DISTRIBUTED COMPUTER SIMULATION OF DATA COMMUNICATION NETWORKS: PROJECT REPORT III

Shun Cheung
Jack W. Carlyle
Walter J. Karplus

September
CSD-850037

Abstract

This is the third in a series of reports on continuing studies of discrete-event simulation using distributed processing (e.g., on networks of minicomputers or networks of microcomputer workstations), with application to performance prediction for data communication networks. Issues examined include synchronization mechanisms, task allocation algorithms, simulator implementation approaches, and deadlock prevention methods.

Table of Contents

page

1. Introduction 1
 1.1 Background 1
 1.2 Recent Progress in the Project 2
2. Alternatives in the Roll Back Mechanism 3
 2.1 Side Effects of Selected Saved States at Check Points 3
 2.2 Delayed Cancellation (Lazy Cancellation) 5
3. Model for Distributed Simulation 6
 3.1 The Model of a Node 6
 3.2 Deadlock Due to Incorrect Modeling 7
4. Model Partitioning, Load Balancing, and Object Allocation 9
 4.1 Process: The Basic Unit which Forms a Simulation 9
 4.2 The Load Balancing and Object Allocation Problem 11
  4.2.1 Main Sources of Simulation Overhead 11
  4.2.2 Guidelines for Object Allocation 12
  4.2.3 A Heuristic Allocation Algorithm 14
5. Implementation of Distributed Simulator 15
 5.1 Next Event Time (NET) 16
 5.2 Queues 18
  5.2.1 Input/Output (I/O) Queues 19
  5.2.2 Input Queues 20
  5.2.3 Output Queues 21
  5.2.4 Arrival Queues 22
 5.3 Message Format 24
6. Other Deadlocks and Flow Control 24
 6.1 Pipe Read Deadlock 24
 6.2 Pipe Write Deadlock 25
 6.3 Buffer Usage Deadlock 26
7. The Current Status of the Project and Upcoming Work 27

References 29

1. Introduction

1.1 Background

This is the third in a series of documents reporting on work in progress in the project ''Distributed Computer Simulation of Data Communication Networks'' (DCSDCN). This is a continuing project, which was initiated with support from the State of California and Doelz Networks, Inc., under the University of California/Microelectronics Innovation and Computer Research Opportunities (UC-MICRO) program.

The DCSDCN project is broadly directed toward understanding and implementing discrete-event systems simulation via distributed processing in networks of computers, with specific application to the simulation of data communication networks, using as a test case the Doelz network architecture [Doel84] (an extended slotted ring for communication). A long-range objective of the DCSDCN project is to develop simulation tools, executable via distributed processing on microcomputer workstations, usable in field-engineering situations where performance of proposed data communication network installations must be estimated.

It is a well-known problem that discrete event simulation is very computer-time consuming in general. As networks of microcomputer workstations are rapidly emerging as a medium-scale computing resource, it is now possible to consider more detailed simulations using these distributed environments. A number of different approaches to distributed discrete event simulation have been discussed and compared in our previous report [Cheu85]. The *model partitioning* method can exploit the inherent concurrency in the model, and we believe that it is an appropriate approach to realize distributed simulation of ring type communication networks. In our implementation, the *roll back*

mechanism in the *Time Warp* method invented by Jefferson and Sowizral [Jeff83, Jeff85a, Jeff85b] is used to maintain the correctness of the event sequence.

The simulator is developed on the LOCUS system at UCLA. LOCUS is a network-transparent version of UNIX†. When a simulation run begins, a number of pipes are set up for inter-process communications. The processes are then forked and migrated to different sites (computers) on the network. Hence the simulation is executed in a genuine distributed environment.

1.2 Recent Progress in the Project

A heuristic algorithm for model partitioning and load balancing has been developed. An initial version of the user interface for simulation model preparation is also completed. A significant portion of the simulator has already been built and tested on LOCUS; the basic feasibility of distributed simulation on a network-transparent operating system is therefore verified. In particular, the development tasks completed to date include: the command generation, simulation forward, and roll back mechanism in terminal nodes. We are planning to finish the implementation of host nodes, and the response message handling mechanism in terminal nodes, in the near future.

In this report, we emphasize the completed portions of the distributed simulator. In Section 2, we compare alternatives in the roll back mechanism. Section 3 discusses the differences between a regular model and a model for distributed simulation. The model partitioning, load balancing, and object allocation problem and a proposed solution are discussed in Section 4. In Section 5, the implementation is described. One of the major difficulties in distributed systems in general is the deadlock problem, usually due to the lack of centralized control. Several sources of deadlock in distributed

† UNIX is a trade mark of Bell Laboratories.

simulation have been identified, and suggested solutions are discussed in Sections 6. Finally, in Section 7, we outline our future plans for this project.

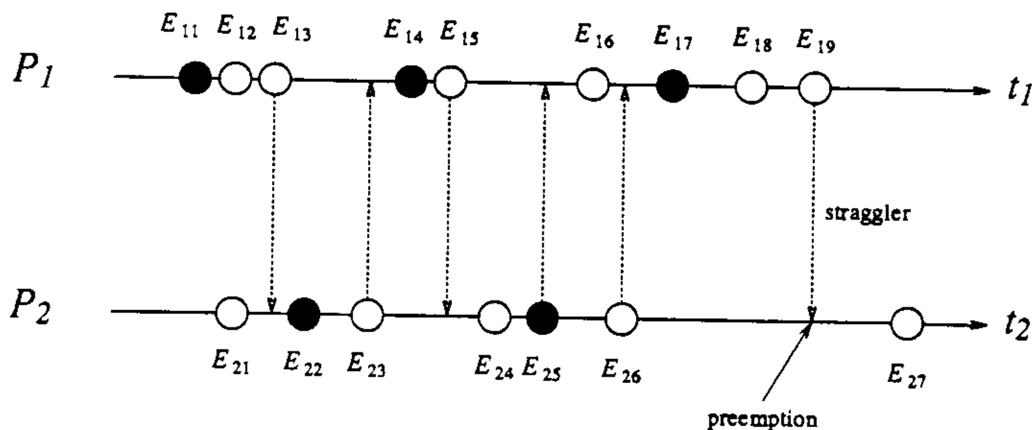
2. Alternatives in the Roll Back Mechanism

2.1 Side Effects of Selected Saved States at Check Points

Simulation is guaranteed to progress in Time Warp because, according to the principle of causality, an event with a later simulation time cannot affect another one with an earlier simulation time. If we take a snapshot of the processes at any point in real time, the event which has the earliest simulation time in that snapshot can never be preempted at any point in the (real time) future. Therefore, the simulation progresses after this event has been simulated [Jeff83, Jeff85a]. Subsequently, the system will enter a new snapshot, and another event will have the earliest simulation time.

One of the main problems with Time Warp is its potential for large memory usage. A significant amount of memory is needed to save the previous states. As a result, in actual implementations of Time Warp, the states of a process are usually saved only ''every so often'' at certain *check points*, not after the completion of each event. The tradeoff is that when a preemption occurs, on the average, the preempted process will need to roll back further into the past to reach a saved state. The ''optimal'' spacing of check points depends on a number of factors such as the amount of memory available, the characteristics of the model, etc. There does not seem to be any straight-forward algorithm to determine how often states should be saved. Unfortunately, there is a very undesirable side effect when only selected states are saved: since an event can now cause a roll back to a state prior to that event's simulation time and generate anti-messages, a preemption could initiate a chain of roll backs. This may be considered as a minor violation of causality. An example of this problem is shown in Figure 1, where every

third state is saved. P_1 and P_2 are two processes; axes t_1 and t_2 are their simulation times.



A circle indicates an event.
A darkened circle indicates an event whose state is saved.
The dashed arrows represent inter-process messages.
The axes are the simulation time of the processes.

Figure 1: Chain Roll Backs due to Selected Saved States

Assume that in real time, event E_{27} is simulated on P_2 before message E_{19} from P_1 arrives. (The relation between real and simulation time among several processes can only be shown in a three-dimensional graph.) Since E_{19} has an earlier simulation time, it will preempt P_2 and cause a roll back to E_{25}, the latest saved state before the preemption time. During this roll back, an anti-message is sent for E_{26}. This anti-message causes P_1 to roll back to E_{14} and send an anti-message to cancel E_{15}, which causes P_2 to roll back to E_{22} and so on. It is somewhat surprising that not only P_2 has to roll back past E_{25}, but also P_1, which initiated the preemption, has to subsequently roll back too. These unnecessary roll backs are clearly a waste of computation time. This problem is worsened if this chain of roll backs reaches a point which is earlier than the current

global virtual time (GVT). † (In Figure 1, for example, the GVT of the two-process system at the time of the preemption is the simulation time of E_{19}.) Since saved states earlier than the current GVT are usually expunged as soon as possible to free up memory spaces for future state saves, the old states which the simulation attempts to reach might no longer be available, and anti-messages might have been sent to cancel old positive messages which do not exist any more. These events, of course, are considered errors and will cause the simulation to be aborted. Another possibility is that if the preemption takes place early in the simulation, before and old saved state is expunged, the chain roll back could reach as far back as the initial state. In this case, when the simulation finally continues to simulate forward, it will repeat the previous forward steps, going though the same events and rolling back again to the initial state without any real progress. This is considered as a special type of deadlock. The problems described above would not occur if every state was saved. Consider the example in Figure 1 again, the straggler sent by E_{19} will cause P_2 to return to E_{26}. No subsequent anti-message will be sent to P_1. (The positive message E_{26} is sent before the corresponding state is saved. Therefore restoring the saved state does not cause an anti-message to be sent for E_{26}.) P_2 will resume its forward simulation from E_{26}, and P_1 will simply continue from E_{19} after sending the straggler message.

2.2 Delayed Cancellation (Lazy Cancellation)

The concept of *delayed cancellation* (lazy cancellation) was introduced by Jefferson *et.al.* [Jeff85c]. When a roll back occurs, it is not necessary to immediately cancel every positive message which was sent after the restored state had previously been reached. Instead, the process should simulate forward again. When a new outgoing

† A good description of the concept of global virtual time can be found in [Jeff85b]. We have also provided a brief summary on it in a previous report [Cheu85].

message is generated, the process will then need to check whether an identical message has been sent before or not. If so, the new message should be discarded. Anti-messages should be generated only for those messages sent before the roll back but would not have been sent under the new condition. New positive messages should of course be sent too. For example, in Figure 1, with delayed cancellation, when the preemption occurs, P_2 will roll back to E_{25}, but no anti-message will be sent for E_{26} at this point. P_2 will simulate forward again from E_{25}. Since everything remains the same, a new E_{26}, which is identical to the previous one, will be generated and then discarded. No anti-message will be sent for E_{26} at all; hence no unnecessary roll backs will take place. This will not only save computation time by reducing unnecessary roll backs but also avoid the deadlock problems described in the previous section. However, in the cases where anti-messages should be sent, the receiving processes will find out the "bad news" at a later time due to the postponement in delayed cancellation. This is a small price to pay, considering the advantages of delayed cancellation.

3. Model for Distributed Simulation

3.1 The Model of a Node

The model of a node contains an input queue, an arrival queue, a response queue, and a *node time*. Except for the last node in a process where a special output queue for delayed cancellation is created, the output queue of a node is the same as the input queue of the following node on the ring. Therefore, the connection from one node to another in a process is formed by these input/output queues. Moreover, an output from a node automatically enters the input queue of the following node.

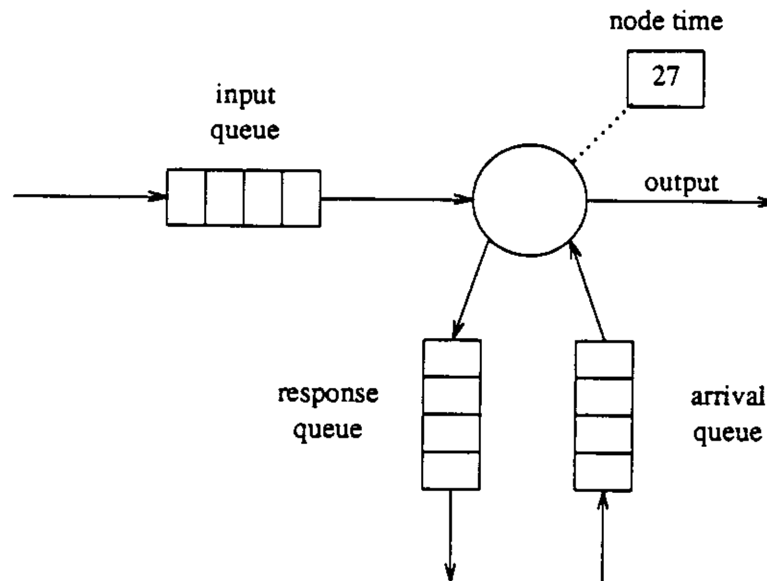


Figure 2: The Model of a Node in the Network

The terminal arrivals to a node are merged together and stored in the arrival queue (in a non-decreasing time stamp order). Since several terminals may receive inputs at the same time, commands in the arrival queue may have time stamps with the same value. The node time indicates the time stamp of the last message leaving that node. No message leaving the node from the output queue should have a time stamp less than or equal to the value of node time.

3.2 Deadlock Due to Incorrect Modeling

Figure 3 shows a simple ring network with four nodes which send messages to one another. Nodes 1 and 2 are assigned to computer I and nodes 3 and 4 are assigned to computer II. Assume that there is only one priority level for the messages, and a message already on the ring has privilege over a newly arrived message on a siding queue (not shown) to occupy an available slot. Consider the following situation: when

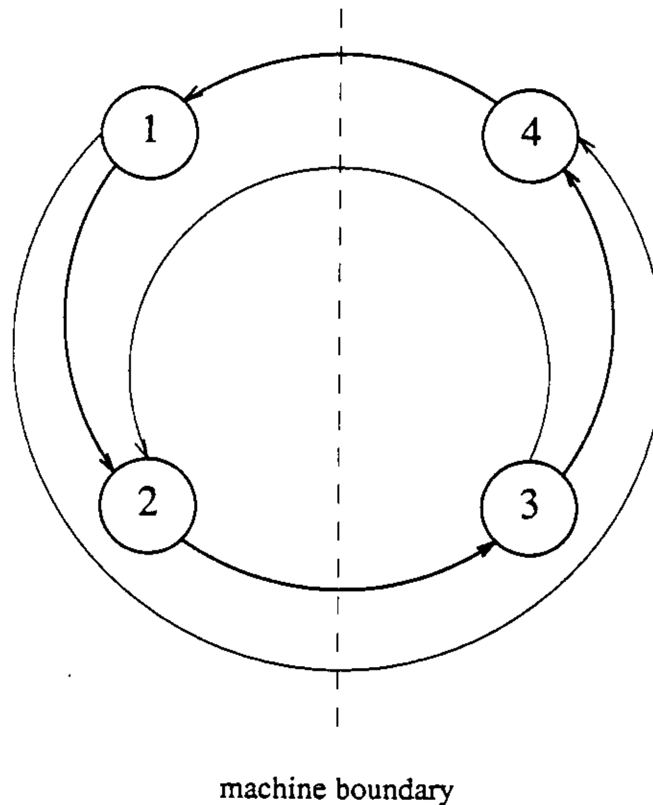


Figure 3: Deadlock Due to Interfereing Roll Backs

the simulation begins at time = 1, message A arrives node 1 and is heading for node 4 through nodes 2 and 3; at the same time, message B originates from node 3 and its destination is node 2. When A arrives (from a terminal, for example), there is no other message competing for the time slot at 1 so that it gets on the ring immediately. Similarly, B gets on the ring without any delay and occupies the time slot at 1. When message A arrives node 3, since it is a message already on the ring and occupies time slot 1, an anti-message will be sent to cancel message B. A will continue occupying time slot 1 and goes on to node 4. In the mean time, message B will preempt message A at node 1 for the same reason, and an anti-message will be sent for message A. After both cancellations have taken place, the simulation will be in its initial state again and this

cycle repeats. This deadlock problem is a consequence of incorrect modeling. In the real network, there is a ''conveyor belt'' of fixed time slots. We may imagine this as one time slot (frame) which goes around the ring. One possibility is that this time slot originates from one node and cycles around. When it returns to the starting node, a new slot will be put on the ring. This system is less fair because there are nodes which have better chances to obtain resource, i.e., empty slots, than others. This reveals the solution to our deadlock problem: the distributed simulation model should not allocate slots by considering which message is already on the ring during the simulation. It should instead compare the originating node number of the messages. When arriving at the same time, a message from a node where a time slot passes through earlier will have priority to occupy that slot. For example, in Figure 3, if we assume that a time slot originates from node 1 and goes around the ring through nodes 2, 3, and 4, message A, which originates from node 1 would be able to preempt message B, but not vice versa. Hence the deadlock condition described above would not occur.

4. Model Partitioning, Load Balancing, and Object Allocation

4.1 Process: The Basic Unit which Forms a Simulation

A convenient way to view a simulation model is to consider it as a group of processes, or objects. Each one of these processes corresponds to one or several physical objects in the model. These processes interact with one another through messages. Each process has an input queue and an output queue for the incoming and outgong messages respectively. A process also receives arriving messages from the outside world; within the context of ring network simulation, these messages are commands from terminals or responses from computers connected.

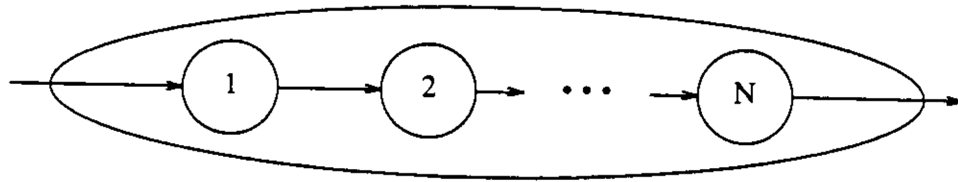


Figure 4: A Process which Contains a Number of Tandem Nodes

It is possible to have only one network node simulated by each processor (physical CPU). (In fact, more than one processor may be used to simulate one node, although a break down to such an extreme will normally slow down rather than speed up the simulation due to a large number of inter-process messages and synchronization overheads.) However, since the number of nodes in a realistic network is often much larger than the number of processors in a distributed simulator, it is typically necessary to assign several nodes to each processor. Each network node may be considered as an independent unit and assigned to a process; a number of processes will therefore be allocated to a computer (or processor). In a regular Time Warp implementation, processes on the same computer interact with one another by sending messages, just as do processes assigned to different sites. However, when the model is a ring network and adjacent nodes in the ring are assigned to a common machine, it is possible to consider all of these nodes as one process. (Object allocation will be discussed in the following Section.) The main advantage of this grouping is that nodes assigned to the same computer will communicate in a much simpler manner; hence the overhead involved is significantly reduced. The tradeoff is that several nodes are now grouped into one unit so that a preemption to the first node in a series will cause every node to roll back together. (In fact, the first node is the only one which may be preempted. This problem is discussed in Section 5.1.) We do not consider this as a serious disadvantage because when several processes are connected in tandem, if the first one is preempted, it is very

likely that the remaining ones will subsequently be preempted anyway.

4.2 The Load Balancing and Object Allocation Problem

In multi-processor applications, task (object) allocation is a common fundamental problem. The main objective is to allocate the tasks to the processors so as to fully utilize available resources and speed up the computation. Unfortunately, it has been shown that the load balancing problem is NP-complete in terms of complexity theory; i.e., the optimal allocation can only be determined after every possible alternative has been checked and compared. Although there are schemes such as the branch and bound method which can eliminate the cases which are ''clearly'' not leading to the optimal allocation, the time it requires to consider the remaining possibilities even for a medium-size simulation problem could still be much longer than the time to perform the actual simulation. Therefore, efficient sub-optimal allocation methods are often desired.

4.2.1 Main Sources of Simulation Overhead

In distributed simulation systems using Time Warp for synchronization, the major sources of overhead are roll backs and inter-processor communications. A roll back occurs when a processor with small simulation time † sends a message to another one with larger simulation time. The farther apart the two simulation times are, the more the leading processor needs to roll back. Therefore, it is desirable to minimize the differences of the simulation times among the processors.

Assume the real time needed to process each message is the same. Let λ_i be the message arrival rate to object i. $\frac{1}{\lambda_i}$ is therefore the mean inter-arrival time; i.e., on the

† In Time Warp's terminology, simulation time is usually referred to as *Local Virtual Time* (LVT).

average, the simulation time of object i is advanced by $\frac{1}{\lambda_i}$ sec. for every incoming message processed. This can also be regarded as the rate at which simulation time grows on object i. If several objects are assigned to processor p, the rate at which simulation time grows on p is simply $\frac{1}{\sum_{i \in p} \lambda_i}$. If the number of processors available is M, the ideal simulation time growth rate is:

$$\text{ideal growth rate} = \frac{M}{\lambda}, \text{ where } \lambda = \sum_{all\ i} \lambda_i$$

Therefore, it is desirable to assign the objects to the processors in a way such that each will have (almost) the same simulation time growth rate.

The need to minimize inter-processor communication in additional to load balancing makes the allocation problem more complex. Several heuristic job allocation methods for general distributed simulation using Time Warp for synchronization have been suggested by Samadi [Sama85]. In a single-loop ring network model, there is inherently a lot of communications, but they are possible only among neighboring nodes. It is therefore reasonable to group adjacent nodes and assign them to the same processor in order to minimize the inter-processor communications. This goal is best achieved, of course, in a single-processor environment. An effective sub-optimal allocation scheme should therefore find a good compromise between concurrency gained and communication overhead in a distributed environment.

4.2.2 Guidelines for Object Allocation

We have made the following assumptions to simplify the allocation problem:

1. The number of objects, N, is much greater than the number of processors, M; i.e.,

$N \gg M$.

2. There is a single loop in the network and all of the hosts are connected to the same network node. This assumption will be relaxed so that ring networks with more complex topologies can be considered.

3. No object has a very large load which dominates the computation time.

Since the host node has to process each request and then generate the corresponding responses, it needs to process much more events than the terminal nodes. Therefore, one (or more) processor(s) should be dedicated to simulate the events in the host node and the computers connected. Each terminal node, however, processes relatively few events; usually, several neighboring terminal nodes are assigned to one processor to balance the load.

(a) Partitioning N nodes into M processors

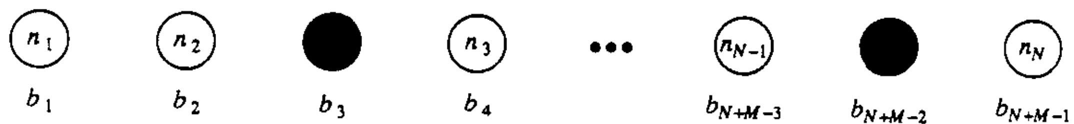


Figure 5: (b) Selecting M-1 balls from a set of N+M-1

Assume M processors are left after dedicated processors have been assigned to simulate the host node and there are N terminal nodes. With the host node removed, the ring is broken and the remaining nodes are connected as tandem nodes in a line. These N nodes should be assigned to the M processors such that all nodes assigned to a processor are adjacent to at least one other node assigned to that processor. This situation is shown in Figure 5a. The nodes are numbered n_1 through n_N. M - 1 partitions are needed to separate these N nodes into M groups. This problem is equivalent to selecting M - 1 balls from a set of N + M - 1 as shown in Figure 5b. The balls selected are colored black. The number of ways to select M - 1 balls from a set of N + M - 1 is:

$$\binom{N+M-1}{M-1} = \frac{(N+M-1)!}{N!\,(M-1)!}$$

$$= \frac{(N+M-1)\,(N+M-2)\,\cdots\,(N+1)}{(M-1)!} \approx O\,(N^{M-1})$$

If N=64 and M=8, there are approximately 1.33×10^9 ways to partition the nodes. Assume that a computer can perform 1000 partitionings and comparisons in a second; it requires over two weeks to find the optimal allocation. If there are 16 processors instead of eight, it requires over 100 centuries! Clearly, some more-efficient allocation methods are needed.

4.2.3 A Heuristic Allocation Algorithm

Since the growth rate of simulation time involves reciprocals, let us use the concept of "processor load" instead in the following discussion. The algorithm provided can be applied, with some minor modifications, to minimize the simulation time growth rate among the processors. As mentioned before, an ideal allocation (without considering the communication overhead) is to assign exactly the same amount

of load to each processor. However, this is usually not possible because the sizes of the loads cannot always be grouped into M sets with equal sum. *Optimal allocation* is therefore defined to be the feasible allocation which is *closest* to the ideal. (Currently, we assume that the "closest" means the allocation which has the smallest variance. The ideal allocation, according to its definition, always has zero variance.)

Our heuristic allocation method attempts to approach the ideal allocation by assigning slightly above or below-average load to each processor (since the exact average is not always possible). With the nearest neighbor constraint, the allocation begins with either the first node, n_1, or the last node, n_N, which are both adjacent to the host node in the network model. Nodes are assigned to a processor until the total load is greater than or equal to the ideal. Allocation will then continue recursively with the next node and the next processor in the same manner until either the nodes or the processors are exhausted. Moreover, if the total load on the first processor is greater than average, the algorithm also checks the below-average case by removing the last assigned node from the first processor and then continue to the second processor. The algorithm has been implemented as a procedure in C. Experimental results indicate that this algorithm usually provides very near optimal allocations, especially when $N \gg M$. It should be noted that the complexity of this algorithm is $O(N \times 2^M)$, and it may become very time consuming when M is large. However, the allocation of a 64-node, 16-processor example only needs about one minute of computation time on a modern mini computer. A less complex allocation algorithm will probably be needed if over 20 processors are used.

5. Implementation of Distributed Simulator

A simplified flow chart for distributed simulation of ring networks is shown in Figure 6. For each event in the simulation, the following steps are repeated: (1) The

input queue of each process is checked; arrived input messages, if any, are read. (2) If there are straggler messages, the process will roll back to a state prior to the time stamp of the earliest straggler. (3) If the message is an anti-message, the corresponding positive message will be deleted. (4) The process will then determine the most imminent event, (5) simulate it, and (6) send output messages if there are any.

5.1 Next Event Time (NET)

In discrete event simulation, sorting the event list usually takes up a significant portion of the computation time. In distributed simulation, after the model has been partitioned, the event list for each sub-model becomes much shorter. In the current implementation, each process contains several network nodes. Therefore, the events among these nodes need to be sorted, and the ''earliest'' one will be simulated next. The sorting is achieved in a two-level manner: events in each node are sorted, which generates the *next event time* (NET) for each node. The most imminent node is then determined by sorting the nodes according to their individual next event time.

In the first level, the next event time of a node is determined by three factors: the time stamp of the slot (message) at the front of the input queue, the time stamp of the front element of the arrival queue, and the node time. Roughly speaking, the next event time of a node is the smaller one of the time stamp values between the front elements of the two incoming queues. But it must be strictly greater than the node time unless the current node is the destination of the front element of the input queue. In the second level of sorting, the nodes are checked in the order of their physical locations. If two or more nodes in a process have the same NET, the node with the smallest node number will be simulated first. This guarantees that there will not be any preemption caused by one node to another inside a process. Internal preemption is prohibited when several

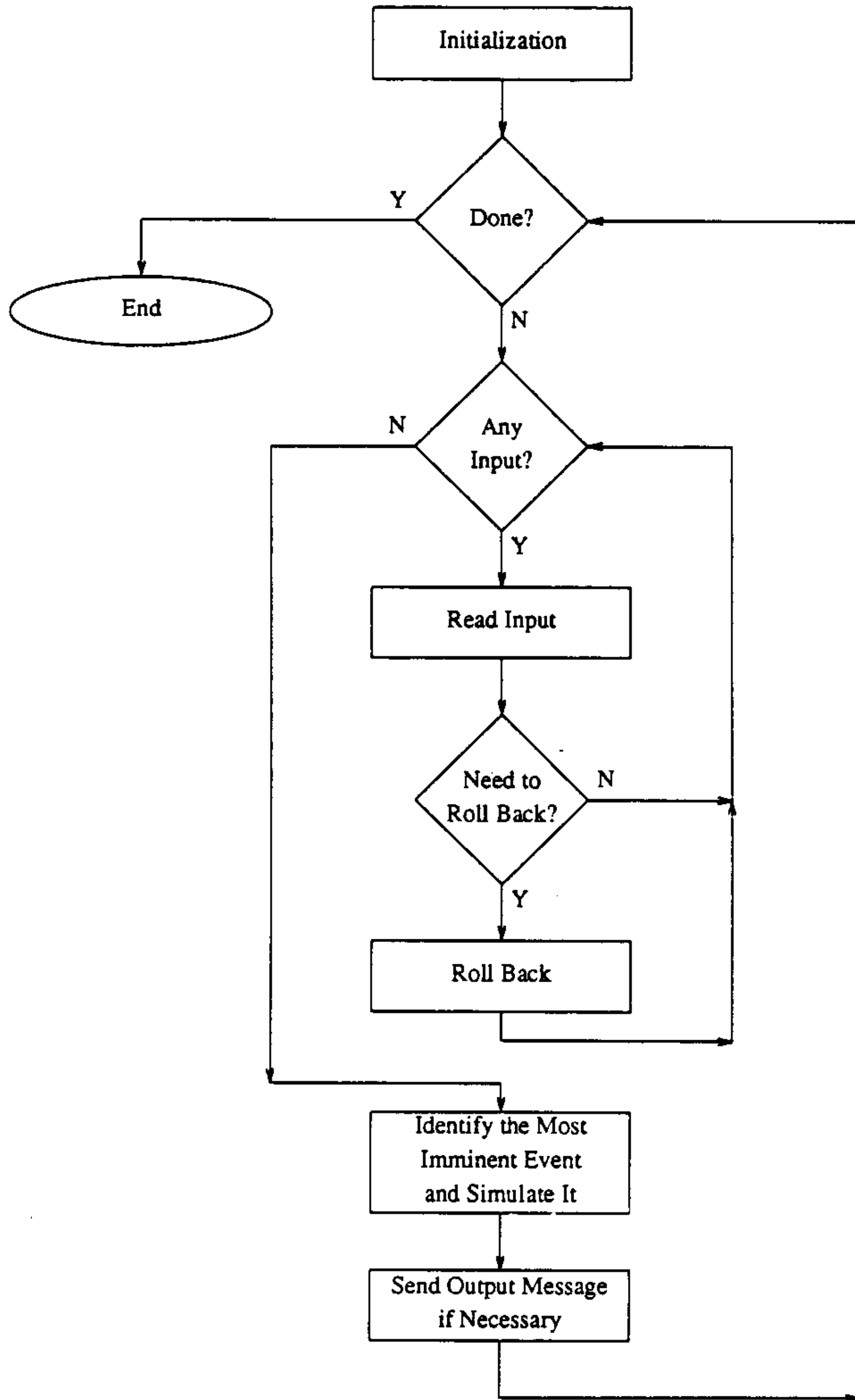


Figure 6: The Flow Chart for a Process

nodes are grouped into one process and their states are saved together, as in this implementation. An internal preemption will cause all the nodes to roll back to a previous state, unless an external straggler arrives to break this sequence, the simulation will simply repeat the same events and cause an identical internal preemption; i.e., the simulation is trapped in a deadlock.

5.2 Queues

There are five types of queues in each process representing a group of neighboring terminal nodes: input/output queues, input queues, output queues, arrival queues, and response queues. The first three types of queues form the connections between nodes in the ring. Except for the input queue to a process, i.e., the input queue to the first node in a process, or immediately after a roll back, messages will arrive a queue in strictly increasing time stamp order. Hence each new element arriving a queue will automatically become the tail element of that queue. Moreover, no anti-message should ever enter these queues. Arrival queues are part of the command generation mechanism; they are inherently different from those queues described above. Response queues have not been implemented and will not be discussed in this report.

The model of a typical queue is shown is Figure 7. The front of a queue is defined to be the next element to be dequeued. The tail is the last element enqueued. If every element has been removed, the pointer *qfront* will be NULL. However, pointer *qtail* will not be NULL unless nothing has ever been enqueued in the available history. There is also a pointer *qhead* which points to the very first element (in the available history) of that queue. Queues are implemented as linked lists. Hence memory can be allocated in a flexible manner. Moreover, elements may be added to or removed from the middle of a queue without much overhead. To save the state of a queue, it is not

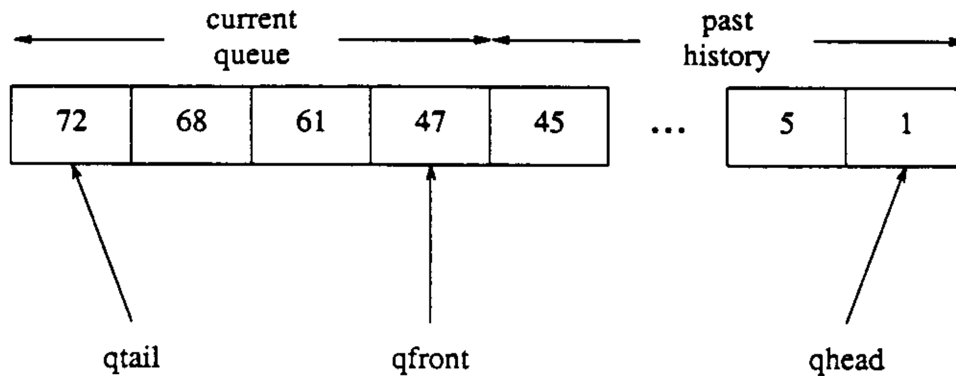


Figure 7: Queue Representation

necessary to copy each queue element once for each state saved; only two pointers *qfront* and *qtail* which point to the current queue front and tail elements should be saved.

5.2.1 Input/Output (I/O) Queues

The queues between adjacent nodes in the same process are called input/output queues because they form connections between each pair of nodes. Messages are sent from one node to its successor node through the i/o queue in between.

When a roll back occurs, the qfront and qtail pointers of each i/o queue in the process will be replaced by the corresponding values for the saved state. Queue elements which are after the restored qtail (the after-roll-back qtail) will be discarded. Some of these discarded elements might be needed again later on in the simulation, but they will always be regenerated by the previous nodes. In other words, the cancellations among nodes in a process are not delayed. This is a consequence of having several nodes in a process.

5.2.2 Input Queues

An input queue is the connection between the input pipe and the first node of a process. Unlike the i/o queues described in the previous section, messages arriving an input queue are not necessarily in increasing time stamp order because the processes are not synchronized and can roll back. Moreover, some arriving messages may be anti-messages.

When a new message arrives, the process needs to determine whether a roll back is needed. A simple way is to compare the time stamp of the arriving message with that of the front element of the input queue. If the time stamp value of the new element is smaller, a roll back may be considered necessary. (Note that this is a necessary but not sufficient condition to require a roll back; however, a simulation which makes unnecessary roll backs remains logically correct although wasteful of computation time.) During a roll back, the entire process must return to a state whose time stamp is less than (earlier than) that of the new element. It is an error if no saved state far enough into the past is available. A straggler message which does not cause a roll back should be placed in the middle of the input queue. (Therefore, the input queue may be considered to be a FIFO device with respect to simulation time, but not with respect to real time.) It should also be noted that in a roll back, only the old qfront pointer is restored; the qtail pointer for the input queue remains unchanged. Hence no input queue elements will be discarded during roll backs.

If the new arrival is an anti-message, the corresponding positive message will be searched for and both messages will be canceled. However, if the positive message is not found, an error has occurred and the simulation will be aborted because in this implementation, there are no alternative paths for message transmission. Hence it is impossible for an anti-message, which is always generated after the corresponding

positive message, to arrive earlier. A remaining problem is that when a positive message is canceled from the middle of an input queue, the pointers for the saved states should be checked. Those pointing to the message to be deleted should be adjusted to the previous or next message.

5.2.3 Output Queues

An output queue forms the connection between the last node and the output pipe of a process. The output queue serves both as a buffer for output messages leaving the process and as a storage for previous output messages in saved states. Each message in the output queue has a ''sent'' tag which indicates whether that message has already been sent or not. Messages join the output queue from its tail; the pointer *outqtail* always points to the last element enqueued. The process *sendout* copies messages pointed by *outqfront* and sends them to the next process. The ''original'' copies of these messages remain in the output queue, but their sent tags are flipped to ''1.'' Hence they automatically become the past history of that queue. There is also a pointer *lastsent* which points to the last message sent.

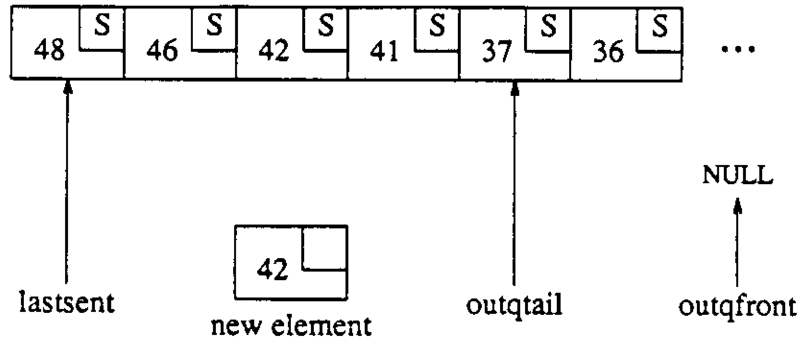
Delayed cancellation complicates the mechanism for the output queue; each output message sent needs to be marked. Normally, lastsent points to the predecessor of the element pointed by outqfront, which is the next message to be sent. After a roll back, previous pointers for outqfront and outqtail are restored; only those messages which should not have been sent will be canceled. Output queue elements which have not been sent and are between the after-roll-back outqtail and the before-roll-back outqtail will be deleted. There are several different situations depending on the relative position of the pointers; a typical case is shown in Figure 8. The number in each queue element is its time stamp. An ''S'' in the box at the upper right hand corner indicates the ''sent'' flag

has been set. An ''A'' signifies that it is an anti-message. Since messages must join the output queue in increasing time stamp order, when the new version of the message with time stamp 42 arrives after a roll back as shown in Figure 8(a), it also implies that message 41 should not have been sent before. Therefore, its ''sent'' flag is turned off and the ''anti-message'' flag is turned on as shown in 8(b). When an anti-message is actually sent, the copy in the output queue is discarded. Hence no record of that message would exist in the history of that process any more as shown in 8(c). The new message at time 42 is discarded because an identical copy has already been sent before.

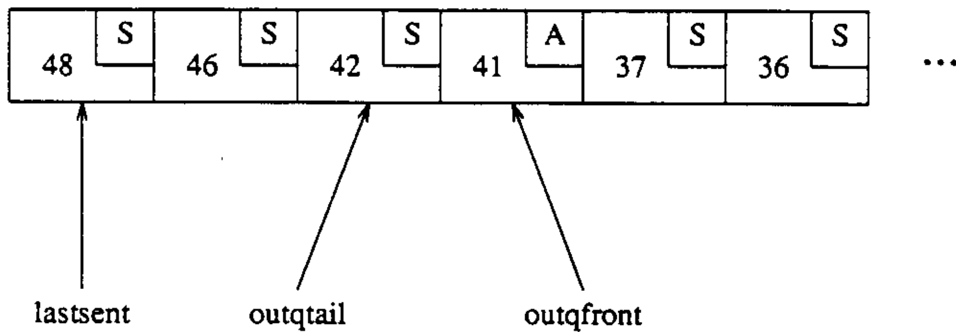
5.2.4 Arrival Queues

Unlike the three types of queues described above, the arrival queue is for new commands arriving the network through a terminal node. The number of elements in an arrival queue is fixed and is equal to the number of terminals connected to that node. Each element contains the next arriving command from a particular terminal, and these elements are sorted in increasing time stamp order. The head element of the arrival queue is the next command entering the node. After an command has ''arrived,'' a new arrival from that terminal will be generated randomly according to a predefined distribution to replace the removed element, and the queue is resorted.

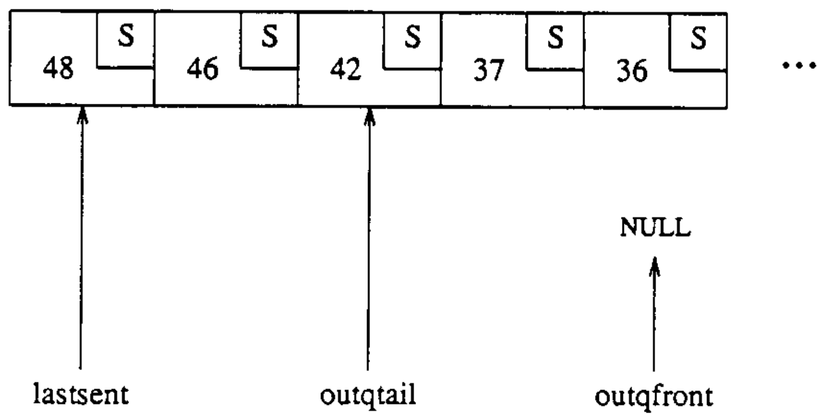
Old arrivals are saved in the *oldarrival* queue. These past values will be needed again in two situations. After a roll back, rather than randomly regenerating command arrivals for the roll-backed period, previous arrivals are reused. This is preferable when delayed cancellation is used. If new random arrivals are generated, it is unlikely that they will be identical to the arrivals generated before; hence the old messages will need to be canceled before new messages are sent. Since both sets of arrivals are statistically equivalent, it is therefore a waste of time to replace the old ones. These old arrivals are



(a) The Output Queue before the Cancellation



(b) The Output Queue after the Cancellation



(c) The Output Queue after Sending the Anti-Message

Figure 8: Delayed Cancellation at the Output Queue

also needed when responses from computers arrive. The time stamps of both the commands and the response messages are needed to calculate the ''round-trip'' delays.

5.3 Message Format

<table>
<tr><td colspan="4">time stamp</td></tr>
<tr><td colspan="3">source</td><td>destination</td></tr>
<tr><td>+/-</td><td>prior</td><td>S</td><td>message id</td></tr>
</table>

Figure 9: Message Representation

In the simulation, each message is represented as three consecutive unsigned words as shown in Figure 9. The first word is the time stamp. The second word consists of the source and destination addresses of the message; they are usually i.d. of terminals or computers. The last word consists of a one-bit flag indicating whether the message is regular (positive) or anti (negative), a two-bit priority value, another one-bit flag to show whether the message has been sent or not, and the remaining part is the message i.d., which is a number needed to distinguish messages with the same source/destination pair.

6. Other Deadlocks and Flow Control

6.1 Pipe Read Deadlock

As mentioned before, pipes are used for inter-process communication in the current implementation. Unfortunately, in UNIX (and hence LOCUS), when a process reads from an empty pipe, it will wait idly until something is written into (the other end

of) the pipe.† If every process in the ring happens to be reading from an empty pipe at a certain point, clearly, each process will be waiting forever and the simulation deadlocks. Although this problem is unlikely to occur frequently, it must be resolved. Moreover, it is also undesirable for any process to remain idle for a long time and wait for a message to arrive. One solution to this problem is to make the reading process responsible for checking the input pipe before it executes a read command. For example, before a read, the reading process can write a dummy token into the input end of its input pipe so that there is always something to be read. Since the pipe is a FIFO device, if the first item read is the dummy, then the pipe was empty before the dummy was entered. Otherwise, the process should continue reading until the dummy finally appears.

6.2 Pipe Write Deadlock

There is a "dual" deadlock problem for pipe writes. When a process writes to a pipe, if the pipe is full, the process will remain idle and wait until something has been read from the other end of the pipe so that there is room available to complete the write. Therefore, if every pipe is full when each process on the ring is attempting to write, the processes will be waiting forever and the system deadlocks.

One way to limit the number of messages in a pipe is to use permission tokens. These tokens are issued by the message-receiving process to the sending process; they travel through a separate pipe in the opposite direction to the message pipes. Initially, the maximum number of permission tokens is available; it should be less than the number of messages the pipe can hold. Every time before the sending process puts a message in its output pipe, it is required to obtain a premission token. If there is no token available at a certain time, the process should postpone sending the message and

† The timeout function *select* is designed to avoid hang up during pipe reads, but it has not been implemented on LOCUS for multiple-site applications.

process some other events. It is very important that a process does not busy wait for a token; otherwise, the pipe write deadlock problem is unchanged. A permission token indicates that there is room in the output pipe and is discarded as soon as it is obtained. During the simulation, every time after a process reads a message from its input pipe, it will send a permission token back to its predecessor process. Therefore, the sum of the number of available tokens and the number of messages in the pipe is a constant.

Permission tokens do not really introduce much additional overhead to the simulation because it is necessary to acknowledged each message anyway if GVT is to be calculated periodically for memory management purpose [Sama85]. The acknowledgment message can simply serve as a token also so that no extra messages will be sent in the ''backward'' direction.

6.3 Buffer Usage Deadlock

When the output queue of a process is full, this process should neither generate any more arrivals nor accept any inputs unless the destination of the current front element of the input queue happens to be n itself. If this problem appears all around the network, the simulation will not be able to progress. This situation is similar to the *indirect store and forward deadlock* problem described by Kleinrock [Klei76], and is a result of having too many messages in the simulation. A possible solution to this problem is to restrict the arrival of new commands once the output queue has been filled up beyond a certain threshold. Before every buffer becomes full, command arrivals will be reduced or even prohibited until a sufficient amount of messages have left the ring. It will still be possible for individual queues to become full, but not for all of them to be filled up at the same time. Hence packets will be able to move around and eventually leave the ring. However, when there are ''amplifying'' devices such as computers which

accept one-packet commands and generate multiple-packet responses, the number of packets may be multiplied by a significant factor when they pass through these devices. It will be necessary to set up tighter thresholds or make available more buffers for outputs from computers in the ring. This problem will be further investigated once the implementation of the simulator has reached a point so that it can be understood more thoroughly.

7. The Current Status of the Project and Upcoming Work

A significant portion of the distributed simulator has already been implemented and tested. This includes the simulate forward, roll back, cancellation, and arrival generation mechanisms for terminal nodes. We are planning to complete the implementation of the simulator module for host nodes in the near future. In the mean time, a special benchmark which contains terminal nodes only is used to test the simulator. Messages are sent from one terminal to another. The nodes in the model are separated into two processes. These processes are migrated to two VAX 11/750 computers so that the simulator can be tested in a real distributed environment.

Asynchronous distributed simulations are in general difficult to debug. The main problem is that the sequences of the events assigned to different computers may be executed in (slightly) different orders during separate runs. When an error is discovered, there is no guarantee that it will appear again in subsequent simulation runs although the initial conditions are the same. Therefore, it is both difficult to identify errors and to correct them. Our current approach is to save a brief description of the intermediate events into data files and then trace them manually afterwards. This method is rather time consuming and tedious but seems to be effective.

After the implementation of the simulator for host nodes, we are planning to realize the memory management, deadlock prevention, and flow control functions. These are necessary features for a realistic distributed simulator for networks. Finally, we would like to add statistical analysis capabilities for simulation result studies.

References

[Cheu85] Cheung, S., J. W. Carlyle, and W. J. Karplus, ''Distributed Computer Simulation of Data Communication Networks: Project Report 2,'' Department of Computer Science, University of California at Los Angeles, Los Angeles, California, Tech. Rep. CSD-850013, March, 1985.

[Doel84] Doelz, M.L. and R.L. Sharma, ''Extended Slotted Ring Architecture for a Fully Shared and Integrated Communication Network,'' in *Proceedings of the MIDCON 1984 Conference,* Dallas, Texas: September 12, 1984.

[Jeff83] Jefferson, D. and H. Sowizral, ''Fast Concurrent Simulation Using the Time Warp Mechanism Part I: Local Control,'' The Rand Corporation, Santa Monica, California, Tech. Rep. Rand Note N-1906AF, 1983.

[Jeff85a] Jefferson, D. and H. Sowizral, ''Fast Concurrent Simulation Using the Time Warp Mechanism,'' in *Proceedings of the Conference on Distributed Simulation 1985,* San Diego, California: Society for Computer Simulation, January, 1985.

[Jeff85b] Jefferson, D., ''Virtual Time,'' *ACM Transactions on Programming Languages and Systems,* Vol. 7, No. 3, July, 1985, pp. 404-425.

[Jeff85c] Jefferson, D. *et. al.*, ''Implementation of Time Warp on the Caltech Hypercube,'' in *Proceedings of the Conference on Distributed Simulation 1985,* San Diego, California: Society for Computer Simulation, January, 1985.

[Klei76] Kleinrock, L., *Queueing Systems, Volume II: Computer Applications*, New York: John Wiley and Sons, 1976.

[Sama85] Samadi, B., *Distributed Simulation, Algorithms and Performance Analysis*: Ph.D. Dissertation, University of California, Los Angeles, February, 1985.