A STUDY AND ANALYSIS OF PERFORMANCE OF DISTRIBUTED SIMULATION

M. Seetha Lakshmi

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-32 August 1987

Abstract

This report describes the experimental study done on the performance of a distributed algorithm, proposed by Chandy and Misra for simulating networks of servers and queues.

ACKNOWLEDGEMENTS

This reaearch was sponsored by the National Science Foundation through the grant number MCS77-09812 to the University of Texas at Austin. The author would like to thank Professors J. Misra and K.M Chandy for suggesting this area of research and for the many hours spent in providing valuable guidance during the research and writing process.

April, 1979

CHAPTER 1

INTRODUCTION

This report describes the experimental study done on the performance of a distributed algorithm, proposed by Chandy and Misra[1], for simulating networks of servers and queues. The algorithm has some unique features which distinguish it from the conventional event driven simulation algorithms. The following sections briefly give the motivation for distibuted programming and distributed simulation. A detailed description of the distributed simulation algorithm appears in a subsequent chapter.

Tools for predicting the performance of systems are very valuable. Analytic modelling and simulation are the most widely used tools for this purpose. Simple systems can be effectively modelled using analytic methods; but as the systems' complexity increases they tend to be mathematically intractable and defy solutions by analytic methods. Under such circumstances simulation techniques are resorted to. Systems can be modelled at any level of detail using simulation techniques. Many real systems can be identified as a network of servers and queues. Simulation of the physical system results in simulating the activities of the real system in the queueing network. One major problem with

simulation techniques is to determine how to terminate the simulation early enough and yet obtain some performance metric with desired confidence. It is necessary to continue the simulation for a long time in order that the effect of the initial conditions disappears and the system reaches a steady state. This imposes heavy computation time requirement. Another reason for heavy computation requirement can be attributed to the sequential nature of the conventional simulation algorithms. The conventional (centralized) event driven simulation algorithm maintains a clock and a single list of events which represent the ensuing activity at each server/queue. The events are ordered in chronological order according to their time of occurence. Processing of one event may generate some new events which have to be inserted into the event list. This results in a large fraction of the time being spent in manipulating the event list. Study of complex systems is often limited by the computational requirements of the simulation algorithms. Hence it is important to decrease the computation time of the simulation algorithms.

1.1 DISTRIBUTED PROGRAMMING

One obvious way to achieve speed improvement is to develop parallel algorithms.The proliferation of cheap and powerful microprocessors and the advances made in the field

of computer networks provide an attractive solution to the implementation of parallel algorithms. The cost effectiveness of processors in a network environment can be fully exploited if we run distributed programs on them.

Distributed programs constitute a special class of parallel programs. A distributed program is a collection of processes which work on a common problem by communicating with each other only through messages and without sharing any global variables. There is no central process to control and synchronize other processes.

1.2 DISTRIBUTED SIMULATION

In many queueing systems, different servers work simultaneously. This inherent parallelism can be exploited by developing distributed algorithms for simulation. The distributed solution to discrete event simulation, in addition to increasing the execution speed, also provides a one to one correspondence between the real system and the simulator. It will be more cost effective than the conventional algorithm designed for uniprocessors, if the memory requirements and the overhead due to interprocessor communication is minimized.

The goal of the distributed simulation is to a) partition the system being simulated into relatively

independent subsystems which communicate with each other in a simple manner by passing messages, and b) simulate each sybsystem on a different processor. A natural way of doing this, for simulating a queueing system, is to allow each processor to model a single server and the associated queue. The processors are interconnected according to the topology of the system being simulated.

Jobs enter the system from a source at arbitrary time intervals, get processed by some servers and then move to some neighbouring servers. Eventually a job leaves the system by getting absorbed by the sink. The arrival and departure of a job to the server are represented by messages of the form <t,j>, in the simulator, where j is a job name and t is a time. If the ith processor sends a message <t,j> to the (i+1)th processor at any time during the simulation, it means that, the job j leaves the server i and arrives at server (i+1) in the real system at time t. All processors repeat the following cycle: wait for a message on all the input lines; simulate the servicing for that job with earliest arrival time and output a message to each processor that is directly connected to it. In each cycle the processor also collects enough information to compute the statistics related to the server it models.

1.3 RELATED WORK

The field of distributed simulation is relatively new. A few distributed simulation algorithms have appeared in the literature. They are yet to be implemented on actual distributed computer systems.

The simulation algorithm whose performance is studied in this report is part of the research done by Chandy and Misra [1,2,3]. The algorithm has the following properties: 1) it is based upon a collection of identical asynchronous logical processes which have no shared variables and which communicate only by transmitting messages, 2) there is no global controlling process to synchronize these processes, and 3) the system is deadlock free. Proof of correctness and absence of deadlock is given in [3].

Peacock et al [9] define different methods for simulating queueing systems in a network of microprocessors. There is, however, no data on the performance of this algorithm.

Holmes [7] has designed parallel algorithms for two specific applications, namely discrete event simulation of feed forward networks and graph problems. He suggests a method for allocating processors to servers, in order that the processors are optimally utilized. These algorithms are

yet to be implemented.

1.4 PERFORMANCE OF DISTRIBUTED SIMULATION

We can obtain accurate performance figures for the distributed simulation algorithm by implementing it on a network of processors. However, in the absence of such a facility at the University of Texas, Austin, the algorithm is simulated on a uniprocessor (DEC1Ø) and its performance is evaluated. This approach introduces two levels of simulation. At the lower level there is a collection of processes simulating a queueing system. At the upper level there is a simulator which simulates the effect of the lower level system by simulating a collection of cooperating concurrent processes. The lower level simulator is concerned with predicting the performance (thruput, utilization, queue length distribution, etc.) of the queueing system while the upper level simulator is concerned with the performance of the processes. For instance, if each process is modelled on a microprocessor, the upper level simulator monitors the performance of the processors, such as fraction of time all processors are busy, the total number of messages transmitted by the processors, etc.

1.5 PREVIEW OF CHAPTERS

Chapter 2 provides a detailed description of the distributed simulation algorithm. It discusses the three types of processes of which the algorithm is comprised and explains how these processes together simulate any arbitrary queueing network model. Chapter 3 explains how the distributed simulation algorithm is implemented/simulated on a uniprocessor. In particular it describes the two level simulation, mentioned in section 1.4, in greater detail. It also mentions some properties of implementation. The results of the experiments and their analysis appear in Chapter 4. A summary and conclusion of this report can also be found in this chapter.

CHAPTER 2

DISTRIBUTED SIMULATION

2.1 INTRODUCTION

The distributed simulation algorithm is described in detail, in this chapter. The algorithm can be used to simulate any arbitrary queueing network. It is composed of a collection of subsystems known as processes. The processes communicate with each other only by sending and receiving messages. All messages are of the form <t,n>, where t denotes a time and n the number of jobs leaving/arriving at t. The transmission time of messages between processes is assumed to be negligible compared to the processing time of the messages. There are three different types of processes -delay, fork and merge processes.The delay process simulates a single server and the associated queue, while the fork and merge processes route the jobs to different servers according to the topology of the physical system being simulated.

The subsequent sections describe the function of each process, the interconnection, and the communication protocols. The queueing system being modelled is represented by a directed graph consisting of edges and

nodes. The nodes correspond to servers and the edges correspond to the flow of jobs. The processes in the simulator simulate the nodes. Hence the terms "process" and "node" are interchangably used in the following discussion. To avoid confusion between the system to be simulated and the system (distributed simulation algorithm) which performs the simulation, the former will be called the physical/real system and the latter will be called the simulator.

2.2 SOME NOTATIONS FOR DESCRIBING DISTRIBUTED PROGRAMS

We need some special notations for describing certain operations performed by distributed programs. The processes which comprise the distributed program communicate only by sending and receiving messages. There are, however, no implicit buffers for message transmission; nor are there any global variables. Two commands SEND and RECEIVE are introduced here. These are similar to the input-output commands proposed by Hoare [5] for communicating sequential processes.The SEND command takes as arguments the message and an identification of the process to which the message must be sent. The RECEIVE command takes as arguments an identification of the process from which message is to be received and a set of variables which get the values of the message.

For example, a process i wishing to send a message to process j will issue the output command SEND (j, <message>); and process j wishing to receive a message from process i will issue the input command RECEIVE (i, <variables>). All messages in our simulator are of the form <t,n> where t is a non-negative real number and n is a non-negative integer. A message <t,n> with n>0 indicates that n number of real jobs leave/arrive at time t. A message <t,n> with n=0 implies the absence of any real job until time t. The message <t,0> is said to be concerned with NULL jobs and hence are called NULL messages. As a result of the message transmission, variables in the process executing the input command receive the corresponding values in the message. Execution of the input command by process j will be completed only if process i issues the corresponding output command, and vice versa.

Another operation that is often useful in distributed programming is a parallel command. A parallel command consists of a set of commands which may be executed in any arbitrary order (or even simultaneously, when possible). In our distributed simulation programs we restrict the constituents of a parallel command to be only input commands or only output commands.

For example, the parallel command issued by process k which wishes to input from two other processes i and j is

denoted by

[RECEIVE (i,<variables>) || RECEIVE(j,<variables>)]

The programmer should guarantee that the same variables do not appear in both the input commands. The parallel command issued by process i wishing to output to two other processes j and k is denoted by

[SEND(j,<msg>) || SEND(k,<msg>)].

The execution of the parallel command will be completed only when all the constituent commands are completed.

2.3 DELAY PROCESS

A delay process models a single server in the physical system. It has an input line and an output line through which all communications take place. The schematic of a delay process is shown in fig 2.3.1.

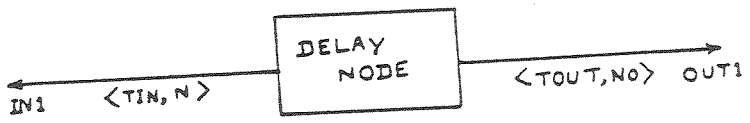


fig 2.3.1

The delay process maintains certain local variables, namely

TIN ~ time of arrival of the last job to this node.
(t value of the last message received).

TOUT ~ time of departure of the last job from this node (t-value of the last message output).

N ~ number of jobs which arrived at time TIN (n-value of the last message received).

J ~ total no. of real jobs yet to be processed by this node.

CLOCK ~ the earliest time at which a real job will be processed, when there is a real job waiting to be processed.

The delay process waits for a message on its input line. A message of the form <TIN,N> indicates that N jobs (N>Ø) arrive at this server at time TIN in the physical system. On receiving a message the delay node simulates the servicing of each job and outputs a message to the process that is connected to its output line. Occasionally NULL jobs (messages of the form <t,Ø>) arrive at the delay node. These do not correspond to any real job. A message of the form <t,Ø> in the simulator implies the absence of any job until time t at the corresponding server in the real system. The messages concerning NULL jobs are necessary for the avoidance of deadlock in the simulator.

The delay node on receiving a tuple <TIN,N> computes the output tuple <TOUT,No> as follows:

```
if N>Ø then TOUT = max(TIN,TOUT) + SERVICE_TIME ; No = 1;
if N=Ø then TOUT = TIN + SERVICE_TIME ; No = Ø;
```

When N=0, SERVICETIME represents the service time of the next real job.

The actual algorithm for the delay process appears below. In this algorithm i is the name of the process that is connected to the input line and k is the name of the process that is connected to the output line of the delay process j.

ALGORITHM 2.3.1

```
    PROCESS j ::
        var
            TIN, TOUT, CLOCK : real;
            J, N : integer;
            (* all variables are initialized to 0 *)
        repeat
            if TIN<TOUT then
                                (* next event is an arrival *)
                                begin
                                  (* input message *)
                          L:      RECEIVE(i,<TIN, N>);
                                  J:= J+N;
                                  (* advance CLOCK if
                                     possible *)
                                  if (J=N) and (N>0) then
                                  CLOCK:= max (CLOCK,TIN);
                                end
```

```
else if (TIN>=TOUT) and (J>0)  then
                    (* next event is a departure
                       of a real job *)
                    begin
                      (* prepare message *)
                      TOUT:= CLOCK+SERVICE_TIME;
                      CLOCK:=TOUT;
                      (* output message *)
              LL1:    SEND(k,<TOUT,1>);  J:= J-1;
                      SERVICE_TIME := NEXT(SERVICE_TIME);
                      if (J=N) and   (N>0)  then
                      CLOCK:=  max (CLOCK,TIN);
                    end
          else if (TIN>=TOUT)  and (J=0) then
                    (*  no real job *)
                    begin
                      (* prepare NULL message *)
                      TOUT:=TIN+SERVICE_TIME;
                      (* output NULL message *)
              LL2:    SEND(k,<TOUT,0>);
                    end
forever.
```

As long as TIN<TOUT, the delay process keeps receiving messages and maintains a count of number of jobs. Whenever TIN>=TOUT and there are real jobs to be processed (ie., J>0), then a message corresponding to a real job is

output; if J=Ø then a message of the form (t,Ø), corresponding to a NULL job is output.

The delay process should also compute the statistics such as thruput,mean queue length, mean wait time etc. of the server it models. This is not shown in algorithm 2.3.1, since we are only interested in the performance of the distributed simulation algorithm. However the method proposed by Chandy[2] can be adopted for computing queue statistics.

2.4 FORK PROCESS

A fork process has a single input line and two output lines through which incoming messages are routed. Schematically a fork node is represented as in fig 2.4.1. There is no processing

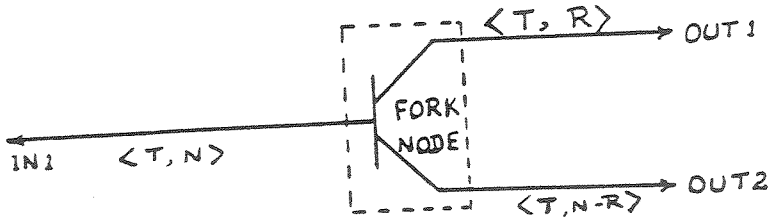


fig 2.4.1

done at this node. That is, the arrival time and the departure time of a job arriving at the fork node are the same. The output path taken by a job arriving at the fork node depends on the branching probabilities of the physical system being simulated.

The function of the fork process is captured in the following algorithm where i is the identity of the process from which the fork process can receive messages; k and m are the processes to which the fork process can send messages.

ALGORITHM 2.4.1

```
      PROCESS j ::

          repeat
            (* input message *)
      L:    RECEIVE(i,<TIN,N>);
            (* determine the number of jobs going to
               each output line *).
            compute R a random number, Ø<=R<=N;
            (* output messages to processes k and m *)
      LL:   [ SEND(k,<TIN,R>) || SEND(m,<TIN,N-R>) ];
          forever.
```

The fork process first waits for a message on its input line. When a message arrives, it determines the number of jobs going to each output line and then outputs on both its output lines simultaneously by issuing a parallel command. Note that, in the event that all the jobs that arrive at an instant take the same output path, the fork process sends out a NULL job along the other output line. This is to inform the process connected to the latter line

that no real job can be expected on that line until the specified time. When a NULL job arrives at a fork node, it sends out a NULL job on both the output lines.

2.5 MERGE PROCESS

A merge process receives input on two lines and sends output on a single line as shown in fig 2.5.1. No processing is done on the jobs that arrive at this node. However, the merge process orders the messages in chronological order of their t-values before sending them out. This ensures that the sequence of messages leaving a merge node matches the chronological order of jobs leaving the corresponding merge node in the physical system.

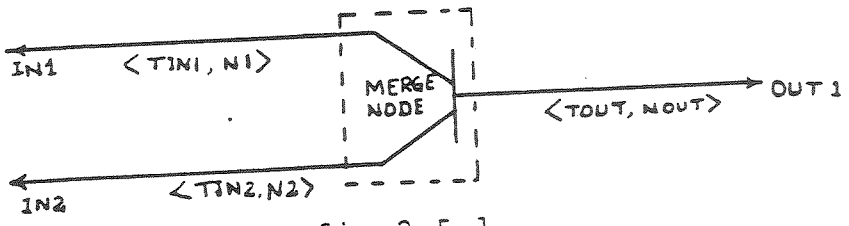


fig 2.5.1

In the algorithm 2.5.1, for merge process, i and k are the identities of the processes from which the merge node can receive messages. ~~M~~ is the identity of the process to which the merge node can send messages.

The local variables used by the merge process are:

TIN1 ~ t-value of the last message on input line 1.

TIN2 ~ t-value of the last message on input line 2.

N1 ~ number of jobs arriving at TIN1 from line 1.

N2 ~ number of jobs arriving at TIN2 from line 2.

```
ALGORITHM 2.5.1
 PROCESS  j ::
   L1:    [RECEIVE(i,<TIN1,N1>)  ||  RECEIVE(k,<TIN2,N2>)];
          repeat
              if TIN1<TIN2 then LL1: SEND(1,<TIN1,N1>);
                                L2: RECEIVE(i,<TIN1,N1>);
              else if TIN1>TIN2 then LL2: SEND(1,<TIN2,N2>);
                                L3:       RECEIVE(k,<TIN2,N2>);
              else
              if TIN1=TIN2 then LL3: SEND(1,<TIN1,N1+N2>);
                           L4:     [ RECEIVE(i,<TIN1,N1>)
                                   || RECEIVE(k,<TIN2,N2>) ];
          forever.
```

The merge process first waits for messages on both its input lines. On receiving the two messages it compares their t-values and outputs the message with the smallest t-value. A message appearing on an input line will be output only when its t-value is less than that of the message arriving on the other input line. This is important for the correct operation of the simulator, because it is

possible for many consecutive messages appearing on one line to have t-values less than that of the message appearing on the other line. When the t-values of the messages on both input lines are same then a message of the form <TIN1,N1+N2> is output and the process waits for messages on both input lines by issuing a parallel command.

2.6 INTERCONNECTIONS

The interconnection of the processes is determined by the topology of the network being simulated. For example, the queueing network shown in fig 2.6.1a is represented by a collection of delay, fork and merges processes which are interconnected as in fig 2.6.1b.

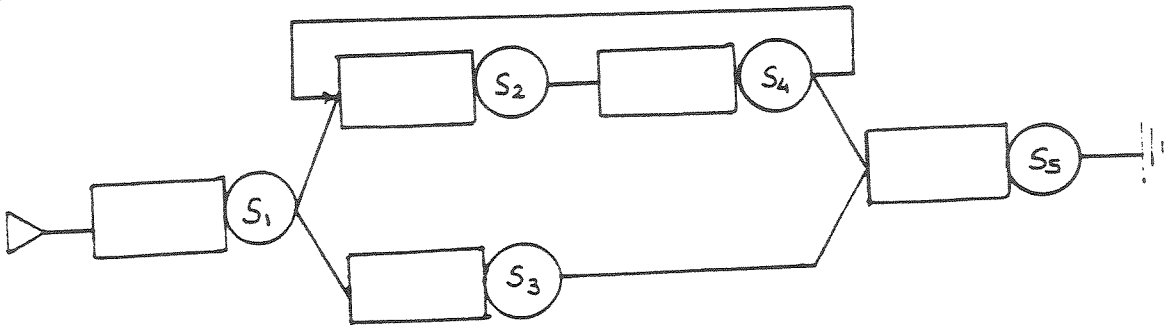


fig 2.6.1a

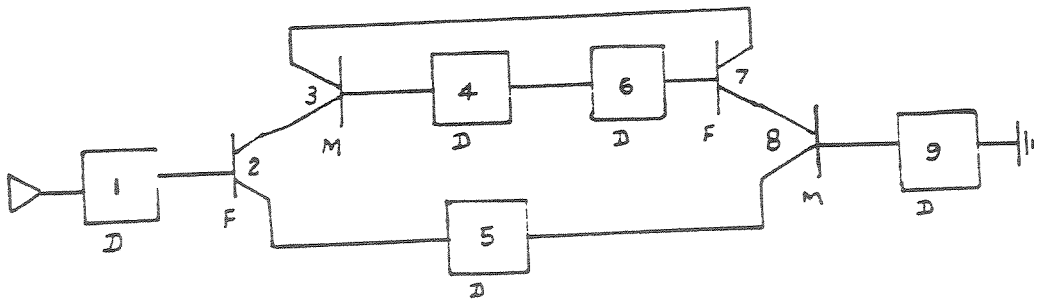


fig2.6.1b

Each server/queue in the physical system is modelled by a delay process. When jobs from one server go directly to only one other server then the corresponding delay processes are interconnected. But when a server feeds to more than one other server (eg.,server 1 in fig 2.6.1a) fork processes are introduced. Merge processes are introduced when more than one server feed to a single server. The interconnection among processes indicate the direction of message flow.

2.7 COMMUNICATION PROTOCOLS

A distributed program is a collection of processes which work on a common problem by communicating with each other only through messages. Since there is no global variables nor a control program, in order to maintain synchronization among the processes we have to establish a set of communication protocols.

In our simulator communication occurs whenever

1) one process names another in a RECEIVE command, as a source of input 2) that other process names the first as destination for output in a SEND command and 3) the target variables of the input command RECEIVE match the value of the message in the output command SEND.

The effect of the SEND and RECEIVE command pair is to assign the value of the message in the output command to the variables in the input command.

If the above three requirements are not satisfied then one of the two commands will wait for the other. We call a process to be 'starved' if it is waiting for the completion of an input operation; if it is waiting for the completion of an output operation it is said to be 'blocked'.

When a process issues a parallel command it will be allowed to proceed only when all the outstanding constituent commands are completed. The order in which the constituent commands are completed is of no concern; but the completion of individual commands is subject to the three conditions mentioned in the last paragraph.

For example, the implementation of the parallel command

[RECEIVE(i,<variables>) || RECEIVE(k,<variables>)]

appearing in the merge process will allow the merge process to continue only after both processes i and k have issued the output command SEND(j,<message>).

The interpretation of these protocols will slightly vary when we introduce buffers on the edges connecting the processes. This is explained in the next section.

2.8 BUFFERS

It is readily seen that the work done by the fork and merge processes is much less compared to the work done by a delay process. Hence when processes are interconnected to simulate a network,the fraction of time spent by fork and merge processes in waiting for the completion of input/output commands will be more than that of delay processes. Also depending on the branching probabilities more number of jobs may flow into one patn than another. This may cause processes in one path to be blocked while those in another path will be starved -- though only temporarily. This may result in poor performance.

In order to keep as many processes simultaneously busy as possible, we may introduce buffers on edges. The number of buffers to be provided in each edge will have to be tuned to an optimal value. Increasing the buffer size beyond a certain value may not produce any appreciable improvement in performance.

The communication protocols will now have to be slightly modified. When buffers are introduced the processes no longer communicate directly, but through the buffers. Each process has to know the identity of the buffer pool from/to which it can receive/send messages. Thus a process j wishing to input from process i will now

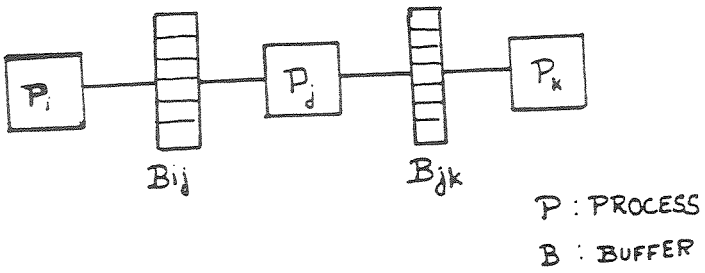


fig 2.8.1

have to address the buffer pool Bij on the edge ij. The interconnection of processes through the buffers is schematically shown in fig 2.8.1. The process j will starve only if all the buffers in the pool Bij are empty. Likewise process j wishing to output messages to process k will send the messages to the buffer pool Bjk. Process j will be blocked only if all the buffers in Bjk are already full. Buffer management and synchronization in this case are similar to those of the classical producer/consumer environment. Proven solutions are readily available. Note that buffers can also be introduced as processes; in that case the same protocols apply.

2.9 THE SIMULATOR

In the light of the above discussion, if a network of queues has to be simulated, we first have to identify the queueing network as a collection of delay, fork and merge nodes and establish the interconnections. Then we have to introduce a source which will simulate the arrival of jobs

at a rate representative of the physical system. A sink will have to be introduced as a final node in the network to absorb the jobs which leave the system. The next step is to assign these processes to different processors. More than one process may be simulated by a single processor depending on how busy we want the processor to be and also on the available memory. Once the simulation starts it continues until many jobs leave the system or statistics with the desired degree of confidence are collected. It is also possible to obtain statistics at any intermediate time. In a conventional simulation this is done by halting the simulation temporarily when the clock reaches a certain value and reporting all the queue statistics. But in tne case of distributed simulation there is no global clock and the local clocks of the processes are not synchronized. Hence it is not possible to halt the simulation at a random time and obtain any meaningful statistics. The only way to get queue statistics at any arbitrary time is to stop each delay process individually when its local clock crosses the specified time and output the desired information regarding that server.

CHAPTER 3

IMPLEMENTATION DETAILS

3.1 INTRODUCTION

The goal of this report is to study the performance of the distributed simulation algorithm described in chapter 2. One way to evaluate the performance of this algorithm is to actually implement it on a network of processors and observe its performance characteristics. An alternate method is to simulate the distributed algorithm on a uniprocessor. The latter approach is taken here. That is, the distributed simulation algorithm itself has in turn been simulated on DEC1Ø in Pascal. This simulator of a simulator works as follows. It simulates the three processes (delay,fork, and merge) and with the help of a monitor program it creates a network of communicating processes and maintains synchronization among them. It has to be emphasized that this monitor program to control and synchronize the processes is needed only because of the presence of two levels of simulation; had the algorithm been implemented in a real network of processors there would be no need for such a control program. This chapter gives a detailed description of the monitor and the data structure maintained by it, the simulation of the three processes, and the performance statistics collected.

3.2 THE MONITOR AND ITS DATA STRUCTURE

The monitor program receives as input, the description of the real queueing system to be simulated. It then configures a network of processors and assigns a process (delay, fork, or merge) to each processor. After establishing the interconnection among the processors, it initiates and supervises the simulation. It also gathers relevant statistics and outputs them at the end of tne simulation. In effect the monitor creates a set of virtual processors such that each one of them can model a single node in the queueing network. When processors issue interprocessor communication commands, the monitor takes control and performs the actual exchange of messages between the processors.

Crucial to the operation of the monitor (and nence to our simulation) is the data structure that is used to store information about the virtual processor network. The data structure maintained is a table; an entry in the table has 13 fields which contain all the information regarding a single virtual processor. The diferent fields in tne table are described below:

field no	description
1	an identification for the virtual processor.
2	the type of process (delay,fork, or

	merge) that runs on this processor.
3	the status of the processor (BUSY/READY/WAITING).
4,5	id of the processors from which it can receive messages.
6,7	id of the processors to which it can send messages.
8,9,10,11	boolean fields indicating if (and why) the processor is waiting.
12	pseudo-program counter (explained in section 3.3).
13	local data for the process running on this processor.

An example of an entry in the table is shown in fig 3.2.1. Fields 5,7,9,11 are not applicable to certain processes, since some processes have only one input line or one output line.

1	2	3	4	5	6	7	8	9	10	11	12	13
PROCE-SSOR ID	TYPE OF PROCESS	STATUS	IN1 IS CONNEC-TED TO	IN2 IS CONNEC-TED TO	OUT1 IS CONNEC-TED TO	OUT2 IS CONNEC-TED TO	WAITING FOR MESSAGE ON		WAITING TO SEND ON		PSEUDO PROGRAM COUNTER	LOCAL DATA FOR PROCESS
							IN1	IN2	OUT1	OUT2		
5	MERGE	WAITING	3	4	6	—	YES	YES	NO	—	L1	TIN1, N1, TIN2, N2

fig 3.2.1

The table has to be rather elaborate. It is possible to deduce the information in some fields from those in other fields; but since the monitor will have to check each entry in the table many times, it is more eficient to store all the information explicitely.

The entries in the table are made by the monitor. The function of the monitor is abstracted as shown below:

```
MONITOR::
      begin
        form table from input data.
        REPEAT
           1. go through the table and give control to the
              processors which are ready.
           2. exchange messages between pairs of processors
              which are waiting for the completion of SEND/
              RECEIVE commands and then pass control to the
              two processors in turn.
           3. collect statistics.
         UNTIL end-of-simulation.
      end.
```

The input to the program is a description of the queueing network that is to be simulated by the distributed simulation algorithm. It contains such information as the number of nodes,type of each node and their interconnection. For example, if the queueing network shown in fig 2.5.1 were

to be simulated then the input to the simulator would appear as follows:

```
9                    {no of nodes}
1 delay              {type of nodes}
2 fork
3 merge
4 delay
5 delay
6 delay
7 fork
8 merge
9 delay
source  1 in1    {interconnections}
1 out1  2 in1    {in1,in2 are input
2 out1  3 in2    {lines  of a node;
2 out2  4 in1    {out1,out2 are output
3 out1  5 in1    {lines of a node;
4 out1  8 in2
5 out1  6 in1
6 out1  7 in1
7 out1  3 in1
8 out1  9 in1
9 out1  sink
```

The monitor, as it reads the input, fills the table. It also validates the input for any possible inconsistency

in the description.

After initializing the table, the monitor starts the simulation of the queueing system. It repeatedly goes through the table sequentially and when it finds the status of a virtual processor to be READY (initially all are READY) it passes control to it. 'Passing the control to a virtual processor' means that the process which is assigned to that processor is excuted. The virtual processor is said to relinquish control back to the monitor when the process issues a SEND or RECEIVE command. The monitor then marks the status of this processor as WAITING and checks the next entry in the table.

The status of the processors can be READY, BUSY or WAITING. The state diagram for process i is given in fig 3.2.2.

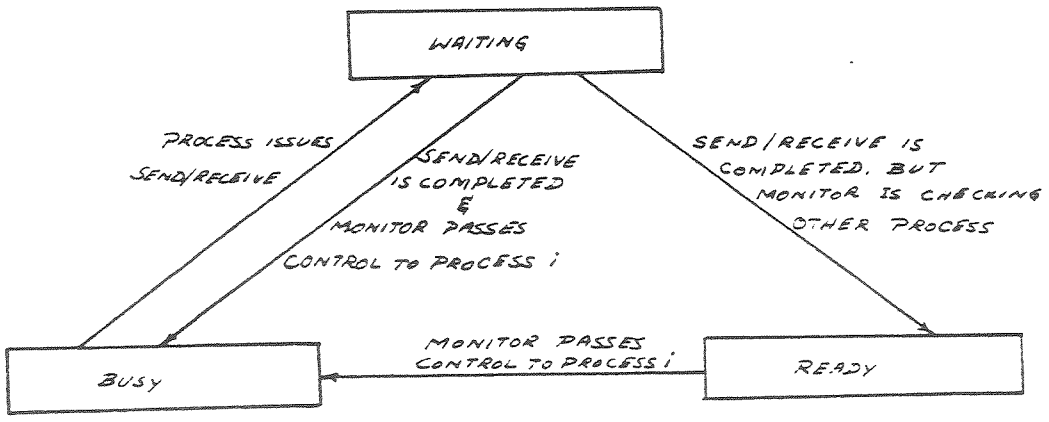


fig 3.2.2

When the monitor finds a processor to be waiting for the completion of a SEND/RECEIVE command it checks to see if there is another processor waiting to RECEIVE from / SEND to the former; if there is such a processor it exchanges the messages between them, marks the second processor READY and passes control to the first processor. The status READY is required only in our two level simulation. In an actual network environment the processors will either be BUSY or WAITING. Each processor will remain in the BUSY state for a time period dependent on the type of process that is running on it. This is discussed in section 3.3.

When the monitor checks all the entries in the table once, it is said to have completed one cycle. At the end of each cycle it gathers statistics about the virtual processors. The monitor also keeps count of the number of jobs that leave the underlying queueing system simulator. It stops the simulation and outputs the statistics regarding the distributed simulation, when enough jobs have left the system.

3.3 IMPLEMENTATION OF DELAY,FORK, AND MERGE PROCESSES

If we work in an actual network environment then the processes will be programs running on real processors, and it will be easy to block and restart them according to the

communication protocols. The program counter will always give the point from which a process has to be resumed after the communication is completed. To simulate this effect in our approach, we introduce a pseudo program counter with each entry in the table. Whenever a process issues a SEND/RECEIVE command the associated pseudo program counter takes on a specific value and control is passed back to the monitor.After the commands are completed (by the monitor) the process continues from the point determined by its pseudo program counter. The value that the pseudo program counter can take corresponds to one of the labels L's or LL's in the algorithms in chapter2. The processes are implemented as procedures and the value of the pseudo program counter acts as an entry point.

Some details about the simulation of actual execution time of the three processes: It can be observed from the algorithms in chapter2 that the amount of work done by a process between consecutive interprocess command varies for different processes. For example the delay process performs more work after it has completed a SEND command than other processes. To account for this fact we adopt the following conventions: the virtual processor with delay process, remains in BUSY state for two cycles after completing a SEND command and for one cycle after completing a RECEIVE command, while the virtual processor with fork or

merge process remains BUSY for one cycle after completing any interprocessor command.

3.4 COLLECTION OF STATISTICS

In order to evaluate the performance of the distributed simulation algorithm, diferent parameters are measured during our simulation. These include

1) number of virtual processors needed. (this depends on the number of nodes in the queueing system to be simulated.)
2) number of cycles completed by the monitor at the end of the simulation. (turnaround time.)
3) total number of messages transmitted in the system.
4) number of messages which correspond to NULL jobs.
5) fraction of time each delay node was BUSY/WAITING.
6) fraction of time n delay nodes were simultaneously BUSY/WAITING.
7) number of real jobs processed by each delay process.

How these parameters are related to the performance of the algorithm is the topic of the next chapter.

3.5 MODULARITY IN IMPLEMENTATION

The implementation of this two level simulation has been done in a modular fashion. The module decomposition is done in such a manner that the set of requirements which are likely to change are captured within a module while the module interface is built around the set of requirements which are unlikely to change. For example, the requirements of a delay process may change to accommodate different service discipline at the server. But the interface between the delay process and other modules remains the same.

The hiearchy of modules is shown in figure 3.5.1. There are 4 distinct layers. Procedures in each layer can call only other procedures in the adjacent inner layer.

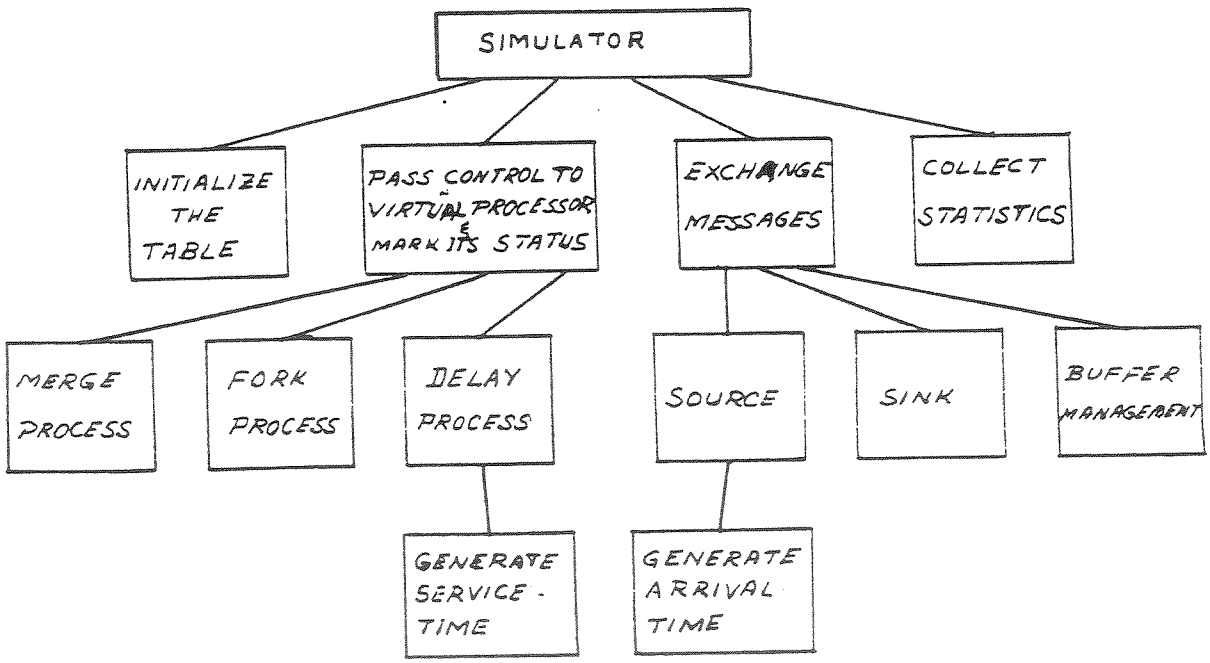


fig 3.5.1

The program can easily be modified to accommodate larger networks and to study the effect of number of buffers, service rates, branching probabilities, etc., on the performance of the algorithm, by suitably changing certain parameters of the program.

CHAPTER 4

EXPERIMENTAL RESULTS AND ANALYSIS

4.1 INTRODUCTION

Several different queuing networks were simulated using the distributed simulation algorithm. Especially effects of varying 1) the topology of the network, 2) number of buffers on each edge, 3) service rate of the servers, and 4) branching probabilities, on the performance of the algorithm are observed. The results are presented in this chapter.

4.2 PERFORMANCE METRICS

The parameters measured by the upper level simulator, described in section 4 of chapter 3 are directly related to the performance metrics of our algorithm, such as turn around time,speed up factor, efficiency,interprocessor communication overhead,processor utilization etc. These are defined in the following sections.

4.2.1 TURNAROUND TIME

Turn around time of a simulation algorithm is defined as the time taken to simulate a given number of events. In our studies the turn around time is measured under the assumption that delay node takes 3 units of time while fork and merge nodes take one unit time each, to simulate a job (which consists of two events, an arrival and a departure). The number of cycles completed by the monitor at the end of simulation gives the turn around time of the distributed simulation algorithm. (simulation ends when 1000 jobs arrive at the sink).

In order to compare the performance of the distributed algorithm with that of the sequential simulation algorithm, we have to evaluate the turn around time of the sequential algorithm. This is described below.

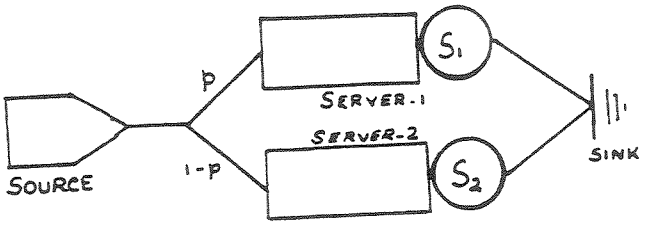


fig 4.2.1

Consider the network shown in fig 4.2.1. The jobs arriving from the source go to one of the two servers depending on the branching probabilities. Let 'b' be the time taken by the sequential algorithm to determine the path taken by an arrival. Also let 's' be the time taken to

simulate the activities corresponding to a single job at a server and 'm' be the time involved in doing the merge operation. Then the turn around time of the sequential simulation algorithm is given by

$$T = n*b + n1*s + n2*s + n*m = n*(b+s+m)$$

where n is the total number of jobs simulated

n1 is the number of jobs that went to server1

n2 is the number of jobs that went to server2

To be compatible with our distributed algorithm, we assume b=1, s=3, and m=1. One of the parameters measured in our experiments is the number of real jobs that went to each delay node. From this and the topology of the queuing network we can analytically evaluate the turn around time of the sequential algorithm.

4.2.2 SPEED UP FACTOR

We use this parameter as the major criterion in evluating the performance of the distributed algorithm. The speed up factor is defined as the ratio of turn around time of sequential algorithm to that of the distributed algorithm,

$$\text{ie., speed up factor} = \frac{\text{time taken to simulate n events using a single processor}}{\text{time taken to simulate n events using N such processors}}$$

Ideally the speed up factor should be equal to N, the number of processors used by the distributed algorithm. Note that, while simulating queuing networks the speed up factor depends on the topology of the network.

4.2.3 EFFICIENCY

We define efficiency as the ratio of actual speed up factor achieved to the ideal speed up factor,

$$\text{ie., efficiency} = \frac{\text{speed up factor measured from experiments}}{N}$$

4.2.4 INTERPROCESSOR COMMUNICATION OVERHEAD

The cost effectiveness of the distributed simulation will depend on the amount of interprocessor communication involved. In our algorithm each departure of a job from a server corresponds to an interprocessor communication. When a job arrives at a fork node it generates two interprocessor

communication requests, since we always output on both output lines of the fork node simultaneously. If there are many branches in the network that is being simulated, there will be many NULL messages generated and hence the overhead due to interprocessor communication may be excessive.

The total number of transmitted messages is counted in all the experiments. This gives a direct measure of the overhead due to interprocessor communication.

4.2.5 UTILIZATION

Since processors are becoming increasingly inexpensive processor utilization is not considered as a major performance characteristic. Besides, with our distributed algorithm, a processor utilization of 100% will not always mean that the processor is doing useful work all the time. It is likely (in networks with many branches) that for an appreciable part of the time the processor is busy processing NULL jobs. However, the information obtained on processor utilization may be used in optimally assigning more than one process to a processor and in buffer assignment.

The rest of this chapter describes the different experiments and the analysis of the results.

4.3 EXPERIMENTS

4.3.1 TANDEM NETWORKS

A tandem network is a set of servers connected in series as shown in fig 4.3.1a. The logical equivalent (which is a collection of delay nodes) is shown in fig 4.3.1b.

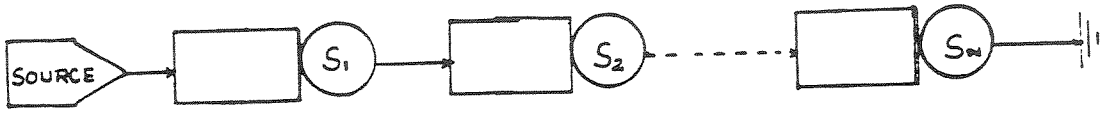


fig 4.3.1a

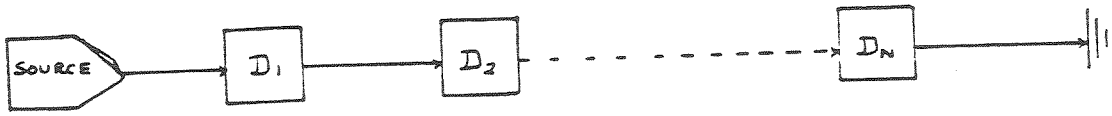


fig4.3.1b

Experiments were conducted for upto 5 servers in series. Table 4.1 shows the results obtained. It has been observed that the turn around time is independent of the number of servers in the network. Also ideal speed up factor of N seems to be achievable for this type of networks.

Table 4.1

Processing of 1000 jobs in a tandem network

no of servers	no of processors	turn around time of distributed simulation	turn around time of centralized simulation	speed up factor	efficiency
1	1	3003	3000	~ 1	~100%
2	2	3006	6000	~ 2	~100%
3	3	3009	9000	~ 3	~100%
4	4	3009	12000	~ 4	~100%
5	5	3012	15000	~ 5	~100%

Further experiments with tandem networks reveal that the number of buffers on the edges does not affect the performance.

4.3.2 ARBITRARY FEEDFORWARD NETWORKS

Here we study networks in which servers are interconnected in any arbitrary manner, except that there is no feedback path. Four different cases are studied under this category.

Case a:

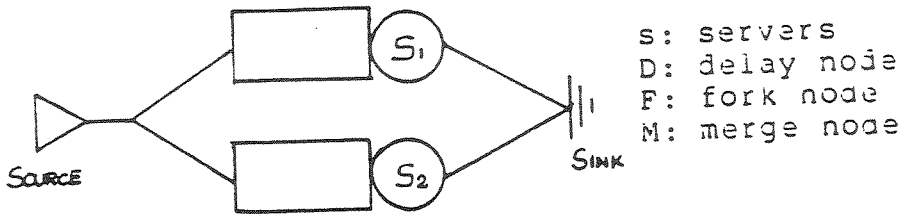
All the servers have the same service rate, the branching probabilities (of fork nodes) are equal and there

are 10 buffers on each edge.

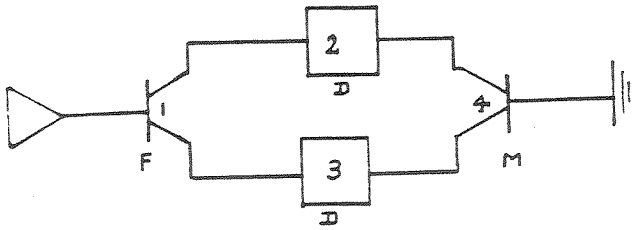
Nine different networks shown in figures 4.3.2.1 through 4.3.2.4 were simulated. Table 4.2 lists the speed up factor and the efficiency obtained in each case.

a: physical system

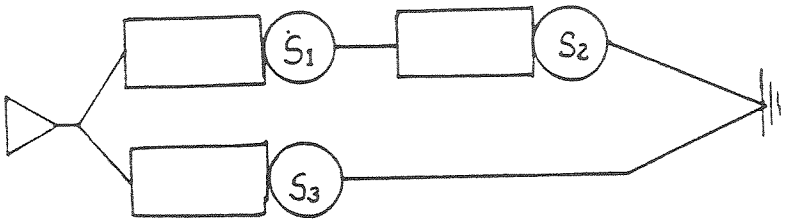
b: logical system



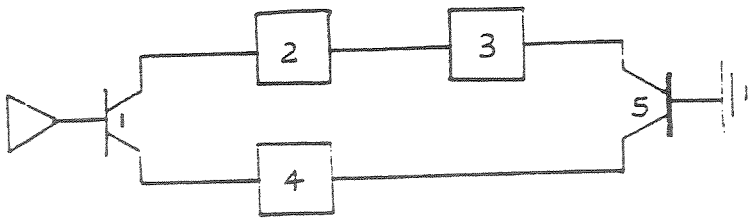
1a



1b

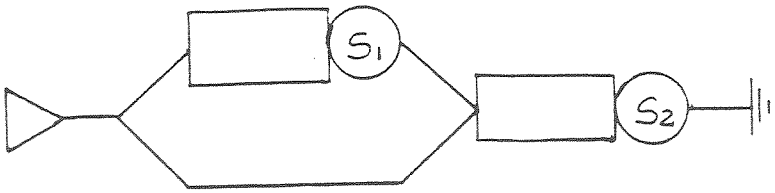


2a

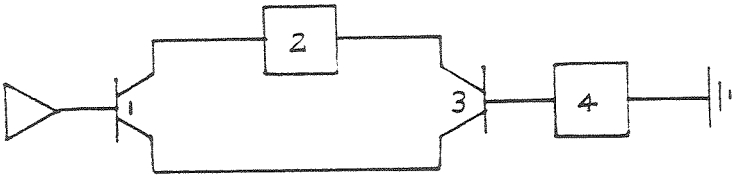


2b

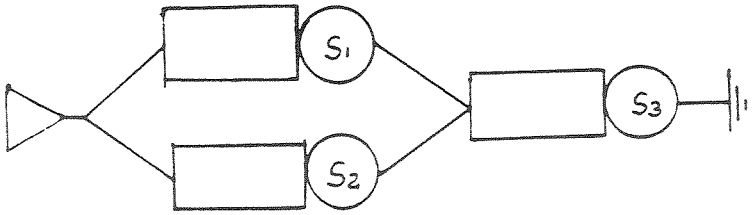
fig. 4.3.2.1



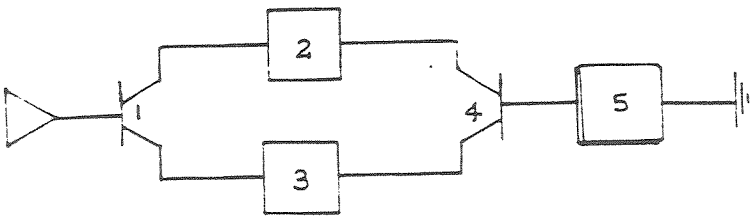
3a



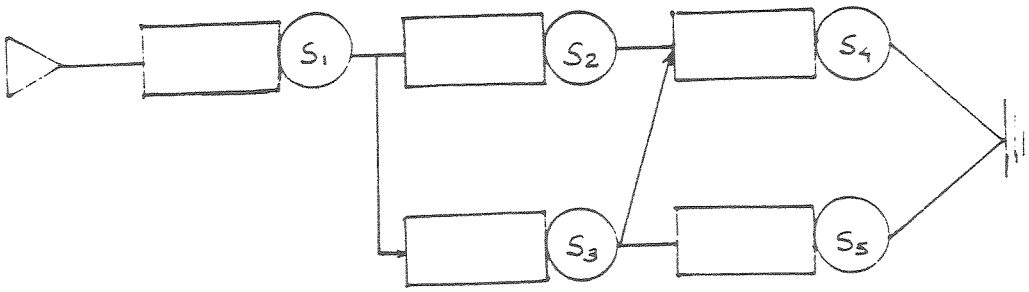
3b



4a

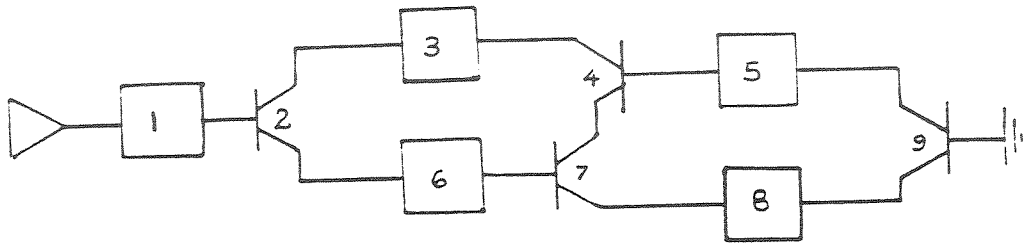


4b

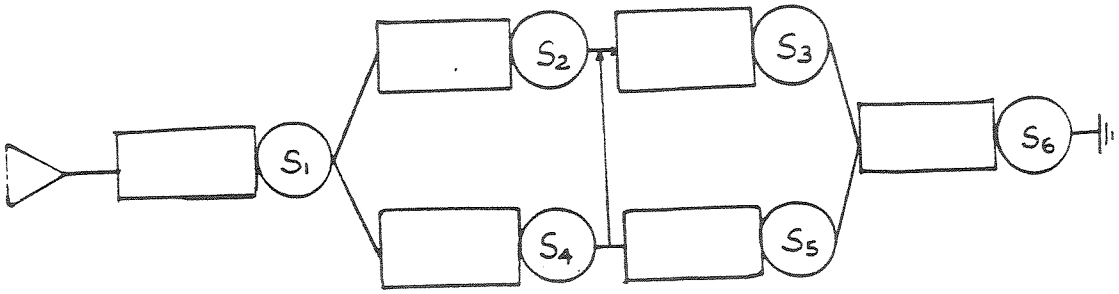


5a

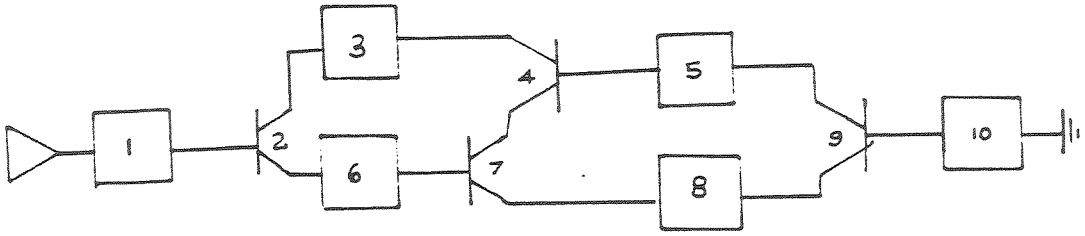
fig. 4.3.2.2



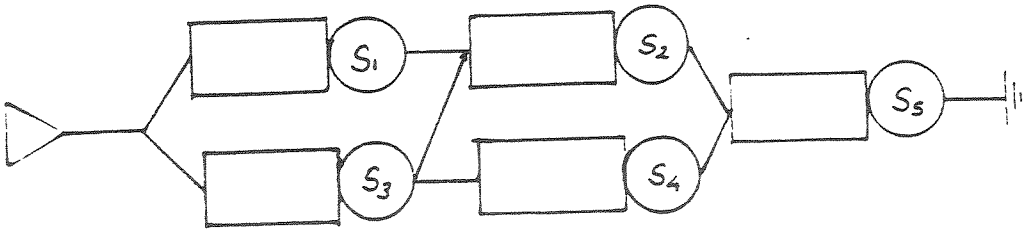
5b



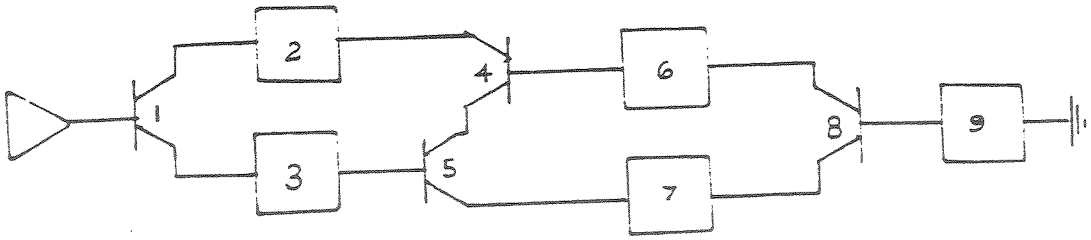
6a



6b

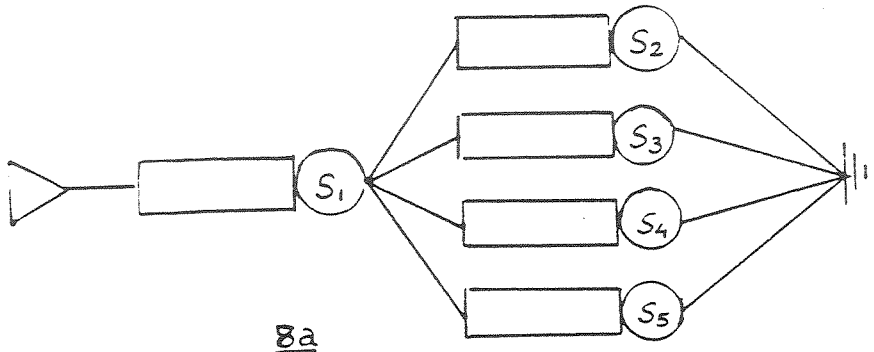


7a

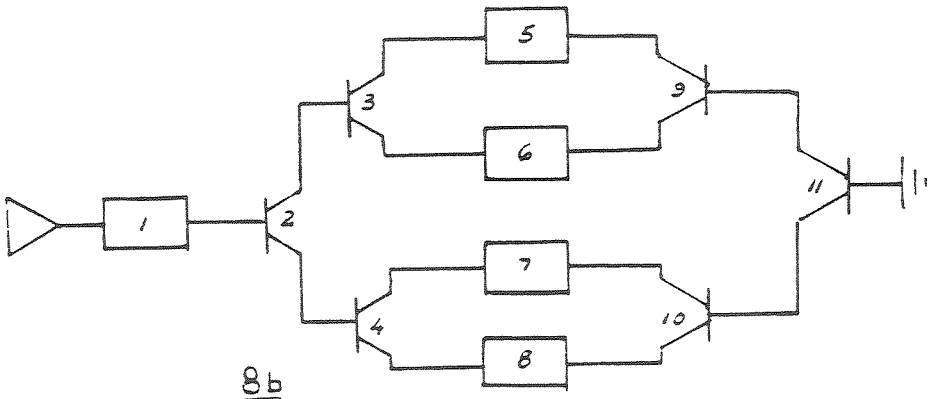


7b

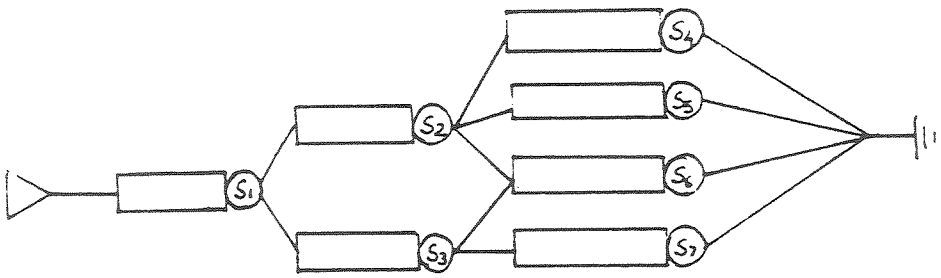
fig. 4.3.2.3



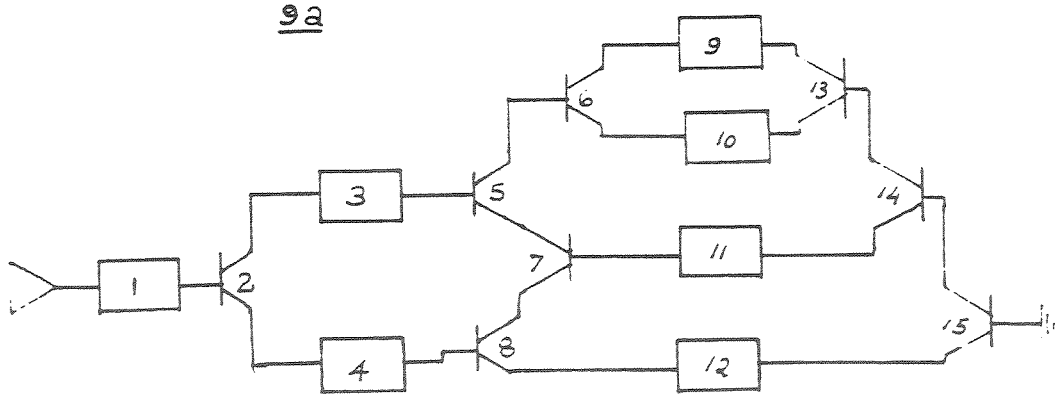
8a



8b



9a



9b

fig. 4.3.2.1

Table 4.2

performance of feedforward networks

network no	no of processors	speedup factor	efficiency
1	4	2.48	62 %
2	5	3.23	65%
3	4	2.19	55%
4	5	2.71	54%
5	9	4.32	48%
6	10	5.23	52%
7	9	4.98	55%
8	11	3.96	36%
9	15	5.33	36%

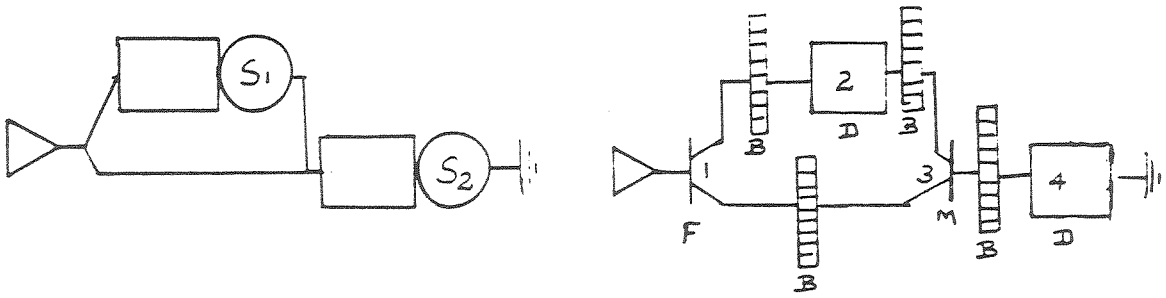
Other parameters, such as number of messages transmitted, processor utilization, etc. can be found in pages 55-63.

From table 4.2 it seems that efficiency is (inversely) proportional to the number of branches in the network. A possible explanation for this could be the following: Since we introduce a processor (fork node) for every two way branch, the number of processors required by the distributed simulation increases. Also the processors which simulate the servers in the branches spend an

appreciable amount of time processing NULL jobs, thus increasing the turn around time. Hence efficiency which is the ratio of turn around time of sequential algorithmr to (turn around time of distributed algorithm * number of processors used by the distributed algorithm), decreases.

Case b:

Here the effect of varying the number of buffers is studied.



physical system logical system

fig 4.3.3

The network shown in figure 4.3.3 is simulated and the number of buffers on each edge is varied from 1 to 20. The turn around time for each case is tabulated in table 4.3.

Table 4.3

Effect of number of buffers

no of buffers	turn around time	% decrease in turn around time
1	3547	0%
2	3175	10.49%
3	3061	13.7 %
4	3030	14.58%
5	3031	14.58%
6	3023	14.77%
7	3023	14.77%
8	3020	14.86%
9	3020	14.86%
10	3027	14.66%
15	3014	15.03%
20	3010	15.14%

The results show that the percentage decrease in turn around time is not appreciable as the number of buffers is increased beyond 3.

Case c:

Same as case a, but the branching probabilities and the service rate may be arbitrary.

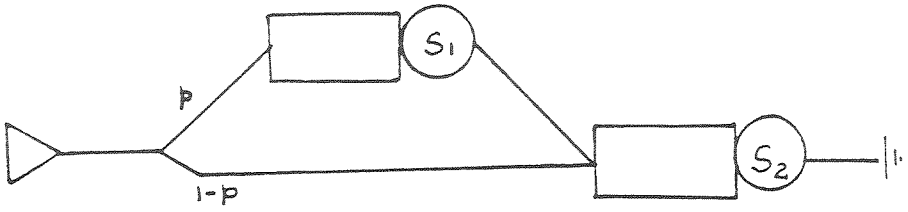


fig 4.3.4

The network shown in figure 4.3.4 is simulated, with the branching probability p varying from Ø.2 to Ø.8. The results show that the turn around time is not affected by the branching probability. This is to be expected since irrespective of the branching probability the total number of messages (real + NULL) sent to delay node 2 is the same.

It has also been observed that changes in the service rate of different servers do not have any effect on the turn around time.

4.3.3 NETWORKS WITH FEEDBACK -

A few networks with feedback paths have been simulated. The performance of the algorithm for feedback networks is rather poor.

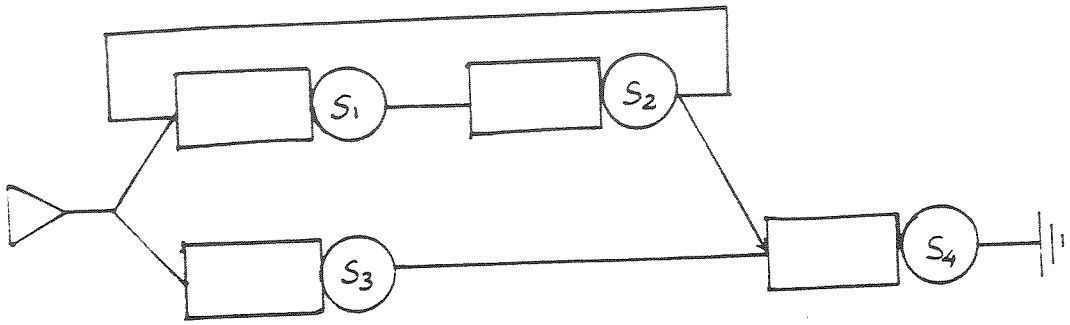


fig 4.3.5

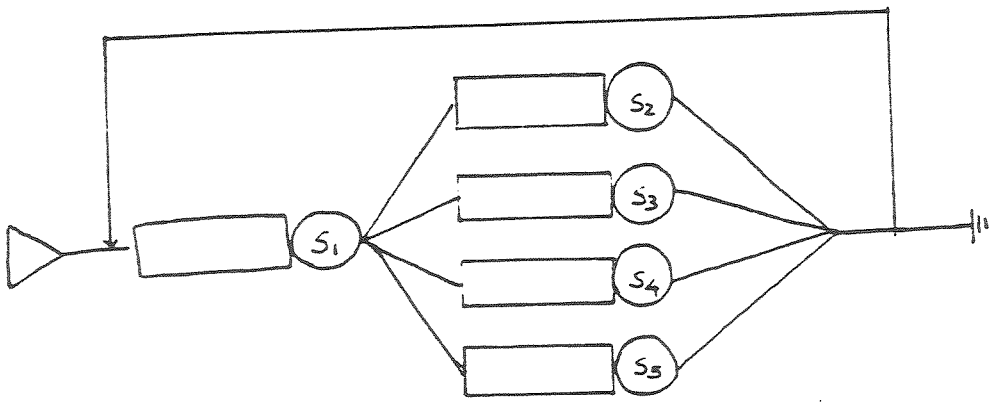


fig 4.3.6

For the network shown in fig 4.3.5 the efficiency achieved was 30% . But, for the network of figure 4.3.6 the efficiency is less than 1% . Also the number of messages with NULL jobs increases tremendously. While simulating feedback networks the processors tend to starve for atleast 50% of the time, in some cases upto 80% of the time. Suggestions for improving the efficiency of the algorithm appear in the next section.

4.4 SUMMARY

The performance of the distributed algorithm for discrete event simulation has been evaluated in this report. We defined the performance of the distributed simulation in terms of efficiency which combines the turnaround time of distributed simulation, turnaround time of sequential simulation, and the number of processors required by distributed simulation. Cost effectiveness was not explicitely considered, though some measure of processor utilization and communication overhead was revealed. No attempt was taken to compute the queue statistics, however, there are some readily available methods.

While simulating tandem networks by the distributed algorithm, turnaround time equal to the theoritical upper bound (which means 100% efficiency) was achieved. Processors were also found to be fully utilized, when each processor in the distributed system was allowed to simulate one server in the queueing network.

When arbitrary feedforward networks were simulated, eventhough 100% efficiency was not obtained, significant improvement in turnaround time was feasible. The performance of distributed simulation of these kind of queueing networks seemed to be inversely related to the number of branches in the network. This can be attributed

to the increased number of processors and NULL jobs generated in these systems. The processors, however, were not observed to be highly utilized. Hence a possible improvement in performance can be achieved by judiciously assigning more than one process to a processor. This, apart from increasing the processor utilization, effectively reduces the number of processors required by distributed simulation, thus increasing efficiency.

It was also discovered that the number of buffers required to improve the performance was very small. This plus the fact that our algorithms for delay, fork, and merge processes were in general quite small emphasize that the memory requirement for each processor is not high.

The above results clearly seem to point in favor of distributed simulation of a class of queuing networks (without feedback path).

The performance of distributed simulation has been observed to be poor while simulating networks with feedback. This is due to the fact that the number of NULL messages in the system formed an appreciable fraction of the total number of messages transmitted. Also the processors in a loop were found to be starved most of the time. These factors clearly call for a significant improvement in the algorithm if it were to be used for simulating any general

complex network.

A possible approach to increase the efficiency of distributed simulation algorithm for simulating feedback networks is to reduce the amount of time the loop is saturated with NULL messages. If we could devise a method to detect the situation in which only NULL messages are circulating within the loop, then the real jobs which are waiting to get into the loop can be forced in. The method discussed by Dijkstra and Scholten[4] could be applied.

To conclude: distributed computer systems are economically attractive due to the availability of processors at low cost. In such systems, it is turnaround time and not processor utilization which is considered a major performance criterion. The performance advantage of distributed simulation, of any queueing network without feedback, over conventional sequential simulation is highly appreciable that we can adapt the distributed approach for descrete event simulation. The percentage improvement in turnaround time, obtained while simulating queueing systems with feedback path is very small. This suggests the need for further refinement in our algorithm or an alternate solution.

BIBLIOGRAPHY

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

2. 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

3. 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.

4. Dijkstra, E.W and Scholten, C.S., "Termination Detection for Diffusing Computations," unpublished manuscript.

5. Hoare, C.A.R., Communicating Sequential Processes, "CACM, Vol.21, No. 8, Aug. 1978.

6. Hoare, C.A.R. and Kaubisch, W.H., "Discrete Event Simulation Based on Communicating Sequential Processes," unpublished manuscript.

7. Holmes, V., Ph.D Thesis, University of Texas at Austin, 1978.

8. Iglehart, D.L., "The Regenerative Method for Simulation Analysis," Current Trends in Programming Methodology, Vol. III. Software Modeling and Its Impact on Performance, (Chandy, K.M. and Yeh, R.T., editors) Prentice Hall, Englewood Cliffs, N.J., 1978- 52-71.

9. Peacock, J.D., Wong, J.W. and Manning, E., "Distributed Simulation Using Network of Microcomputers," Computer Networks, Vol.3, No.1, Feb. 1979.

10. Jensen, K., and Wirth, N., PASCAL User Manual and Report, pringer-Verlag, New York, 1974.