A Distributed Approach to Queueing Network Simulation

J. Kent Peacock, J. W. Wong, and Eric Manning

Computer Communications Networks Group and
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1

Abstract

Discrete simulation is a widely used technique for system performance evaluation. The conventional approach to discrete simulation (e.g., GPSS, Simscript) does not attempt to exploit the parallelism typically available in queueing network models. In this paper, a distributed approach to discrete simulation is presented. It involves the decomposition of a simulation into components and the synchronization of these components by message passing. This approach can result in the speedup of the total time to complete a given simulation if a network of processors is available. The architecture of a microcomputer network suitable for distributed simulation is described and some results concerning the distributed approach are presented.

INTRODUCTION

In recent years, queueing network models have been used extensively to study the performance of computer systems (9) and computer communication networks (19). A simple example of queueing network models is shown in Figure 1. It consists of a collection of servers organized in such a way that customers move from one server to another in order to complete their service requirements.

Queueing network models can broadly be classified as open or closed. In an open network, like the one shown in Figure 1, customers can enter or leave the network, and the total number of customers in the network is a random variable. Open networks find applications in packet-switched communication networks (19). In a closed network, customers can neither enter nor leave the network, and the total number of customers in the network is a constant. Closed networks are suitable for studying interactive computer systems (9).

There are two common approaches to system performance evaluation using queueing network models: analytic modelling and discrete simulation. Between the two, discrete simulation has the advantage of being able to model and study systems at an arbitrary level of complexity (17). However, it is also generally more costly to develop and run the simulation programs.

The basic technique in discrete simulation is to repeatedly advance the simulation time (which is an abstraction of real time in the simulated system) to the time of next event and simulate the changes in system state according to the event type. Simulation programs implementing this technique are conventionally designed to execute in sequential machines. There are two basic approaches to the organization of these programs. The first approach, as used in Simscript (15), is event scheduling. Its underlying structure attempts to invoke the event routines at the appropriate simulated time instants. A program in Simscript therefore contains a

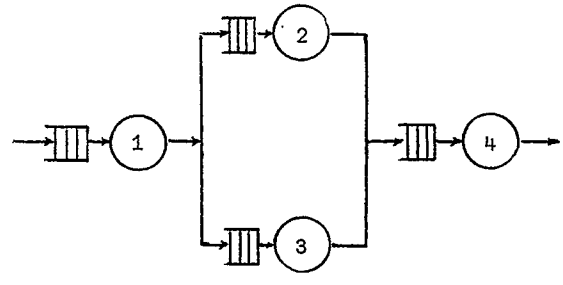


Figure 1. Queueing Network Model

CH1437-3/79/0399-0406$00.75
1979 Winter Simulation Conference

group of subroutines which characterize the state changes for the different types of events, and a timing routine which processes events by calling these subroutines. The second approach is the one used in GPSS (12). is process interaction. Its underlying structure attempts to move customers to the completion of their activities. Transactions (or customer records) are moved from the future event chain to the current event chain at the appropriate event times. The current event chain is then scanned and the active transactions are moved as far as possible towards the completion of their activities.

Neither approach attempts to exploit the parallelism typically available in the simulation of queueing network models. Such parallelism is a result of the servers in the network operating in parallel. A natural approach to exploit this parallelism is to treat the simulation of each server as a separate component (3,4,6,18). Each component is responsible for processing the events related to its server, and the actions of these components are synchronized so that the simulation is carried out correctly. This distributed approach can potentially result in a speed-up of the total time to perform a given simulation if a network of processors is available. Economical developments of such networks are becoming more and more viable with the availability of low-cost microprocessors.

In this paper, we first present the distributed approach to discrete simulation developed by Bryant (3,4), Chandy and Holmes (6), and by the authors (18). We then evaluate the merit of this approach with respect to the improvement in total time to complete a given simulation. Our evaluation is based on experiments performed on a network of microcomputers developed at the University of Waterloo.

DISTRIBUTED APPROACH TO DISCRETE SIMULATION

Decomposition into Components

As mentioned previously, one technique to perform simulation in a distributed manner is to decompose the simulation into components and develop a method to synchronize the action of these components. There is no fixed rule to do the decomposition, although a natural approach is to treat each server in the network model as a separate component. However, for the case of a packet-switched network where a number of switching computers are connected together by a collection of communication channels, it may be more convenient to treat each switching computer and its outgoing channels as a component.

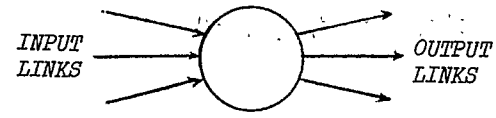


Figure 2. A Component of the Simulation

Interconnection Graph

The movement of customers in the network model is represented by events sent among the various components. The function of each component is then to receive customer arrival events, process them, generate customer departure events, and sent these events to other components. It also performs the necessary synchronization functions and collects statistical data for output purposes.

Since each component is a producer and consumer of events, we find it convenient to characterize the inter-relationship of the components by an interconnection graph. Each component in the simulation is represented by a node in this graph. If component i generates events to be consumed by component j, then there is a link from node i to node j. The general picture of a component is shown in Figure 2. Input links are for the reception of customer arrival events, while output links are for the departure of customers to other components. In an open network, a component which generates external arrivals is represented by a node with no input links. Similarly, the sink which absorbs all departures from the network is represented by a node with no output links.

The interconnection graph of the network model in Figure 1 is shown in Figure 3.

Synchronization of Components

For the purpose of synchronization, we assume that the components communicate with each other by message passing. Since events are processed in non-decreasing

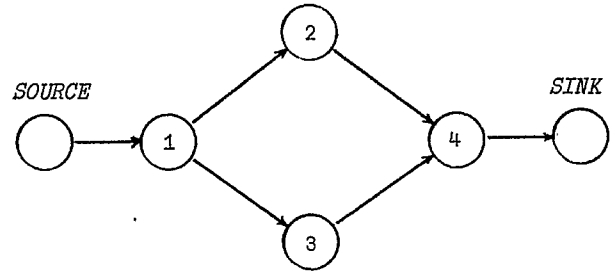


Figure 3. Interconnection Graph

simulation times, before a component accepts an arrival event, it must ensure that no other arrivals can occur at an earlier time. The acceptance of the next arrival event from one of the input links forms the core of the synchronization function. For the case of a single input link (e.g., component 2 in Figure 3), the synchronization is trivial. As long as component 1 generates departures to component 2 in non-decreasing event times, component 2 can process them as they arrive. The situation becomes more complicated when there is more than one input link (e.g., component 4 in Figure 3). This component must perform a merging function which determines the arrival with the earliest event time across all the input links. The operation is still quite straight-forward when there exists one or more arrivals on each input link. In some cases, however, there may not be an arrival on one of the input links for some period of time. During this period, the component cannot proceed with its part of the simulation, and the amount of parallelism is reduced. To illustrate such a phenomenon, consider the example shown in Figure 4. 99% of the customers leaving component 1 are directed to component 3. Component 2 is therefore idle most of the time. When there is no departure from component 2, component 4 cannot accept its next arrival from component 3 even though the event times in other components happen to be arranged in such a way that no other arrivals can occur earlier. When there is no customer across an input link, we say the link is "empty".

Link Time Solution (3,6,18)

One technique to improve the amount of parallelism is to have the components send timing information to each other. As an example, consider the model shown in Figure 4. Suppose the links from node 1 to node 2 and from node 2 to node 4 are both empty, and component 1 has just received an arrival at simulation time 1Ø. Component 1 can send a message to component 2, informing him that there is no departure from component 1 before simulation time 1Ø. Likewise, component 2 can send the same information to component 4. Component 4 can then use this information to carry out

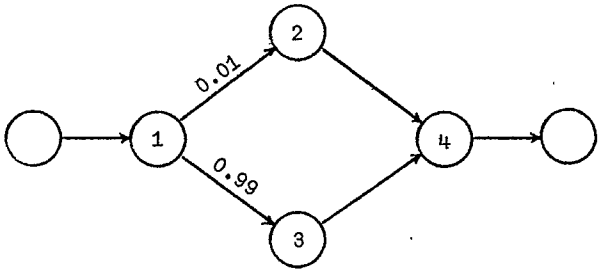


Figure 4. Empty Link Problem

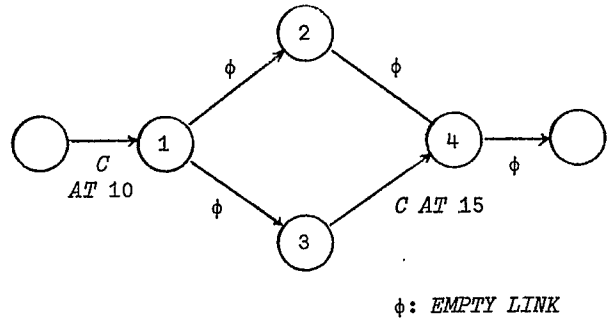


Figure 5. Blocked Node Example

the merging process. If the first arrival from component 3 is at simulation time < 1Ø, then component 4 can process it right away. If not, component 4 must wait for more information from component 2 before it can proceed, and we say that component 4 is "blocked". An example of a blocked node is shown in Figure 5.

We now characterize the merging operation by defining a link time across each link in the interconnection graph. When the link is non-empty, the link time is simply the arrival time of the next customer on that link. When the link is empty, the source end is responsible for maintaining the link time as the greatest lower bound on the next departure across the link, according to the information it possesses. In performing the merging operation, a node first determines the input link with the lowest link time. If this link is non-empty, then it accepts the next arrival from this link, otherwise it is blocked. A blocked node becomes unblocked only when the source end can calculate a sharper lower bound, or generate a customer across the empty link.

Unfortunately, with certain combinations of network topology and customer placement, a node may not be able to do either of the above and the simulation halts in a deadlock situation. Figure 6

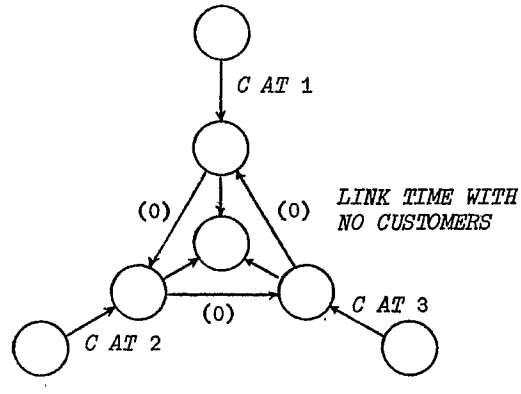


Figure 6. Deadlock Example

illustrates a simple example of deadlock which occurs at initialization, when all the empty links have link time 0. The deadlock is among nodes 4, 5, and 6 which are blocking each other in a circular fashion. The following theorem, proved independently by Chandy and Holmes (5) and the authors (18), gives the necessary and sufficient conditions for a deadlock to occur:

Theorem: A deadlock exists if and only if there is a cycle consisting of empty links which all have the same link time, and of nodes which are blocked because of these links.

This theorem gives us the means to find a solution to the deadlock problem. We first observe that acyclic graphs present no problem since the necessary condition that a cycle exists is not fulfilled. Also, if we insist that each customer must not have zero service time at any node, we can break the necessary condition that the link times around the cycle of empty links are identical.

Pseudo-Time-Driven Effect

Although deadlock can be avoided by insisting on a minimum service time at every server, the presence of cycles may cause severe degradation in the efficiency of the distributed approach. As an example, consider the model shown in Figure 7. Suppose the minimum service time is 1. The link times on the empty links will increase alternately by 1 until the link time on the lower empty link is 1000, at which time the customer is accepted to node 1.

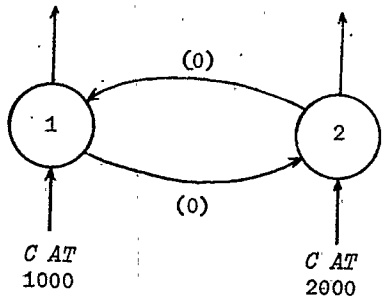


Figure 7. Pseudo-time-driven behaviour

HARDWARE AND SOFTWARE

Hardware

In this section, we describe the experiments we have thus far performed to assess the utility of the distributed approach to simulation. We first give an overview of the hardware and software, then describe the general structure of the simulator. Next we discuss the factors which affect a simulation's performance, and finally present the experimental results.

The hardware which we have available consists of a homogeneous network of DEC PDP-11/03 microcomputers, each of which has 56K bytes of memory and an average execution rate of approximately 200,000 instructions per second. We also use a PDP-11/45 with mass storage for program development and support.

All of these machines are interconnected by a fixed time-division-multiplexed bus developed at the University of Waterloo, called the SIBUS (1). The SIBUS allows 15 subscriber machines to each send a one-word, addressed message every 6.4 microseconds. The SIBUS is not a direct memory access (DMA) device, which means that a processor must load every message to be sent into the device registers.

Software

The PDP-11/45 is used to run the UNIX (2) operating system from Bell Labs, and primarily provides a pleasant environment for programming, providing file system support, editing, and compilation facilities. Because the instruction sets of the 11/45 and 11/03 are virtually identical except for memory management facilities (the 11/03 has none), code produced by compilers on the 11/45 will run on an 11/03 without modification other than the use of a different runtime support package. We have chosen to use the "native" programming language of UNIX, the C programming language (14), because of its simplicity, power, and the excellent library provided under UNIX. To load programs into the microcomputers, we make use of the SIBUS and the Satellite Processor System (2) extension to UNIX.

Since we want to explore various assignments of simulation components to processors, we need the capability to multiprogram the microprocessors. This facility has been provided by a small operating system kernel, called the Stand-Alone Executive (SAE), written at the University of Waterloo by Randall Howard (11). SAE provides primitives to do process creation, blocking and unblocking, timeouts, and input/output through appropriate device drivers.

Our primary mechanism for communication between processes is message-passing, although processes in one machine can also communicate via shared memory. Message passing is implemented with two basic primitives, "nsend" and "nrec", and by the use of a queue structure associated with each process. Nsend and nrec stand for non-blocking send and receive respectively, where by non-blocking we mean that a

process doing an nsend is free to continue execution as soon as the message is put on the receiver process's queue, and one doing an nrec continues to execute even if no message is waiting to be received. We have also found it useful to supply a blocking receive primitive, brec, which blocks until a message is available before returning to the caller. However, we have not incorporated an explicit blocking send primitive.

The call "nsend(toproc, tchan, buffer, nbytes)" causes a message containing the data to which "buffer" points, of length "nbytes", to be put on the message queue for process "toproc". The message also has a header which contains the identity of the sending and receiving processes and the value of "tchan". The "tchan" parameter is a 16-bit mask indicating one or more virtual channels on which the message may be received. The call "nrec(fromproc, rchan)" returns a pointer to the first message on the message queue from process "fromproc" for which the bitwise "and" of "rchan" and "tchan" is non-zero. If there is no such message, nrec returns a NULL pointer. To receive a message from any source, nrec is called with the special value "STAR" as the first argument. The call "brec(fromproc, rchan)" provides a blocking receive primitive, implemented using nrec, which blocks until a satisfactory message has been received. The selection and implementation of these basic primitives evolved from consideration of the MOOSE operating system nucleus, developed by Meisner (16).

On top of the message passing primitives, we have the routines which perform the synchronization of the components. In particular, the call "nev(min_event)" returns the time of the next external input to the component which arrives before time "min_event", if there is one. If it is certain that there is no external event before "min_event", an invalid time value (such as -1) is returned, to allow the component to simulate internally up to time "min_event". Note that the call may block if there is insufficient information to determine the next external input before "min_event". The call "gen(dest, dtime)" sends an event to component "dest" at time "dtime". For our discussion, we have assumed that events contain no other information besides their arrival or departure times from a component.

At the top level, we have the code which does the actual simulation of a component of the system, which interfaces to lower levels using the "nev" and "gen" calls. For our evaluations, we have chosen to simulate a queueing network consisting of simple first-come, first-served servers with uniform service time distributions and fixed branching probabilities.

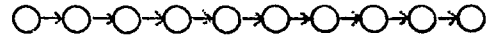
This completes a bottom-up description of the hardware and software used to implement our distributed simulation methods.

EXPERIMENTAL RESULTS

In this section, we present results of some of our experiments using the link time algorithm. We chose a number of simple topologies to simulate, such as a tandem queue, the model in Figure 5, and a loop. With each topology, we then ran a number of simulations using different assignments of nodes to processors, and measured the time taken for each simulation to complete. In order to include a more conventional approach in this comparison, we also ran a simulation in a single PDP-11/03 processor using the same process structure, that is, a separate process for each component of the simulated system, but using shared memory for inter-process communications and a simple earliest-event-time-first method of scheduling the processes.

The first topology considered was a tandem queue with 10 nodes, and service times selected uniformly from the set { 0, 1, 2, 3 }. The graph for this network and the last finishing times for a number of different assignments of nodes to processors is shown in Figure 8. The assignment of nodes to processors is indicated by the lines enclosing groups of nodes, where the interconnections have been deleted for clarity.

The results obtained are very encouraging, since the time to completion steadily decreases as the number of proces-

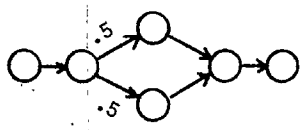


PROCESSORS	*ASSIGNMENT*	*TIME*
1	*CONVENTIONAL APPROACH*	77
1	(o o o o o o o o o o)	102
2	(o o o o o) (o o o o o)	58
3	(o o o o) (o o o) (o o o)	43
4	(o o o) (o o o) (o o) (o o)	36
5	(o o) (o o) (o o) (o o) (o o)	32
6	(o o) (o o) (o o) (o o) (o) (o)	30
7	(o o) (o o) (o o) (o) (o) (o) (o)	29
8	(o o) (o o) (o) (o) (o) (o) (o)	25
9	(o o) (o) (o) (o) (o) (o) (o) (o) (o)	21

Figure 8. Tandem Queue

sors goes up. However, enthusiasm should be tempered by the realization that this particular topology is probably the best possible configuration for the distributed approach to be successful. The reason for this claim is simply that no link time messages are ever necessary. Hence if the link time algorithm is refined so that it only sends the necessary messages, the number of messages passed is minimal. The two refinements which accomplish this are simply that link time messages are not sent when the node can still process arriving events, and that link time messages with the same link time as the last departed event are not sent to that event's destination. Besides the fact that the times decrease monotonically with additional processors, we note that the simulation with two processors completes faster than the one using the conventional approach, but that the link time algorithm run in a single processor is substantially more expensive.

The results for the next example, a network with a single fork, are given in Figure 9, where the service time distributions are the same as in the tandem queue



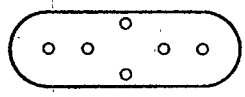
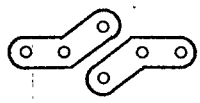
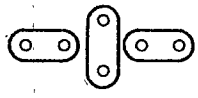
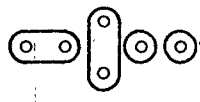
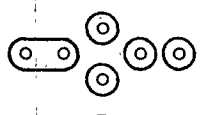
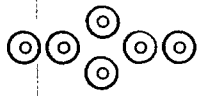
PROCESSORS	*ASSIGNMENT*	*TIME*
1	*CONVENTIONAL APPROACH*	153
1		209
2		124
3		100
4		93
5		84
6		84

Figure 9. Forked Queue

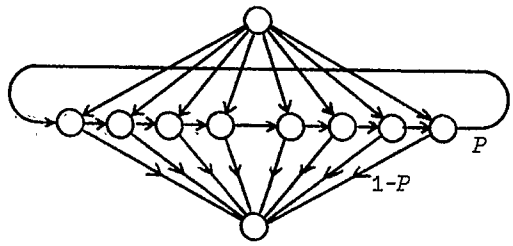
example. The completion times for this case follow the same sort of pattern as the first example, in that the use of two processors beats the conventional approach, and the times generally decrease as the number of processors increases. The surprising thing about this example was that there were almost no link time messages passed through the fork, due to the same modification which eliminated unnecessary messages in the first example. The node at which the forks rejoin allows a queue to build up on one edge if the other edge is empty, and simply delays processing until the next arrival on the empty edge. This works rather well if the branching probabilities at the fork are nearly equal, but would result in large queue buildups if the probability of taking one of the forks is very small. In this case one might want to cause the node at which the fork takes place to send a link time message on a low-probability edge after every k departures, where k is perhaps the largest queue buildup one is willing to tolerate at the joining node.

The last example is a the loop topology shown in Figure 10. In this case, the service times for the 8 nodes in the loop are taken from the set { 1, 2, 3 }, and the source inter-departure times from { 0, 1, 2, 3 }. Also, the branching probability of each edge in the cycle is p and that of each edge into the sink node 1-p. Statistics were collected for two values of p, p=.50 and .75 .

We note that in the first case, little is gained by distributing the simulation, relative to the conventional approach. The reason for this is that the sink node, which has to merge 10 inputs, is a bottleneck. There are about twice as many link time messages received at the sink node than event arrivals, so the overhead is considerable.

In the second case, the events circulate around the cycle more because of the higher probability of staying within the cycle, so the conventional method has to process more events. So, the second case appears more promising since the times for more than 2 processors are less than with the conventional approach. Note that with 9 processors, the time is roughly the same as the p=.50 case, because the sink node is still the bottleneck. The increased work in processing more events is distributed among nodes which were doing less work in the first case.

The fact that we are able to obtain a monotonic decrease in finishing time as the number of processors is increased in every case is a strong indication that indeed our global message passing is sufficiently fast relative to local message passing and shared memory to make distributed simulations with our particular distributed system viable. If this were not the case,



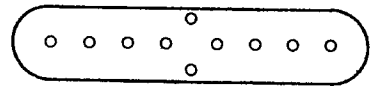
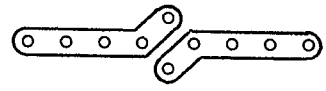
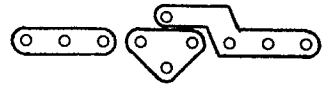
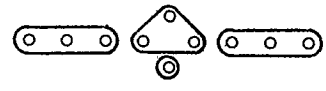
PROCESSORS	ASSIGNMENT	TIME P=.50	TIME P=.75
1	CONVENTIONAL APPROACH	31	48
1		57	73
2		41	47
3		35	38
4		31	31
5		31	29
9		28	27

Figure 10. Loop Example

then we would expect the completion times to decrease and then increase, or in the worst case, to begin increasing going from one processor to two. (In fact, this effect was observed using earlier versions of the link time algorithm and message-passing code which were less efficient.)

CONCLUSIONS

The distributed approach to discrete simulation has been evaluated by experimtents performed on a network of microcomputers. It was found that for some topologies of queueing network models, this approach results in a speed-up in the total time to complete a given simulation. However, for other topologies, especially those with loops, the speed-up may not be significant. It was also observed that the amount of message passing has a significant effect on the performance of the distributed approach. It is therefore not beneficial to use the link time algorithm to simulate models that can potentially get into the pseudo-time-driven behaviour illustrated in Figure 9. However, there are modifications to the link time algorithm which promise a solution to this problem, for example, Bryant's so-called "time acceleration" (4).

BIBLIOGRAPHY

1. Banks, W., Private communication.

2. The Bell System Technical Journal, The Unix Time-Sharing System, Vol 57, No 6, Part 2, (July-Aug. 1978).

3. Bryant, R.E., "Simulation of Packet Communication Architecture Computer Systems", MIT/LCS/TR-188, Massachusetts Institute of Technology, Cambridge, Massachusetts, (Nov 1977).

4. Bryant, R.E., "Simulation on a Distributed System", submitted to First International Conference on Distributed Systems.

5. Chandy, K.M., Holmes, V., Private communication, (Feb. 1978).

6. Chandy, K.M., Holmes, V., and Misra, J., "Distributed Simulation of Networks," Technical Report 81, Department of Computer Sciences, University of Texas at Austin, Texas 78712, also submitted to Computer Networks.

7. Chandy, K.M. and Misra, J., "A Non-Trivial Example of Concurrent Processing: Distributed Simulation," Technical Report 82, Department of Computer Sciences, University of Texas at Austin, Texas 78712, also in Proceedings COMPSAC, Chicago, Nov. 16-18, 1978.

8. Chandy, K.M. and Misra, J., "Specification, Synthesis, Verification, and Performance Analysis of Distributed Programs; A Case Study: Distributed Simulation," Technical Report 86, Department of Computer Sciences, University of Texas at Austin, Texas 78712.

9. Denning, P. J., and Buzen, J. P., "The Operational Analysis of Queueing Network Models", ACM Computing Surveys 10, 3 (Sept. 1978), pp 225-261.

10. Emshoff, J.R., and Sisson, R.L., Design and Use of Computer Simulation Models, Macmillan, New York, (1970).

11. Howard, R.J., Private communication.

12. IBM, General Purpose Simulation System System/360 User's Manual, GH 20-0326, White Plains, N.Y., (1970).

13. Irland, M.I., "Analysis and Simulation of Congestion in Packet-Switched Networks", Ph.D. thesis, University of Waterloo, (1977).

14. Kernighan, B.W., Ritchie, D.M., The C Programming Language, Prentice-Hall, Toronto, (1978).

15. Kiviat, P.J., Villanueva, R., Markowitz, H.M., Simscript II.5 Programming Language, Consolidated Analysis Centers Inc., Los Angeles, California, (1973).

16. Meisner, N.L., "Process Management and Communication Facilities for Distributed Operating Systems", Master's Thesis, University of Waterloo, Waterloo, Ontario (1979).

17. Muntz, R.R., "Analytic Modeling of Interactive Systems", Proc. IEEE 63, 6 (June 1975), pp 946-953.

18. Peacock, J.K., Wong, J.W., and Manning, Eric, "Distributed Simulation using a Network of Micro-Processors", Proc. Third Berkeley Workshop on Distributed Data Management and Computer Networks, (Aug 1978), and Computer Networks, Vol 3, No 1, pp 44-56, (Feb 1979).

19. Wong, J. W., "Queueing Network Modeling of Computer Communications Networks", ACM Computing Surveys 10, 3 (Sept 1978), pp 343-351.