A NONTRIVIAL EXAMPLE OF CONCURRENT PROCESSING:

DISTRIBUTED SIMULATION

By

K. M. Chandy

J. Misra

September 1978 TR-82

DEPARTMENT OF COMPUTER SCIENCES

THE UNIVERSITY OF TEXAS AT AUSTIN

A NONTRIVIAL EXAMPLE OF CONCURRENT PROCESSING: DISTRIBUTED SIMULATION

K. M. Chandy
J. Misra

The University of Texas at Austin

Abstract

This paper is concerned with the pros and cons of writing distributed applications software. Most applications software is highly sequential due to the sharing of variables. Here we focus attention on one such application: discrete-event simulation. We show how to develop distributed software for this application by taking a radically different view of the application. We outline proofs that our distributed software is correct. Our goal is to develop guidelines for writing distributed applications software.

1. Writing Distributed Applications Software

The continued development of distributed systems has been due to (1) the continued decline in processing costs and (2) the increasing geographically distributed nature of inputs to information processing systems. Most of the previous work on distributed systems has been concerned with architectures and operating systems. The key issue of distributed applications software has not received sufficient attention. Our goal is to focus on distributed applications software running on well understood distributed architecture and operating systems. It is recognized that the major cost of information systems lies in applications software; we wish to focus on this fact in distributed systems as well. We shall use a message communication network as the architecture and operating system base on which we shall develop distributed applications software.

1.1 Why Choose Message Communication Networks?

Firstly, anything which can be done using a loosely coupled collection of processors which communicate only through messages can also be done using a tightly coupled set of processors with shared memory. Secondly, there are several Message Communication Networks (MCN) in existence. Hence MCNs are a practical basis to develop ideas about distributed applications.

1.2 Distributed Applications Programming

Most applications programs are written as a series of sequential programs. Relatively recently [4] a great deal of attention has been paid to concurrent programming in which several processes may be in operation simultaneously. Distributed programs are special cases of concurrent programs in which communication between processes is exclusively through messages. There are no shared variables. For a distributed program to be efficient the time spent in communication between processes should be small relative to the time spent in processing messages and data. The problem is to write applications software as a collection of loosely coupled processes.

Many applications are not suitable for distributed programming. A common approach to the creation of parallel programs is to attempt to recognize parallelism in existing programs and to then restructure the programs to make the parallelism evident. This method does not work in many cases because a variable may be shared by several processes. In this paper we consider an application (discrete-event simulation) which has always been solved in a highly sequential fashion because one variable, the "clock" or "time" variable is shared by all processes. We show how to create a truly distributed applications program despite the wide, intensive sharing of this variable. The development of this distributed program stems from viewing the application from a radically different point of view, this view-point may be useful in other applications as well.

1.3 Difficulties with Distributed Applications Programming

(1) If the program is not highly structured and many portions of the program share a large number of variables then the program is unsuitable for distribution.

(2) It is harder to prove a distributed program correct than it is to prove the correctness of a conventional sequential program. This is because to show correctness of distributed programs we have to consider issues such as deadlocks and flooding the network with messages; these issues are not considered in sequential application programs. In this paper we provide one example of proving correctness of a distributed program.

The key contributions of this paper are: (1) It lays stress on the importance of distributed applications software (not merely architecture and operating systems) (2) we consider an application, viz. discrete-event simulation, which has been solved in a highly sequential fashion and show how to create a distributed program by taking a radical view of this application (3) we show how to ensure the correct behavior of this distributed system without any form of centralized control (4) we formally prove the correct behavior of this distributed system elsewhere [2].

Our overall goal is to develop a methodology for the design of distributed applications software. The example provided here should give the reader some idea about the advantages and the difficulties in writing distributed software.

2. An Example of Distributed Programming: Simulation of Networks

We consider a network of work stations each of which could process jobs. Jobs enter the system from a source at arbitrary time intervals, get processed by some work station and then move to some neighboring work station for more processing. Due to the presence of loops in the network, a particular job may get processed more than once by a work station. Eventually, a job leaves the system by reaching the sink. The service time associated with processing a job at a work station may depend on the job and the work station. A queue may build up at a work station if more jobs arrive before one job is completely processed. Jobs are processed First Come First Serve (FCFS). Since one work station may receive inputs from two or more work stations, jobs in the queue may come from different work stations and get ordered by their arrival times. An example of such a network is shown in Fig. 1, where A,B,C,D are work stations and a directed line between two work stations indicates that a job could (possibly) go from one to the other in the direction of the line.

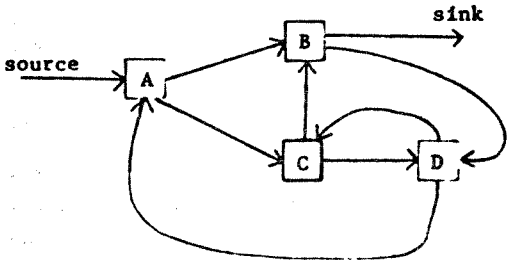
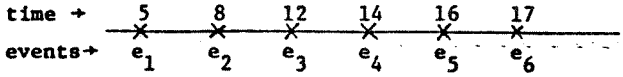


Figure 1

Assume that a job J_1 arrives at time 5 and requires 7 units of processing by A. It will then depart A at time 12 and suppose it goes to C. Suppose J_2 arrives at A at time 8 then it must wait till time 12 to get processed. Assume that J_2 takes 2 units of processing at A and then goes to B. Assume J_1 takes 5 units of processing at C and J_2 takes 2 units of processing at B. Both go to D. A picture of the functioning of the system is shown below.



$e_1 \equiv J_1$ arrives at A

$e_2 \equiv J_2$ arrives at A

$e_3 \equiv J_1$ departs A to go to C

$e_4 \equiv J_2$ departs A to go to B

$e_5 \equiv J_2$ departs B to go to D

$e_6 \equiv J_1$ departs C to go to D

Note that D will process J_2 first and then J_1.

The events happen at discrete instants in time. We assume that no time is taken for a job to travel from one work station to the next.

A conventional simulator maintains an event-list, which is a list of expected future events ordered in an increasing sequence of expected time of occurrence. The simulator removes the first event from the event list and moves the simulation clock to the time associated with that event. Now the occurrence of this event may create new events with associated times, each of which is entered into the event list. It is possible that occurrence of an event e_1 may cause event e_2 to happen next; hence the simulator can not remove more than one event from the event-list at any time. The strict serial nature of this algorithm was a challenge to deriving a distributed algorithm where processes could operate in parallel.

Clearly, the work stations in the real system are operating in parallel; for instance B,C process J_2, J_1 in parallel during times 14 through 16. However, the synchronization is achieved in real time. Any effort to duplicate this parallelism in the simulator would result in a shared variable "clock" or a central process which schedules events in the proper order - another implementation of the event list. Our solution dispenses with a central clock by maintaining clocks at each individual process, where a process corresponds to a work station of the real system.

2.1 A Special Case; Assembly Line

It is easier to explain the algorithm when the network structure is a tandem. We thus have work stations 1,2,...,n, where the ith work station receives input jobs from (i-1)th work station and outputs jobs to (i+1)th work station. Source is the 0th work station and sink the (n+1)th work station. In the simulator, we associate one process with each work station; message transmission in the simulator is along the same direction as the job flow in the real system. As in conventional simulation, we assume that jobs are produced by the source by a random number generator and service times at work stations are arbitrary random variables.

The messages transmitted in the simulator are of the form <t,j> where j is a job name and t is a time. If ith process transmits <t,j> to (i+1)th process at any step during simulation, it means that, the job j departs the ith work station (arriving at the (i+1)th work station) in the real system, at time t. Thus every job bears a time stamp which identifies the time occurrence of an event involving that job in the real system.

Given this interpretation of the messages, it is quite straightforward to specify the logic of each process.

```
p_i :(ith process)
  Var D; {time of departure of the last job from
          this work station = time stamp of the
          last message output by p_i}
      A; {arrival time for the next job}
      j; {the identity of next job}
  Initially D:=0;
  loop
     input <A,j> ;  {from p_(i-1)}
     D:=max(A,D) + next(service time){next returns
                                      the next ser-
                                      vice time}
```

output <D,j> {to p_{i+1}}

collect statistics {for histogram and other output from the simulator}

forever

Correctness of the logic of p_i can be seen as follows.

The jobs go from source to sink visiting each work station in order. Thus we note the following.

(1) Each process p_i receives input <A,j> at the jth cycle, where A is the arrival time of the jth job at the ith work station in the real system.
(2) D, at the beginning of the jth cycle, is the departure time of (j-1)th job from work station i in the real system.
(3) Hence the jth job can not be processed by the ith work station until max(A,D). Processing will start at this time.
(4) Thus departure time of the jth job from the ith work station is max(A,D) + next (service-time).

We have thus effectively associated a clock D, with every process which simulates the functioning of that particular work station up to D. Since different processes may have different D's, processes are not synchronized in terms of the real time. In fact, if we were to stop the simulator at some step and examine the snapshot of the system (last arrival and departure times at each process), we will most likely find that the snapshot corresponds to no physical snapshot of the real system - some processes may have simulated further and gone ahead of other processes; i.e., their clocks may have advanced further. The only synchronization is between neighboring processes where a message sent from one process to the next, might advance the clock of the latter process.

We summarize the important aspects of the proposed algorithm.

1. There is no shared variable.
2. Messages are transmitted between neighboring processes and that constitutes the only shared information.
3. Synchronization is only in a logical sense, unrelated to the physical synchronization through time in the real system.
4. Logic of each individual process is simple. Memory requirement of individual processes is minimal. Only the necessary data is transmitted in each message, prefixed with a time stamp.
5. Structure of the simulator is identical to the structure of the real system.
6. Correctness of the simulator can be shown by proving that each process produces a correct message (i.e., if it produces <t,j>, j will actually depart at time t in the real system) assuming that it receives a correct message. Thus correctness of each individual process is sufficient to guarantee correctness of the entire system, provided there is no deadlock in the simulator.
7. The simulator is deadlock-free as long as the source keeps on producing jobs and the sink keeps accepting them. This fact is obvious from the tandem structure of the network. In general, absence of deadlock proofs may have to rely on structure of the network as well as on the internal logic of individual processes.
8. Since snapshot in the simulator does not usually correspond to a snapshot in the real system, it is not entirely straightforward to compute certain statistics such as queue length distribution. These statistics however can be computed on-line with a bounded amount of storage, see [1] for details.

2.2 Difficulty of Algorithm Design for Arbitrary Networks

It may seem that the algorithm given in the previous section is also applicable to arbitrary networks. Unfortunately this is not so. To see why, consider a process that has two input lines. Clearly, the process must receive input on both lines before producing any output; otherwise if it receives an input $<t_1,J_1>$ on one input line and the other line is empty, it can not assert that the next job to be processed by this process would be J_1, since $<t_2,J_2>$ may appear on the other input line at a later simulation step where $t_2 < t_1$. In the real system of course, this problem does not arise since J_2 would arrive (physically) prior to arrival of J_1.

Now consider a loop which has a process P having two input lines L_1 and L_2; line L_1 is in the loop (see Figure 2). In all reasonable real systems every loop must have such a process, otherwise jobs can not enter the loop. We claim that P can not proceed with simulation since it requires input from both L_1 and L_2 and since it can not produce any output, line L_1 will never have any message transmitted over it. In fact all the processes in the loop would be individually deadlocked. A possible solution is to somehow identify that L_1 is in a loop and hence P should proceed without waiting for input from L_1. This is correct only if it can be guaranteed that no message can appear on L_1 unless P produces some output. A solution given in [3] sends around a message to detect if such a deadlock has occurred. This may slow down simulation. We propose an alternate solution which solves this problem.

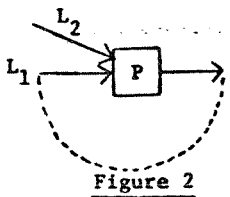


Figure 2

It may seem that the algorithm given for the tandem case would be applicable to acyclic networks. Again this is not so, unless we permit every process to have a buffer of unbounded length. Consider the network given in Figure 3.

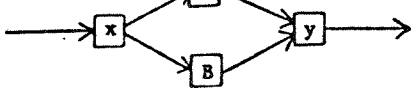


Figure 3

Suppose x sends job J_1 at t=5 to A. J_1 takes 10 units of processing at A.
x sends job J_2 at t=6 to B. J_2 takes 1 unit of processing at B.
x sends job J_3 at t=7 to A.
x attempts to send J_4 at t=8 to A.

Note that y would read input from both lines $<15,J_1>$ on (A,Y) and $<7,J_2>$ on (B,Y). It will process J_2. Next it would attempt reading from (B,Y) again. However x is blocked attempting to output to A, which is blocked since its input buffer has one item (J_3) and its output buffer has one item (J_4). Hence all nodes would halt.

A solution is to allow processes to have unbounded buffers. Thus A could keep reading inputs as long as input is being sent to it from x. Such a solution appears in [3]. Our algorithm solves the problem with a bounded buffer.

2.3 The General Algorithm: Time EXchange System (TEXS)

A concept that is crucial to the solution of the general problem is the notion of a "null" job, to be denoted by $\emptyset$. A message $<t,\emptyset>$ on a line indicates that no job will appear until time t on this line. A message $<t,J>$, $J\neq\emptyset$, indicates that job J appears on the line at time t and no other job will appear until time t on this line. Such an assertion can be made in our algorithm since the times produced by each process are monotone nondecreasing; a fact that we prove in the next section.

Let P be a process that has more than one output line. Whenever P produces $<t,J>$, $J\neq\emptyset$, on some output line, it produces $<t,\emptyset>$ on every other output line. This is valid since the next output of P to any other process must be at a time at least as high as t.

Processing of a null job should take 0 time; however this would fail to avoid the deadlock in a loop, as explained in the previous section. If $<t,\emptyset>$ is the next message to be processed at process P and service time for the next (real) job at P is s, then P can certainly output $<t+s,\emptyset>$ on every output line; this is easily seen to be valid since the next real job can not arrive before t and hence can not depart before t+s.

A second key idea is to suppress the identity of jobs and only count the number of jobs which arrive at time t. Thus each message would be of the form $<t,n>$, $n \geq 0$ which denotes that,

(i) exactly n real jobs arrive at time t along the given line at this process (n=0 denotes that no real job arrives at time t) and,
(ii) next job, real or null, arrives at time strictly greater than t.

It is possible to keep only a count since identity of jobs does not affect the simulation procedure. A process accepts inputs from all input lines and processes those having the least time stamp. Thus $<t,n_1>$,$<t,n_2>$ on two different input lines would be logically equivalent to $<t,n_1+n_2>$, i.e., n_1+n_2 real jobs arrive at this process at time t. We will show in the next section that this scheme permits a process to produce jobs in a strict monotone time sequence: a fact that is important for showing absence of deadlock.

We now show how an unbounded buffer can be replaced by a bounded (one-word) buffer for the solution of the second problem discussed in the previous section. Suppose a process P receives the message $<t_1,n_1>$ and determines the next departure time to be T. Now any sequence of input messages $<t_2,n_2>,<t_3,n_3>\ldots<t_r,n_r>$, can be replaced by $<T, \sum_{i=2}^{r} n_i>$ provided $t_r < T$, for the purposes of simulation. Clearly none of the jobs can be processed until time T and hence their individual arrival times are of no consequence† as long as they are before T. Thus a process may accept inputs (in the proper order) as long as the input times are less than the next departure time and simply keep a count of the number of jobs so received. A formal description of the algorithm and its proof appears in [2].

3. System Properties

In this section, we show that any arbitrary network of processes as defined in the previous section, is deadlock-free, provided every loop has a process which has nonzero service-time. For correctness of individual processes and the general problem of designing correct distributed systems, reader is referred to [1].

It is important to distinguish between system deadlock and process deadlock. System is deadlocked when no process can proceed, i.e., every process is waiting for either input or output such that no pair of processes are waiting for input-output from each other. A process is deadlocked if it is waiting and would never get out of the waiting state. It is possible that a process may be deadlocked, but the system is not deadlocked. It is likely that in our case, the system may not be deadlocked, but it may not be computing anything useful (such as a null job going around a loop). Hence we need to establish that no process will deadlock; then the source can produce an arbitrarily large number of jobs.

We first note an important property of our system: the sequence of messages $<t_1,n_1>$, $<t_2,n_2>$ $\ldots<t_r,n_r>$ output along a line have strictly increasing time stamps, i.e., $t_i < t_{i+1}$. This can be shown by induction on the total number of messages output in the system; note that if any process receives input having the strict monotonicity property, it produces outputs having strict monotonicity property.

Define a process to be starved, if it is waiting to input and blocked, if it is waiting to output. Observe that no process is simultaneously blocked and starved. At any point during simulation, we associate a time t with every line in

† The only statistics that depends on individual arrival times is the queue length distribution. As shown in [1], it can be computed on-line with a bounded amount of storage.

the network: t is the time stamp of the last message sent along the line (sent from the process at the tail of the line and received by the process at the head of the line).

We make the following observations without proving them here.

1. If a process P is blocked waiting to output on line out then for any input line in

$$t_{in} \geq t_{out}.$$

Every loop has a process for which $t_{in} > t_{out}$.

2. If a process P is starved waiting to input along line in, then for any output line out,

$$t_{in} \leq t_{out}.$$

Every loop has a process for which $t_{in} < t_{out}$.

3. If a process u is blocked waiting to output to process v and v is starved then,

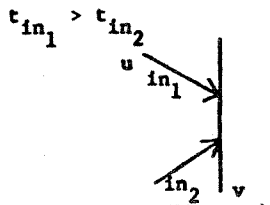


Figure 4

4. If a process u is starved waiting to input from process v and v is blocked then,

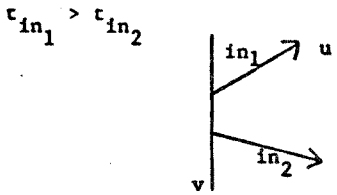


Figure 5

Theorem: System is never deadlocked.

Proof: Intuitively, if there is deadlock then there is a loop of blocked or starved processes or a (pseudo) loop of starved and blocked processes. There can not be a loop of blocked/starved processes each of which is waiting on the other, since the times on lines must be monotone nonincreasing/nondecreasing and there must be a discontinuity in time across some process.

The only possibility then is a pseudo loop of the following type.

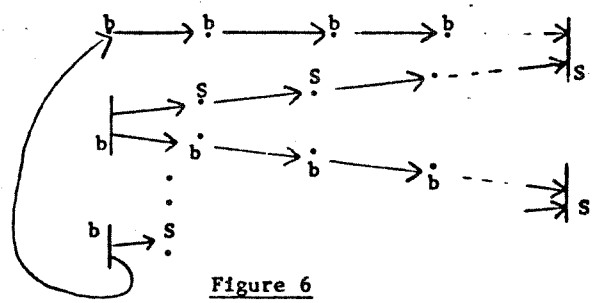


Figure 6

where b denotes a blocked and S denotes a starved process. Again such a pseudo loop is impossible since times are monotone nonincreasing across blocked processes and monotone nondecreasing across starved processes and there is discontinuity at the boundary processes.

Corollary: Assuming that every service-time > ϵ, where ϵ is a positive real number, no process will deadlock.

4. Discussion

We have shown that any distributed program synthesized according to a certain discipline is deadlock-free. Furthermore, the memory requirements for implementing the logic of processes is minimal. Every process behaves identically independent of the structure of the network. Synchronization among neighbors is achieved through time stamps affixed to a message.

Though the discussion has been couched in terms of simulation, it is not hard to see that the same technique is applicable to any system where every process can determine its (local) time. It may then output messages bearing the proper time stamp. Buffer length, in theory, could depend on the times t_{in}, t_{out}. However, we have found that in almost all cases, a bounded buffer suffices.

We are currently investigating the use of message communication networks in the solutions of other application problems.

5. Acknowledgement

We are grateful to Professors Manning, Wong and Peacock of the University of Waterloo for helpful discussions. This work has been extended in an orthogonal direction by Mr. Vic Holmes [5]. This work was supported by NSF Grant MCS77-09812. We are grateful to Ms. Debbie Davis for typing and editorial comments.

References

1. K. M. Chandy, Victor Holmes and J. Misra, "Distributed Simulation of Networks," Computer Networks, December 1978.

2. K.M. Chandy and J. Misra, "Issues in the Design of Correct Distributed Systems," Technical Report, Computer Sciences Department, University of Texas, 1978.

3. J.K. Peacock, J. W. Wong and E. Manning, "Distributed Simulation Using a Network of Microcomputers," Computer Networks, December 1978.

4. W.H. Kaubisch and C.A.R. Hoare, "Discrete Event Simulation Based on Communicating Sequential Processes," Comm of ACM (to appear).

5. Victor Holmes, Ph.D. Dissertation, University of Texas, in Preparation.