := link.no.msg

        -- Initialize the msg counters for user processes
        u.think.time[node] := 0
        u.send.nbr.msgs[node] := 0
        u.sending.words[node] := 0
```

```
        u.reading.words[node] := 0
        no.sending.words[node] := 0
        ni.rest.msg[node] := 0

        -- Initialzie the boolean flags for user & net-in filling
             -- network buffer
        u.filling[node] := FALSE
        ni.filling[node] := FALSE
      }}}
      node := node + 1              -- increment counter

}}}
{{{ schedule first block end
SEQ
  newtime := block.len              -- set newtime to end block
  -- schedule the termination action at time newtime
  prq (sched,evs,term,newtime,(trace/\2))

}}}
{{{ initialize the simulation clock
clock :=  0                         -- set clock to time zero
}}}

{{{ create the set of network processes (servers)
SEQ
  node := 0
  prior := 0
  WHILE (node < n.nodes)
    SEQ
      {{{ Create the user process (ug)
           -- get the entity id, assign it to this process, assign
           -- the node this id, let stats know start state, and
           -- assign the process to the start state.
      SEQ
        ent(get,sys)
        ug[node] := sys
        entity[sys][node.id] := node
        ens(get,sys,ug.think,prior,(trace/\16))

        ug.state[node] := ug.think
      }}}
      {{{ Create the user process (ur)
          -- get the entity id, assign it to this process, assign
          -- the node this id, let stats know start state, and
          -- assign the process to the start state, and place
          -- process on Block queue

      SEQ
        ent(get,sys)
        ur[node] := sys
        entity[sys][node.id] := node
        ens(get,sys,ur.block.uf,prior,(trace/\16))
        ur.state[node] := ur.block.uf
        prq(sched,blockq[node],sys,clock,(trace/\4))
```

```
        }}}

        {{{ Create the user front process (uf)
            -- get the entity id, assign it to this process, assign
            -- the node this id, let stats know start state, and
            -- assign the process to the start state, and place
            -- process on Block queue

        SEQ
          ent(get,sys)
          uf[node] := sys
          entity[sys][node.id] := node
          ens(get,sys,uf.block,prior,(trace/\16))
          uf.state[node] := uf.block
          prq(sched,blockq[node],sys,clock,(trace/\4))
        }}}

        {{{ Create the network receiver (ni)
            -- get the entity id, assign it to this process, assign
            -- the node this id, let stats know start state, and
            -- assign the process to the start state, and place
            -- process on Block queue

        SEQ
          ent(get,sys)
          ni[node] := sys
          entity[sys][node.id] := node
          ens(get,sys,ni.sleep,prior,(trace/\16))
          ni.state[node] := ni.sleep
          prq(sched,blockq[node],sys,clock,(trace/\4))
        }}}

        {{{ Create the network transmitter (no)
            -- get the entity id, assign it to this process, assign
            -- the node this id, let stats know start state, and
            -- assign the process to the start state, and place
            -- process on Block queue

        SEQ
          ent(get,sys)
          no[node] := sys
          entity[sys][node.id] := node
          ens(get,sys,no.sleep,prior,(trace/\16))
          no.state[node] := no.sleep
          prq(sched,blockq[node],sys,clock,(trace/\4))
        }}}
        node := node + 1
    }}}

  }}}

opus = sim.sim
```

```
{{{ run one block
SEQ
  realclock ? stimer
  ens (reset,dummy,dummy,clock,(trace/\16))
  run := TRUE
  WHILE run
    SEQ
      {{{ get next event, action and node
      SEQ
        prq(next,evs,sid,clock,(trace/\2))   -- get next event notice
        act := entity[sid][action]           -- get the action id
        node := entity[sid][node.id]         -- get node it is for
        {{{ if trace/\1 print action         -- trace if necessary
        IF
          (trace /\ 1) <> 0
            SEQ
              IF
                act = s.term
                  write.full.string(screen,"block end ")
                act = ur.close.mail
                  write.full.string(screen,
                    " user process closes mail msg")
                act = ug.time.out
                  write.full.string(screen,
                    "user process just timed-out")
                act = ug.xfer
                  write.full.string(screen,
                    "user just moved msg to nbuff")
                act = uf.produce
                  write.full.string(screen,
                    "user front just filled ubuff")
                act = uf.consume
                  write.full.string(screen,
                    "user front just removed word from ubuff")
                act = ni.get.link
                  write.full.string(screen,
                    " net process received a word on the link")
                act = ni.xfer
                  write.full.string(screen,
                    " net process removed a word from the link")
                act = no.xfer
                  write.full.string(screen,
                    " net process just filled link")
                act = no.word.received
                  write.full.string(screen,
                    " word was removed from link")
                TRUE
                  write.full.string(screen,"@!%&@#!@ ")
              write.full.string(screen," with id ")
              INTwrite(sid,4)
              write.full.string(screen," at time ")
              INTwrite(clock,6)
              write.full.string(screen,"*c*n")
          TRUE
```

```
        SKIP
    }}}
  }}}

  {{{ Process BOUND EVENTS
  IF
    act = ug.time.out
      {{{ time has expired for user process ug to run
      SEQ
        {{{ Kill the control entity
        ent(put,sid)
        }}}
        {{{ Leave u.think state / enter u.block state
        SEQ
          ens(leave, ug[node], ug.think, clock, (trace/\16))
          ens(enter, ug[node], ug.block, clock, (trace/\16))
          ug.state[node] := ug.block
        }}}
        {{{ Move user process ug from proc.running to BLOCK Queue
        SEQ
          prq(sched,blockq[node],ug[node],clock,(trace/\4))
          proc.running[node] := FALSE
        }}}
      }}}
    act = ur.close.mail
      {{{ time has expired for user process ur to read a mail msg
      SEQ
        {{{ Kill the control entity
        ent(put,sid)
        }}}
        {{{ Leave ur.read.mail state / enter ur.block.uf state
        SEQ
          ens(leave, ur[node], ur.read.mail, clock, (trace/\16))
          ens(enter, ur[node], ur.block.uf, clock, (trace/\16))
          ur.state[node] := ur.block.uf
        }}}
        {{{ Move user process ur from proc.running to BLOCK Queue
        SEQ
          prq(sched,blockq[node],ur[node],clock,(trace/\4))
          proc.running[node] := FALSE
        }}}
        {{{ Leave msg.traffic state if last word of msg received
        IF
          u.reading.words[node] = 0
            SEQ
              ens(leave,msg.header[node],msg.traffic,
                     clock,(trace/\16))
              ent(put,msg.header[node])
          TRUE
            SKIP
        }}}
      }}}
    act = ug.xfer
      {{{ time expired for user process ug to fill nbuff w/msg
```

```
    SEQ
      {{{ Kill the control entity
      ent(put,sid)
      }}}
      {{{ Leave ug.fill.nbuff state / enter ug.block state
      SEQ
        ens(leave, ug[node], ug.fill.nbuff, clock, (trace/\16))
        ens(enter, ug[node], ug.block, clock, (trace/\16))
        ug.state[node] := ug.block
      }}}
      {{{ set u.filling false if last word of msg was xferred
      SEQ
        IF
          u.sending.words[node] = 0
            u.filling[node] := FALSE
          TRUE
            SKIP
      }}}
      {{{ Move user process ug from proc.running to BLOCK Queue
      SEQ
        prq(sched,blockq[node],ug[node],clock,(trace/\4))
        proc.running[node] := FALSE
      }}}
    }}}
  act = uf.produce
    {{{ time expired for user front to fill ubuff with word
    SEQ
      {{{ change uf state
      SEQ
        ens(leave,uf[node],uf.fill.ubuff,clock,(trace/\16))
        ens(enter,uf[node],uf.block,clock,(trace/\16))
        uf.state[node] := uf.block
      }}}
      {{{ change ni state(tell ni that it can fill ubuff now)
      SEQ
        ens(leave,ni[node],ni.block.uf,clock,(trace/\16))
        ens(enter,ni[node],ni.block.ubuff,clock,(trace/\16))
        ni.state[node] := ni.block.ubuff
      }}}
      {{{ kill control entity; wait on contingent event
      ent(put,sid)
      }}}
      {{{ move uf proc to block queue & set proc.running false
      SEQ
        prq(sched,blockq[node],uf[node],clock,(trace/\4))
        proc.running[node] := FALSE
      }}}
    }}}
  act = uf.consume
    {{{ time expired for user front to get word from ubuff
    SEQ
      {{{ change uf state
      SEQ
        ens(leave,uf[node],uf.remove.ubuff,clock,(trace/\16))
```

```
        ens(enter,uf[node],uf.block,clock,(trace/\16))
        uf.state[node] := uf.block
      }}}
      {{{ change ur state(move ur to block so it can read next)
      SEQ
        ens(leave,ur[node],ur.block.uf,clock,(trace/\16))
        ens(enter,ur[node],ur.block,clock,(trace/\16))
        ur.state[node] := ur.block
      }}}
      {{{ kill control entity; wait on contingent event
      ent(put,sid)
      }}}
      {{{ move uf to block queue and set proc.running to false
      SEQ
        prq(sched,blockq[node],uf[node],clock,(trace/\4))
        proc.running[node] := FALSE
      }}}
    }}}
  act = ni.get.link
    {{{ a word has arrived on the link
    SEQ
      IF
        ni.state[node] = ni.sleep
          {{{ message header on link
          SEQ
            {{{ get msg header & dest, set ni.rest.msg counter
            SEQ
              header := nlink[node]
              ni.rest.msg[node] := entity[header][n.words] + 1
            }}}
            IF
              entity[header][fdest] = node
                {{{ message is local  (let uf run first)
                SEQ
                  ni.block[node] := ni.block.uf
                  ni.decode[node] := ni.put.h.ubuff
                  ubuff.nheaders[node]:=ubuff.nheaders[node]+1
                  prq(sched,ubuff[node],header,
                    dummy,(trace/\4))
                }}}
              TRUE
                {{{ message is for another node
                SEQ
                  ni.block[node] := ni.block.nbuff
                  nbuff.nheaders[node]:=nbuff.nheaders[node]+1
                  prq(sched,nbuff[node],header,
                    dummy,(trace/\4))
                  IF
                    nbuff.nwords[node] = max.nbuff
                      ni.decode[node] := ni.put.h.nbuff +
                        ni.insert.msg.wait
                    TRUE
                      ni.decode[node] := ni.put.h.nbuff +
                        ni.insert.msg.no.wait
```

```
                  }}}
                {{{ enter block state
                SEQ
                  ens(leave,ni[node],ni.sleep,clock,(trace/\16))
                  ens(enter,ni[node],ni.block[node],
                    clock,(trace/\16))
                  ni.state[node] := ni.block[node]
                }}}
                {{{ kill control entity
                ent(put,sid)
                }}}
              }}}
            TRUE  -- ni is waiting on the link
              {{{ one word of the message is on link
              SEQ
                {{{ enter block state
                SEQ
                  ens(leave,ni[node],ni.wait.on.link,
                    clock,(trace/\16))
                  ens(enter,ni[node],ni.block[node],
                    clock,(trace/\16))
                  ni.state[node] := ni.block[node]
                }}}
                {{{ set time to decode word to time needed to place
                    -- word in nbuff or ubuff
                IF
                  ni.block[node] = ni.block.ubuff
                    ni.decode[node] := ni.put.w.ubuff
                  nbuff.nwords[node] = max.nbuff
                    ni.decode[node] := ni.put.w.nbuff +
                           ni.insert.msg.wait
                  TRUE
                    ni.decode[node] := ni.put.w.nbuff +
                           ni.insert.msg.no.wait
                }}}
                {{{ kill control entity
                ent(put,sid)
                }}}
              }}}
      }}}
    act = ni.xfer
      {{{ time expired to move a word from link to buffer
      SEQ
        IF
          ni.rest.msg[node] > 0
            {{{ message not complete, enter wait on link
            SEQ
              ens(leave,ni[node],ni.state[node],
                clock,(trace/\16))
              ens(enter,ni[node],ni.wait.on.link,
                clock,(trace/\16))
              ni.state[node] := ni.wait.on.link
            }}}
          TRUE
```

```
          {{{ complete message received, go back to sleep
          SEQ
            ni.filling[node] := FALSE
            ens(leave,ni[node],ni.state[node],
              clock,(trace/\16))
            ens(enter,ni[node],ni.sleep,clock,(trace/\16))
            ni.state[node] := ni.sleep
          }}}
      {{{ kill control entity (prev node will send a get.link)
      ent(put,sid)
      }}}
      {{{ move net-in process from proc.running to BLOCK Queue
      SEQ
        prq(sched,blockq[node],ni[node],clock,(trace/\4))
        proc.running[node] := FALSE
      }}}
    }}}
  act = no.xfer
    {{{ time expired to move a word from buffer to link
    SEQ
      {{{ schedule next node to receive word
      SEQ
        {{{ set control entity, next node to get word off link
        SEQ
          entity[sid][action] := ni.get.link
          entity[sid][node.id] := succ[node]
        }}}
        {{{ compute time to transmit down the line
        SEQ
          rng.get(os.time, os, (trace/\32))
          newtime := (os + cwxmit) + clock
        }}}
        {{{ schedule the control entity
        prq(sched,evs,sid,newtime,(trace/\2))
        }}}
      }}}
      {{{ leave no.fill.nlink state / enter no.busy state
      SEQ
        ens(leave,no[node],no.fill.nlink,clock,(trace/\16))
        ens(enter,no[node],no.busy,clock,(trace/\16))
        no.state[node] := no.busy
      }}}
      {{{ move net-out process from proc.running to BLOCK Queue
      SEQ
        prq(sched,blockq[node],no[node],clock,(trace/\4))
        proc.running[node] := FALSE
      }}}
    }}}
  act = no.word.received
    {{{ successor node received the word on link
    SEQ
      nlink.online[succ[node]] := link.no.msg
      ens(leave,no[node],no.busy,clock,(trace/\16))
      ens(enter,no[node],no.sleep,clock,(trace/\16))
```

```
        no.state[node] := no.sleep
        ent(put,sid)
      }}}

    act = s.term
      {{{ end this block
      SEQ
        run := FALSE
        newtime := clock + block.len
        ens(cpu,dummy,dummy,clock,(trace/\16))
        prq (sched, evs, sid, newtime,(trace/\2))
      }}}
    TRUE
      {{{ illegal control code
      SEQ
        write.full.string(screen,"Illegal control code*c*n")
        STOP
      }}}
  }}}

  IF
    act <> s.term
      {{{ Process CONTINGENT EVENTS
      SEQ
        {{{ update BLOCK and READY Queues
        prq(length,blockq[node],blockq.len,dummy,(trace/\4))
        i := 0
        WHILE  i < blockq.len   -- do all items on block queue
          SEQ
            prq(next,blockq[node],sid,dummy,(trace/\4))
            IF
              sid = ug[node]
                {{{ update user process generator
                IF
                  u.think.time[node] > 0
                    {{{ run before sending mail
                    SEQ
                      entity[sid][action] := ug.do.work
                      prq(sched,readyq[node],sid,
                        clock,(trace/\4))
                    }}}
                  u.sending.words[node] > 0
                    {{{ currently sending a message
                    SEQ
                      entity[sid][action] := ug.send.mail
                      prq(sched,readyq[node],sid,
                        clock,(trace/\4))
                    }}}
                  u.send.nbr.msgs[node] > 0
                    {{{ send start of new mail message or block
                    IF
                      ((NOT ni.filling[node])  AND
                        (nbuff.nwords[node] < gen.msg.can.fit))
                        SEQ
```

```
            u.filling[node] := TRUE
            entity[sid][action] := ug.send.mail
            prq(sched,readyq[node],sid,
              clock,(trace/\4))
          TRUE --msg can't be moved,go on BLOCK Queue
            prq(sched,blockq[node],sid,
              clock,(trace/\4))
        }}}
      TRUE
        {{{ run the application program
        SEQ
          entity[sid][action] := ug.do.work
          prq(sched,readyq[node],sid,
            clock,(trace/\4))

          rng.get(nbr.msgs, len,(trace/\32))
          u.send.nbr.msgs[node] := len

          rng.get(proc.time,think,(trace/\32))
          u.think.time[node] := think
        }}}
    }}}
  sid = ur[node]
    {{{ update user process receiver
    SEQ
      IF
        ur.state[node] = ur.block
          {{{ move ur to READY queue
          SEQ
            entity[sid][action] := ur.get.mail
            prq(sched,readyq[node],sid,
              clock,(trace/\4))
          }}}
        TRUE
          {{{ place ur back on Block Queue
          prq(sched,blockq[node],sid,
            clock,(trace/\4))
          }}}
    }}}
  sid = uf[node]
    {{{ update user front process
    SEQ
      IF
        ni.state[node]=ni.block.uf -- ni has priority
          SEQ
            entity[sid][action] := uf.put.ubuff
            prq(sched,readyq[node],sid,
              clock,(trace/\4))
        (ur.state[node] = ur.block.uf) AND
           (ubuff.nwords[node] > 0)
          SEQ
            entity[sid][action] := uf.get.ubuff
            prq(sched,readyq[node],sid,
              clock,(trace/\4))
```

```
          TRUE
            prq(sched,blockq[node],sid,
              clock,(trace/\4))
      }}}
    sid = ni[node]
      {{{ update net-in process
      IF
        (((ni.state[node] = ni.block.ubuff) AND
          (ubuff.nwords[node] < max.ubuff)) AND
          (NOT u.filling[node]))
          SEQ
            ni.filling[node] := TRUE
            entity[sid][action] := ni.put.ubuff
            prq(sched,readyq[node],sid,
              clock,(trace/\4))
        (((ni.state[node] = ni.block.nbuff) AND
          (nbuff.nwords[node] < max.nbuff)) AND
          (NOT u.filling[node]))

          SEQ
            ni.filling[node] := TRUE
            entity[sid][action] := ni.put.nbuff
            prq(sched,readyq[node],sid,
              clock,(trace/\4))
        TRUE -- net-in can't run,go back on BLOCK Queue
          prq(sched,blockq[node],sid,clock,(trace/\4))
      }}}
    sid = no[node]
      {{{ update net-out process
      IF
        ((nlink.online[succ[node]] = link.no.msg)  AND
          (nbuff.nwords[node] > 0))
          SEQ
            entity[sid][action] := no.send.word
            prq(sched,readyq[node],sid,
              clock,(trace/\4))
        TRUE
          prq(sched,blockq[node],sid,clock,(trace/\4))
      }}}
    TRUE
      write.full.string(screen,
        "Illegal control entity on BLOCK queue*c*n")
  i := i + 1
}}}
{{{ set one process running if necessary/possible
prq(length,readyq[node],len,dummy,(trace/\4))
IF
  (len > 0) AND (proc.running[node] = FALSE)
    SEQ
      {{{ get next action and proc,set proc.running TRUE
      SEQ
        prq(next,readyq[node],sid,len,(trace/\4))
        act := entity[sid][action]
        proc.running[node] := TRUE
```

```
}}}
{{{ perform the contingent event
IF
  act = ug.do.work
    {{{ set user proc running its application
    SEQ
      {{{ Determine operating system delay for run
      SEQ
        rng.get(os.time, os, (trace/\32))
      }}}
      {{{ Determine time to run before u.time.out
      SEQ
        IF
          u.think.time[node] > max.proc.time
            SEQ
              newtime := (max.proc.time + os)+clock
              u.think.time[node]:=
                u.think.time[node] - max.proc.time
          TRUE
            SEQ
              newtime := (u.think.time[node] + os)+
                clock
              u.think.time[node] := 0
      }}}
      {{{ Create control entity &
          --Schedule u.time.out
      SEQ
        ent(get,sys)
        entity[sys][node.id] := node
        entity[sys][action] := ug.time.out
        prq(sched, evs, sys, newtime, (trace/\2))
      }}}
      {{{ Leave ug.block / enter ug.think state
      SEQ
        ens(leave,ug[node],ug.block,
          clock,(trace/\16))
        ens(enter,ug[node],ug.think,
          clock,(trace/\16))
        ug.state[node] := ug.think
      }}}
    }}}
  act = ug.send.mail
    {{{ let user process fill nbuff with mail
    SEQ
      IF
        u.sending.words[node] = 0
          {{{ starting new msg
          SEQ
            {{{ Generate random length of msg
            SEQ
              rng.get(msg.len, len, (trace/\32))
            }}}
            {{{ Generate distance the msg should
                -- travel (number of links)
```

```
        SEQ
          rng.get(msg.dist, dist, (trace/\32))
          dest := (node + dist) REM n.nodes
        }}}
        {{{ Create MSG HEADER
        SEQ
          ent(get, header)
          entity[header][fdest] := dest
          entity[header][n.words] := len
            -- does not include header
          ens(enter,header,msg.traffic,
            clock,(trace/\16))
        }}}
        {{{ Place msg in nbuff (header in
            -- nbuff, update buffer counters)
        SEQ
          prq(sched, nbuff[node], header,
            prior, (trace/\4))
          nbuff.nheaders[node] :=
            nbuff.nheaders[node]+1
          nbuff.nwords[node] :=
            nbuff.nwords[node]+1
        }}}
        {{{ Update counters
        SEQ
          u.send.nbr.msgs[node] :=
            u.send.nbr.msgs[node]-1
          u.sending.words[node] := len
        }}}
        u.think.time[node] := u.header.gen
        send.time := u.put.h.nbuff
      }}}
    TRUE
      {{{ currently sending words of a msg
      SEQ
        nbuff.nwords[node] :=
          nbuff.nwords[node] + 1
        u.sending.words[node] :=
              u.sending.words[node]-1
        u.think.time[node] := u.word.gen
        send.time := u.put.w.nbuff
      }}}
{{{ Determine time to move msg to nbuff
SEQ
  rng.get(os.time,os,(trace/\32))
  newtime := (send.time + os) + clock
}}}
{{{ Create control entity & Schedule xfer
SEQ
  ent(get,sys)
  entity[sys][node.id] := node
  entity[sys][action] := ug.xfer
  prq(sched,evs,sys,newtime,(trace/\2))
}}}
```

```
      {{{ Leave ug.block / Enter ug.fill.nbuff
      SEQ
        ens(leave,ug[node],ug.block,
          clock,(trace/\16))
        ens(enter,ug[node],ug.fill.nbuff,
          clock,(trace/\16))
        ug.state[node] := ug.fill.nbuff
      }}}
    }}}
  act = ur.get.mail
    {{{ let user process read mail msg waiting
    SEQ
      IF
        u.reading.words[node] = 0
          {{{ get next header from ubuff, set
              -- counters, set read time
          SEQ
            prq(next,ubuff[node],header,
              dummy,(trace/\4))
            u.reading.words[node]:=
              entity[header][n.words]
            ubuff.nwords[node] :=
              ubuff.nwords[node] - 1
            ubuff.nheaders[node]:=
              ubuff.nheaders[node] - 1
            msg.header[node] := header
            -- ent(put,header)
            read.time := u.read.header
          }}}
        TRUE
          {{{ update counters, set read time
          SEQ
            ubuff.nwords[node] :=
              ubuff.nwords[node] - 1
            u.reading.words[node]:=
                   u.reading.words[node]-1
            read.time := u.read.word
          }}}
      {{{ compute time to read the msg
      SEQ
        rng.get(os.time,os,(trace/\32))
        newtime := (read.time + os) + clock
      }}}
      {{{ create control entity & schedule transfer
      SEQ
        ent(get,sys)
        entity[sys][node.id] := node
        entity[sys][action] := ur.close.mail
        prq(sched,evs,sys,newtime,(trace/\2))
      }}}
      {{{ leave ur.block / enter ur.read.mail state
      SEQ
        ens(leave,ur[node],ur.block,
          clock,(trace/\16))
```

```
      ens(enter,ur[node],ur.read.mail,
        clock,(trace/\16))
      ur.state[node] := ur.read.mail
    }}}
  }}}
act = uf.get.ubuff
  {{{ let user front proc get next word in ubuff
  SEQ
    {{{ compute time to read the msg
    SEQ
      rng.get(os.time,os,(trace/\32))
      newtime := (uf.get + os) + clock
    }}}
    {{{ create control entity & schedule transfer
    SEQ
      ent(get,sys)
      entity[sys][node.id] := node
      entity[sys][action] := uf.consume
      prq(sched,evs,sys,newtime,(trace/\2))
    }}}
    {{{ leave uf.block / enter uf.remove.ubuff
    SEQ
      ens(leave,uf[node],uf.block,
        clock,(trace/\16))
      ens(enter,uf[node],uf.remove.ubuff,
        clock,(trace/\16))
      uf.state[node] := uf.remove.ubuff
    }}}
  }}}
act = uf.put.ubuff
  {{{ let user front proc put next word in ubuff
  SEQ
    {{{ Create control entity to transfer word
    SEQ
      ent(get,sys)
      entity[sys][node.id] := node
      entity[sys][action] := uf.produce
    }}}
    {{{ Determine time needed to make transfer
    SEQ
      rng.get(os.time,os,(trace/\32))
      newtime := (uf.put + os) + clock
    }}}
    {{{ Schedule the transfer
    prq(sched,evs,sys,newtime,(trace/\2))
    }}}
    {{{ leave uf.block / enter uf.fill.ubuff
    SEQ
      ens(leave,uf[node],uf.block,
        clock,(trace/\16))
      ens(enter,uf[node],uf.fill.ubuff,
        clock,(trace/\16))
      uf.state[node] := uf.fill.ubuff
    }}}
```

```
  }}}
act = ni.put.nbuff
  {{{ let net-in proc fill nbuff w/ word on link
  SEQ
    {{{ Create control entity to transfer word
    SEQ
      ent(get,sys)
      entity[sys][node.id] := node
      entity[sys][action] := ni.xfer
    }}}
    {{{ Determine time needed to make transfer
    SEQ
      rng.get(os.time,os,(trace/\32))
      newtime := (ni.decode[node] + os) + clock
    }}}
    {{{ Schedule the transfer
    prq(sched,evs,sys,newtime,(trace/\2))
    }}}
    {{{ update word counters
    SEQ
      nbuff.nwords[node] := nbuff.nwords[node]+1
      ni.rest.msg[node] := ni.rest.msg[node] - 1
    }}}
    {{{ leave ni.block.nbuff/ enter ni.fill.nbuff
    SEQ
       ens(leave,ni[node],ni.block.nbuff,
         clock,(trace/\16))
       ens(enter,ni[node],ni.fill.nbuff,
         clock,(trace/\16))
      ni.state[node] := ni.fill.nbuff
    }}}
    {{{ schedule control entity for previous node
    SEQ
      ent(get,sys)
      entity[sys][node.id] := prev[node]
      entity[sys][action] := no.word.received
      prq(sched,evs,sys,clock,(trace/\2))
    }}}
  }}}
act = ni.put.ubuff
  {{{ let net-in proc fill ubuff w/word from link
  SEQ
    {{{ Determine time needed to make transfer
    SEQ
      rng.get(os.time,os,(trace/\32))
      newtime := ni.decode[node]+(clock + (3*os))
    }}}
    {{{ Schedule the transfer
    SEQ
      ent(get,sys)
      entity[sys][node.id] := node
      entity[sys][action] := ni.xfer
      prq(sched,evs,sys,newtime,(trace/\2))
    }}}
```

```
        {{{ update word counters
        SEQ
          ubuff.nwords[node] := ubuff.nwords[node]+1
          ni.rest.msg[node] := ni.rest.msg[node] - 1
        }}}
        {{{ leave ni.block.ubuff/ enter ni.fill.ubuff
        SEQ
          ens(leave,ni[node],ni.block.ubuff,
            clock,(trace/\16))
          ens(enter,ni[node],ni.fill.ubuff,
            clock,(trace/\16))
          ni.state[node] := ni.fill.ubuff
        }}}
        {{{ schedule control entity for previous node
        SEQ
          ent(get,sys)
          entity[sys][node.id] := prev[node]
          entity[sys][action] := no.word.received
          prq(sched,evs,sys,clock,(trace/\2))
        }}}
      }}}
    act = no.send.word
      {{{ let net-out process place word on link
      SEQ
        IF
          (no.sending.words[node] > 0)
                      -- still sending a msg
            {{{ put word on link (decrement counters)
            SEQ
              no.sending.words[node]:=
                no.sending.words[node]-1
              nbuff.nwords[node] :=
                nbuff.nwords[node] - 1
              nlink.online[succ[node]] :=
                link.word.msg
            }}}
          TRUE              -- send start of msg
            {{{ put header on link,decrement counters
            SEQ
              {{{ move the header from nbuff to
                  -- nlink, update sending.words
              SEQ
                prq(next,nbuff[node],header,
                  prior,(trace/\4))
                no.sending.words[node] :=
                  entity[header][n.words]
                nlink[succ[node]] := header
              }}}

              {{{ update counters controlling buffers
                  -- and links
              SEQ
                nbuff.nheaders[node] :=
                  nbuff.nheaders[node] - 1
```

```
                                nbuff.nwords[node]:=
                                  nbuff.nwords[node]-1
                                nlink.online[succ[node]] :=
                                  link.head.msg
                              }}}
                            }}}
                        {{{ Determine time needed to do the transfer
                        SEQ
                          rng.get(os.time, os, (trace/\32))
                          newtime := no.put.word + ((6 * os) + clock)
                        }}}
                        {{{ Create control entity & Schedule event
                        SEQ
                          ent(get,sys)
                          entity[sys][node.id] := node
                          entity[sys][action] := no.xfer
                          prq(sched, evs, sys, newtime, (trace/\2))
                        }}}
                        {{{ Leave no.sleep / enter no.fill.nlink
                        SEQ
                          ens(leave,no[node],no.sleep,
                            clock,(trace/\16))
                          ens(enter, no[node], no.fill.nlink, clock,
                             (trace/\16))
                          no.state[node] := no.fill.nlink
                        }}}
                      }}}
                    TRUE
                      write.full.string(screen,
                        "Illegal action on READY queue  *c*n")
                  }}}
              TRUE
                SKIP
            }}}
          }}}
        TRUE
          SKIP
{{{ print time elapsed
SEQ
  realclock ? ftimer
  etimer := ftimer MINUS stimer
  durance := (REAL32 ROUND etimer)*(0.000064(REAL32))
  write.full.string(screen,"*#07")
  write.full.string(screen,"*#07")
  write.full.string(screen,"*c*n")
  write.full.string(screen,"Elapsed time for this block is ")
  REAL32write(durance,6,2)
  write.full.string(screen," seconds*c*n")
}}}
{{{ dump the accumulated statistics
ens (dmp,dummy,dummy,clock,(trace/\32))
}}}
{{{ dump the priority queues
SEQ
```

```
            prq (dump, evs, dummy, dummy, (trace/\8))
            node := 0
            WHILE node < n.nodes
              SEQ
                prq (dump, ubuff[node], dummy, dummy, (trace/\8))
                prq (dump, nbuff[node], dummy, dummy, (trace/\8))
                prq (dump, readyq[node],dummy, dummy, (trace/\8))
                prq (dump, blockq[node],dummy, dummy, (trace/\8))
                node := node + 1
          }}}
        }}}
      opus = sim.quit
        SKIP
      TRUE
        {{{ error
        STOP
        -- display an error from here. This path should never be taken.
        }}}
  }}}
:

PROC xnetrun()
  {{{ control the simulation
  {{{ Get the parameters
  PROC cnv.si(VAL INT len, VAL []BYTE str, INT val)
    {{{ convert an integer string to the integer value
    INT i, dval:
    SEQ
      val := 0
      i := 0
      WHILE ((i < len) AND ((str[i] < '0') OR (str[i] > '9')))
        i := i + 1
      WHILE ((i < len) AND ((str[i] >= '0') AND (str[i] <= '9')))
        SEQ
          dval := (INT str[i]) - '0'(INT)
          val := (10*val) + (INT dval)
          i := i + 1
    }}}
  :

  PROC get.params(CHAN screen, keyboard, []INT P)
    {{{ prompt for the parameters
    INT ch:
    INT i, len, val:
    INT distr:
    [80]BYTE str:

    SEQ
      {{{ print blank lines
      write.full.string(screen,"*c*n*n")
      }}}
      {{{ GET the # nodes and speed of the links (10 or 20 MHz)
      {{{ # of nodes in the system
      write.full.string(screen, "Number of NODES in the system (1-32) ==> ")
```

```
read.string(keyboard,screen,len,str)
cnv.si(len,str,P[17])
}}}
{{{ speed of the link
write.full.string(screen, "Link speed (10 or 20) ==> ")
read.string(keyboard,screen,len,str)
cnv.si(len,str,P[16])
IF
  P[16] = 10
    P[16] := 30
  TRUE
    P[16] := 15
}}}
}}}
{{{ GET the size of the buffers (nbuff, ubuff) and max words/msg
{{{ max words in a msg
write.full.string(screen, "Max No. of words in a msg ==> ")
read.string(keyboard, screen, len, str)
cnv.si(len, str, P[0])
P[0] := P[0] + 1                -- account for the message header
}}}
{{{ network and user buffer sizes
write.full.string(screen, "Network buffer size (MAX 2000) ==> ")
read.string(keyboard, screen, len, str)
cnv.si(len, str, P[2])
write.full.string(screen, "User buffer size (MAX 2000) ==> ")
read.string(keyboard, screen, len, str)
cnv.si(len, str, P[3])
}}}
}}}
{{{ Explain the distribution codes
SEQ
  write.full.string(screen,"*c*n*n")
  write.full.string(screen,"Distribution Codes:*c*n")
  write.full.string(screen,
    " Uniform  Negative Exponential   Constant *c*n")
  write.full.string(screen,
    "    1              2                   3 *c*n")
}}}
{{{ GET the distribution, mean, and seed (# msgs to send at once)
{{{ distribution # msgs to send
write.full.string(screen,"*c*n")
write.full.string(screen,"Number of messages to send at one time *c*n")
distr := invalid.distr    -- set to an invalid distr.type
WHILE (distr <> const) AND ((distr <> nexp) AND (distr <> unif))
  SEQ
    write.full.string(screen,"  -- Distribution Code: ")
    read.string(keyboard,screen,len,str)
    cnv.si(len,str,distr)
P[19] := distr
}}}
{{{ mean # msgs to send
write.full.string(screen,"  -- Mean: ")
read.string(keyboard,screen,len,str)
```

```
cnv.si(len,str,P[13])
IF
  P[13] > P[0]      -- if the mean is greater than the maximum
    P[13] := P[0]   -- set mean to max
  TRUE
    SKIP
}}}
{{{ seed for # msgs to send
write.full.string(screen, "  -- Seed: ")
read.string(keyboard, screen, len, str)
cnv.si(len, str, P[8])
IF
  P[8] = 0
    P[8] := 37
  TRUE
    SKIP
}}}
}}}
{{{ GET the distribution, mean, and seed (# words in a msg)
{{{ distribution # words in a msg
write.full.string(screen,"*c*n")
write.full.string(screen,"Number of words in a message *c*n")
distr := invalid.distr   -- set to an invalid distr.type
WHILE (distr <> const) AND ((distr <> nexp) AND (distr <> unif))
  SEQ
    write.full.string(screen,"  -- Distribution Code: ")
    read.string(keyboard,screen,len,str)
    cnv.si(len,str,distr)
P[20] := distr
}}}
{{{ mean # words in a msg
write.full.string(screen, "  -- Mean: ")
read.string(keyboard,screen,len,str)
cnv.si(len,str,P[14])
}}}
{{{ seed for msg length
write.full.string(screen, "  -- Seed: ")
read.string(keyboard, screen, len, str)
cnv.si(len, str, P[9])
IF
  P[9] = 0
    P[9] := 61
  TRUE
    SKIP
}}}
}}}
{{{ GET the distribution, mean, and seed (destination length)
{{{ distribution # links a msg should travel
write.full.string(screen,"*c*n")
write.full.string(screen,
  "Number of links a message should travel *c*n")
distr := invalid.distr   -- set to an invalid distr.type
WHILE (distr <> const) AND ((distr <> nexp) AND (distr <> unif))
  SEQ
```

```
    write.full.string(screen,"  -- Distribution Code: ")
    read.string(keyboard,screen,len,str)
    cnv.si(len,str,distr)
P[22] := distr
}}}
{{{ mean # links a msg should travel
write.full.string(screen, "  -- Mean: ")
read.string(keyboard,screen,len,str)
cnv.si(len,str,P[15])
}}}
{{{ seed for the operating system delay
write.full.string(screen,"  -- Seed: ")
read.string(keyboard,screen,len,str)
cnv.si(len,str,P[1])
IF
  P[1] = 0
    P[1] := 37
  TRUE
    SKIP
}}}
}}}
{{{ GET the distribution, mean, and seed (operating system delay)
{{{ distribution for operating system delay
write.full.string(screen,"*c*n")
write.full.string(screen,"Operating System Delay *c*n")
distr := invalid.distr   -- set to an invalid distr.type
WHILE (distr <> const) AND ((distr <> nexp) AND (distr <> unif))
  SEQ
    write.full.string(screen,"  -- Distribution Code: ")
    read.string(keyboard,screen,len,str)
    cnv.si(len,str,distr)
P[21] := distr
}}}
{{{ mean operating system delay
write.full.string(screen,"  -- Mean: ")
read.string(keyboard,screen,len,str)
cnv.si(len,str,P[12])
}}}
{{{ seed for the operating system delay
write.full.string(screen,"  -- Seed: ")
read.string(keyboard,screen,len,str)
cnv.si(len,str,P[10])
IF
  P[10] = 0
    P[10] := 83
  TRUE
    SKIP
}}}
}}}
{{{ GET the distribution, mean, and seed (user process run time)
{{{ distribution for time to create a word (user process time)
write.full.string(screen,"*c*n")
write.full.string(screen,"Time to process between generating msgs*c*n")
distr := invalid.distr   -- set to an invalid distr.type
```

```
    WHILE (distr <> const) AND ((distr <> nexp) AND (distr <> unif))
      SEQ
        write.full.string(screen,"  -- Distribution Code: ")
        read.string(keyboard,screen,len,str)
        cnv.si(len,str,distr)
    P[18] := distr
    }}}
    {{{ mean time to create a word
    write.full.string(screen,"  -- Mean: ")
    read.string(keyboard,screen,len,str)
    cnv.si(len,str,P[11])
    }}}
    {{{ seed for user process time
    write.full.string(screen, "  -- Seed: ")
    read.string(keyboard, screen, len, str)
    cnv.si(len, str, P[7])
    IF
      P[7] = 0
        P[7] := 61
      TRUE
        SKIP
    }}}
    }}}
    {{{ GET # blocks and the block length
    write.full.string(screen, "*c*n*n")
    write.full.string(screen, "Number of blocks ==> ")
    read.string(keyboard, screen, len, str)
    cnv.si(len, str, P[4])
    write.full.string(screen, "Block duration ==> ")
    read.string(keyboard, screen, len, str)
    cnv.si(len, str, P[5])
    }}}
    {{{ GET trace values
    write.full.string(screen, "TRACE VECTOR value ==> ")
    read.string(keyboard, screen, len, str)
    cnv.si(len, str, P[6])
    }}}
    {{{ print blank lines
    write.full.string(screen,"*c*n*n")
    }}}
  }}}
:


}}}
INT i:
INT clock:
BYTE ch:
INT kint:
INT dummy:
INT len:
SEQ
  write.full.string(screen,"Simulation Of An Occam Network (1988) *c*n*n ")
  get.params(screen,keyboard,params)
```

```
{{{ initialize the priority queue objects
-- the priority queue objects are self initializing

INT c,c1,c2,c3,node:            -- ASSIGN THE PRIORITY QUEUES
SEQ
  node := 0                     -- node #'s start at 0
  c := 1                        -- ubuff queues #'s start at 1
  c1 := max.nodes * 1           -- nbuff queues start at max.nodes
  c2 := max.nodes * 2           -- ready queues start at 2 * max.nodes
  c3 := max.nodes * 3           -- block queues start at 3 * max.nodes

  WHILE node < max.nodes        -- For each node:
    SEQ
      ubuff[node] :=  c         -- start at 1  (note: evs is queue 0)
      nbuff[node] :=  c + c1    -- get next queue number for this node
      readyq[node] := c + c2    -- get next queue number for this node
      blockq[node] := c + c3    -- get next queue number for this node
      node := node + 1          -- get next node number
      c := c + 1                -- increment by one
}}}
{{{ initialize the RNG objects
SEQ
  rng.init(nbr.msgs,distr.gen.msgs,params[13],params[8])
  rng.init(proc.time,distr.proc.time,params[11],params[7])
  rng.init(msg.len,distr.msg.len,params[14],params[9])
  rng.init(os.time,distr.ostime,params[12],params[10])
  rng.init(msg.dist,distr.msg.dist,params[15],params[1])
}}}

xnetsim(sim.init,clock)

kint := 1
WHILE (kint <= n.blocks)
  {{{ Run simulation for another block
  SEQ
    xnetsim(sim.sim,clock)
    write.full.string(screen,"*c*n")
    write.full.string(screen,"BLOCK #")
    INTwrite(kint,3)
    write.full.string(screen,"*c*n")
    kint := kint + 1
  }}}

xnetsim(sim.quit,clock)

{{{ terminate the statistics process
SEQ
ens (quit,dummy,dummy,clock,(trace/\16))
}}}
{{{ terminate the priority queue objects
SEQ
  prq(quit,evs,dummy,dummy,(trace/\2))
  i := 1
  WHILE i < max.sys.queues
```

```
          SEQ
            prq(quit,i,dummy,dummy,(trace/\4))
            i := i + 1
      }}}
      {{{ terminate the RNG objects
      SEQ
        rng.quit(proc.time,(trace/\32))
        rng.quit(nbr.msgs,(trace/\32))
        rng.quit(msg.len,(trace/\32))
        rng.quit(os.time,(trace/\32))
        rng.quit(msg.dist,(trace/\32))
      }}}
      write.full.string(screen,"End program execution*c*n")
      keyboard ? ch
    }}}
  :

  PAR
    PAR i = 0 FOR 5
      c.rand(to.rand[i],from.rand[i])

    PAR i = 0 FOR max.sys.queues
      c.prq(to.prq[i],from.prq[i],screen)

    SEQ
      c.stats(to.stats,from.stats,screen)
    SEQ
      xnetrun()
:
```

LIST OF REFERENCES

1. Banks, Jerry, and Carson, John S. Discrete-Event System Simulation. Englewood Cliffs: Prentice-Hall, 1984.

2. Comfort, J.C. and Raja Gopal, R., "Environment Partitioned Distributed Simulation With Transputers". Proceedings of the 1988 Winter Simulation Conference, pp. 103-108, Feb. 1988.

3. Garcia, Albert B., Shaw, Wade H., "Transient Analysis Of A Store-And-Forward Computer-Communications Network" Proceedings of the 1986 Winter Simulation Conference. pp. 752-760, Washington, D.C., Dec. 1986.

4. INMOS Transputer Reference Manual. INMOS Ltd. 72-TRN 006-03, Bristol, UK, 1987

5. MacDougall, M.H. Simulating Computer System Techniques and Tools. Cambridge: The MIT Press, 1987.

6. Shannon, R. E. System simulation: The Art and Science. Englewood Cliffs: Prentice-Hall, 1975.

7. Silberschatz, Abraham, and Peterson, James L. Operating System Concepts. New York: Addison Wesley Publishing Company, 1988.

8. Stuck, Bart W. "Calculating the Maximum Mean Data Rate in Local Area Networks". Computer, May 1983, pp. 72-76.

9. Qiang, Li, William B. Feild Jr., and Donald Klein. "Implementation of a Transputer Ring Network and a Deadlock Prevention Algorithm". Proceedings from the 3rd U.S. Occam User Group Meeting, Sept. 1987.

10. Qiang, Li, William B. Feild Jr., and Donald Klein. "Channel Design Primitives in Occam". Proceedings from the 3rd U.S. Occam User Group Meeting, Sept. 1987.

VITA

Master's Thesis Title:

A SIMULATION OF A MESSAGE PASSING PROTOCOL
FOR A NETWORK OF TRANSPUTERS

Janice R. Glowacki

Born in Miami, Florida, on January 11, 1963

Attended Sable Palm Elementary, John F. Kennedy Junior High, Highland Oaks Junior High, and North Miami Beach Senior High, all in Miami, Florida.

Attended Diablo Valley College in Pleasant Hill, California, from September 1980 to December 1981.

Attended Florida International University in Miami, Florida, from August 1982 to December 1986 at which time a Bachelor of Science degree was awarded. Majored in Computer Science.

Attended Florida International University in Miami, Florida, from January 1987 to date for graduate studies.

Member of Association of Computing Machinery.

Author of "A Simulation of a Store-and-Forward Distributed Network of Transputers" for the Proceedings of the 1988 Winter Simulation Conference (San Diego, California).

School of Computer Science

September 16, 1988

Janice R. Glowacki

Janice R. Glowacki