DISTRIBUTED SIMULATION OF NETWORKS*

TR-81 1978

K.M. Chandy
Victor Holmes
J. Misra

Department of Computer Science
University of Texas
Austin, Texas 78712

* Supported by NSF Grant MCS77-09812

Abstract

A potentially valuable attribute of message switched networks is that all processors in the network can cooperate in solving a common problem. This attribute has not received sufficient attention in the literature probably because it is hard to partition most programs into processes which communicate exclusively by exchanging messages. The problem of partitioning programs and assigning them to processors in message switched systems becomes acute when programs appear to be inherently sequential. In this paper we use a message switched network to solve a problem that has always been solved in a highly sequential fashion. The specific problem that is studied is discrete-event simulation though key concepts can be extended to other areas of message-switched problem-solving. There are no shared variables in message switched networks. The shared variable "clock" typically used in simulation algorithms, does not appear in the proposed scheme; instead each process maintains an internal clock that is not usually synchronized with clocks of other processes. The case where the network is a tandem of servers is considered in detail in this paper. The core ideas reported here were significantly developed and radically extended by a group at the University of Waterloo under the direction of Professors Manning and Wong.

1. INTRODUCTION

1. The Problem of Distributed Programming: A Simulation Example

The advent of inexpensive and increasingly powerful minicomputers has increased pressure to create parallel programs for a variety of applications. We are interested in distributed programs which are special cases of parallel programs; a distributed program is a collection of parallel processes which only communicate with each other by passing messages. A key factor in distributed programs is that concurrent processes have no shared variables. One method for generating distributed programs is to recognize parallelism in programs written for serial machines and to then reconstruct the program using parallel constructs. However, many existing algorithms do not lend themselves to distribution by this method. In these cases radically different algorithms are required. We are interested in the creation of distributed algorithms in general; however, in this paper we shall restrict attention to distributed algorithms for event driven simulations. The concepts we use in distributing simulations can be applied to other important areas. Our goal is to use message switched networks to solve parallel problems.

1.1 The Importance of Distributed Simulation: the problem of parallel time

a) The apparently sequential nature of simulation

Conventional simulation algorithms appear to be inherently sequential for the following reason. The key data structure in event driven simulations is the event list, which is a list of expected future events ordered in increasing order of expected time occurrence. Simulations proceed by processing the next (i.e., top) event in the event list and by moving the simulation clock to the time of occurrence of this event. In general, events must proceed in strict chronological order. The processing of one event may

result in the spawning of several new events, one of which may be the next event to be processed. Thus, to ensure strict chronological order, events can be processed only one at a time, resulting in an apparently sequential program. Parallelism can only be achieved by changing the structure of the event list so as to capture the independence as well as the interdependence of the processes being simulated. Thus simulation provides an interesting problem for the creation of distributed algorithms.

Our goal is to construct a simulator consisting of several parallel processes with no shared variables. Communication is permitted only by passing messages between processes. This implies that each process must maintain its own clock (time) and hence the strict chronological sequence of events that occurs in the real system must be realized in the asynchronous simulator solely by messages between processors.

b) Importance

Simulations are widely used in analyzing systems including job-shops, computers and communication networks. Computational expense and/or the real time required to run a significant simulation may inhibit the use of simulation in these areas. Parallelism may reduce this problem.

1.2 Desirable Criteria for Distributed Algorithms

a) Inter-Processor communication

Communication between processors may be a sizeable overhead in distributed processor systems: hence inter-processor communication should be kept to a minimum.

b) Memory

Each processor in a distributed system has access to its local memory only. Hence the memory available to each processor may be less than the amount available to conventional processors. Memory requirements should be kept to a minimum.

c) Natural representation

Simulations are rarely checked for correctness in a formal manner. This is because simulations are not usually specified formally. In many cases the informal specification for a simulator is an existing system such as a manufacturing job shop (though the values of parameters must be changed more easily and with less expense in the simulation than in the real system). It is important that simulators be natural representations of the systems to be simulated because even though a simulation is not formally proved to be correct, a decision-maker is likely to believe in a simulation if he can easily see the correspondence between events and processes in the real and simulated systems. An obvious correspondence between real and simulated systems is also less likely to lead to errors. Very often, the system being simulated is a parallel processor system; in this case the correspondence between reality and the simulation model is more obvious if the simulator is also a parallel processor system.

d) Correctness

An interesting problem in its own right is to formally specify a simulator and to then prove that the simulator meets specifications. This problem becomes doubly interesting when the simulator is a distributed program. In this paper we prove that the distributed simulation is

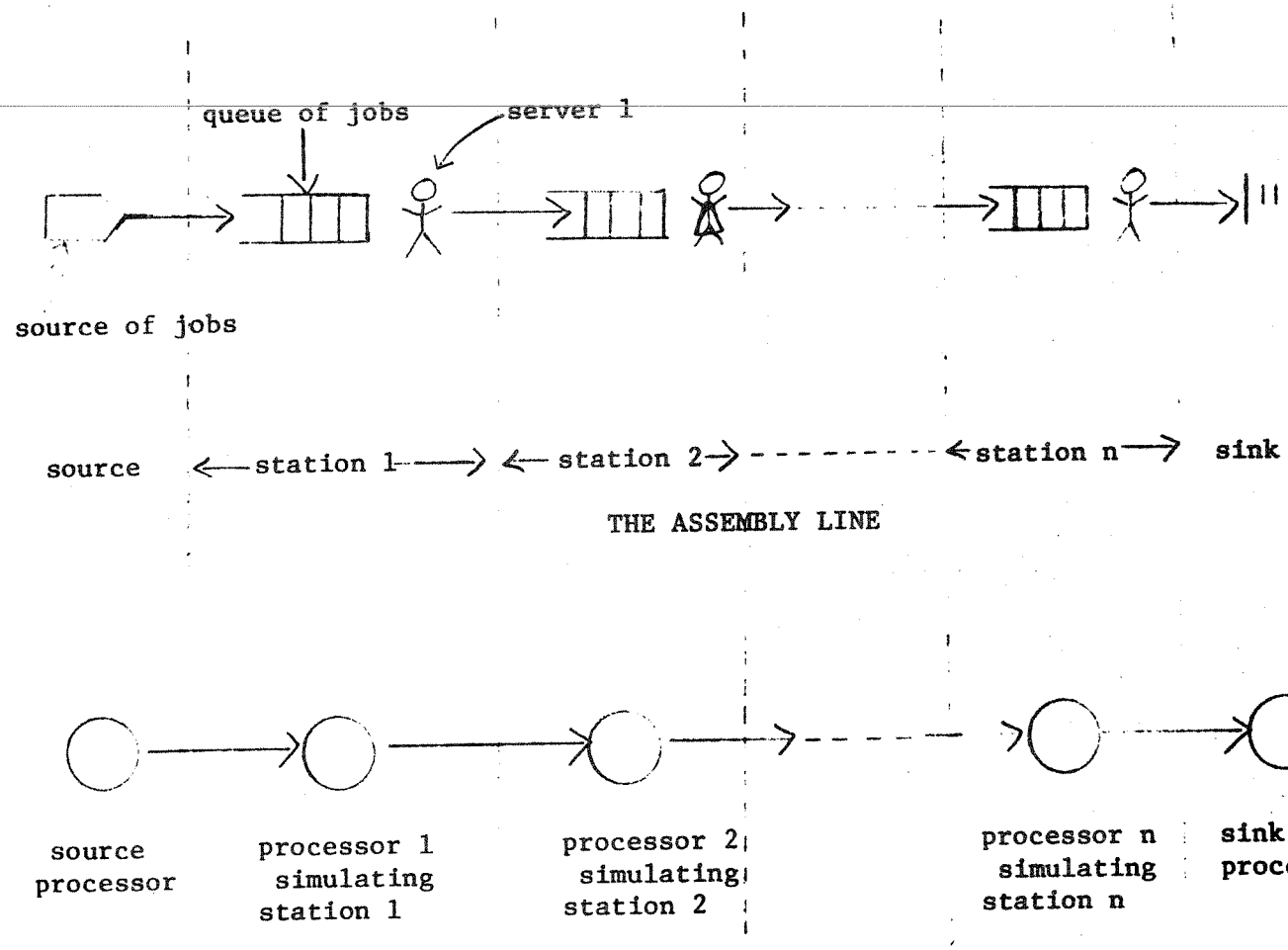
correct. We shall show that every process in the distributed simulation will continue to run until the simulation is over (i.e., deadlock is impossible) and that the simulator faithfully represents the real system. For a detailed formal study of correctness of distributed programs see [CHANDY and MISRA].

1.3 An Example: Assembly Lines

To illustrate the key concepts associated with distributed simulation we shall use the simple example of an assembly line. An assembly line (figure 1) consists of a series of n work-stations. Jobs enter the assembly line at work-station 1; when a job has been processed at work-station i it proceeds to work-station i+1, i = 1,2,....,n, until it is completed and leaves the system. Service times at different stations are arbitrary random variables. There are buffers between stations. A server at a station takes one job from its input buffer, processes that job and then places the job in its output buffer (which is the input buffer to the next station). All stations are assumed to be served in a First-Come-First-Served (FCFS) manner. Initially, we will assume that buffers have infinite capacity. Our objective is to find queue length and wait time distributions.

1.3.1 Conventional simulations

In conventional simulations the key data structure is the event list which consists of the departure times from all the work-stations and from the job source. To process the next event we (a) determine the next work-station (say work-station i) to have a departure and the time t of the departure, (b) move the simulator clock up to t, (c) remove this event (departure at t from work-station i) from the event-list, (d) decrement the number of jobs in work-station i by 1 and



SIMULATION OF THE ASSEMBLY LINE

The distributed simulation of an assembly line

Figure 1.

increment the number in work-station i+1 by 1, (e) if the number of jobs at work-station i is non-zero, compute the time of the next departure from i and insert this event into the event list, (f) if there is currently no event in the event list corresponding to a departure from i+1 compute the time of the next departure from i+1 and insert this event into the event list.

In conventional simulations the behavior of any subsystem may depend upon the state of any other subsystem (or possibly upon the entire system). For instance, the service time at the ith work-station may depend upon the numbers

of jobs at work-station 1,....,n. The generality of the conventional simulation approach has resulted in its widespread use. However, it is difficult to run such a simulator on multiple processors.

1.3.2 Distributed simulations: Time Exchange Systems

There is a limited class of problems which lends itself naturally to distributed simulation. The goal of distributed simulation is to (a) partition the system being simulated into relatively independent subsystems which communicate with each other in a simple manner (such as passing customers or jobs from one subsystem to another), and (b) simulate each subsystem on a different processor. For instance, in the assembly-line example, if the service time at each work-station were independent of all other work-stations, then each work-station could be treated as an independent subsystem and simulated on a different processor. If all the parts of a system are mutually interdependent then distributed simulation provides no advantage, because the overhead involved in communicating (information about the status of several subsystems to other subsystems) negates the advantage derived from concurrent processing. In a forthcoming paper we shall discuss the simulation of systems with interdependent subsystems.

The crucial issue in distributed simulation is that of simulating time on multiple processors. Two methods are suggested: the first method is not a distributed program because it has a global variable.

(a) Time Driven Simulation: A Method That is Not a Distributed Program

All processors keep the same simulated time. In this case there is a single master clock which drives all the processors. The master clock moves forward in time in discrete steps of fixed size. At each clock step, all the processors transmit information about the events

that happened during that incremental interval of simulated time. In the assembly-line example, in every clock-step, each processor determines if a job leaves the corresponding work-station and communicates that information to the next processor downstream. The advantage of this approach is that the correspondence between the simulator and the simulated system is obvious. The disadvantage is common to all time-driven simulations: the clock-step must be short for purposes of accuracy, but short clock-steps result in long simulations.

(b) TEXS: Time EXchange Systems

In our method a different clock is associated with each connection between subsystems. For example, in the assembly-line case (figure 2), there is a clock (number 0) associated with the connection from the source to the first work-station, and a clock (number i, i = 1,....,n) associated with the interface between the ith work-station and the next one downstream. Each clock moves forward in time in an asynchronous manner. Thus we cannot take a "snapshot" of the entire system by stopping all processors at the same instant. Each processor maintains the clocks for all the connections going out of it (figure 2).

The basic idea is simple. All processors continuously repeat a two-step cycle. (1) Whenever an event occurs on the output line of a processor it sends a message to each processor that it is directly connected to. This message includes the current clock-time of the line connecting the two processors and a description of the event which occurs at that time. (2) Each processor inspects the messages corresponding to all of its input lines and based on these messages it determines the next event on its output line and moves its output lines forward in time to this event. Now, the next cycle starts with the new interface times.

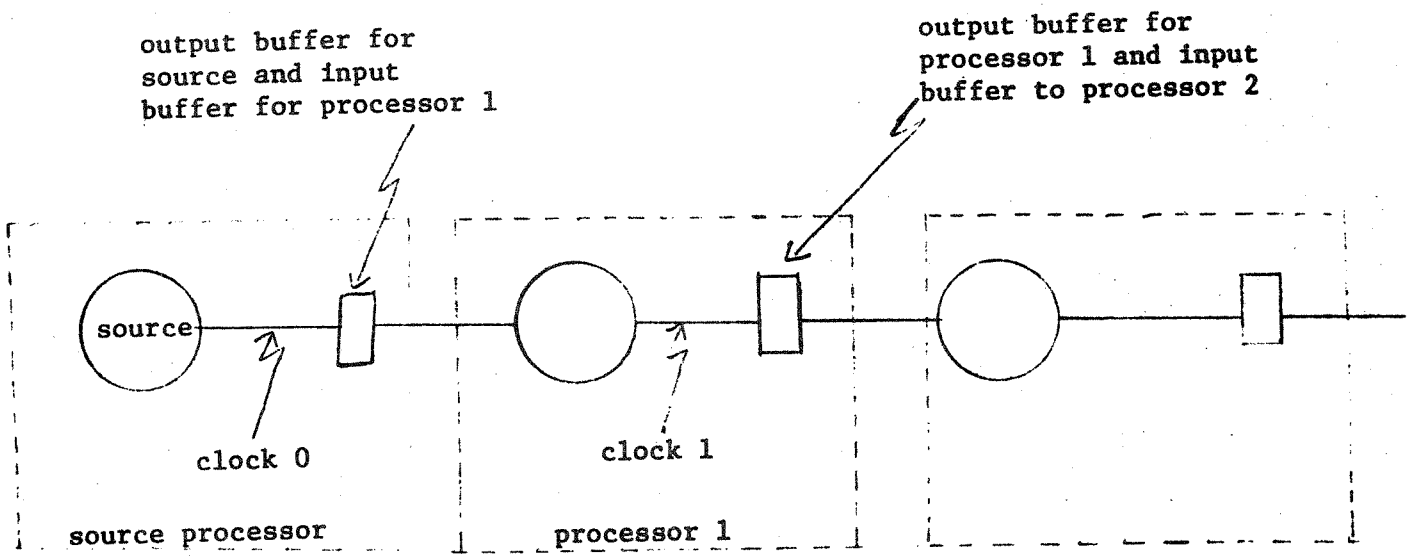


Figure 2.

In the assembly-line example, the clock for the interface between the ith and i+1th processors is moved forward in discrete steps through a sequence of times $t_1, t_2, \ldots$, which are the instants at which jobs move from the ith station to the (i+1)th. The ith processor (i = 1,...,n) computes the time at which the next job will depart from the ith work-station and enter the (i+1)th station; this time is clock-time of the interface between the ith and (i+1)th stations. Given the sequence of arrival times into a work-station (i.e., the sequence of values of the input line's clock) a processor merely computes the sequence of departure times from the station (i.e., the sequence of values of the output line's clock).

Consider an arbitrary work-station. Let A_j, S_j, and D_j be the arrival, service and departure times (respectively) of the jth job, j = 1,2,... . Since the ith job cannot begin to get service until it arrives and the previous job departs, a formal specification of departure times is:

$D_0 = 0$ (assumed for convenience)

$D_i = S_i + \text{maximum } \{A_i, D_{i-1}\}, \quad i = 1,2,3,...$

The ith processor has three local variables: A, D, S which are the arrival, service and departure times for this job. (Processors also keep local variables for computing statistics, but these are not discussed.) There is a one word buffer that the ith processor writes into and the (i+1)th processor reads from. We call this buffer the (i, i+1) buffer. We assume the existence of a protocol which ensures that the (i+1)th processor waits to read the buffer until the ith processor has written into it.

The ith processor repeats the following four-step cycle:

(1) Compute S (The service time is normally computed using the random number generator).

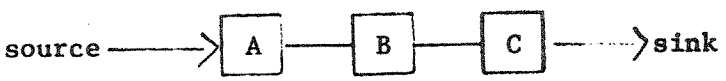
(2) Read the (i-1, i) buffer into A

(3) D:=S + max(A,D)

(4) Write D into the (i, i+1) buffer

An Example

Assume that we have three processes A,B,C connected in sequence as shown below, to a source and a sink.



Source produces 4 jobs 1,2,3,4 at times shown below. Service-times for each of the jobs on each processor is also given in the following table.

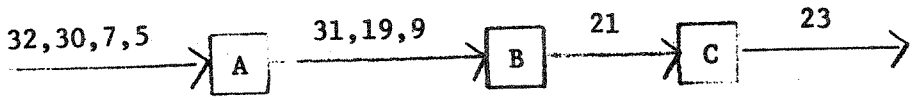
node \ job	1	2	3	4
source (production time)	5	7	30	32
A	4	10	1	5
B	12	15	2	7
C	2	3	1	4

Processing Times

Sequences of departure times corresponding to different jobs, produced by the various nodes are shown below.

node \ job	1	2	3	4
source	5	7	30	32
A	9	19	31	37
B	21	36	38	45
C	23	39	40	49

If we were to stop all the processes in the simulator at some point in time and take a "snapshot" of the simulator we may get the picture shown below.



The sequence of departure times produced by each process is shown on the corresponding line. The source has produced all jobs including job 4; node A has processed up to job 3 and has output it to B; B has processed job 1 and has output it to C; C has finished processing job 1 and output it to the sink. There are 1,2,0 jobs waiting to be processed by A,B,C respectively. As a next step A,B can process their next input jobs, possibly in parallel. Note

that each processor is working at its own point in simulated time. If we were to stop the real system at some point in time, all processes of the real system would be at that point in time. A radical difference between distributed and conventional simulations is that snapshots of distributed simulations do not correspond to snapshots of conventional ones.

We discuss initial and termination conditions and the computation of statistics later; our goal here is to provide a simple intuitive explanation of TEX Systems.

The advantage of TEX Systems is that many processors can work in parallel in a pipe-lined fashion. The primary problem is that it is difficult to determine the state of the system (including such information as queue-lengths) at any instant of real-time whereas this is easily done in conventional simulations. This difficulty is serious since many statistics (such as queue-length) are defined over time. We next discuss a method for solving this problem.

2. Queue-length distributions

It is often necessary to determine the length of time in a simulation that a buffer contains n or more jobs, n = 1,2,.... Once again consider the simulation of a single queue. Let Q(n) be the length of time in a simulation that there are n or more jobs at a service station (waiting for or receiving service). We can specify Q(n) formally in terms of the sequences of arrival times A_1, A_2 and departure times $D_1, D_2, \ldots$.

2.1 Specification

Let t be a non-negative real variable representing time and let $x_n(t)$, n=1,2,... be indicator functions which can take on the values 0 or 1. Define $x_n(t)$ in the following way. Set $x_n(t)$ to 1 if there exists some i such that $A_i \leq t < D_{i-n+1}$; otherwise set it to 0.

$x_n(t) = 1$ if and only if there are n or more jobs in the queue at time t; this is because $x_n(t) = 1$ implies that for some i, the ith, (i-1)th, (i-n+1)th jobs depart after t and all of these jobs arrived before time t. Hence

$$Q(n) = \int_{t=0}^{T} x_n(t).dt$$

where T is the length of the simulation.

2.2 The problem with distributed simulation

In our simulation a processor gets a job from a processor upstream and immediately passes the job on to the next processor downstream. Hence we have to employ some ingenuity in estimating how many jobs there would be in the real system at some time t. An obvious solution is to keep a record of all arrivals and departures, and to deduce queue-lengths from this log. However, this solution is infeasible due to the tight constraints on available memory in mini and microcomputers. We present a solution which does not require storage of more than n+1 variables; this memory requirement (for typical n) is within the capability of most configurations.

2.3 Computation of queue-length distribution

We show how Q(N), for some specific N, may be computed on-line by a processor. The processor will maintain two local variables SUM and CURSOR, and after the ith cycle,

$$SUM = \int_{0}^{CURSOR} x_N(t).dt, \text{ and}$$

$$CURSOR = \max(A_i, D_{i-N+1}).$$

Initially, $A_0 := D_{-N} := D_{-N+1} = \ldots D_0 := 0$; SUM:=0; CURSOR:=0;

Thus definitions of SUM and CURSOR are trivially satisfied initially. At the beginning of the ith cycyle the processor reads the next arrival time A_i. Then it computes the following quantities.

(1) $D_i := S_i + \max(A_i, D_{i-1})$; {ith departure time}

(2) $W_i := D_i - A_i$; {ith wait time; needed to compute the histogram}

(3) if $A_i \leq D_{i-N+1}$ then SUM:=SUM + D_{i-N+1} - max(CURSOR,A_i);

(4) CURSOR:=max(A_i,D_{i-N+1});

A proof that SUM and CURSOR are computed correctly by this program, appears in Appendix A. We note that SUM holds the correct value of Q(N) up to at least time A_i at the end of the ith cycle, since CURSOR $\geq A_i$.

Note that the algorithm given above requires a history of the last N departure times to be kept - we call this a "window" of length N. After each new arrival, the window is moved once by discarding the first departure time and appending the new departure time. Clearly Q(1), Q(2)... Q(N-1) can be computed without need for any extra information if we are computing Q(N). Furthermore, this technique is independent of the particular structure of the network, e.g., tandem structure considered so far. It can be applied in the general case whenever the departure time for a new arrival can be predicted.

Typical output of a simulation program is the statistics collected at every node at prespecified times $T_1, T_2, \ldots, T_r$. These are the instants when a conventional simulation can take a snapshot and output the proper statistics. As we have shown, the proposed algorithm does not directly produce snapshots which correspond to snapshots of the real system. However this is not a major problem. Every processor could keep a table of $T_1, T_2, \ldots, T_r$. Statistics corresponding to source T_j are recorded when $A_i \leq T_j \leq D_i$. Thus the processor need only compare the next T

against the arrival and departure times to determine if the statistics should be recorded and output.

3. An Overview of TEXS

We now present an overview of how TEX Systems work in general networks. Crucial to the general solution is the notion of a "null" job: a fictitious job created by the simulator solely for its own operation. Null jobs do not appear in the real system. A special indicator needs to be used to differentiate a null job from a real job. This notion is of fundamental importance in avoiding deadlock in the general case. Creation of null jobs makes the simulator differ in a nontrivial manner from the real system.

We show, with an example below, that the general problem is not just a trivial extension of the assembly line case, considered earlier. In the case of an assembly line, every subnetwork having one input line and one output line preserves the FCFS property; any job J_1, which enters the subnetwork before job J_2 must leave the subnetwork ahead of J_2. This property is not preserved in the general case even when the network has no loops. Consider the subsystem shown in Figure 3 and where A, B and C are queues with First Come First Served service, jobs branch at X and merge at Y. Assume that we wish to carry out a distributed simulation with one processor assigned to each node. Thus the (logical) interconnection between processors is exactly as shown in Figure 3.

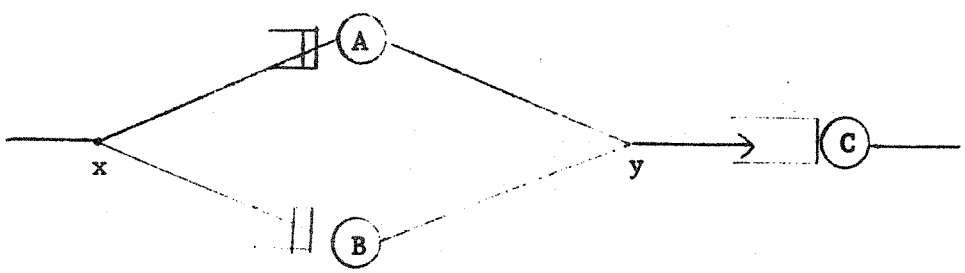


Figure 3

The difficulty is to ensure that the correct chronological sequence of jobs is input to C, while all processors maintain local information; this problem is illustrated by the following example: assume that the system is initially empty and that jobs 1, 2 and 3 enter branch point X at times t = 1, 2 and 5 respectively (figure 4). Assume that jobs 1 and 3 are directed to A while 2 is directed to B. Let the service times for these jobs 1, 2 and 3 be 3, 1 and 2 respectively. As shown in Figure 4, job 2 will arrive at C before job 1. However, if we use exactly the same method used in the assembly line example, processor C will receive messages regarding job 1 first (from A), then regarding job 2 (from B) and finally,

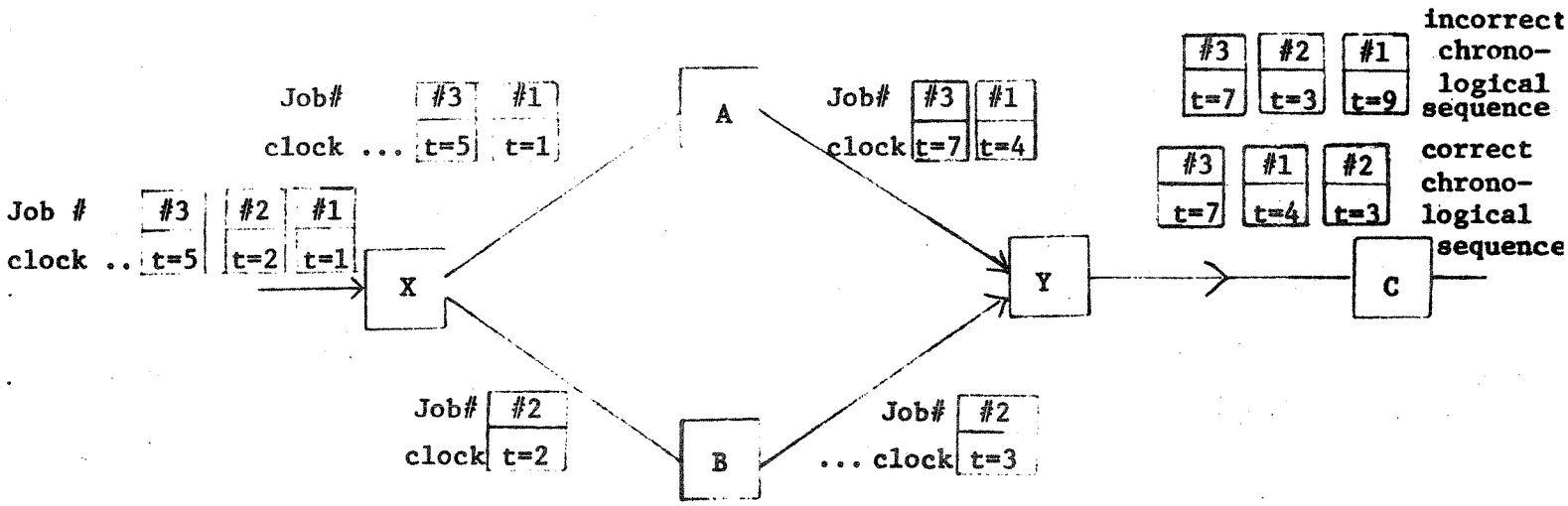


Figure 4.

regarding job 3. Thus the sequence of messages received by processor C in the simulator does not match the chronological sequence of jobs arriving at work-station C in the real system.

The solution to the problem is obvious if we insist on a separate clock for each line between processors and if we move all clocks as far ahead in time as possible, based on the local information at hand. When job 1 arrives at t=1 to node X and is directed to A, we move the clock on line (X,A) up to t=1 and send a message to processor A stating that a job will arrive at t=1; we will also move the clock along line (X,B) to t=1 stating that no job will arrive at B until t=1. Processor A moves the clock on the line (A,Y) up to t=4 because the next job arrives at that time. Processor B moves the clock along line (B,Y) up to t=2, and sends a message stating that no job can leave B before t=2 (this is because no job can enter B until t=1 and the next service time at B will be 1 unit). Now processor Y inspects the clocks on its input lines and moves the clock along its output line to the minimum of the values of the input clocks. Thus the key information exchanged between processors is the clock-time of the connecting lines and each processor attempts to move the clocks of its output lines as far forward as possible. This method ensures that messages flow in the right order. The algorithm is discussed more formally in another paper [4].

The key points of our method are the following.

(1) There is a separate clock (time) associated with each line connecting processors

(2) Processors transmit messages including time information

(3) Each process attempts to move the output clocks as far ahead in time as possible based upon currently available information

(4) The output message on a line may state that no job will arrive on that line between the current clock-time and some future time. The use of no-job messages is crucial to the correct operation of distributed simulators.

(5) Since the sequence of clock times on a line are monotone increasing, merging of two lines at a processor can be achieved on-line based on the well known merging algorithm which appears in merge-sort, for instance.

4. Earlier Work

Techniques for parallel simulations, with centralized clocks were reported by Parent, et al (2) and Kaubisch and Hoare (3). To the best of our knowledge the concept of a distributed clock simulation was discovered by the authors. The distributed clock simulation concept has been radically extended by Peacock, Wong and Manning (1).

5. Summary and Discussion

We have proposed a distributed algorithm for an important problem which is typically solved in a sequential manner. Distributed algorithms have the advantage that many processors may operate in parallel; since there are no shared variables (except message buffers between two connecting processors) protocol issues for shared data access are largely avoided. We have suggested an algorithm for the general network whose formal description and proof appears elsewhere (5).

We have assumed that each processor has a buffer size large enough to store all incoming messages prior to processing. In practice, the buffer size can be quite small if all simulator processors take approximately the same time to handle a job (i.e., compute its departure time and maintain statistics); in this case jobs arrive at approximately the same rate at which they are processed.

A formal solution to the problem with very short buffers, is desired since memory can be a scarce resource in minicomputers; this solution must formally prove that no processor will be deadlocked with buffers permanently empty or full.

Deadlock is an important issue in any distributed algorithm. Usually avoidance of deadlock is considered a global property and is solved by methods which are based upon collection of global information. Solutions to this problem based upon collection of global information and unbounded buffers appears in (1). A general solution which avoids deadlock with bounded buffers and without collecting any global information is clearly desired; this problem is currently under active investigation.

The algorithm proposed here has been implemented on 2 NOVA processors with communication capability at the University of Texas. A major problem in implementation has been the short (16 bit) word length in the computer. This makes it impossible to generate a long nonrepeating random sequence unless several words are used. We anticipate this to be the major problem in implementing simulation algorithms (conventional or the proposed scheme) on a microcomputer.

Acknowledgement

1. The initial ideas on distributed programs were developed by the authors in an attempt to parallelize inherently sequential problems in an attempt to use a large parallel pipelined computer, the Texas Instruments ASC. We are grateful to TI for giving us a grant of time on their machine in 1976.
2. The ideas presented here were discussed in a course taught at the University of Waterloo by K. M. Chandy in Summer 1977. We are grateful for this support from the University of Waterloo and to course participants for valuable discussions.
3. We are especially grateful to Professors Manning, Wong and Peacock of the University of Waterloo for pointing out that these core ideas could be significantly extended to general networks (3) by considering deadlock detection and avoidance schemes, and for their continued cooperation.
4. We are grateful to the National Science Foundation for support under grant MCS77-09812.

References

1. J. K. Peacock, J. W. Wong and E. Manning, "Distributed simulation using a network of microcomputers," Computer Networks (this issue).

2. M. Parent, F. Prunet, J. M. Dumas and Y. Moreau, "Graphical models and the LAM hardware discrete event simulator," in Computer Performance (eds. K. M. Chandy and M. Reiser), North-Holland, 1977.

3. W. H. Kaubisch and C. A. R. Hoare, "Discrete event simulation based on communicating sequential processes," to appear in CACM.

4. K. M. Chandy and J. Misra, "A nontrivial example of concurrent processing: distributed simulation," Proceedings of COMPSAC '78, available from IEEE.

5. K. M. Chandy and J. Misra, "Issues in the design of correct distributed systems," Technical Report, Computer Sciences Department, University of Texas, 1978.

Appendix A: A proof of correctness of queue length computation.

We show that the algorithm given in Sec. 2.3 is correct, i.e., at the end of the ith cycle,

$$SUM = \int_0^{CURSOR} x_N(t).dt, \text{ and}$$

$$CURSOR = \max(A_i, D_{i-N+1}).$$

We consider a slightly altered form of the algorithm that includes the parts that modify SUM and CURSOR.

Initially,

$$A_0 = D_{-N} = D_{-N+1} = \ldots \; D_0 := 0;\; SUM := 0;\; CURSOR := 0$$

Thus trivially the definitions are satisfied.

At the beginning of ith cycle:

(1) read A_i and compute D_i;

(2) if $A_i \leq D_{i-N+1}$ then $SUM := SUM + D_{i-N+1} - \max(CURSOR, A_i)$;

(3) $CURSOR := \max(A_i, D_{i-N+1})$

We will show that the steps (2), (3) properly update SUM and CURSOR to meet the given definitions. Step (3) clearly maintains the definition of CURSOR. In order to show that step (2) properly updates SUM,

(i) We will assume that prior to step (2),

$$SUM = \int_0^{CURSOR} x_N(t).dt,$$

$$CURSOR = \max(A_{i-1}, D_{i-N})$$

(ii) and show that after step (2),

$$SUM = \int_0^{\max(A_i, D_{i-N+1})} x_N(t).dt \ .$$

Proof is based on the following lemma.

Lemma: Prior to step 2, if A_i > CURSOR then

$$x_N(t) = 0, \ CURSOR < t < A_i.$$

Proof: Suppose $x_N(t) \neq 0$ for some t, CURSOR < t < A_i.

Then there is some j such that

$$A_j \leq t \leq D_{j-N+1}.$$

$$A_j \leq t, \ t < A_i \Rightarrow A_j < A_i \Rightarrow j < i.$$

Hence CURSOR < t, $t \leq D_{j-N+1} \Rightarrow$ CURSOR < $D_{j-N+1} \leq D_{i-N}$.

However CURSOR $\geq D_{i-N}$ prior to step 2. Contradiction!

In order to prove correctness of step (2), we consider the following 3 cases one of which holds prior to executuion of step (2).

Case a) $A_i > D_{i-N+1}$: Then A_i > CURSOR

Then, $x_N(t) = 0$, CURSOR < t < A_i, from the lemma.

$$\max(A_i, D_{i-N+1}) = A_i$$

New value of SUM (according to definition) $= \int_0^{A_i} x_N(t).dt = \int_0^{CURSOR} x_N(t).dt + \int_{CURSOR}^{A_i} x_N(t).dt$

$$= SUM + 0 = SUM$$

Note that the value of SUM remains unchanged in step (2) in this case, as required.

Case b) $A_i \leq D_{i-N+1}$, $A_i \leq$ CURSOR: Here CURSOR = $D_{i-N} > D_{i-N+1}$

Then, $x_N(t) = 1$, CURSOR $\leq t \leq D_{i-N+1}$ since

From the definition, $x_N(t) = 1$, $A_i \leq t \leq D_{i-N+1}$.

$\max(A_i, D_{i-N+1}) = D_{i-N+1}$

New value of SUM (according to definition) = $\int_0^{D_{i-N+1}} x_N(t).dt$

$$= \int_0^{\text{CURSOR}} x_N(t).dt + \int_{\text{CURSOR}}^{D_{i-N+1}} x_N(t).dt = \text{SUM} + (D_{i-N+1} - \text{CURSOR}),$$

provided $D_{i-N+1} \geq$ CURSOR.

Case c) $A_i \leq D_{i-N+1}$, $A_i >$ CURSOR:

Then,

$$x_N(t) = \begin{cases} 0, & \text{CURSOR} < t < A_i, \text{ from the lemma.} \\ 1, & A_i \leq t \leq D_{i-N+1}, \text{ from definition.} \end{cases}$$

$\max(A_i, D_{i-N+1}) = D_{i-N+1}$

New values of SUM (according to definition) =

$$\int_0^{D_{i-N+1}} x_N(t).dt = \int_0^{\text{CURSOR}} x_N(t).dt + \int_{\text{CURSOR}}^{A_i} x_N(t).dt + \int_{A_i}^{D_{i-N+1}} x_N(t).dt$$

$$= \text{SUM} + 0 + (D_{i-N+1} - A_i).$$

In cases b) and c) SUM must change to SUM + $D_{i-N+1} - \max(\text{CURSOR}, A_i)$, which is accomplished in step (2).