DISTRIBUTED COMPUTER SIMULATION OF A DATA COMMUNICATION NETWORK

Shun Cheung

July 1987
CSD-870039

UNIVERSITY OF CALIFORNIA

Los Angeles

Distributed Computer Simulation of a Data Communication Network

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science

by

Shun Cheung

1986

TABLE OF CONTENTS

page

1 INTRODUCTION 1
1.1 Background 1
1.2 The Benchmark Network 5
1.3 The Objective and Scope of this Research 8
1.4 Outline of the Dissertation 9

2 SYNCHRONIZATION METHODS 10
2.1 Synchronous Methods 12
2.1.1 Fixed Time Increments 13
2.1.2 Variable Time Increments 14
2.1.3 The Virtual Ring Method 14
2.2 Asynchronous Methods 15
2.2.1 Object Oriented Simulation 16
2.2.2 Synchronization Methods Which Prevent Preemptions 17
2.2.2.1 The Time Packet Method 20
2.2.2.2 The Link Time Algorithm 21
2.2.2.3 The Null Message Method 24
2.2.2.4 The Deadlock/Recovery Method 25
2.2.2.5 The Safe Forward Simulation Time Method 26
2.2.3 The Time Warp Method 27
2.2.3.1 State Saves and Recovery 28
2.2.3.2 The Global Virtual Time 33
2.3 Conclusions 34

3 SPECIALIZED ALGORITHMS 36
3.1 Simulation Model Specification 36
3.2 Problems with Model Partitioning 36
3.3 Queueing Network Models 37
3.3.1 State Saves and Recoveries of a Queue 38
3.3.2 The Software Model of a Node Connected to Terminals 40
3.3.3 The Software Model of a Node Connected to Hosts 43
3.4 Congestion Control 49
3.5 The Role of the Central Controller 51
3.5.1 Algorithms for GVT Evaluation 52
3.6 Periodic State Saves 56
3.6.1 Chain Roll Backs 57
3.6.2 Delayed (Lazy) Cancellation 59
3.6.3 GVT Decrease 60
3.7 Conclusions 61

4 MODEL PARTITIONING 63

4.1 Main Sources of Simulation Overhead 63
4.2 Guidelines for Object Allocation 65
4.3 A Heuristic Allocation Algorithm 67

5 SIMULATOR DEVELOPMENT 73
5.1 Modules in a Simulator Component 74
5.2 Simulation cycles 76
5.2.1 Simulation Cycle of a Host Simulator 76
5.2.2 Simulation Cycle of a Terminal Simulator 78
5.3 Event Generation 80
5.4 Message Arrivals 86
5.5 Pipes 87
5.6 State Saves 88
5.6.1 State Save Frequency 88
5.6.2 Location of Check Points 91
5.6.3 Conclusions 94
5.7 Message Format and Representation 94
5.8 Deadlocks 96
5.8.1 Deadlock Due to Incorrect Modeling 96
5.8.2 Buffer Usage Deadlock 98
5.9 Conclusions 99

6 EXPERIMENTS AND RESULTS 100
6.1 Network Performance Evaluation 100
6.2 Characteristics of the Benchmark Network 101
6.3 The Experiments and Results 102
6.4 Simulation Results Analysis 111
6.4.1 Estimating the Steady State Response Time 111
6.4.2 Generating Confidence Intervals 114
6.4.3 A Simple Analytical Model 121
6.5 Performance of the Distributed Simulator 123

7 CONCLUSIONS 126
7.1 Background and Scope 126
7.2 Achievements and Contributions 127
7.3 Results and Conclusions 129
7.4 Future Research and Extensions 130

References 132

ABSTRACT OF THE DISSERTATION

Distributed Computer Simulation

of a Data Communication Network

by

Shun Cheung

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1986

Professor Jack W. Carlyle, Co-Chair

Professor Walter J. Karplus, Co-Chair

Performance evaluation through computer simulation is a very important step in the planning of modern data communication networks. In view of the advent of network of micro-computer workstations, it is attractive to investigate the possibility of carrying out these time and memory-consuming computations in this type of new environment in a distributed manner.

This dissertation describes the conceptual development and implementation of a distributed discrete-event system simulator for data-communication networks. A slotted-ring network model is used as a benchmark; the model includes multi-terminal (user) nodes and host (cpu) modes. Inherent concurrency and other special features of this type of models have influenced the design of the simulation software, with attention to the selection of appropriate mechanisms for distributed processing, task

assignment, synchronization, memory management, congestion control, and data collection. Synchronization is carried out using state-saving and rollback strategies, based on the time-warp concept, particularized to queueing-network model applications; state information is maintained with the aid of pointer manipulations in a collection of queues. Permits are used to throttle the flow of inter-processor messages in order to prevent congestion in the communication medium and to restrain memory usage in the simulators. A model partitioning algorithm has been developed for ring type network models.

To demonstrate feasibility and correctness of the methodology, a distributed simulator is implemented on an existing minicomputer network supporting distributed processing, with a view toward the longer-range goal of developing tools usable in computing environments consisting of distributed workstations. The experimental results suggest that medium-scale distributed simulation using a network of mini-computers or workstations is feasible and is a promising approach especially for large models which exceed the capacity of single machines.

CHAPTER 1

INTRODUCTION

1.1 Background

In the 1970's, data communication networks have evolved rapidly and gain important economic significance. They are used in many aspects in our everyday life such as telephone conversations, financial transections, file and data transfers, electronic mail, video signal communication, etc. More recently, due to the increasing demand of intra and inter-office communications, local area networks (LAN) have especially become very popular. A common type of LAN is the ring topology. In the early 80's, Doelz Networks implemented an extended-slotted ring architecture based on the Pierce loop [Pier72] which hierarchically enhances the slotted ring architecture so that it is suitable for long-distance communication in addition to local area service [Doel84]. Their design also takes into consideration the current trend of integrating various types of traffics which have very different demands on the same network hardware. The goal is to have one general-purpose network architecture so that the inter-network interface overhead and hardware cost can be significantly reduced. In the planning and design phase of these costly and complicated network installations, there are many tradeoffs to be considered and many parameters to be optimized. It is important to evaluate the performances of alternative designs under various normal and unusual situations such that eventually a network with appropriate performance will be built under cost

constraints.

There is a number of methods to evaluate the performance of data communication networks. If a prototype or an existing network is available, it may be tested under various representative conditions (e.g. different traffic loads), and the network's performances can be measured. This method is, generally speaking, the most accurate one but is often not feasible in the planning and design stage when usually no actual hardware is available. The two most common alternatives are mathematical performance analysis and digital computer simulation. Analytical techniques have been developed to near maturity at the present. They have the advantage of simplicity. Once an analytical solution is available, different parameter values may be substituted into the expression such that performances under different conditions may be obtained with little extra effort. Moreover, the relation between the performance and the parameters can be observed from the expression so that it can be understood more thoroughly. Unfortunately, closed form analytical solutions are often possible only with idealized and simplified assumptions. Moreover, they usually provide steady state results only; transient behaviors cannot be obtained analytically except for some very simple situations. When accurate or transient results are desired, it is impossible to use analytical methods alone. Digital computer simulation is therefore often a necessary mean for performance evaluation of data communication networks. (We are not suggesting that analytical methods have limited applicabilities. In fact, the contrary is true. Frequently, a combination of both methods is used so that the advantages of the two can be combined.)

Simulation of communication networks belongs to the discrete-event simulation category in which all space and time variables are discretized; that is, state changes are assumed to take place instantaneously at certain times. Every state change is considered as an event. Each event in the simulation has a simulation time at which that event "occurs." In conventional discrete-event simulation on one CPU, the events are sorted into an event list in increasing simulation time order. The most imminent event is always simulated next. New events generated are inserted, according to their simulation times, to appropriate positions in the list.

Compared to analytical methods, the main advantages of simulation are its generality and accuracy. Any model between a highly simplified rough approximation and a detailed description can be used. More importantly, in the model, parameters such as job arrival rates and service rates may be assumed to be random variables with virtually any probability distribution. Hence distributions which closely approximate the reality rather than those which provide tractable solutions may be used. These are the reasons why computer simulations can provide very accurate predictions not possible through analytical methods. Moreover, since a simulation run is an abstract of the events in the real world, it can also predict transient behaviors of a network under normal and especially unusual conditions. For example, after a connection fails in a network, it would take some time before the routing and flow control mechanisms can adapt to the new topology. In the mean time, traffic in the network might become congested. This type of crucial transient behavior can be thoroughly studied and understood through simulations.

Unfortunately, these advantages of computer simulation do not come for free. Because of the probabilistic nature of simulation, small-sample data results would provide little information. Instead, a large amount of data should be collected so that some statistically meaningful conclusions can be drawn before a simulation run is terminated (or additional runs are determined to be unnecessary). In addition to that, when the model is complex, a simulation run usually goes through a sufficiently long period of *transient phase*. During this period, the behavior of the system being simulated may be significantly different from that in the steady state. Theoretically, the effect from this period will affect the behavior of the system for an indefinitely long period of time, but this effect decreases as time increases. Hence it will be negligible after a certain point when the system may be considered to be in steady state. If one wants to study the steady state behavior of a system, all result data from the transient phase should be discarded, and a simulation run must be long enough so that enough data from the steady state period are collected. With all these factors, it is not surprising why simulation is well known to be very computer time consuming.

Since the 1970's, due to the advent of modern VLSI technology, the price of moderately powerful integrated circuit chips has been falling rapidly. This trend initiated the popularity in research of multi-processor systems and distributed computation. In the past few years, networks of mini-computers and micro-computer workstations are becoming very popular. Some of the modern workstations, using state-of-the-art processors, have computing power very close to that of large mini-computers but are available for just a fraction of

their price. Since there is much inherent concurrency in the simulation model of networks, it becomes attractive to consider conducting time-consuming simulations in these newly available distributed environments.

It should be understood that in the research presented here, we are not proposing to attempt to optimize execution time by solving application problems in a distributed manner. Using a powerful mainframe or super-computer can be more efficient, at least in execution time, but may not be conveniently available to users in a field-engineering situation, for instance. Such users may wish to conduct simulations as an aid in planning or installing data communication network products, and the computing resources conveniently at hand to them are likely to be workstations. If the workstations are networked in such a way that some distributed processing capabilities are available, how can this resource be harnessed to run the simulations? We seek a constructive response to this question; our goal is thus to verify feasibility and practicality in smaller-scale simulations, and easy extensibility to larger application problems when more processors can be added to the distributed computing system.

Note that we use the term "network" here in reference to two distinct systems, both of which may contain computers and terminals, but need not have any physical similarities: (1) the data communication network being simulated, and (2) the actual computing network on which the simulation is executed. The distinction should be clear from the context.

1.2 The Benchmark Network

The Doelz network [Doel84] is an extended slotted-ring (ESR) network based on the Pierce Loop [Pier72]. An example with three network nodes (NN) is shown in Figure 1.1. Connected to the first node is a number of computers (hosts). The other nodes are connected to terminals.

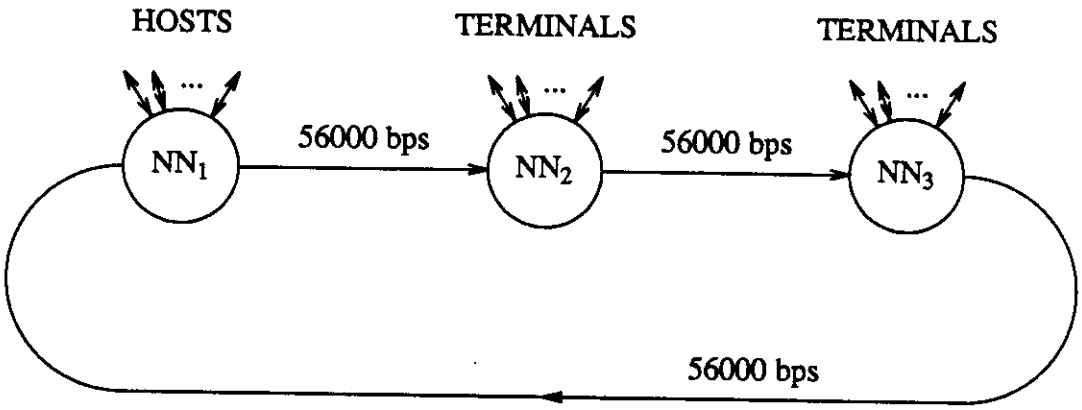


Figure 1.1: A Doelz ring network with three network nodes

The philosophy behind the design of this network [Doel84] is to have a fully shared and integrated network for multiple purposes. The design goals and requirements can be summarized as following:

1. Usually, a ring network implies that it is for local area communications. However, the ESR is a homogeneous architecture for local as well as wide area communications.

2. The transmission wire and nodal hardware should be fully shared by data transmission and network management control functions.

3. The network must potentially be able to provide very fast response time so that it is suitable for future integration of digitized voice transmission along with synchronous and asynchronous data communications.

In the Doelz network, transmission lines in a basic loop are connected by lower-level intelligent nodes (called "Elite-One"). Packet assembly, disassembly, queueing, packet switching, data generation, and repeating are all performed by these nodes. Packet switching is supported by virtual circuits set up at the time of service request. Since many virtual circuits can share a physical transmission line, this architecture is very efficient and cost effective.

A small packet size, namely 12 bytes (96 bits), is chosen mainly for the advantage of very fast response time. Among the 12 bytes, five are overhead bytes (for header and error checking) and seven are data bytes, but two packets may be combined into one long block which still contains five bytes of overhead while the data portion is increased to 19 bytes. To further improve the response time for voice packets, there are four levels of user-defined priorities. Packets which require fast response time may be assigned higher priorities. In regular slotted-ring networks, when a node receives an occupied slot, unless that node happens to be the destination, the slot will bypass the node unaltered. In the Doelz network, however, if a packet in a siding queue has higher priority than that of a packet in an arriving slot, the packet in the siding queue will seize that slot while the other packet has to wait in a buffer for the next available slot. Multiple rings with the ESR architecture may be connected together with higher-level network nodes (called "Esprit-One").

These nodes are connected by trunk ESRs to form a two-level hierarchical Doelz network.

There is no constraint in the topology of a Doelz network. Although the physical configuration is a ring, a designer may choose any logical topology to minimize the cost of the transmission medium. The reliability of the Doelz network is enhanced by its fault-tolerant design with redundancy in critical hardware components such as the higher-level nodes. The ring itself has the self-healing ability. When a line is out of service, the ring can reconfigure itself by adding a stand-by link segment.

1.3 The Objective and Scope of this Research

To summarize the introductory discussion of Section 1.1, the objective of our work is to develop application-directed simulation tools executable via distributed processing on an environment which is becoming widely available, namely a network of minicomputers and/or microcomputer workstations; the application is to estimate the performance of proposed data communication network installations. A collection of technical problems, including model specification and representation, processor synchronization, simulation load partitioning, and practical implementation difficulties, must be resolved before this goal can be achieved. We will address these in detail in the following chapters.

After obtaining appropriate solutions to these problems, to test the applicability and to verify the feasibility of our methodologies, we have

implemented a distributed simulator on the Olympus network at UCLA, using as a benchmark the Doelz network. The Olympus network consists of several VAX † mini-computers and runs the LOCUS operating system [Pope85], which is a network-transparent distributed extension of UNIX. ‡ The LOCUS/VAX environment is appropriate for our work for at least two reasons: (1) LOCUS is a contemporary example of a general-purpose distributed operating system which is also being offered for networks of workstations (e.g., IBM PC/ATs), and (2) conducting initial experiments on mini-computers, rather than micros, makes it possible for us to run benchmark experiments on a single machine as well as on several machines for comparison.

To limit the amount of effort required for implementation, our simulator is designed mainly for ring networks and benchmarks introduced in Section 1.2 above. However, we believe that the design can be expanded and generalized, using the methodology developed, to a more general-purpose distributed simulator for data communication networks.

We realize that there is much overhead in distributed simulation. Because of the computing power of workstations, it is unreasonable to expect a network of them will provide an execution time faster than that obtainable from powerful mainframe systems; as stated before, we seek feasibility first rather than optimization. Moreover, general-purpose distributed operating systems such as LOCUS are not designed to optimize execution of communication-

† VAX is a trademark of Digital Equipment Corporation.

‡ UNIX is a trademark of AT&T Bell Laboratories.

intensive distributed applications. The point is that such operating systems are becoming widely available and therefore will provide convenient environments on which applications can be carried out. In the future, the methodology developed can be adapted as these distributed or parallel computer systems evolve and provide better support to distributed simulation.

1.4 Outline of the Dissertation

In Chapter 2, the problems in distributed discrete-event simulation, especially the synchronization problem, are introduced, followed by a survey of existing solutions and a careful comparison of their merits. Chapter 3 explains the components in the simulation models and extensions to the selected methods such that they become more efficient for the simulation of queueing networks. The congestion problem and a solution are also presented. Chapter 4 provides an algorithm for model partitioning and assignment. Its effectiveness is demonstrated by an example. Chapter 5 discusses the experimental work involved and Chapter 6 shows various simulation results, which verify the correctness of the simulator. Finally, the conclusions, contributions, and future research directions are presented in Chapter 7.

CHAPTER 7

CONCLUSIONS

7.1 Background and Scope

In view of the increasing demand on performance evaluation of data communication networks through digital computer simulation and the gradual popularity of networks of micro-computer workstations as a new computing environment, it is reasonable and necessary to consider carrying out simulation in this type of new environments. Problems such as model specification and representation, simulation load partitioning and allocation, synchronization among processors, and congestion control must be solved before distributed simulation can be realized. There is much work on these problems is general settings, and assuming that specialized architectures will be available. The research described in this dissertation is directed toward the development of a distributed simulator, accepting models represented as queueing networks, for a ring-type data communication network in particular, seeking an integrated solution which is implementable and extensible on general-purpose systems, and embedding needed mechanisms in the application code where the general-purpose system does not specifically provide high-level support.

7.2 Achievements and Contributions

Before a simulation run can begin, it is important to prepare an accurate model in the correct format for input to the simulator. To solve the model specification problem, we have implemented an interactive program which prompts a user for the necessary parameters. This program greatly reduces the probability of neglecting important information or using meaningless parameters in the simulation model.

Several synchronization methods for asynchronous distributed simulation are already available. We have developed a number of criteria to evaluate their respective merits and then carried out careful studies and comparisons among these methods. We concluded that the roll back scheme in the Time Warp method [Jeff85a] is the most promising approach because it (1) can exploit the maximum amount of inherent concurrency, (2) introduces relatively few extra message transfers, and (3) requires no tight interactions with a central controller. We have therefore adopted this synchronization method in our simulator. A significant portion of recent research in Time Warp is directed toward the implementation and future extension of a special-purpose distributed operating system for discrete-event simulation [Jeff85c]. Application programs are executed on top of this operating system on a special-purpose multi-processor system. In our work, the synchronization mechanism is built into the application program, and it runs on a general-purpose distributed operating system which provides low-level support for distributed processing. Hence we are able to consider special properties in the simulation models and achieve gains in simplicity by adapting the state save, roll back, and recovery schemes for simulation of communication networks. A

potential problem of Time Warp is large memory usage and slow progress. To reduce memory usage, we have developed a state saving scheme specially for queueing network models which saves pointers to representative queue elements rather than copying the entire queue for each state saved. Experimental results indicate that even with the 20 host/terminal model, the simulation will need only a few thousand queue elements.

One major factor which affects the performance of a distributed simulator is the balancing of simulation load. A heuristic allocation method for ring type networks has been developed. Although our small-scale experimental test cases do not actually require allocation algorithms, a number of examples have been created just to test the performance of the algorithm; the overall result is very promising.

While it is important to leave the processors asynchronous and exploit as much inherent concurrency as possible from the model, when the simulation times on different processors become too far apart, the memory could be exhausted due to a large number of saved states and very long queues. Therefore, there should be a mechanism which prevents the simulator from extracting more inherent concurrency than it can handle and keeps the processors approximately synchronized. We have developed such a method, which uses "permits" to control memory usage.

Major issues in distributed simulation were, in this manner, studied and (conceptually) resolved; however, a real test requires that they should be put into practice to demonstrate their feasibility and applicability. Moreover,

detailed problems often cannot be discovered until an actual implementation is carried out and testing is done. Therefore, we applied our methodology and developed a distributed simulator on an existing distributed operating system. The correctness of our approach and its basic algorithms have been verified in this way by experimentation.

7.3 Results and Conclusions

From the experiments, we can conclude that distributed simulation of data communication networks using a general-purpose distributed computing environment is indeed feasible based on the methodology we developed. The advantage of this approach is that a simulator can operate without dedicated hardware or system software, and can be transported to workstations as general-purpose distributed operating systems become available for them. However, efficiency inevitably suffers because, at least in this case, neither the operating system nor the processors provide specialized support for intensive inter-processor communications. The inter-process messages are short in our particular application (they are 12 bytes long in this simulator and could have been even shorter had a compact coding scheme been used), but they demand fast inter-processor communication. Pipes in UNIX are designed for the opposite purpose: transmission of a large amount of data without stringent speed requirements. If the main objective of distributed simulation is to improve absolutely the execution speed using multiple processors, one would of course want to have a multi-processor whose architecture is designed for this purpose, for example, using shared-memory to speed up inter-processor

communications. If the available hardware is a network of workstations, the viewpoint is rather different: as the application model size increases relative to the resources of one workstation, there may not be enough memory to store the entire model for example, and distributed simulation becomes attractive. When speed up is not the primary concern, in this sense a general-purpose distributed computer system will be a feasible choice.

7.4 Future Research and Extensions

Although most of the methodology and algorithms discussed in this dissertation were developed for models of any type of communication networks, some solutions were simplified by taking advantage of the special properties in ring network models. These algorithms can be enhanced. For example, the model partitioning method can be extended for general distributed simulation of communication networks.

As support for high-level distributed processing features evolves in distributed computer systems, the performance of our simulator can be measured and optimized with respect to different alternatives (e.g., state save frequency, allocation schemes, communication protocols, etc.).

Since state save frequency has major impacts on the performance of a distributed simulator, it would be helpful, if possible, to carry out some analysis to approach an optimal frequency, under various conditions. This is expected to be a difficult problem, but at least some study with further results would be helpful.

It has been determined that inter-processor communication introduces very large overheads in distributed simulation. Of course, there is inherently much communication inside a network model. When the model is partitioned, this becomes inter-processor communication, and at this point, no model partitioning scheme can totally avoid this dilemma. A potential alternative is to divide the model into weakly interacting parts, if possible, and assign them to different processors. Furthermore, inter-processor communication can also be reduced by replacing actual communication with probabilistic inter-processor interactions. The coupling probabilities may or may not be updated at run time. Some result accuracy would be compromised for simulator performance. The feasibility and merit of this and other approaches is an interesting topic for further studies.

CHAPTER 2

SYNCHRONIZATION METHODS

In conventional discrete event simulation on one processor, the events are sorted into an event list according to the simulation times these events are scheduled to take place. The event which has the smallest simulation time is always simulated next, and new events generated are inserted into the list, also according to their simulation times. In distributed simulation, this is not necessarily the case. Instead, the simulation operations should be assigned to the processors in a way that execution can be carried out in parallel. Events may or may not be processed according to their respective simulation times. There are currently two approaches to distribute the operations: (1) Function partitioning: assign different functions, e.g., random number generation, event processing, statistics collection, data input/output, etc. to different processors [Shep85, Wyat85] and (2) Model partitioning: partition the model, and each portion is simulated by a processor. Each processor will have its own event list. The second option can be further classified into a number of different methods [Brya77, Peac79, Chan79, Chan81, Misr86, Kris85, Jeff85a].

Function partitioning has the advantage of not involving the event synchronization problem. Since all of the events are simulated by one processor, they will be simulated in the same order as the corresponding events occur in reality so that the events are automatically synchronized.

Unfortunately, function partitioning can only exploit a limited amount of concurrency because functions in simulation can usually be classified into just a few types in this manner. Hence only a limited number of processors can be used to carry out the distributed simulation. However, it should be understood that function partitioning can be combined with model partitioning, which will be discussed in detail, to fully exploit all existing concurrency. This combination is attractive when a lot of processors are available relative to the size of the simulation model (so that additional processors will no longer improve the execution speed when using the model partitioning method only), and optimization of execution speed is an important goal.

When the model is partitioned and assigned to different processors, the processors should cooperate with one another such that events on different processors will be simulated in a way to reflect the correct situation in reality. A number of synchronization methods have been proposed; they can be classified into the following categories:

I. Synchronous Methods

   a. Central controller with fixed time increments

   b. Central controller with variable time increments

   c. The Virtual Ring method

II. Asynchronous methods

   a. The Time Packet method

b. The Link Time method

c. The Null Message method

d. The Deadlock/Recovery method

e. The Time Warp method

A good synchronization technique should have the following characteristics: (1) as much distributed control as possible, (2) maximize parallelism, reduce sequential processing, and (3) minimize inter-processor messages, especially messages to a centralized location.

2.1 Synchronous Methods

In a distributed environment, discrete event simulation may be carried out either synchronously or asynchronously. When the processors are synchronized, that is, the simulation times of different processors progress at the same pace, the events will be simulated in the same order as their counterparts in the real world, just like conventional discrete event simulation on one processor. Hence there will not be any synchronization (preemption) problems. The synchronization among processors may be carried out either in a centralized or distributed manner. If a central controller is used, simulation time may be advanced in either fixed increments (time driven) or variable increments (event driven).

2.1.1 Fixed Time Increments

When the time increment is fixed, the central controller will send a control message to each processor at the beginning of every time period. Upon receiving this message, each processor should determine whether it has events scheduled to be simulated in this period and, if so, carry them out. After the simulations for that period are completed, each processor will return a message to the controller signaling that it is ready to begin a new period. When the controller collects all termination control messages, it will start a new period by advancing its clock and sending control messages again.

There are two major drawbacks of the Fixed Time Increment method. Since control messages have to be sent in both directions in each time increment, the central controller can easily become a bottleneck, especially when the number of processors is large. In addition to that, the size of the time increment is very difficult to select. In each period, it is necessary to consider the worst case and wait until the last processor to complete its simulations, hence introducing a lot of processor idleness. When a large increment is used, more than one event might have to be simulated in each period. If the processors need to send messages to one another and generate new events which should also be simulated in the same period, the events could be simulated out of order and the simulation results will be incorrect. This problem can be avoided if the time increment is small enough so that each processor needs to simulate at most one event in each period. Unfortunately, this also implies that many more control messages will be necessary due to the

increase of time periods. Worst of all, since simulation events generally appear at random times, when the increment is small, it is likely that in many periods, there are no events to simulate and the processors will just remain idle.

2.1.2 Variable Time Increments

An alternative is the event driven approach. Like conventional simulation, the event which has the smallest simulation time has to be determined, in this case, in a two-level manner: each processor sorts its own list and then reports its local minimum to the central controller. The controller will determine which event has the smallest simulation time (among all processes) and initiate its simulation. The main problem with this approach is that a significant portion of the model's inherent concurrency is lost because the event which has the smallest simulation time in the global sense has to be determined sequentially although the simulation itself is carried out distributedly. Moreover, there will be a lot of processor idleness because processors also need to consider the worst case to guarantee that events will be simulated in the correct order.

2.1.3 The Virtual Ring Method

The Virtual Ring synchronization algorithm developed by Peacock, Wong and Manning [Peac79] is an event-driven synchronous method. Event synchronization is carried out in a distributed manner such that no central controller will become a potential bottleneck. The processors are mapped on a virtual ring on which synchronization messages are sent. Each processor

should keep track of the minimum event time among messages in its event list. The rank of a processor is defined to be the number of processors whose minimum event time is smaller than its own minimum event time. If the rank of a processor is 0, the first event on its event list is the earliest event in the entire simulation at that point and therefore should be simulated. After that event has been processed, a control message will be sent around the virtual ring to re-synchronize the processors; that is, to update the rank information.

There is a problem in re-synchronization. For example, processor A just finishes simulating an event and sends a message to B. If the time stamp of this message is smaller than the event time of the first event in B, it will generate a new event which becomes the first event in B. If the rank of B is reevaluated before this message arrives, the new rank of B (as well as the new rank of other processors) might be incorrect. As a result, events could be simulated in the wrong order which affects the correctness of the simulation. Moreover, since the rank of a processor is determined in a sequential manner, a large amount of inherent concurrency will be lost.

2.2 Asynchronous Methods

The main advantage of synchronous synchronization methods is their similarity to the real world situations. However, it seems to be inevitable that a significant amount of concurrency has to be lost because of the need to carry out some kind of sequential sorting. When the processors are not synchronized, ideally, they may simulate forward as long as there are events to be processed, regardless what the progresses of the other processors may be. Since each

processor maintains its own event list, available events in a processor will be simulated in the correct order. Events on different processors, however, could be simulated in a different order from the way the corresponding events occur in the real system. Since event dependencies are often of partial order, this change in order does not always affect the correctness of the simulation. In fact, it can potentially exploit even more concurrency in the model than what is available in the real system. However, the correctness of the simulation will be violated when a *preemption* occurs. This problem will be discussed in the following sub-section.

2.2.1 Object Oriented Simulation

In asynchronous distributed simulation, a convenient way to specify a system is to represent the model as a collection of objects. Each object represents a portion of the model. These objects interact with one another through sending messages. Most of these messages are event messages which indicate the completion of events in the simulation. They travel along arcs connecting the objects. Others are control messages needed to maintain the correctness of the simulation. The number of objects in a realistic model is in general (much) larger than the number of processors. It is therefore necessary to map several objects to each processor in a way to optimize the execution of the simulation. This problem will be discussed in Chapter 4.

Each object and every event message must carry a time stamp which is necessary to synchronize the events. The time stamp of an object is the simulation time when the most recent event in that object terminated.

Sometimes, a message is sent at that point and its time stamp will have the same value as that of the object. (We are assuming that there is no delay when a message is sent from one object to another.) This time stamp should be approximately equal to the time when the corresponding event occurs in the actual system. However, since the objects are not synchronized, some of them will be ahead and some will be behind in simulation time. Therefore the order of arrival among messages from different inputs to an object is not always the same as that in the actual system. This creates the message synchronization problem in asynchronous distributed simulation. Some inputs with "earlier" (smaller) time stamps may be received by an object after it has seen inputs with "later" (greater) time stamps; i.e., messages with earlier time stamps *preempt* messages with later time stamps. This situation is shown in Figure 2.1 where a message from object B arrives object A before a message from C, but t_B, the time stamp of the message from B, is greater (later) than t_C. If all input messages were consumed simply in the order received, the progress of the simulation might become incorrect in relation to what would have taken place in the real world, and the simulation results would be wrong and useless.

2.2.2 Synchronization Methods Which Prevent Preemptions

In the late 1970's and the early 1980's, a number of methods such as Time Packet [Brya77], Link Time [Peac79], Null Message [Chan79], Deadlock/Recovery [Chan81] and Safe Forward Simulation Time [Kris85] were proposed to solve this preemption problem. These methods were later considered to be *conservative methods* by Jefferson and Sowizral [Jeff83]

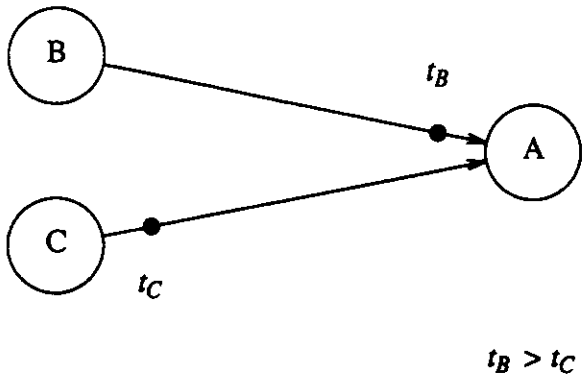


Figure 2.1: Preemption

because they attempt to prevent preemptions in the expense of reducing concurrency and increasing processor idleness. †

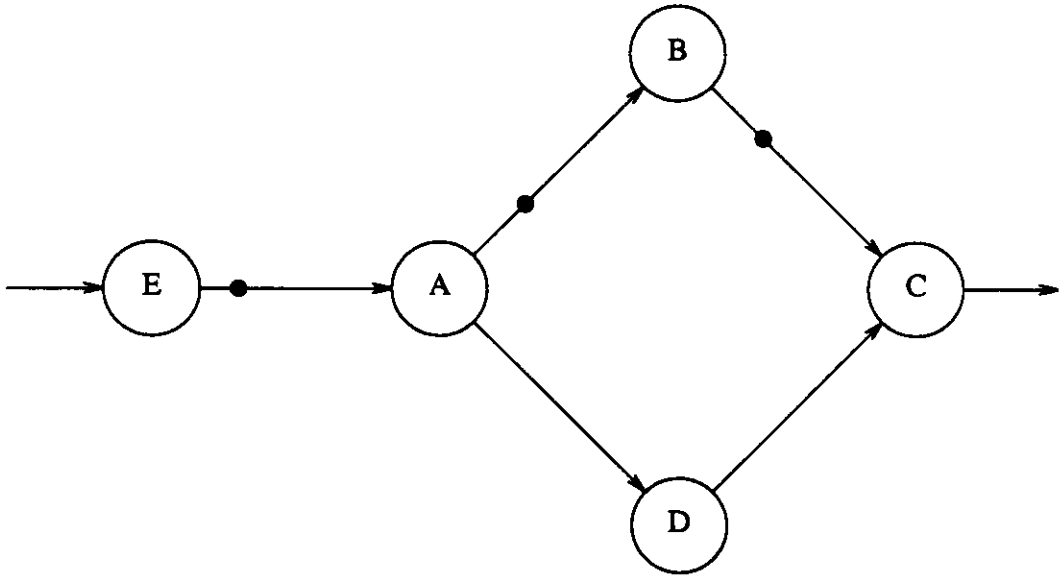
The following requirements guarantee that input messages will be processed in increasing time stamp order and hence the correctness of the simulation:

1. The order of the messages going from one object to another through an arc must be the same as that of the corresponding messages in the actual system. That is, a message must have a larger time stamp value than its predecessor. (Messages coming from different arcs, however, may arrive in a different order.)

2. If there are more than one incoming arc to an object, there must be at least one message at each arc so that the most imminent message can be identified and then processed. Otherwise, the object will be considered

† The Safe Forward Simulation Time method was introduced after Jefferson and Sowizral had classified the conservative methods, but it belongs to that category.

blocked and none of the available messages should be consumed.



Assume that each arc may hold at most one message.

Figure 2.2: A Deadlock

Although these two requirements are sufficient (but not necessary) to guarantee the absence of preemption, there are situations where they cannot be applied. For example, if there are a number of incoming arcs to a certain object, there is not always at least a message available at each arc. When one or more input arcs are empty, the object is considered blocked. The simulation of a blocked object has to be suspended until messages arrive to unblock it. If one or more arcs rarely have any incoming messages, the object will frequently be blocked. This is certainly not a desirable situation. Moreover, if there are no messages whatsoever from a particular arc, the object will be permanently blocked and hence deadlocks. Figure 2.2 shows an example of deadlock where each arc has finite storage and is allowed to contain at most one message.

Object C cannot consume the message from B because it could be preempted by a possible future message from D. B cannot consume the message from A because its output arc is occupied. For the same reason, A cannot consume the message from E. As a result, no messages can be sent from A to D and then to C, and C will never be able to consume the message from B. Several of the conservative methods discussed in the following sections attempt to resolve the deadlock problem by artificially introducing additional messages to unblock objects. Others search for additional information to replace the second requirement when it cannot be fulfilled.

2.2.2.1 The Time Packet Method

Bryant [Brya77] proposed the *time packet* method. Time packets are messages sent from one object to another along the arcs but carry no information other than a time stamp. The value of this time stamp, *tout*, is evaluated in following manner:

$$tout = (\min_{1 \le k \le n} tlast_k) + delay$$

where $tlast_k$ is the time stamp of the last message from input arc k

and *delay* is the processing delay of the object.

That is, compare the time stamps of the most recent message from each arc, *tout* is the minimum among these time stamps plus the processing delay of the object. Since consecutive messages on an arc are still required to be in increasing time stamp order, no message with a time stamp smaller than $\min_{1 \le k \le n} tlast_k$ can arrive the object in the future. Hence *tout* is a lower bound for

the time stamp of the next message leaving the object. Each time a new message arrives an object from input arc k, $tlast_k$ should be updated. This message may be consumed if it satisfies the two rules discussed in the previous section, and a new output message will be sent from the object. Otherwise, the object is blocked; *tout* should be updated and a time packet with the new *tout* will be sent to all destination objects. Deadlocks are prevented because the time packets serve as additional messages to unblock objects. The correctness of this method has been proved [Brya77]. However, there are two major problem with it. First of all, a large number of time packets are needed to maintain the synchronization of the objects, especially if the arcs which connect the objects together form cycles. Moreover, the way these time packets are generated reduces the amount of currency in the simulation. Some of the events which can be simulated in parallel will be forced to be simulated sequentially due to the additional dependency introduced by the time packets.

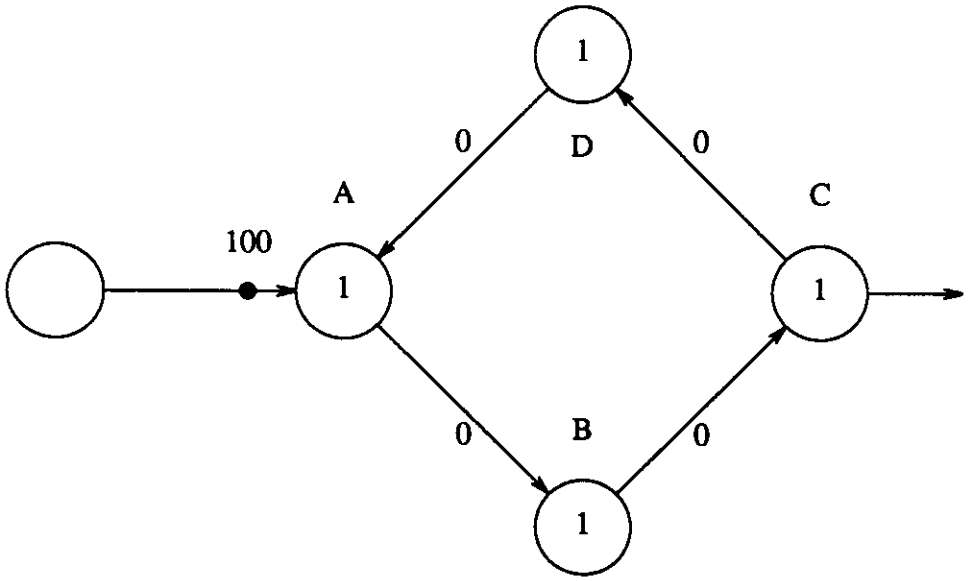
2.2.2.2 The Link Time Algorithm

Peacock, Wong and Manning also proposed the *link time* algorithm [Peac79]. This algorithm is closely related to the time packet method described in the previous section. The link time of an arc is the lower bound for the arrival time of the next message on that arc. When there are messages on a link, the link time of an arc is simply the same as the time stamp of the first message. When there is no message, the source object should evaluate this lower bound by finding the smallest link time among its input arcs and add to it the delay induced by that object.

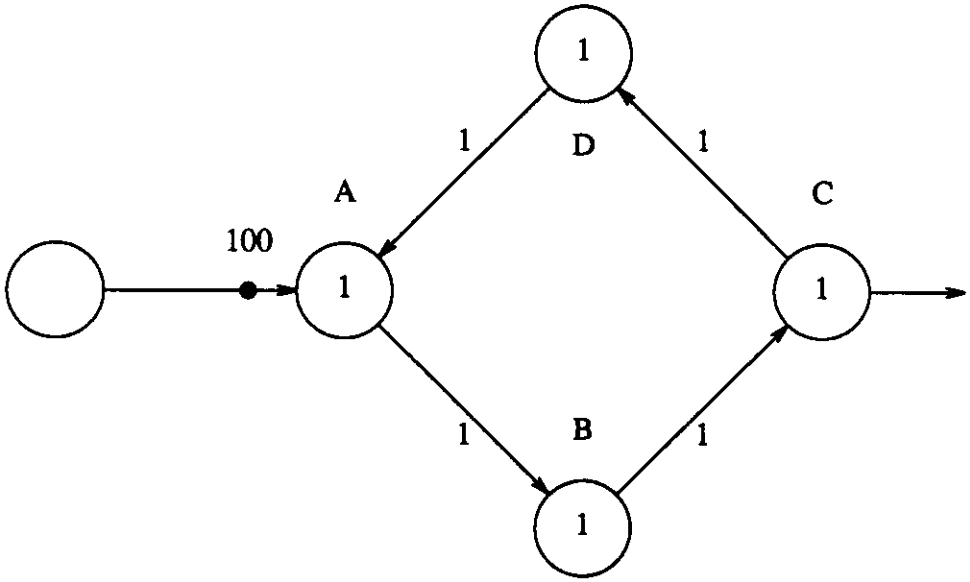
If the arc which has the smallest link time contains messages, the first one may be consumed and the corresponding event will be simulated. If that arc is empty, the next message which has the smallest time stamp might not have arrived. In order to avoid any possible preemptions, this object should be considered blocked, and none of the available messages may be processed. A blocked object becomes unblocked only after a message arrival such that the input arc which has the smallest link time contains a message.

If a series of blocked objects form a certain pattern, it may result in a deadlock; but it can be proved that a deadlock occurs if and only if there is a cycle of empty arcs which have the same link time. Therefore, deadlocks can be prevented if the delay induced by an object is always greater than zero because the link time of any output arc of an object will always be greater than that of any of the input arcs. Hence it is impossible for a cycle of empty arcs to have the same link time.

Although it is deadlock free, this algorithm may still be highly inefficient when the arcs form a cycle. An example is given in Figure 2.3. Assume the delays in objects A, B, C, and D are all 1. Initially, the link times are 0 as shown in Figure 2.3 a. A message with time stamp 100 is available at an input arc to A. Object A is blocked because input arc $\overline{DA}$ is empty; After the first link time update, the new link time of A's output arc $\overline{AB}$ is 1 (0, the minimum input link time, plus 1, the object delay). Actually, following the first round of updates, the link times are now all 1 as shown in Figure 2.3 b. Unfortunately, this update has to be repeated 100 times such that all link times become 100



(a)



(b)

Figure 2.3: A cycle in the link time algorithm

before the new message going to A can be processed.

2.2.2.3 The Null Message Method

Chandy and Misra [Chan79, Seet79] suggested the use of *null messages* to solve the deadlock problem. A null message contains a time stamp but no other information. When an object with multiple output arcs sends an output message through one of the arcs, it should send a null message (with the same time stamp) to each one of the remaining output arcs. Like a time packet, a null message is used to indicate that no message with a smaller time stamp will pass through that arc. The main difference between the two is that time packets are generated when a message arrives an object, but null messages are generated when a message leaves an object.

The major problem with the null message method is that one or more null messages will be generated whenever a message leaves an object with two or more output arcs. This is necessary even though the leaving message is null itself. When there are cycles in the objects, the number of null messages can grow very quickly, leading to a significant amount of processing overhead. Experimental work by Seethalakshmi [Seet79] suggests that this method is an expensive synchronization technique for distributed simulation due to the large number of null messages, but it performs well when the arcs do not form cycles. (Although the number of null messages will increase rapidly when there are cycles among the objects, it will not grow indefinitely if each arc is allowed to store multiple messages. Since the main function of null messages is to provide a lower bound for the time stamp of the next real message arriving

from an arc, all null messages which are waiting in a queue but are not the last element of that queue may be removed. The correctness of the simulation is not affected because the last message, which has the largest (latest) time stamp, always defines the highest lower bound.)

2.2.2.4 The Deadlock/Recovery Method

Chandy and Misra [Chan81] subsequently introduced the Deadlock/Recovery method to avoid the null message overheads. Unlike the other asynchronous methods discussed in the previous sections, it permits the simulation to run into deadlocks instead of generating additional messages to avoid them. The simulator executes a deadlock detection mechanism in parallel to the actual simulation. When a deadlock is detected, the simulation will be halted. A deadlock is resolved in the following manner. Since a preemption occurs when a message with a smaller time stamp arrives an object with a larger time stamp, the event which has the smallest time stamp in the entire simulation cannot be preempted. Hence this event can be simulated without the possibility of generating any incorrect results and the deadlock is resolved. The simulation will continue until the next deadlock occurs and the process repeats. Chandy and Misra suggested that this method may be efficient although occasionally it needs to recover from deadlocks. Some experimental work is needed to demonstrate its efficiency. As pointed out by Jefferson and Sowizral [Jeff83], after a deadlock recovery, the simulation could be in a near-deadlock state. It probably will become deadlocked again very easily. Hence it is not clear that Deadlock/Recovery is an efficient synchronization method.

2.2.2.5 The Safe Forward Simulation Time Method

The Safe Forward Simulation Time Method [Kris85] may be considered to be the most recent addition to the "conservative" synchronization methods. It is similar to the Link Time Method and was developed as an alternative to the function partitioning method. Like other conservative methods, it requires that messages through an arc to be in increasing time stamp order. Each object maintains a local clock time which is the end of the most recent busy period. Since messages must arrive in increasing time stamp order, because of causality, an object cannot send a message with a time stamp smaller than the current local clock time in the future. Each object also maintains a safe forward simulation time (SFST) which is the minimum among the local clock times of its predecessors. That is, an object cannot receive in the future any message less than or equal to the current SFST. Although some inputs may be empty, a process is not considered blocked as long as there are available input messages with time stamps less than or equal to the current SFST because these messages cannot be preempted. However, if all available messages have time stamps greater than the SFST, the SFST must be advanced before any one of them can be safely processed. In this case, A process will send an awakening signal to its predecessor which has the smallest local clock time and request for a local clock time update. If necessary, this predecessor will subsequently send awakening signals to its own predecessors and so on. This process will continue until the SFST of the process which initiated the requests gets updated. When the awakening signals form a cycle, however, the update process would continue forever. The process which initiates the update must be able to detect

the cycle and will simply update its SFST without considering the predecessor which forms the cycle.

The advantages of the Safe Forward Simulation Time method are: (1) It avoids unnecessary blocking and (2) it introduces relatively few synchronization messages in the simulation because SFST updates are carried out only when necessary. However, the overhead per update is considerable because the need for update has to be determined by a descendant, and then a request is sent to the predecessor. Upon receiving this request, the predecessor will determine the new value and send it to the descendant before simulation can progress again. The overall overhead will be increased significantly if processes need to awaken their predecessors frequently. The situation will become worse if each update involves sequently backtracking through many stages of predecessors before a new value is obtained or a cycle is detected. The performance of this synchronization method is yet to be demonstrated experimentally.

2.2.3 The Time Warp Method

From the discussion in the previous sections, we can conclude that the main synchronization problem in asynchronous distributed simulation is preemption. To guarantee the absence of preemptions, an object is considered blocked and its simulation is suspended until it is certain that no preemptions will occur and it is "safe" to continue. This is a waste of computation time and lost of concurrency because objects may frequently be unnecessarily blocked. (Recall that the two conditions in Section 2.2.2 are sufficient but by no means

necessary for the absence of preemptions.) As a matter of fact, a message which arrives an object earlier would usually have a smaller time stamp than another message which arrives later, because computation load should be assigned to the processors in a way such that the simulation times on the processors, although not synchronized, should progress in approximately the same pace. (Simulation load assignment will be discussed in Chapter 4.) Therefore, blocking is really an expensive price to pay in order to avoid a problem which should not occur very often in the first place. Moreover, a cycle of blocked objects will cause the simulation to deadlock. As a result, various synchronization methods introduce additional time messages to unblock the objects. Because a large number of these messages is often generated, the overhead involved is not acceptable.

2.2.3.1 State Saves and Recovery

Since none of these synchronization method is really satisfactory, Jefferson and Sowizral [Jeff83, Jeff85a] developed the *Time Warp* method which solves the fundamental problem, namely preemption, by a different approach. As a result, there will not even be any blocking or deadlock problems. In Time Warp, an object (or a processor) simply simulates forward as long as there are events available, regardless of whether any possible preemptions will occur in the future. Hence an object might run out of events but will never be blocked, and the maximum amount of concurrency can be exploited. Furthermore, messages from an arc are not even required to have increasing time stamp values. To resolve the preemption problem, Time Warp

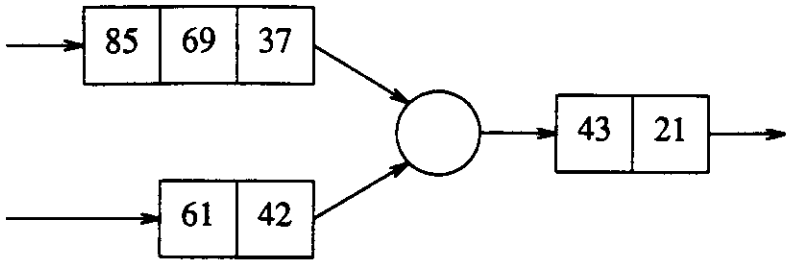
periodically saves the state of objects. When a preemption does occur, the object being preempted will *roll back* to (recover) an old state which has a time stamp earlier than that of the preempting message. Effectively, the object discards the simulation it carried out since that state was saved and pretends as if it had never proceeded from that point. As a result, the message will no longer be preempting and becomes only a regular message.

An object which rolls back might have sent messages to other objects during the part of the simulation it discards. These messages might have provided the destination objects incorrect informations. To resolve this problem, in Time Warp, an object remembers the messages it has delivered. As part of the roll back process, it will send *anti-messages* to cancel the messages which should not have been delivered before. An anti-message is basically an identical copy of the original message, but it carries a negative sign bit (instead of a positive sign bit in the original "positive" message). When an object receives an anti-message, it will first compare the message's time stamp to its own. If the time stamp of the anti-message is larger, no roll back is necessary. If the anti-message has a smaller time stamp, this object will also need to roll back since it has possibly incorrectly processed the corresponding positive message. (We consider this a secondary roll back because it is caused by another roll back.) By rolling back, this object effectively unprocesses a number of messages so that the desired positive message will appear in the object's input queue again. Once both the positive and negative (anti) copies of a message are available in the input queue, they will simply be deleted together. It will appear to the simulation that no such message ever existed. It is also

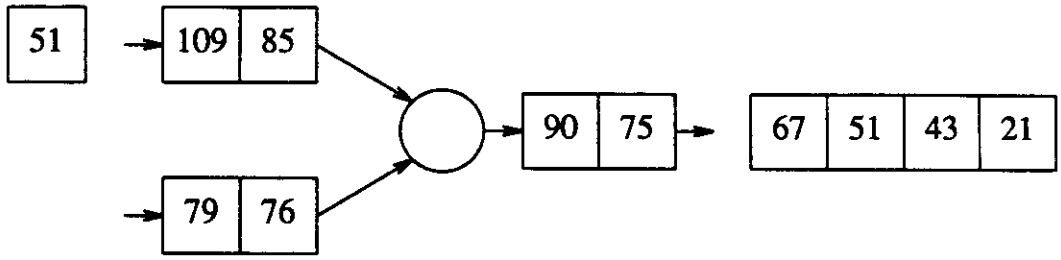
possible that the negative copy of a message actually arrives the destination object first, if alternative paths are available. In that case, the anti-message should be saved in a buffer. When the positive copy finally arrives, both will be deleted.

Figure 2.4 is an example of roll back. There are two inputs to and one output from an object. Each input and output has a queue, and every message in a queue has a time stamp. The state description in Figure 2.4(a) is saved in the simulation. After processing two messages from each input queue and receiving some new arrivals, the new state description is shown in Figure 2.4(b). Messages with time stamps 21, 43, 51 and 67 have been sent and therefore left the output queue. If a new message with time stamp 51 now arrives from the upper input, it will preempt the object and cause a roll back. The saved state in (a) should be recovered. Anti-messages will be sent to cancel the messages the object delivered after the restored state was saved; e.g., output messages with time stamps 21, 43, 51, and 67 belong to this category. The after-roll-back state description is shown in Figure 2.4(c), where an anti-message is represented by a box with the time stamp crossed out.

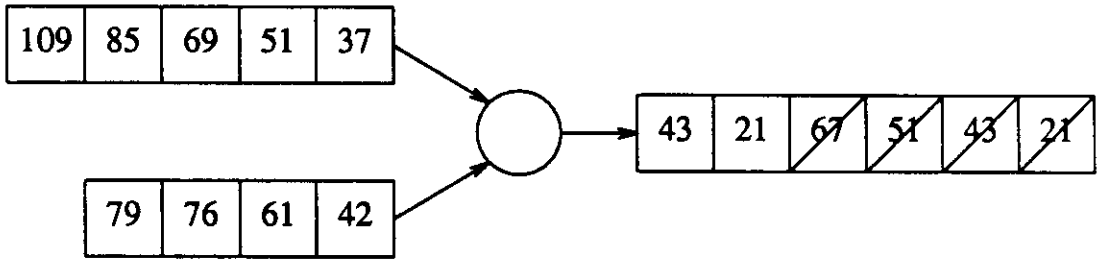
Compared to methods which prevent preemptions, the nature of Time Warp is optimistic because it attempts to simulate forward as far as possible hoping that no preemption will occur, rather then conservatively worrying that there will be preemptions and remaining idle until it is safe to continue. Although it may seem to be highly cumbersome to simulate forward and roll back, send messages and then cancel them, Time Warp is actually quite



(a) A state which is saved.



(b) Later in the simulation, a preemption occurs.



(c) The saved state is recovered.

Figure 2.4: State save and roll back

efficient. It should be understood that a primary roll back is necessary only when a preemption occurs. An object will simply roll back to a state at which it would have been blocked if Chandy and Misra's method had been used. Hence the amount of processor time discarded would be the same as the processor idle time in the other case. (Practically, Time Warp does suffer more overhead than this ideal case when not every state is saved.) In a sense, Time Warp takes a chance and simulates forward without complete information of the other objects. If it is lucky, it will gain some simulation speed. Otherwise, it really does not have much to lose. Again, since we do not expect preemptions to occur frequently, Time Warp should be able to gain this advantage most of the time.

The real computation time overheads in Time Warp are saving the state descriptions and rolling back to old states. When a state is saved, the entire state description of an object should be copied. This might require a considerable amount of memory if the state description is complicated but do not take very much computation time. In a roll back, an object should search its history to determine the latest saved state which resolves the preemption and then recover that state. (An object may roll further back to it past than necessary without affecting the correctness of the simulation. However, it will have to simulate forward again for a longer period of time and cause more secondary roll backs. This is certainly undesirable as far as efficiency is concerned.) A sequential search from the most recent saved state may be employed based on the assumption that it is not necessary to roll very far back into the past. This type of search and recovery scheme also should not be very

computation time consuming. Memory usage, however, could be a potential drawback in Time Warp. As mentioned before, when the state description is complicated, each state save will require a lot of memory. If the history contains a large number of states, memory size will become a problem. The concept of *Global Virtual Time* (GVT) is introduced as a measure of the progress of the simulation and an indication of which part of the history is no longer needed so that memory space can be recycled for future use. Furthermore, it turns out that when the simulation model is a queueing network, the state description will be queues and nodes. It is unnecessary to copy the content of every queue for each state save. This problem will be discussed in detail in Chapter 3.

2.2.3.2 The Global Virtual Time

The progress of the simulation at an object is measured by its simulation time, or *Local Virtual Time* (LVT). The concept of virtual time is explained in detail by Jefferson [Jeff85b]. The LVT of an object is only the apparent progress of that object. It is not an indication of guaranteed progress because an object can be preempted and roll back to an earlier state with a smaller LVT. The real progress is measured by the Global Virtual Time, which is defined to be the smallest time stamp among all objects and messages in-transit in a snap shot of the entire simulation. GVT advances when the object with the smallest time stamp processes an event or the message with the smallest time stamp gets consumed by an object. Hence GVT is a non-decreasing function of real time, and a distributed simulation is guaranteed to progress. Moreover, as mentioned

before, a roll back occurs when a preemption appears, and a preemption is caused by a message with a smaller time stamp arriving an object after another message with a larger time stamp. Hence the message/object which has the smallest time stamp cannot be preempted, and no roll back will need to recover a state saved before the current GVT. (More precisely, the last state saved before the current GVT is still necessary, but any earlier state is not.) Therefore GVT can also be used to indicate which saved state is no longer needed. This concept is very important for memory management of saved states in Time Warp.

2.3 Conclusions

Because of its potential to exploit inherent concurrency in simulation models as well as computation power of distributed computer systems, the model partitioning method is very suitable for assigning computation loads to the processors. Since event dependency are of partial order in general, it is possible to exploit more concurrency by permitting the simulation at different processors to progress in an asynchronous manner. However, because of causality, the simulation on different processors cannot be completely independent from one another. Maintaining synchronization among processors is a topic which attracts much current research interests. In this chapter, we presented a survey of several existing synchronization methods and their individual advantages and disadvantages. We selected the roll back scheme in the Time Warp method as the basis to solve the preemption problem in our simulator because it can potentially exploit the maximum amount of inherent

concurrency, introduces not very many additional control messages, and does not require any synchronous communication with a central controller. The other asynchronous methods are considered to be less efficient than the selected method because they, by one way or another, introduce too many additional messages to maintain synchronization among the processors.

CHAPTER 3

SPECIALIZED ALGORITHMS

3.1 Simulation Model Specification

Before any simulation can begin, an application model must be specified in a computer-readable format. In order to take full advantage of the potential processing power of a distributed computer system, significant portions of any inherent concurrency in the model must be preserved in the specification. In a distributed environment, it is difficult to make direct use of special-purpose simulation languages, since they are typically designed to be translated into a procedural language and executed on a single processor. We have therefore chosen to develop an interactive module which helps users to define the simulation model and the simulation software modules in a general-purpose language, namely C, and to implement the distributed features of our overall simulation software package by using the facilities of LOCUS [Pope85], a distributed extension of UNIX.

3.2 Problems with Model Partitioning

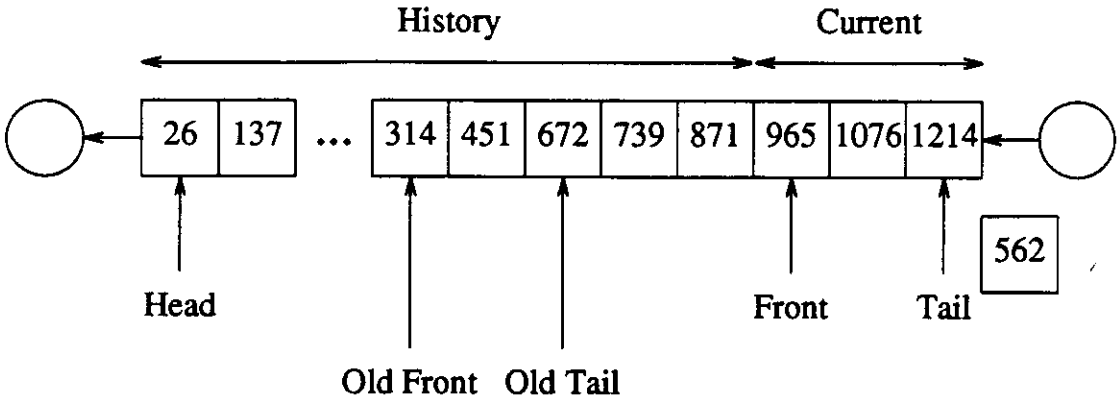
The rule of diminishing return applies to distributed simulation with model partitioning. Even an infinite number of processors is available, as far as execution speed is concerned, it is not always an advantage to partition the model into finer pieces and add more processors to the distributed simulator.

When tightly coupled components are assigned to different processors, the interactions among these processors will introduce too much overhead which counteracts the advantages of distributed simulation. Physical boundaries of components in a model frequently serve as a good guideline for partitioning. For example, when the network in Figure 1.1 is used as a benchmark, and each network node is modeled as a switch with a few queues, the close interaction among the switch and the queues suggests that at least each node should be considered as an object. Any further division might increase the overhead. However, if the precise performance of a network node itself is of interest, for example, the details of the node architecture should be included in the model. In that case, each node can be modeled as a number of objects, and probably assigned to several processors. Model partitioning and allocation will be discussed in detail in Chapter 4.

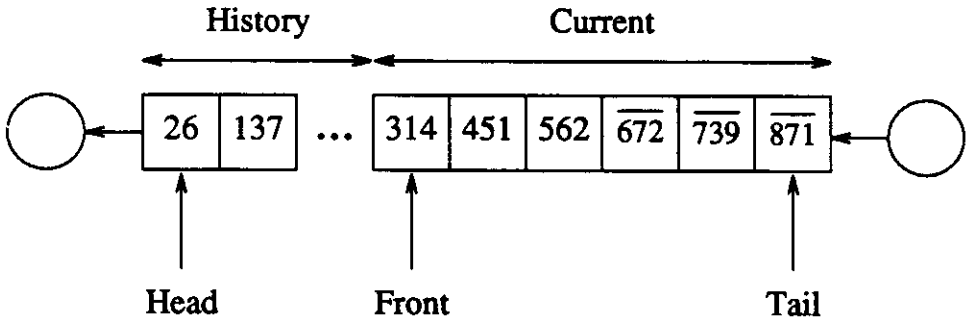
3.3 Queueing Network Models

The purpose of this work is to study distributed simulation techniques; a complicated and precise model for the benchmark is unnecessary. Therefore, each network node may be represented as a switch and a few queues. A ring network such as our benchmark can be conveniently represented by a queueing network model. This type of models has an important advantage when the state description needs to be saved periodically. Instead of copying the entire queue for each state saved, the current queue and the saved history can be combined into a very "long" queue, and only pointers to some representative queue elements should be saved every time.

3.3.1 State Saves and Recoveries of a Queue



(a) Queue representation when a preemption occurs



(b) Queue representation after roll back

Figure 3.1: Queue representation and saved states

The model of a queue is shown in Figure 3.1(a). A queue is implemented as a linked list. Queue elements are in increasing time stamp order. In addition to a time stamp as shown, each element contains some other information such as the sender and receiver i.d., etc. A pair of pointers, *q_front* and *q_tail*, defines the "current" part of the queue; *q_front* points to the next element to leave the queue, and *q_tail* points to the last element entered the

queue. If the queue is currently empty, *q_front* will be null to indicate that there is no next element available for departure; *q_tail* should never be null except for at the very beginning of the simulation. For example, in Figure 3.1(a), the queue currently has three elements with time stamps 965, 1076 and 1214 respectively. These are the elements in the queue of the actual system at the corresponding time. When an element leaves the queue in the simulation, *q_front* will point to the next element, and the linked list is not actually altered. The "departed" element automatically becomes part of the saved history. In Figure 3.1(a), elements with time stamps 26, 137, ... 314, 451, ... 871 are all previous elements which have "left" the queue. To save a state, as mentioned before, it is not necessary to save a copy of the entire queue. Only the current *q_front* and *q_tail* pair should be saved because they are sufficient to define the content of the queue in a state. An example is the saved state pointer pairs, *old q_front* and *old q_tail*. When this state was saved, the then current queue had three elements with time stamps 314, 451 and 672 respectively. When the element with time stamp 562 joins the queue as shown in Figure 3.1(b), since it has a time stamp less than that of the last element departed (i.e., 871), a roll back is necessary. The old state is recovered by replacing the *q_front* and *q_tail* pointers with the *old q_front* and *old q_tail* pair. The new element is then inserted into the queue. Because of causality, the elements with time stamps 314 and 451 cannot be affected by the straggler, but those with greater time stamps; i.e., elements 672, 739 and 871, might no longer be correct, and anti-messages should be sent to cancel them. (There is a chance that the canceled messages are actually not affected and will be

generated again. It seems to be a waste of computation time to cancel and then regenerate messages. This problem will be discussed in Section 3.6.) There is another pointer, *q_head*, which points to the very first element in the available history. It is used when removing no-longer-needed elements from the queue. The scheme described above significantly reduces memory usage because only one "version" of each queue element, represented as a long queue, is available at all times (compared to an entire copy of the current queue per state saved). Computation time is also reduced since mainly pointers are copied for each state save and recovery. Moreover, when there is a modification of the state description, every saved state is modified at the same time. There is no need to change the descriptions individually.

3.3.2 The Software Model of a Node Connected to Terminals

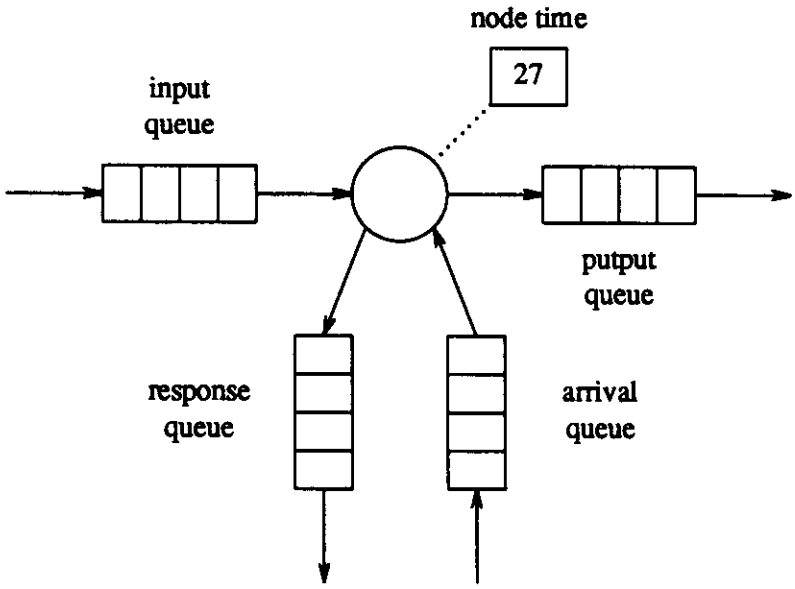


Figure 3.2: Model of terminals connected to a terminal node

Figure 3.2 shows the model of a network node which is connected to terminals in the communication network to be simulated. In a slotted-ring network node without packet priorities, there does not need to be any input buffers. A node simply checks every passing slot; if it is occupied by a packet destinated to that node, the packet will be removed. Otherwise, the slot will bypass the node. When an empty slot is available, the node may transmit a new packet. In the Doelz Network, however, there is a preemptive priority scheme. That is, a packet with higher priority waiting in a siding queue may bump another packet with lower priority off the ring into a local buffer (the input queue) and seizes its slot. The packet which loses its slot will have to wait in the buffer for a later available slot. When packets depart from a network node, they are carried away by the slots which function like a conveyor belt. Pending packets will either wait in the input queue or the arrival queue, and no output buffer is needed. The output queue is only an artificial buffer introduced in the software model so that distributed simulation can be carried out asynchronously; i.e., a sender node may output packets far ahead in real time before a receiver node accepts them (the excessive messages will be stored in the output queue in the simulation); it does not correspond to any physical device in the actual network. The arrival queue represents a siding queue which acts as buffers for input/output packets to/from connected terminals. Usually, many terminals are connected to a network node; their traffic is merged into one stream. Each network node also has a node time, which records the time stamp of the last message processed.

Although each network node can be represented as an object and assigned to a processor, since the number of network nodes in a realistic model is often much larger than the number of processors in a distributed simulator, it is usually necessary to assign several network nodes to each processor, while each node may still be considered as an independent object. In a regular Time Warp implementation, objects on the same processor interact with one another by sending messages; they can also preempt one another, just like objects assigned to different processors. The main difference is that these messages do not require any inter-processor communication so that they will arrive their destinations almost instantaneously. However, when the model is a ring network, our model partitioning algorithm suggests that adjacent network nodes on the ring should be grouped together and assigned to one processor to minimize inter-processor communication. (This problem will be discussed in Chapter 4.) Since network nodes on a ring communicate with neighboring nodes only, it is therefore possible to consider all of these nodes as one larger object. The main advantage of this grouping is that nodes assigned to the same processor will send messages to one another in an even much simpler manner. It is not necessary to save previous messages at both the sender and the receiver, and there will neither be and preemptions nor any anti-messages from one object to another inside a processor. 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 the entire object, i.e., every node assigned to the processor, to roll back together. We do not consider this as a serious disadvantage because when several objects are assigned to a processor, they are synchronized in the sense

that the one which has the earliest event will always be the next object to be processed. Therefore when the first object in tandem is preempted, it is very likely that the remaining ones will subsequently be preempted anyway.

3.3.3 The Software Model of a Node Connected to Hosts

A network node with hosts connected may be modeled as a number of parallel service centers as shown in Figure 3.3. We assume that the hosts on the ring are all connected to the same network node because computers are frequently placed together in a "machine room." If this is not the case, a bypass path should be added to the model for packets which do not interact with the hosts connected to that node. This modification will slightly complicate the model. The two switches NN_{in} and NN_{out} correspond to the same physical device, the network node, which has been bifurcated in the model for representation convenience. Every host consists of a processor, an input buffer and an output buffer, corresponding to actual elements in the network. An artificial merged queue is added to the software model to reassemble output packets leaving a node into the correct order. When a command packet arrives the network node, it will enter the input buffer of its destination host. This command will be processed when its turn arrives, and the corresponding response packets will have to wait in the host's output buffer for available outgoing slots to leave the network node. (Usually, a command is very short, probably just a few letters, but a response may be very long, for example, an entire file.) Responses from different hosts need to compete for outgoing slots in the network. We are currently assuming a FIFO/round robin allocation

scheme. That is, if only one host has response packets available, the response packets will leave the node in a FIFO manner. If several hosts have response packets available, the round robin scheme will be used.

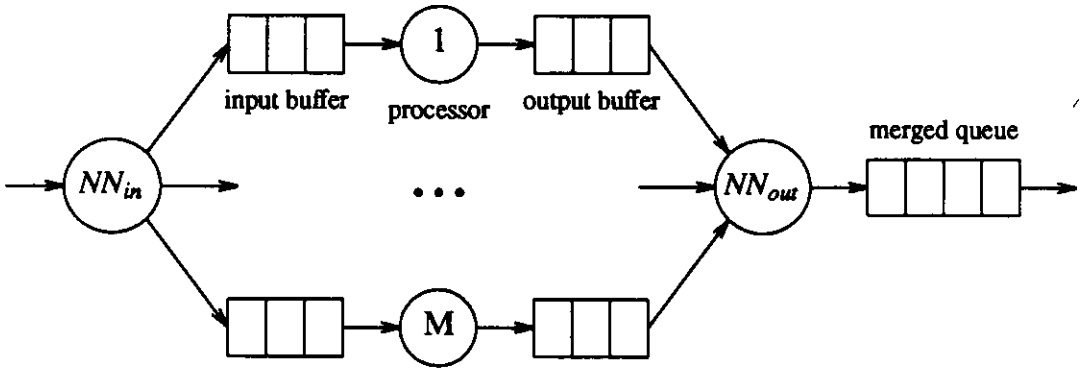


Figure 3.3: Model of Hosts Connected to a Network Node

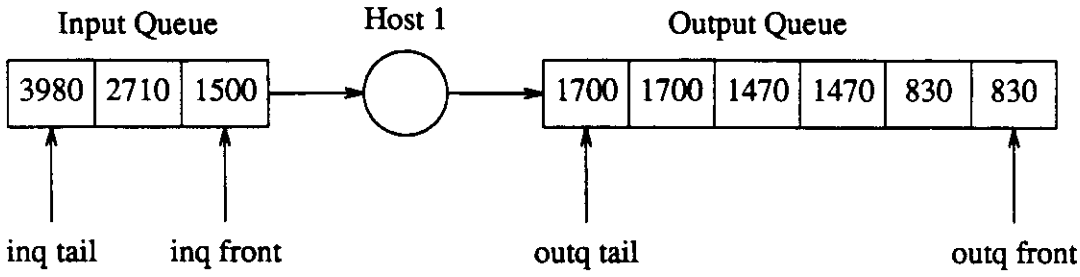
As discussed in Chapter 2, merging messages from parallel nodes is the major synchronization problem in distributed simulation. Since the outputs from the hosts are competing for outgoing slots, a response packet in an output buffer should not be assigned an outgoing slot unless it is certain that no other responses will be competing for that slot or it is this packet's turn to be transmitted. The problem is further complicated since host processing time is a random variable. For example, assume command A arrives the host network node before command B, and the destinations of A and B are hosts 1 and M respectively. Even though the processing times of the hosts have the same distribution, it is possible that the response packets for command B will be available first if the processing time for B happens to be small enough to compensate for the arrival time difference. Response packets might enter the merged queue in an incorrect order, unless, for every other host, response

packets are available so that a time stamp comparison can be made or the earliest command for the host has a larger time stamp than that of the response packets to be merged. (Since the host think time is always positive, the time stamp of a response packet must be greater than that of its corresponding command packet.)

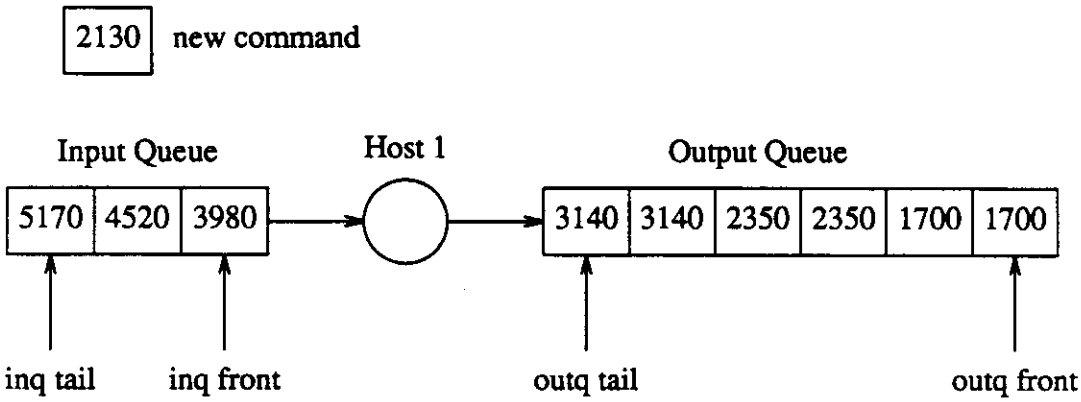
Referring to Figure 3.3, when an input message arrives (it must be a command packet since all hosts are connected to one network node), it will enter the input buffer of its destination host. At this point, this message can no longer affect the activities of the other hosts; a preemption at one host cannot cause another one to roll back. Hence state saves and roll backs of each host may be carried out independently. This is a considerable advantage for parallel nodes in general, and especially in this particular case because the generation of simulated service times requires a number of time-consuming library function calls. However, after a host rolls back output packets from the hosts will probably leave the network node at a different time and in a different order due to a possibly new situation for round robin. Hence the output packets from all of the hosts should be remerged (but not re-generated). We provide for this by keeping separate sets of saved state pointers; namely, the front and tail pointers of the merged queue and the front pointer of each host's output queue are saved periodically as a set. (A front pointer determines which element will leave an output queue and join the merged queue, but does not determine which element will join the output queue.) When one host rolls back, and a saved output queue front is recovered, a corresponding set of output queue front pointers will also have to be recovered, and response packets from different hosts will be re-

merged according to the new situation.

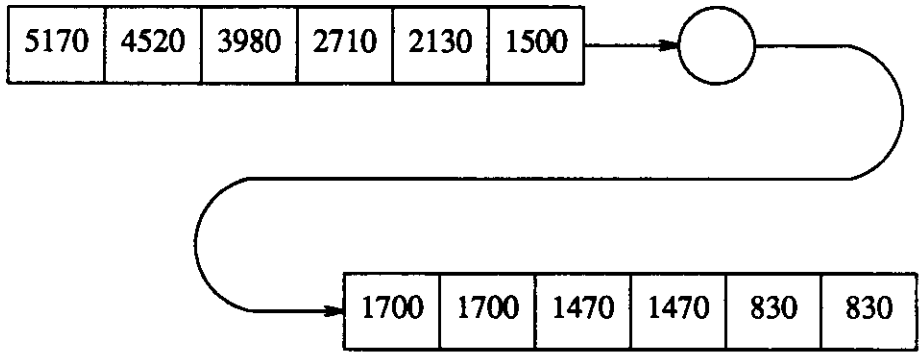
Figure 3.4 is an example of output queue roll back. The state description of Host 1 at a certain time is shown in (a). There are three commands with time stamps 1500, 2710 and 3980 respectively in the input queue to the host. Assume each response consists of two packets; there are six packets (three responses) with time stamps 830, 1470 and 1700 in the output queue. This entire state description is saved. After processing two commands and the departure of two responses, the new state description is shown in (b). If a command with time stamp 2130 now arrives, since its time stamp is smaller than that of the last command processed by the host (namely 2710), a roll back is necessary. When the old state in (a) is recovered, the old output queue pointers will be restored, and elements which were generated after the state save (i.e., those with time stamps 2350 and 3140) will be removed. Some of these removed output queue elements might be regenerated when the simulation progresses forward again. For example, the command with time stamp 1500 cannot be affected by the straggler because of causality. Hence identical response packets with time stamps 2350 will be generated as before (provided that other informations such as the random number seed are saved as part of the state). The straggler will be inserted into the appropriate position in the input queue. However, no elements in the input queue will be deleted during the roll back because their correctness is not affected by the straggler. Moreover, these commands were generated elsewhere and cannot be recreated by the rolling back object. Figure 3.4(c) is the state of the host after the roll back.



(a) First state description of Host 1

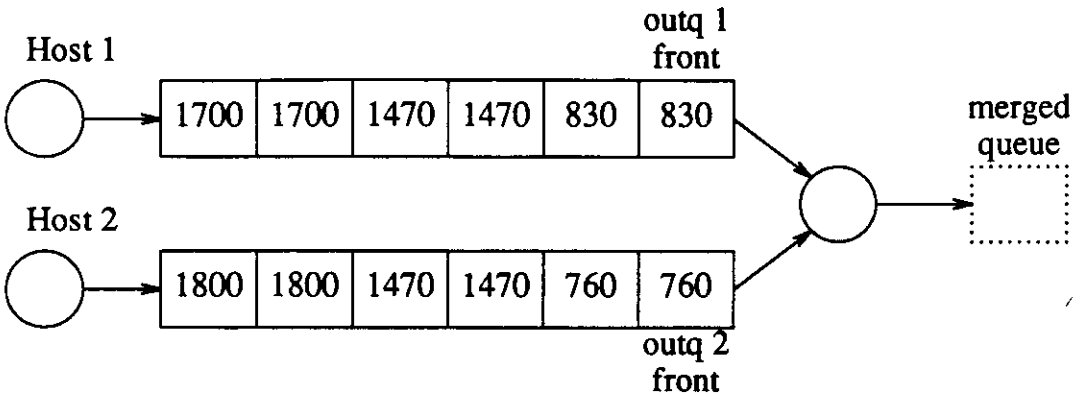


(b) Second state description of Host 1

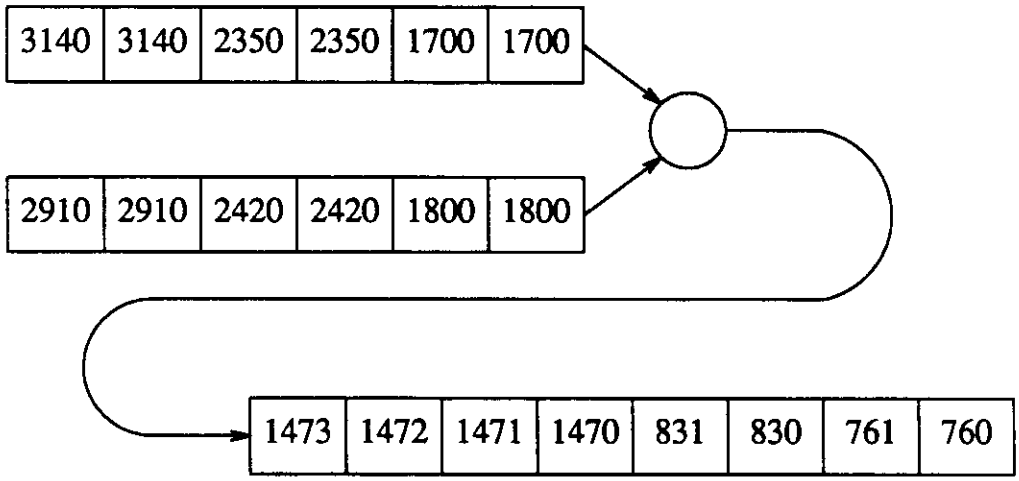


(c) State description of Host 1 after roll back

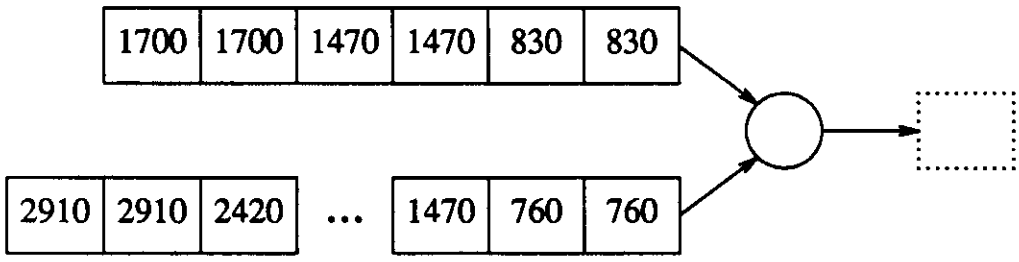
Figure 3.4: An example of host roll back



(a) First state description of the output queues and the merged queue



(b) Second state description of the output queues and the merged queue



(c) State description of the output queues and the merged queue after the roll back

After Host 1 rolls back, the merged queue pointers and host front pointers should also roll back as a set so that output packets can be merged again according to the new situation. Figure 3.5(a) and (b) indicate the relation between the output queues of Host 1 and Host 2 and the merged queue at two different times. They correspond to the situations in Figure 3.4(a) and (b) respectively. The first four elements in each output queue in Figure 3.5(a) are merged and become the eight elements in the merged queue in Figure 3.5(b). For the first pairs of elements in each output queue, they enter the merged queue in the FIFO manner. When both queues have elements available as in the case of the second pairs, they join the merged queue using the round robin scheme. Elements in the merged queue must have different time stamps which represent the outgoing time slots they occupy. Figure 3.5c is the state description after the front pointers have been rolled back. Note that the new contents of the output queue for host 2 are not affected by the roll back, only some old elements are recovered.

3.4 Congestion Control

An asynchronous distributed simulator, like other distributed systems, suffers from congestions. An object which is ahead in simulation time may generate many new message arrivals while another object is still processing messages with much earlier (smaller) time stamps. Since each one of these messages must travel around the ring and eventually occupies memories on other processors, it is possible for one part of the simulator to generate too much traffic to be saved in other parts. The resulting congestion will eventually

cause deadlocks and errors in the simulation. This problem is not unlike the congestion problem in communication networks, where too much traffic could bring the throughput down to zero, though there is significant difference between the two. In communication networks, the buffer which stores a packet is released once the packet has been successfully transmitted; i.e., an acknowledgment is received. In this distributed simulator, however, old states have to be saved; sending of a message only involves the manipulation of queue pointers, but no memory is freed. Memory is released when GVT progresses as described in Chapter 2. Those saved states with time stamps smaller than the GVT are no longer needed and can be removed. Memory may be recycled for future state saves.

One solution to the congestion problem is to throttle the amount of message transfer between neighboring processors. Initially, a sender has a number of *permits* (credits). A permit is "used up" when a message is sent, and it will be returned to the sender when the message is received. If a proper number of initial permits is selected, the number of possible in-transit messages is limited, and the transmission medium will never be overcrowded. Moreover, when the amount of available buffer is low, a receiver will stop returning permits to its predecessor so that the flow of incoming messages can be turned off temporarily. Once new buffers are available, message transfer will be enabled again. Permission credit is a common scheme in flow and congestion control of data communication networks.

A distributed simulator using Time Warp for synchronization is inherently stable. For those objects which are trailing; i.e., their simulation time (LVT) is only slightly ahead of the GVT, a GVT update would relinquish most of their saved states and hence most of the memory. For the objects which are ahead, their LVT will be considerably larger than the GVT. Therefore, a GVT update will free up only a portion of their memory. Since it is likely for the objects which are ahead to have frequent memory shortages and need to wait idly for GVT updates, we can expect that the growth rate of simulation times of the objects will be self-regulated.

3.5 The Role of the Central Controller

Besides the processors which carry out the actual event simulation, one or more processors are necessary to form a central controller which coordinates the event processors in the distributed simulator. The main function of the central controller is to initialize the simulation, synchronize the processors, and terminate the simulation. Processor initialization will be discussed in Chapter 5.

During a simulation run, the central controller is responsible for updating the Global Virtual Time (GVT). A GVT update may be carried out either periodically or only when necessary. For example, when a processor is running out of memory for saved states, it may initiate a GVT update so that its memory can be recycled. If GVT updates are carried out only when needed, fewer of them will be required and hence the total overhead is also reduced. However, memory usage will be more critical. Moreover, it is complicated to handle

situations such as multiple update requests, etc. Since an update is not a very time-consuming operation, we have decided to use periodic updates.

If the distributed simulator consists of only a few processors, one processor may serve as the controller. This processor will communicate with one of the event processors at a time. When a large number of processors are in the simulator, this type of sequential communication will be very inefficient, and the controller could become a bottleneck in the simulator. Some kind of distributed control would be more suitable. For example, the GVT in a snapshot is the minimum among all LVTs. Hence the controller may be arranged to have a (binary) tree structure, where the leaves are the processors. When a GVT update begins, the central controller will send LVT requests which propagate from the root down the links to the processors. Upon receiving the request, each processor will determine its LVT and report it to the controller. The minimum LVT can be determined in a distributed manner. When the LVT values propagate up the tree, two of them will be compared by an intermediate node at each level. Only the smaller one will be propagated upward. Hence the value reaching the root will automatically be the GVT, which will be broadcasted down the tree again to the processors. After an update, the controller will remain idle for a certain amount of time and then initiates a new update.

3.5.1 Algorithms for GVT Evaluation

The concept of Global Virtual Time (GVT) has been discussed in Chapter 2. It is a function of real time. If we take a snapshot of the distributed simulator at a certain real time t_r, GVT is the smallest time stamp value among all objects and in-transit message in that snapshot. According to its definition, GVT is a non-decreasing function of real time. Since it is used to commit simulation results and recycle memory for old state saves, we cannot be overly optimistic while evaluating its value. If a evaluated GVT is larger (later) than the actual value, an excessive number of saved states could be deleted. If the simulation needs to return to one of these states in the future, an error will occur. Moreover, some possibly incorrect simulation results could be committed. However, a conservative value is acceptable, as far as correctness is concerned.

A GVT evaluation involves two types of searching: The LVT of each object and the smallest time stamp among all in-transit messages. The evaluation of LVT of an object is straight forward; it is simply the smallest time stamp among all front elements in the queues in that object. Time stamps of in-transit messages are more difficult to determine. Because of the architecture of the simulator, in general, it is not possible to check the time stamps of messages in transit. A convenient solution is to include them in the LVT evaluation of either the senders or the receivers.

According to its definition, GVT can be determined by searching for the smallest time stamp in a snapshot at time t_r. The approach is very direct and provides the most up-to-date GVT value. However, it implies that event

simulation should be suspended so that a snapshot is frozen when a search is carried out. Although an overall search will not take very much time, the suspension is unnecessary. Moreover, the problem with in-transit messages cannot be easily solved. Since it is usually not necessary to have the most current GVT value anyway, we prefer to use an asynchronous and distributed algorithm which provides a somewhat conservative result while regular event processing is not significantly disturbed.

The main difficulty with including the time stamps of in-transit messages in the LVT evaluation of objects is to ensure that they are considered by some object, usually either the sender or the receiver. After an objet sends a message, it will process other events so that its LVT increases. Assume that this message is going to cause a preemption at the destination. If a LVT evaluation is carried out on the sender after this message has been sent and another evaluation is carried out on the receiver before the message is received; i.e, before the preemption/roll back occurs, the time stamp of the message will not be included in the GVT calculation. Since this message will initiate a roll back, it is possible for the evaluated GVT to be too optimistic (large).

One way to resolve this problem is to require receiver objects to acknowledge messages. A sender should record the time stamp of all outgoing messages and include time stamp values of all unacknowledged messages when evaluating its LVT. After a message arrives, its time stamp will automatically be considered in the receiver's LVT. It is possible that a LVT update on the receiver takes place after the arrival of a message, and the sender's update

happens before the return of an acknowledgment. Hence the time stamp of the message will be included in the LVT of both objects, but this will not cause an error in the final GVT. However, an error will occur if these events take place in a different order. That is, if the LVT of the receiver is evaluated before the arrival of a message and that of the sender is evaluated after the return of the acknowledgment. In this case, the time stamp of the message will neither be included in the sender nor the receiver. Samadi [Sama85] has developed an algorithm which includes; as part of the acknowledgment, the information whether the time stamp of the message was included in the LVT of the receiver or not. Based on this information, a sender will include the time stamp of a message if it was not considered by the receiver (although the acknowledgment has been received). This algorithm is very general, but requires multiple processor modes and additional control messages.

In a queueing network model, objects are connected by queues, and in this application, processors are connected by pipes. Both are FIFO devices. With this additional restriction, there exists a simpler GVT evaluation method [Sama85]. When an object receives a LVT request, it will send a special "boundary" message to each one of its outputs and will include the time stamp of each subsequent outgoing message in its following LVT evaluation, which is carried out after the object has received a boundary message from each one of its inputs. That is, messages sent before the boundary message will be considered by the receiver; those which are after will be included by the sender. This method assumes that (1) messages arrive a receiver in the same order as they were sent and (2) there must be at most one GVT update at a time in the

distributed simulator. If a new GVT update may be initiated before the previous is completed, additional control messages must be added. Otherwise, boundary messages and LVTs from different updates could be confused. Both of these requirements are fulfilled by the characteristics of the current simulator implementation. This will be discussed in further detail in Chapter 5.

3.6 Periodic State Saves

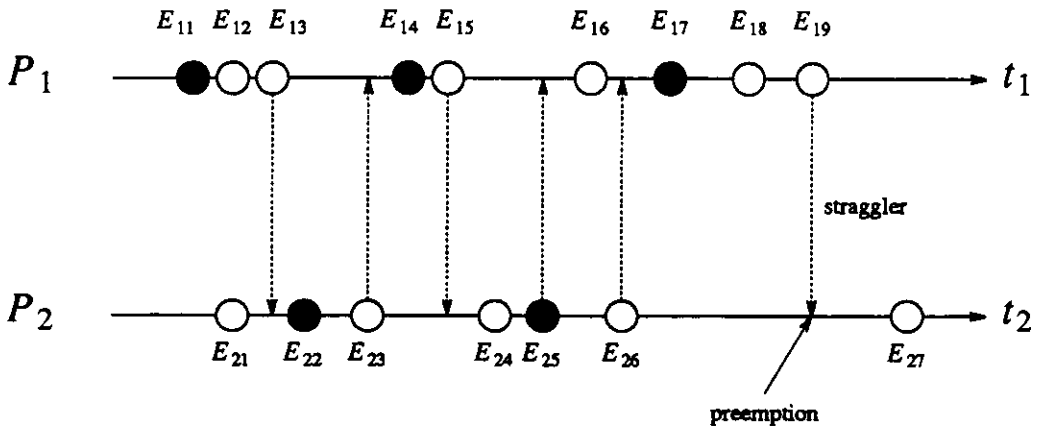
One of the main problems with Time Warp is its potential for large memory usage. A significant amount of memory is needed to save the previous states, especially if the state description of the model is complicated. In actual implementations of Time Warp, the states of a process are usually saved only periodically at certain *check points*, not after the completion of each event. Periodic state saves not only conserves memory but also reduces computation time wasted on state saves and old state searches. 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. This problem will be discussed into further detail in Section 5.6.) Unfortunately, there is a very undesirable side effect when only selected states are saved. A preemption can now cause a roll back to a state prior to the preempting message's time stamp. Strictly speaking, this may be considered as a violation of causality because an event (message) with a later time stamp can affect events with earlier time stamps. In the actual system, a message of course cannot actually affect other events which have

earlier (simulation) times. Although an object might roll back very far into the past, it will process the same events and send the same messages as it did in the previous simulate-forward phase (again, provided that the random number seed and other related informations are saved as part of a state description) until it reaches the time of the preempting message. However, in an implementation, the violation of causality can cause a number of serious problems.

3.6.1 Chain Roll Backs

A preemption can cause a roll back to a state prior to the preempting message's time stamp and send anti-messages, and these anti-messages may initiate secondary and even further roll backs. In the worst case, the simulation can return to the initial state. An example of this problem is shown in Figure 3.6. P_1 and P_2 are two objects; axes t_1 and t_2 are their simulation times. An event occurred on an object is represented by a circle. Assume the state is saved after every third event, which is represented by a darkened circle.

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 objects 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 again to E_{22} and so on. It is somewhat surprising that not only P_2 has to roll back past E_{25},



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 3.6: Chain Roll Backs Due to Selected Saved States

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. The problem is worsened if this chain of roll backs reaches a point which is earlier than the current GVT (Global Virtual Time). (In Figure 3.6, 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 may have been expunged 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 could have been sent to cancel old positive messages which do not exist any more. These problems, 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 could 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 have occur if every state had been saved. Consider the example in Figure 3.6 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.

3.6.2 Delayed (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 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 will simply 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 3.6, 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. Only E_{27} may be affected by the straggler, but there is no violation of causality. This will not only save computation time by reducing unnecessary roll backs but also avoid the deadlocks and errors described earlier. However, in the cases where anti-messages should be sent, the receiving objects will learn the "bad news" at a later time due to the postponement in delayed cancellation. This is a small price to pay, considering the advantages delayed cancellation have.

3.6.3 GVT Decrease

As discussed in Section 2.2.3, simulation is guaranteed to progress because GVT is a non-decreasing function of real time. Assume the GVT at a certain real time t_0 is $gvt(t_0)$. When an object receives a straggler with time t_s (t_s must be greater than or equal to $gvt(t_0)$ according to the definition of GVT), it may have to roll back to reach a saved state with a much earlier simulation time t_b, where $t_b \leq gvt(t_0) \leq t_s$. If the LVT of this object is evaluated again shortly after the roll back, the new LVT, and hence the new GVT, could be less than $gvt(t_0)$. There is an apparent decrease of GVT, which seems to be an error. This problem can appear even though every state is saved, but the farther apart the check points are, the higher the probably this would occur. Although it does not affect the correctness of the simulation, it could be confused with other real GVT decreases caused by actual errors in the simulation.

The GVT "decrease" problem can be avoided if the LVT evaluation algorithm takes the time stamps of preempting messages into account. Again, because of causality, a preempting message cannot affect any events earlier in simulation time than its time stamp. An retraction to further back into the past is purely due to the lack of a suitable saved state. Identical events will be generated again for the extra rolled back period when forward simulation is resumed. Hence LVT really has not rolled back past the time stamp of the preempting message. Therefore, in an LVT evaluation, if the calculated new LVT of an object is smaller than that of the previous value, one or more roll backs must have occurred between the two evaluations. The time stamp of the last preempting message should be recorded, and if the new LVT is less than this time stamp value, it should be set to this value. Since no preempting message can have a time stamp less than the then current GVT, therefore, the new LVT (and hence the new GVT evaluated) cannot be less than a previous GVT, and the GVT decrease problem is resolved.

3.7 Conclusions

Since our simulation model is represented as a queueing network, the selected synchronization method described in Chapter 2 has been extended for this type of models. In particular, synchronization for nodes in series (terminal nodes) and nodes in parallel (host nodes) are of special interest. Specialized algorithms have been developed and discussed in this chapter. In addition to those, there are problems with GVT update and deadlocks. They can be resolved through the use of delayed cancellation. With the major

synchronization problems resolved, a model partitioning and load allocation method will be presented in the next chapter, followed by some detailed implementation problems discussed in Chapter 5.

CHAPTER 4

MODEL PARTITIONING

In distributed computer applications, task (object) allocation is a common fundamental problem. The main objective is to allocate 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 part of the cases which are 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.

4.1 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 a smaller

simulation time † sends a message to another one with a larger simulation time, and the simulation performed in the period retracted is lost. The farther apart the two simulation times are, the more the leading processor needs to roll back, and the more the computation time is wasted. Therefore, it is desirable to minimize the differences of the simulation times among the processors so that fewer roll backs will take place, and even if they do occur, the processors will not need to return very far back into the past.

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 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 processor has a simulation time growth rate closest to this ideal.

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

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 inter-processor communications. This goal is best achieved, of course, in a single-processor environment, where no inter-processor communication is necessary. An effective sub-optimal allocation scheme should therefore find a good compromise between concurrency gained and communication overhead in a distributed environment.

4.2 Guidelines for Object Allocation

We have made the following assumptions to simplify the basic 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 can be relaxed so that ring networks with more complex connection patterns and 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

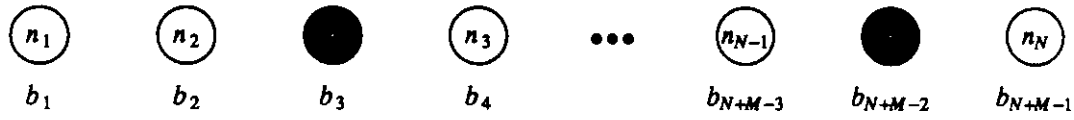


Figure 4.1: (b) Selecting M-1 balls from a set of N+M-1

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 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 4.1a. The nodes are numbered

overheads are relatively small, however. An important advantage is that since many old states are available, a roll back will not need to return very far into the past just to reach a saved state which will resolve the preemption. This situation is shown in Figure 5.3, where (a) indicates that an object with simulation time B being preempted by a message with simulation time A (B > A). Hence the object needs to roll back to a state before A. If states are saved infrequently as shown in Figure 5.3b (where each "x" represents a check point), the simulation needs to roll back to the state saved at time i, and the simulation between i and A is wasted (because the events in this period will be identical to those in the previous simulate-forward phase). If states are saved frequently (Figure 5.3c), the simulation will roll back to time j, and only the simulation between j and A is lost. The major penalty for frequent state saves is memory usage. Usually, the amount of memory used is proportional to the total number of saved states and what is required for each state description saved. Therefore, to determine a desired save frequency, the tradeoff is among the characteristics of the model, the amount of memory available, and the desired progression rate of the simulation. Since we have a special scheme which does not require copying the entire state description (as discussed in Chapter 3), the amount of additional memory per saved state is very limited. We can therefore conclude that it is desirable to have more check points in general. As a matter of fact, when the check points are too far apart, not only the wasted computation time but also memory usage may increase. This problem arises when the simulation periodically needs to return very far back to an old state (e.g., the initial state) so that this state cannot be deleted and no

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 4. 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.

4.3 A Heuristic Allocation Algorithm

Since the growth rate of simulation time involves reciprocals, we therefore use the concept of "processor load" instead in the following discussion. Each additional object assigned to a processor will slow down the simulation time growth rate, and hence increasing the load on a processor. 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 define "closest" to be the allocation which, among all processors, has the

smallest standard deviation. The ideal allocation, by definition, always has zero standard deviation.) The following algorithm generates 2^M sub-optimal allocations, and the best one is then selected from them.

1. Start from the first processor and first network node; i.e., **p** = 1, **n** = 1.

2. Assign node **n** to processor **p**.
   If **n** = N, all nodes have been assigned; terminate.

3. If load of **p** < ideal load, **n** = **n**+1, goto 2;
   else if **p** = M (the last processor), assign remaining nodes to **p**, terminate;
   else if load of **p** = ideal load, **p** = **p**+1, **n** = **n**+1, goto 2;
   else if load of **p** > ideal load, do both:

   a. **p** = **p**+1, **n** = **n**+1, goto 2;

   b. remove node **n** from **p**, **p** = **p**+1, goto 2.

This 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 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 to the next node and the next processor in the same manner until either the nodes or the processors are exhausted. Moreover,

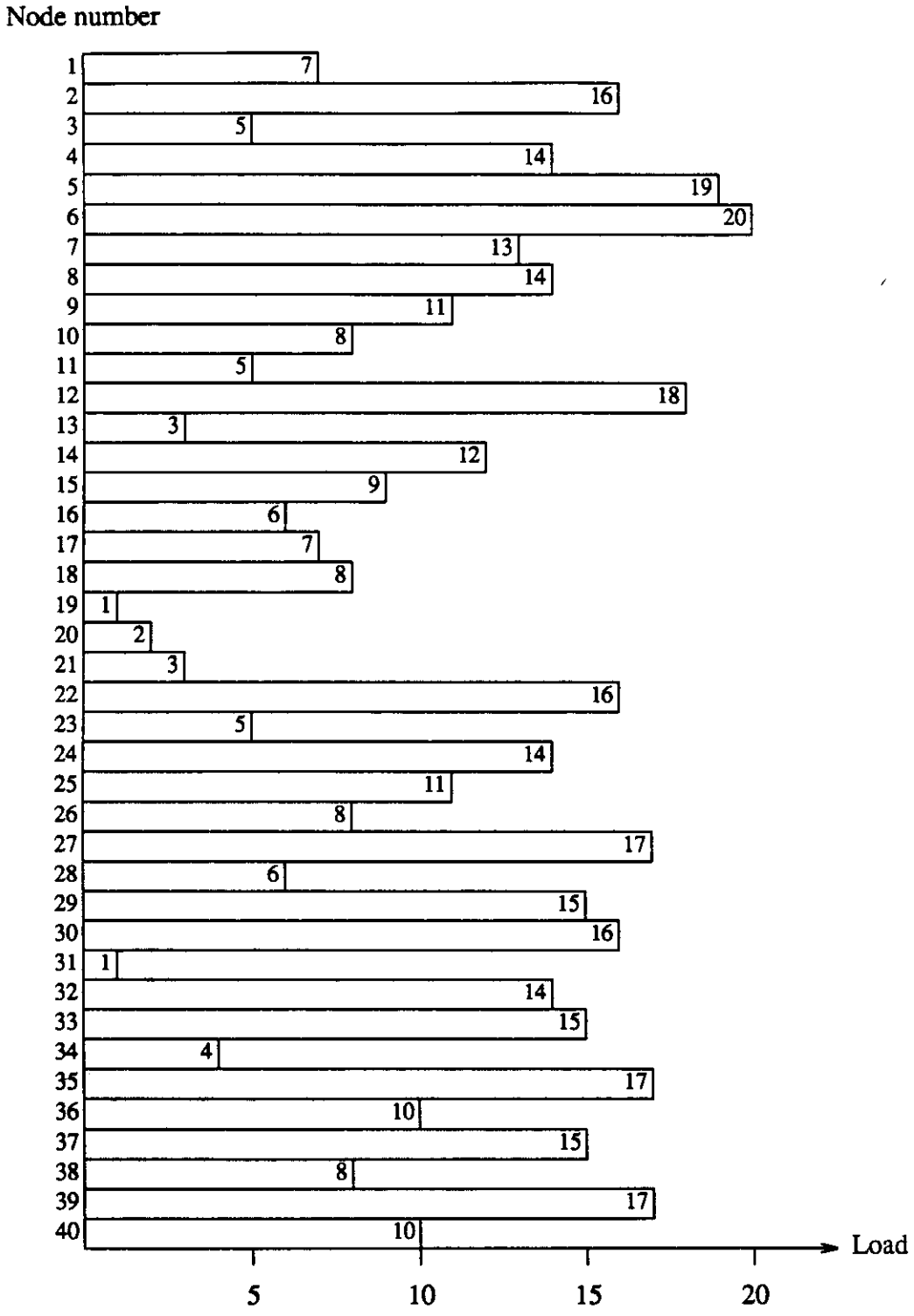


Figure 4.2: Loads of the nodes

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 and is provided in the Appendix. 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.

To demonstrate the performance of the algorithm, we provide an example with 40 nodes and 16 processors. The loads of the nodes are generated randomly with a range between 1 and 20 inclusively; they are shown in Figure 4.2. The resultant allocation and total loads on the processors are provided in Table 4.1.

Although this algorithm usually produces very near-optimal allocations, it does not produce optimal allocations in general. This can be shown by applying the algorithm starting from the last node in the series and continuing in the opposite direction. Some nodes with small loads in the middle of the series may be assigned to a neighboring processor. Frequently it will result in a slightly different allocation although both are usually considered to be very good sub-optimal cases. Table 4.2 shows the loads on the processors in the second case. The processors are listed in the reverse order so that a comparison

Table 4.1: Node assignment and total processor load		
Processor	Nodes assigned	Total load
1	1, 2, 3	28
2	4, 5	33
3	6, 7	33
4	8, 9	25
5	10, 11, 12	31
6	13, 14, 15	24
7	16, 17, 18, 19, 20	24
8	21, 22, 23	24
9	24, 25	25
10	26, 27	25
11	28, 29	21
12	30, 31, 32	31
13	33, 34	19
14	35, 36	27
15	37, 38	23
16	39, 40	27

The standard deviation of this assignment is 3.96.

of the two allocations is clearer. The only difference between the two is that node 20 is grouped together with nodes 16–19 in one case and nodes 21–23 in the other.

This basic allocation algorithm may be extended for a larger set of ring network models which do not completely meet its limitations. For example, if over 20 processors are used, the allocation algorithm should be simplified to reduce the execution time. One approach is to divide both the simulation model and the processors into two groups. Allocation will then be carried out independently in each group. If there are more than one network node which

Table 4.2: Alternative node assignment and total processor load		
Processor	Nodes assigned	Total load
16	1, 2, 3	28
15	4, 5	33
14	6, 7	33
13	8, 9	25
12	10, 11, 12	31
11	13, 14, 15	24
10	16, 17, 18, 19	22
9	20, 21, 22, 23	26
8	24, 25	25
7	26, 27	25
6	28, 29	21
5	30, 31, 32	31
4	33, 34	19
3	35, 36	27
2	37, 38	23
1	39, 40	27

The standard deviation of this assignment is 4.02.

have hosts connected, the allocation for both host and terminal nodes may be carried out together. In that case, the average event processing times for a host node and a terminal node must be known (probably determined experimentally), and the ring may be broken at some convenient point.

CHAPTER 5

SIMULATOR DEVELOPMENT

In order to demonstrate the feasibility of distributed simulation of data communication networks in general and the correctness of our methodology in particular, we have carried out the initial implementation of a special-purpose distributed simulator and applied to the benchmark model. This is the first step in applying the theoretical concepts discussed in Chapter II and III into practicality. As it turns out, a number of practical problems were discovered and subsequently solved. The major problems will be discussed in this chapter.

Our distributed simulator was developed on the Olympus network using the LOCUS operating system at UCLA. LOCUS is a network-transparent extension of UNIX and is running on a group of VAX 11-750 minicomputers. The distributed simulator consists of a number of components. They can be classified into three major categories: central controller, terminal simulator, and host simulator. Each terminal simulator and host simulator subsequently consists of several software modules. This type of hierarchical organization simplifies the development of the simulator.

When a simulation model is loaded, our present central controller forks the required processes, which are migrated to two or more computing sites on the Olympus network, and an actual distributed simulation run can begin;

inter-process messages are then sent through pipes. It should be noted that the central controller is loosely coupled with the processors actually responsible for the simulation; i.e., control messages are sent very infrequently to update the global state of the simulator. Therefore message transfers to and from the controller do not become a bottleneck, when the number of processors is moderate. Each simulator process should first initialize its memory pools and queues. Subsequently, a synchronization signal will be broadcast from the controller to the simulators so that a simulation run can begin.

5.1 Modules in a Simulator Component

Although a host simulator and a terminal simulator have different functions, they both contain the following modules (It should be understood that modules with the same name and function are usually not identical in different types of components):

DEIOQ: Remove a message from a queue.

ENIOQ: Enter a message to a queue.

ENOUTQ: Enter a message to a queue which leaves a process. This module also handles delayed cancellation and generates anti-messages.

MANAGE: Remove no-longer-needed saved states and return the freed memory.

READIN: Read a message (if available) from the input pipe.

ROLL-BACK: Given the time stamp of a straggler, find the latest saved

state which resolves the preemption and then recover that state.

RPTLVT: Evaluate the current LVT and send it to the controller.

SENDOUT: Send messages to another process through the output pipe.

A host simulator also contains the following modules:

FRONT-RB: Roll back the front pointers of the output queues from the hosts.

GAUSSIAN: Generate a pair of normally distributed random numbers.

HOST: Main program for host simulation.

HOST-PROC: Process a host event.

SAVE-FNT: Save the output queue front pointers as a group.

SEARCH-FRONT: Search for the most-recent set of front pointers for a front roll back.

The modules below are for terminal simulators only.

ARRIVAL: Generate a new message arrival for a terminal node.

NXEVN: Determine the next event to be simulated.

RESPONSE: Find the corresponding command for an available response.

TERMINAL: The main program of a terminal simulator.

Moreover, the MANAGE module in a terminal simulator also reports the response times which cannot be revised. Further details of these modules are provided in Appendix ??.

5.2 Simulation cycles

Each processor in the distributed simulator repeatedly carries out a number of functions until the simulation terminates. Each one of these repetition can be considered to be a *simulation cycle*. At the beginning of a cycle, a processor will read all available inputs from its input pipe, and they will be stored in the input queue. After all inputs have been read, they will be processed. Finally, outputs generated will be sent to other processes through the output pipe.

5.2.1 Simulation Cycle of a Host Simulator

The simulation cycle of a host simulator is outlinted below and is also shown as a flow chart in Figure 5.1. Since command processing on the hosts are independent of one another, it is irrelevant that which host is simulated first. Only the merging of response packets should be done in the correct order.

1. While there are commands available in the input pipe, read one input at a time (The inputs must be commands in this case.) and insert it into the input queue of the destination host. If the new command has a time stamp smaller than that of the last command processed by the host, the commands have been processed in an incorrect order, and a roll back is necessary so that commands can be processed again in the correct order.

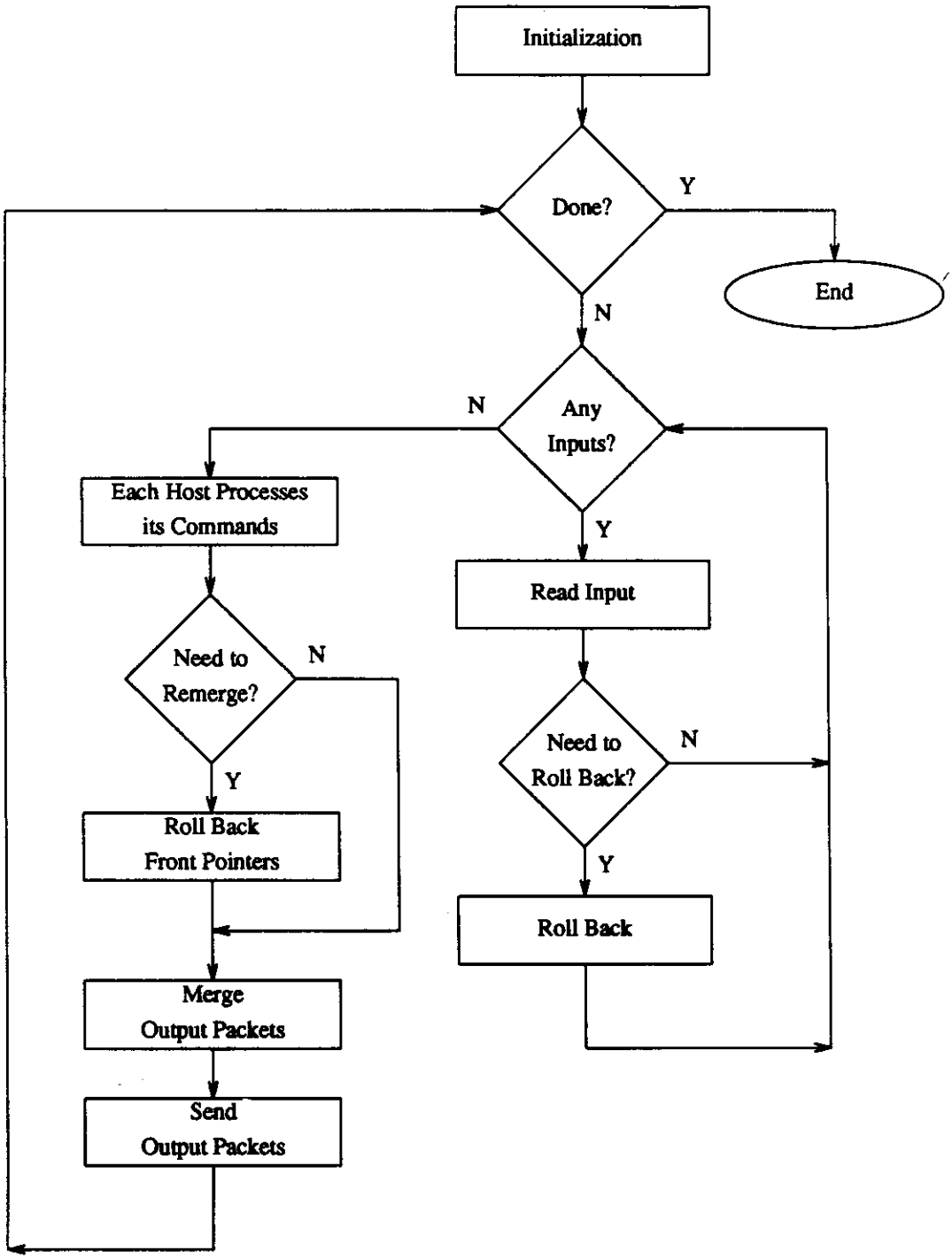


Figure 5.1: A host simulation cycle

This roll back will initiate another roll back of the output queue front pointers and the merged queue pointers; the response packets will be merged again in the right order.

2. For each host in the model, simulate the processing of all commands in the input queue, generate response packets and insert them into the output queue.

3. Among the new responses, if the smallest time stamp is less than that of the tail element in the merged queue, the responses could have been merged in a wrong order. To guarantee correctness, a roll back and remerge is necessary.

4. Merge the response packets according to the FIFO/round robin scheme and then send them to the next processor.

5.2.2 Simulation Cycle of a Terminal Simulator

In addition to processing response packets and by-passing packets, a terminal simulator is also responsible for generating new commands and calculating response times. It needs to compare the time stamps of input packets and packets already inside the object to determine which event should be processed next. The flow chart for the simulation cycle of a terminal simulator is shown in Figure 5.2.

1. While there are inputs available, read one of them at a time from the input pipe. If the time stamp of a new element is smaller than that of the input queue front element and is also smaller than that of the last element

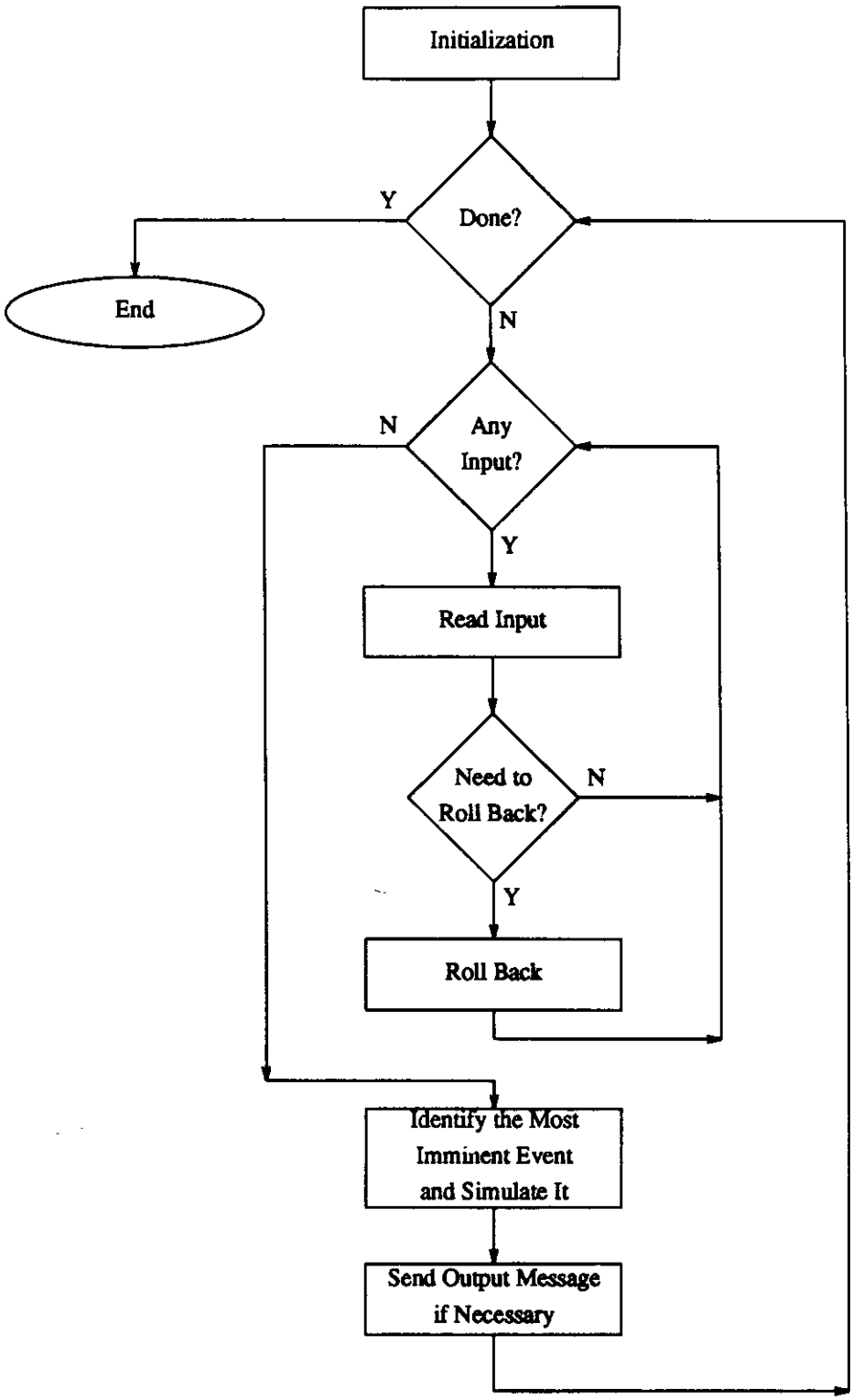


Figure 5.2: A terminal simulation cycle

processed by the first node, a roll back is necessary.

2. After reading all available input messages, compare the time stamps of all events ready to be simulated in the processor and simulate the one which has the earliest time stamp. This is carried out by comparing time stamps of front elements in the queues. If two elements have the same time stamp, their physical dependency will determine the order of simulation. New and by-passing messages will be inserted into the output queue from the last terminal node.

3. Messages in the output queue will be sent to the next processor through the output pipe.

Besider processing the events, each simulator component periodically returns permission tokens to its predecessor (when a certain number of them have been accumulated) and checks for LVT requests from the controller.

5.3 Event Generation

In discrete event simulation, inputs (or external events) are needed to stimulate the model being simulated. There are two general approaches to obtain these inputs. When data from the real world are available, actual events may be used as inputs to the simulation. For example, in the simulation of communication networks, if we can record the times a user issuing commands at a terminal during a certain period, we can use these data as command arrival times in the simulation. (How to obtain a representative set of data is a another issue.) If no such data are available (as in this case), the inputs can be

generated randomly according to certain distributions. It is very important to select appropriate distributions so that the generated events will approximate the behaviors of their counterparts in reality, or the accuracy of the simulation results will be affected. It is also important to generate statistically good random numbers. Our experimental results indicate that replacing a good random number generator by a less sophisticated one, the simulation results could change from good to something completely meaningless.

Usually, random numbers with specific distributions are generated by applying inverse transforms or other manipulations on uniformly distributed random numbers. Uniform random numbers can be obtained from natural phenomena such as white noise and nuclear fusion. They can also be generated using computers; examples of existing methods are the Linear Congruential Generator (LCG) [Law82, Knut81] and the Shift Register Sequence Method [Kirk80]. The numbers these schemes produce are considered to be *pseudo random numbers* because a new "random" number X_{n+1} is a function of one or more of the previous numbers X_n, X_{n-1}, X_{n-2} .... Once an initial value X_0, or a seed, is provided, the entire series of numbers is fixed. (This property is actually an advantage, as we shall see.) When a number which appeared in the series before is generated again, the cycle of random numbers will repeat. The length of a cycle is called the *period* of a generator.

The Linear Congruential Generator is a very common pseudo random number generator. Its general formula is:

$$X_{n+1} = (a\,X_n + c)\ mod\ m \tag{5.1}$$

where a, c, and m are constant parameters. An initial number X_0 should be supplied to initialize the generator, and an infinite series of numbers can be generated recursively. Since the numbers are obtained after "mod m," they must be smaller than m, which is the maximum possible length of the period. A LCG with period m is called a LCG with full period, where every number between 0 and m – 1 appears exactly once in a cycle. It is very undersirable to use a generator with a short period because a repeated series of random numbers will introduce dependencies among results from different sections of the simulation. Therefore, parameters a, c, and m of a LCG should be selected carefully such that the generator will have a full period and also provide numbers whose randomness fulfill statistical requirements [Law82, Knut81]. A shortcoming of the LCG is that its period is limited by the modulus m regardless of the choice of the parameters. If very many random numbers are needed, an m larger than the word size of a machine will significantly lengthen the amount of time needed to evaluate (5.1). The Shift Register Sequence Method does not have this limitation and can efficiently generate numbers with much longer periods. Since we anticipate that fewer than 50,000 random numbers will be required per simulation run, a LCG is suitable in our experiments. The particular generator used in these experiments is:

$$X_{n+1} = (314159269\,X_n + 453806245)\ mod\ 2^{31}$$

The modulus of this generator is 2^{31}, which can be conveniently carried out as a bitwise AND function in C on a machine with 32-bit word size. (This generator has a full period, which is 2^{31}, or 2,147,483,648.)

During a roll back, part of the previous simulation is discarded. Until the point where a straggler changes the input to the object, the events in the new simulate forward phase should be identical to the corresponding ones in the previous phase. Unfortunately, this is not necessarily (and most likely will not be) the case if a new series of random numbers is used to generate the events. Therefore, the purpose of delayed cancellation, which assumes that identical events will be generated, will be defeated. Moreover, because new events may have smaller time stamps, a new LVT and hence a future GVT could decrease. In order to generate identical events in a new simulate-forward phase, the same series of random numbers should be used. When a pseudo random number generator is used, this can be achieved simply by saving the random number seed of each object as part of the saved state. Hence each object should have its own generator and its own series of random numbers.

Since the random number generators for different objects use the same formula, to avoid any correlation among the random numbers for different objects, it is important to initialize them with seeds which are far apart in the period. This is achieved by generating numbers in the entire period and record the every 100,000th number. These numbers will be used as the seeds. Since each generator will produce in the order of several thousand random numbers per simulation run, there will not be any overlaps or correlations among the random numbers in the simulation.

In Chapter 1, we assume that command arrival is a Poisson process and host processing times are normally distributed in our simulation model. A

Poisson process has the following three properties: (1) only one command can arrive at a time (no bulk arrivals), (2) the number of arrivals in different time intervals are statistically independent, and (3) it is time invariant. (Poisson processes are frequently used to model customer or service request arrivals because arrival processes usually fulfill these properties. Moreover, in mathematical analysis, queueing network models are frequently intractable unless the arrival process is Poisson.) In this particular ring network model, it is reasonable to assume that a user can issue only one command at a time (property 1). Moreover, because of the characteristics of the model, simulation of network activities in a period of less than one hour of simulation time is sufficient to provide enough information about the behaviors of the network, we can therefore assume that the command arrival rate does not vary in such a short period of simulation time (property 3). The second property of Poisson processes is more suitable for modeling the combined random behaviors of a large population of users (customers), but is acceptable for the single-user case since his behavior in one period does not necessarily relate to that at other times. When the arrival process is Poisson, the inter-arrival times are exponentially distributed. Exponential random variables can be generated using the inverse transform method [Law82]:

1. Generate a uniformly distributed real number U, $0 < U < 1$.

2. $-\frac{1}{\beta} \ln U$ will be an exponentially distributed random number with parameter β; i.e., the density function of the random variable is $\beta t e^{-\beta t}$.

For host "think times," we assume that it takes, on the average, 1 second to access a file on a disc. The actual host processing time may be longer or shorter than the average, but it is equally likely to be one way or another; a majority of them will be close to the average and only a small percentage will be far away from it. This kind of property can be approximated by the normal distribution. However, the density function f(x) of the normal distribution has positive value for $-\infty < x < \infty$, although it decreases rapidly when x is away from the average. That is, it is very unlikely but possible to generate a negative "think time." This problem is resolved by taking the absolute value of the generated host processing time. Since the probability of generating a negative value is extremely low, this adjustment hardly affects the response time distribution. Normal random variables can be generated using the Polar method [Knut81]:

1. Generate two independent, identically distributed uniform random (real) numbers U_1 and U_2, where $0 < U_1, U_2, < 1$.

2. Let $V_1 = 2U_1 - 1$ and $V_2 = 2U_2 - 1$.

3. $S = V_1^2 + V_2^2$

   If $S \geq 1$, return to step 1 and generate a new pair of U_1 and U_2.

4. Finally, two normally distributed random numbers X_1 and X_2 with mean 0 and standard deviation 1 can be obtained from the following formula:

$$X_i = V_i \sqrt{-2 \frac{\ln S}{S}}, \quad \textit{where } i = 1,2$$

(A proof of this method is provided in Appendix 2.)

5. To obtain normally distributed random numbers with mean m and standard deviation σ, multiply the X_i's by σ and then add m.

5.4 Message Arrivals

In a network, a number of terminals are usually connected to each node. The commands from these terminals will compete for outgoing message slots. We assume that the conflicts will be resolved in a FIFO manner. To represent this property, an arrival mechanism is set up in each terminal node simulator. This mechanism is realized as a list in which every element represents a terminal in the model. Each list element contains a time stamp which is the arrival time of the next command from the corresponding terminal. The list is sorted in increasing time stamp order such that the first element contains the next command (its simulation time and terminal i.d.) to arrive the node in the simulation. After this command is processed, the next command from the corresponding terminal will be generated by obtaining an exponentially distributed random number (with the command arrival rate of that terminal as the parameter) and adding it to the previous time stamp. The new element will be sorted again to maintain increasing time stamp order in the list. After being processed, arrived commands will be saved in an old-arrival queue. This queue serves two purposes. Old commands are needed again when the corresponding response packets are received. The round-trip delay will be determined by subtracting the arrival time from the response time. After a roll back, previous arrivals in this queue may be reused so that it is unnecessary to regenerate the

message arrivals in the new simulate-forward phase.

5.5 Pipes

In UNIX, communication among processes is through pipes, which should be opened before processes are generated. Pipes are originally designed for communications among consecutive processes in one direction only, and there are several properties which make them less suitable for bi-directional communications as required in this application. For example, each pipe has a finite capacity of 4K bytes, which can be filled up by consecutive message transfers. If a process writes to a pipe which is full, this process will busy wait until another process reads from the other end to free up some room for it to complete the write. Similarly, when a process reads from an empty pipe, it will also busy wait until something is written to the other end for it to read. These properties become shortcomings when data transfer is not restricted to one direction only. For example, when the model is a ring network, the processes in the simulator will also be connected as a ring. If every pipe (between a pair of neighboring processes) happens to be full and each process is attempting to write to its output pipe, all of these processes will be busy waiting forever because no process can read from any input pipe. A analogous deadlock occurs when every process is reading from an empty pipe. These two problems can be avoided if we can guarantee that a process will never read from an empty pipe or write to a full pipe. (In UNIX, the I/O control system call "ioctl" can check the number of bytes available in a file (or pipe), but it was not completely implemented in LOCUS when this work was done.)

To prevent reading from an empty pipe, a process should write a dummy token into a pipe before reading from it. This token serves as an delimiter for the last byte read. If the first item being read is the dummy itself, the pipe was empty. Otherwise, the process should continue reading until the dummy is read back. (It is possible that additional new messages arrive after a dummy is written; they will be read in the next simulation cycle.) To avoid filling up a pipe, we use permits to control the message flow. This method has been discussed in Chapter 3 for memory management purposes. The number of permits between two processes limits the maximum number of in-transit messages in a pipe. Hence we can simply set the number to less than the storage capacity of the pipe.

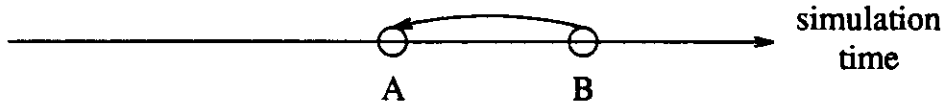
5.6 State Saves

State save and roll back form the central mechanism in a distributed simulator using Time Warp. The way state saves are carried out has direct influence on the correctness, efficiency, and memory usage of the simulation. Two important problems are how often and when states should be saved.

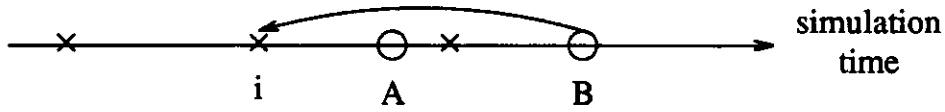
5.6.1 State Save Frequency

The "optimal" state-save frequency is a compromise among several factors. If states are saved very frequently, the total computation overhead for state saving will increase. Moreover, during a roll back, since there are more saved states available, it will in general require a longer search to find the most recent saved state which will resolve the preemption. These additional

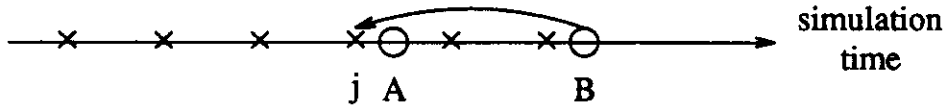
memory can be resued. As the simulation progresses, queue lengths will increase constantly and the memory for queues will eventually be exhausted.



(a) A preemption



(b) Infrequent state saves



(c) Frequent state saves

Figure 5.3: Frequent and infrequent state saves

This is not to say that the more frequently states are saved, the more efficient the simulation will become. For example, when they are generated, packets from the same response carry the same time stamp. There is little advantage gained if the state is saved after processing each one of these messages. Figure 5.3 suggests that state save frequency should be related to the progression rate of the simulation time. The state should be saved again after the simulation time has increased by a certain amount since the previous save. Usually, it is easier to relate the number of state saves to the number of events

processed rather than to the simulation time progressed. When the simulation model is large, the average amount of simulation time progress per event will be small. Hence the state can be saved after processing many (e.g., several hundred) events. When the model is small, the state should be saved more frequently (e.g., after every 50 events). The desired state saved frequency can be determined experimentally with some pilot simulation runs.

5.6.2 Location of Check Points

The second problem is when state saves should be make. Intuitively, it may seem to be irrelevant that a state is saved before or after processing a few events, and it is reasonable to perform a save at some fixed point in each simulation cycle. In reality, the problem is not as straight forward. Since the purpose of state saves is for future recoveries, they should be made at points where the state description is "representative" such that it is convenient for roll backs. When a preemption occurs and a roll back is needed, before a previous state can be recovered, the simulator first needs to determine which state it should return to. Usually, it is the latest state which was saved before the preemption time so that the effect of the preemption can be canceled. To determine this state, it is necessary to compare the preemption time to the time stamp of a certain queue element in the saved state. Therefore, if this element is missing in a certain state; i.e., the pointer to this element is NULL, the saved state is practically useless, and the search has to continue to an earlier state.

Since the simulation is carried out in cycles and new message arrivals are determined by polling, the state description tends to fall in some patterns at different part of a simulation cycle. For example, after all input messages have been processed, the input queue will be empty and the front pointer will be NULL. At the end of a simulation cycle, all output messages should have been sent (unless it is limited by the flow control mechanism) so that the output queue will be empty and its front pointer will be NULL. Below are the detailed descriptions of several different types of roll backs and the corresponding methods to determine which old state should be recovered. They provide the hints on when should states be saved.

When a straggler command enters host H, the roll back will recover an old state, including an old output queue front pointer. Since the content of the output queue of H may be different in the next simulate forward phase, the front pointers of the output queues of the (other) hosts should also roll back together so that the responses in the output queues can be remerged. The front pointer roll back should recover a saved front state whose output queue front pointer for H points to an element with a time stamp less than or equal to that of the recovered output queue front pointer. Hence a search will check the time stamps of the output queue front pointers for H in the saved front states to determine which state to roll back to. Therefore, the front pointers in a saved front state should not be NULL or it will be useless for roll backs. However, in a simulation cycle, the output queues of the hosts are not empty only after commands have been processed and before the merging is completed. Therefore front pointer saves must be carried out during this period.

If host H has not received any commands for a while, it could generate responses packets which have time stamps smaller than those of some responses (from other hosts) which have already been merged. Although no host needs to roll back when this happens, the output queue front pointers and the merge queue description need to roll back such that response packets can be remerged in the correct order. In this case, the front roll back needs to recover a state whose merge queue tail pointer has a time stamp less than that of the earliest new response from H. Therefore, this type of front roll back will need to check the merged queue tail pointers, and the saved states should have different merged queue tail pointers. Fortunately, this also implies that front pointers should be saved after commands have been processed and before their mergering is completed.

Since packets arriving a terminal node are processed in a FIFO manner with priority, the time stamp of a new packet should be compared to that of the most recent packet processed by the node and the time stamp of the front element of the input queue. If the time stamp of the new packet is smaller than both of them it will cause a roll back. A roll back needs to search for a saved state whose input queue front (or tail) element has a time stamp smaller than that of the new element. Therefore, these front elements should not be NULL in the saved states or the search will always need to go back further than necessary to a state whose tail element fulfills the requirement. As a result, the states of a terminal node should be saved after new packets have been read and before all of them have been processed.

5.6.3 Conclusions

Check points should be made at locations which are convenient for roll backs. From the above discussion, we realize that it is another model-dependent problem in general. For queueing network models, it is important to have saved states in which components necessary for roll back comparisons are not missing. Since it has direct effect on the performance and correctness of the simulator, it is important to carry out short experiments to determine the tradeoffs among different check point locations before making substantial simulation runs.

5.7 Message Format and Representation

One of the major functions of the simulator is to manipulate messages in the nodes and queues in the model. The various queues are actually buffers storing messages so that they can be processed in the correct order. Since messages do not always arrive in increasing time stamp sequence, they sometimes need to be inserted into the middle of a queue to maintain the correct order. Moreover, an anti-message may cancel another message in the middle of a queue. Hence queues are represented as linked lists so that insertion and deletion can take place at any part of the queue in a flexible manner.

A message in the simulation contains three fields as shown in Figure 5.4: a time stamp, the source and destination information, and a message i.d.; they are represented as three unsigned long words. The time stamp is a unsigned long word by itself. The source and destination field contains two parts for the

source address and destination address respectively; they are usually i.d. of terminals or hosts. The third field consists of several parts: a one-bit flag indicating whether the message is regular (positive) or anti (negative), a two-bit priority value (for priority levels 0-3), another one-bit flag indicating whether a message in an output queue has previously been sent or not, and the remaining part is the message i.d., which is a number to distinguish different messages with the same source/destination pair.

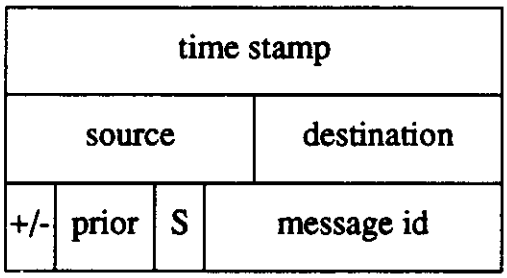


Figure 5.4: Message Representation

In the simulator, memory for the queues are available in memory pools, which are long arrays of structures for the queue elements. Special procedures are developed to obtain and return structure elements to and from the pool such that memory can be reused once released. In addition to the three fields for a message, each structure element contains pointers to the previous and next element in the linked list.

5.8 Deadlocks

Because of the lack of tight centralized control, a distributed simulator may suffer several different types of deadlock problems. The pipe I/O deadlocks have been discussed in Section 5.5. In addition to those, other examples are deadlocks due to incorrect modeling and memory usage.

5.8.1 Deadlock Due to Incorrect Modeling

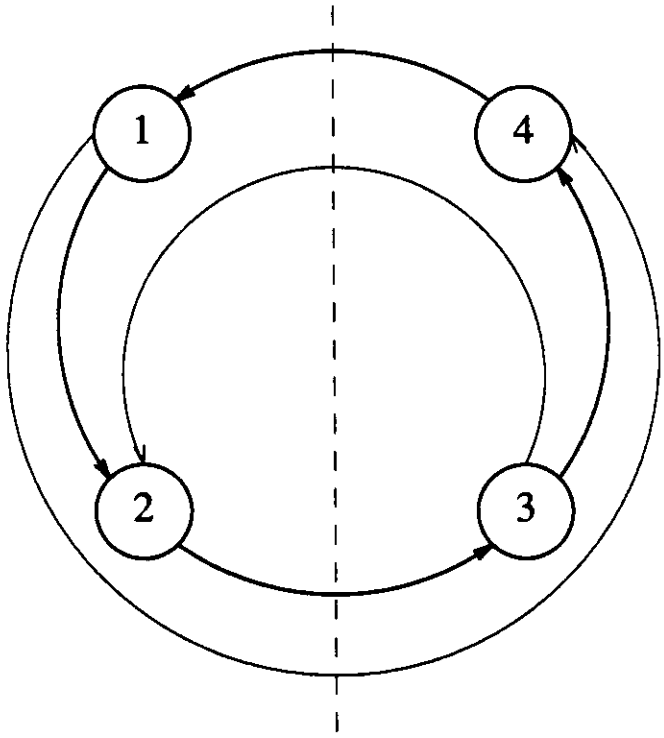


Figure 5.5: Deadlock Due to Interfereing Roll Backs

Figure 5.5 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 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. However, these slots are actually the same one and cannot be used by two messages simultaneously. 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 5.5, 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.

5.8.2 Buffer Usage Deadlock

When the output queue of an object is full, this object should neither generate more arrivals nor accept any inputs unless the destination of the current front element of the input queue happens to be itself. Otherwise, too many messages will exist in the simulator and create congestions. If objects all around the simulator refuse to accept inputs, 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 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. The GVT can then be updated and memory will be released. However, when there are "amplifying" devices such as hosts 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.

5.9 Conclusions

In addition to the theoretical concepts discussed in Chapter 2, 3 and 4, a number of practical problem related to simulator development were discussed in this chapter. The major problems include event generation and processing in the simulator components, state saves and deadlocks. After solving the major problems, an experimental distributed simulator was developed and some initial experiments were carried out. The results of these experiments will be discussed in the following chapter.

CHAPTER 6

EXPERIMENTS AND RESULTS

6.1 Network Performance Evaluation

A number of experiments have been carried out to test the correctness and performance of the distributed simulator as well as the performance of the benchmark communication network. As mentioned in Chapter 1, the basic benchmark model is a ring with three network nodes shown in Figure 1.1. There are, however, many different combinations of host/terminal connections within this basic model. A number of these combinations which produce different amount of traffic on the network are used in the experiments.

The main objective of performance simulation is to obtain information concerning the delay, throughput and queue length distribution of a network. In the experiments, we concentrate on the first issue. Delay can be measured through the response time, which is the elapsed time after a command has been issued at a terminal until the corresponding response packets are received. Response time can be measured in *time units*, which is the time needed to transmit a slot. Since a response usually contains more than one packet, there is a difference between the response times of the first packet and the last packet of a response. Both of them are measured in the simulations. A response time reflects the sum of three delays: (1) Waiting time due to the sharing of hosts,

since commands are competing for the services of hosts. These commands do not need to be originated from different terminals; an earlier command from a terminal may delay another one from the same terminal because of the FIFS service scheme. (2) The host "think time"; i.e., the simulated processing time of the command. (3) Waiting time due to sharing of the network. (The resource being shared are the slots in the ring.) Each response time contains two network access delays, once from the terminal to the host and a second one for the return trip. In a realistic network model, the first two factors dominate the response time. However, since one of the main objectives of a performance study is to observe the network delay under various traffic loads, it should be isolated from the other components in the response time. One solution is to add special features to the simulator to measure this factor separately. Network delay can also be measured indirectly by comparing the differences between the two response times. The minimum possible difference between the two is the number of packets in that response minus one time units because each packet occupies one slot. If there is only one host in the network, the response times difference will always be the minimum. Since the two response times contain the same amount of other delays, any difference greater than the minimum is a result of network sharing.

6.2 Characteristics of the Benchmark Network

In the benchmark network, the hosts are not time shared; commands are processed in a FIFO manner. This assumption is somewhat different from most realistic cases but simplifies the simulator design. Each packet in the network

consists of 12 bytes, among them there are five bytes of overhead and seven bytes of data. Since we assume that the communication medium can transmit 56000 bits/sec, approximately 583.3 packets can be transmitted per second. That is, 1 second = 583.3 time units. We also assume that each response is 300 bytes long; each response will therefore require 43 packets (or slots). If each host delivers a response every two seconds on the average (This is not a very realistic assumption as will be discussed later.), 27 hosts will saturate the network. That is, if there are 27 or more hosts in the model, the delay due to network sharing will grow to infinity.

6.3 The Experiments and Results

The following models are included in the experiments:

1. 2 terminals, 2 hosts
2. 4 terminals, 4 hosts
3. 8 terminals, 8 hosts
4. 12 terminals, 12 hosts
5. 16 terminals, 16 hosts
6. 20 terminals, 20 hosts
7. 22 terminals, 22 hosts
8. 24 terminals, 24 hosts

In each case, every terminal in the model is connected to a dedicated host computer. That is, of course, not a very realistic assumption. However, this series of models produce different amount of loads on the network so that its performance under various conditions can be measured. Moreover, the command arrival rate (Poisson) at the terminals is 0.5/sec. That is, we assume that a user issues a command (and the connected host produces a response) every two seconds on the average. This rate is probably much faster than any realistic situation but produces enough traffic on the network which is also convenient for performance studies. In a more realistic case, five terminals are probably time sharing one host, and the command arrival rate would be 0.01/sec. However, since command arrival is assumed to be a Poisson process, the five terminals can be approximated by one with five times the arrival rate, which is just the type of models used in the experiments here. Therefore, for example, the 12 host/terminal model represents an actual network with 60 terminals and 12 host computers.

Figures 6.1, 6.2 and 6.3 are plots of the response times of the last response packets (observed by the user at terminal 1 of node 1 in the network) against the corresponding simulation times at which these responses are received. The figures are for the 2-host/2-terminal, 12-host/12-terminal and 20-host/20-terminal cases respectively. Since the host processing time (which has the same probability distribution in all of these cases) and host waiting time are the dominating factors in the response time, it is not very clear from the figures that there is any particular relation between the response time and the number of host/terminal pairs. In all three cases, most of the response times are

between 1 and 3 seconds, and only occasionally some very long or short samples appear.

The differences among the three cases are much more distinctive in Figures 6.4, 6.5 and 6.6 where the response time difference between the first and last packets are plotted for the three models. When there are only two host/terminal pairs (Figure 6.4), the difference between the response times is frequently at its minimum; i.e., 42 time units, which is the time needed to transmit 42 packets after the first. Occasionally, there is some delay due to network sharing, but the maximum delay never exceeds 84 slots, which is the maximum possible delay when two hosts are sharing the network in a round robin manner. When the load increases to 12 pairs (Figure 6.5) and 20 pairs (Figure 6.6), fewer samples are at the minimum, and the effect of sharing increases the average difference.

Table 6.1: Average response times

Model	Ave. Response Time (sec.)	Ave. Difference between First and Last Packets (sec.)
2 host/term	1.539	0.075
12 host/term	1.577	0.113
20 host/term	1.747	0.215

When there are 24 host/terminal pairs, the traffic load on the network is very close to its capacity limit. In the simulation, the queue lengths grows continuously. The simulation eventually runs out of memory and terminates. This problem appears again and again when the experiment is repeated. Hence no result data are available for this case.

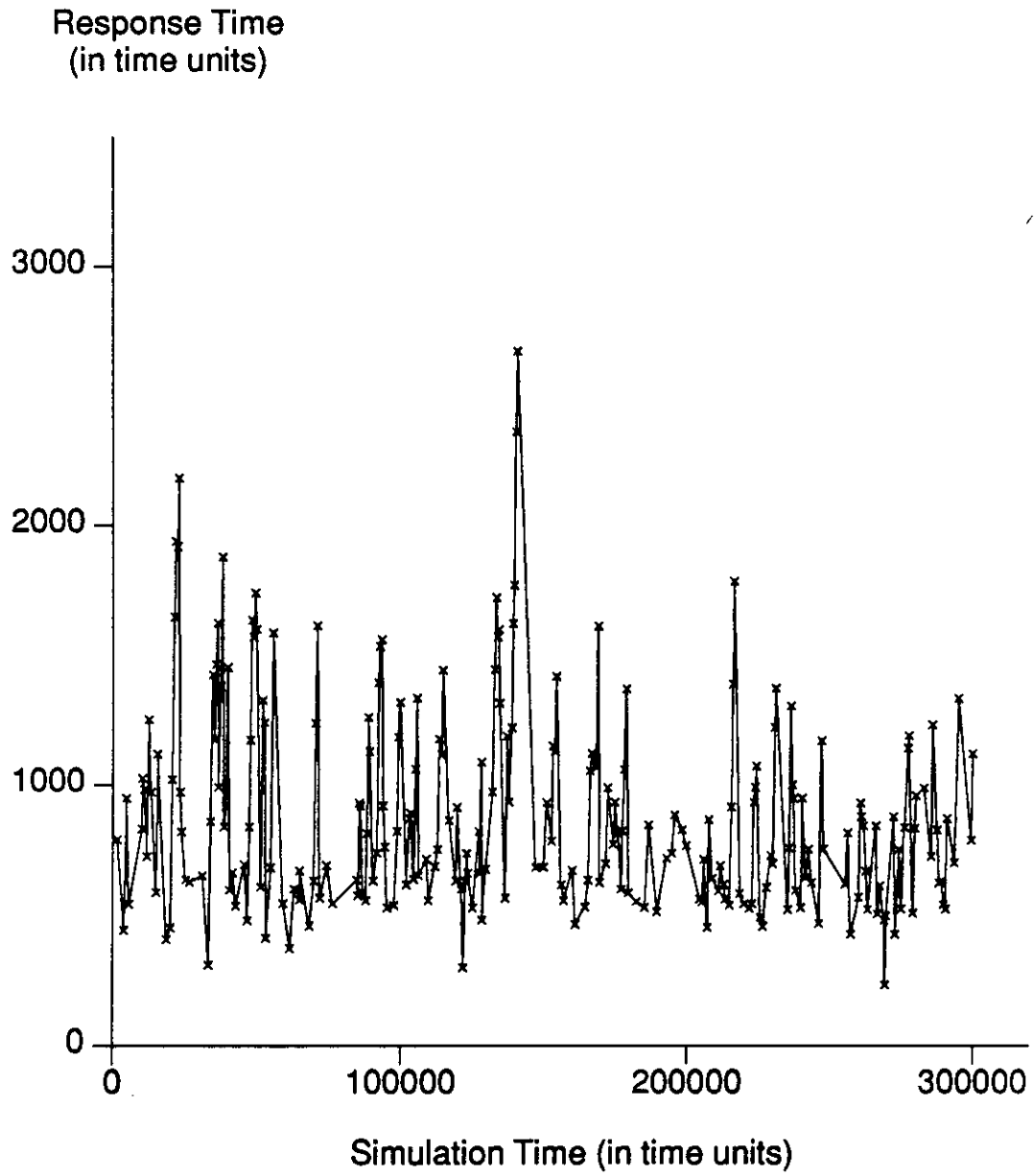


Figure 6.1: Response time of the 2 host/terminal model

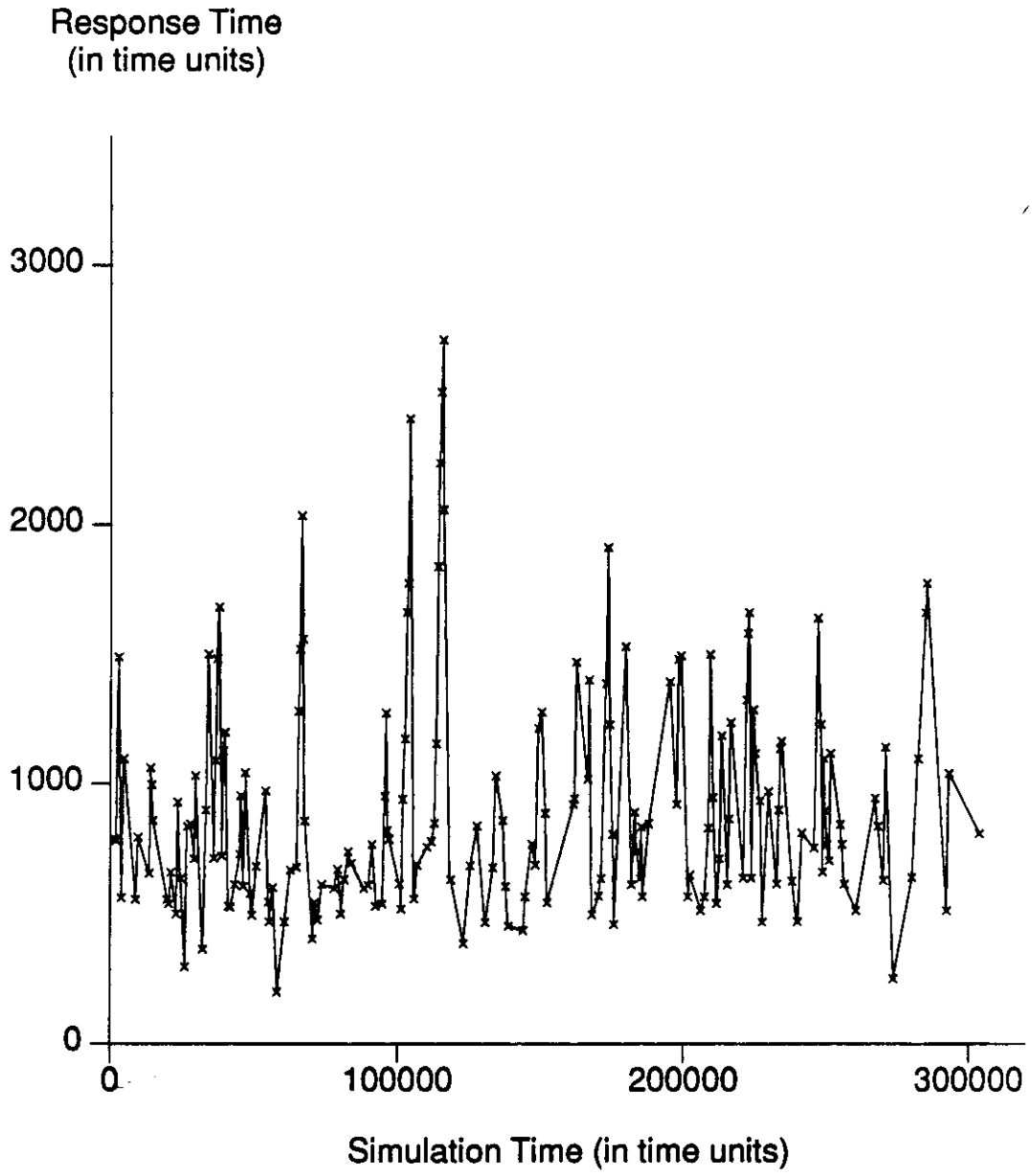


Figure 6.2: Response time of the 12 host/terminal model

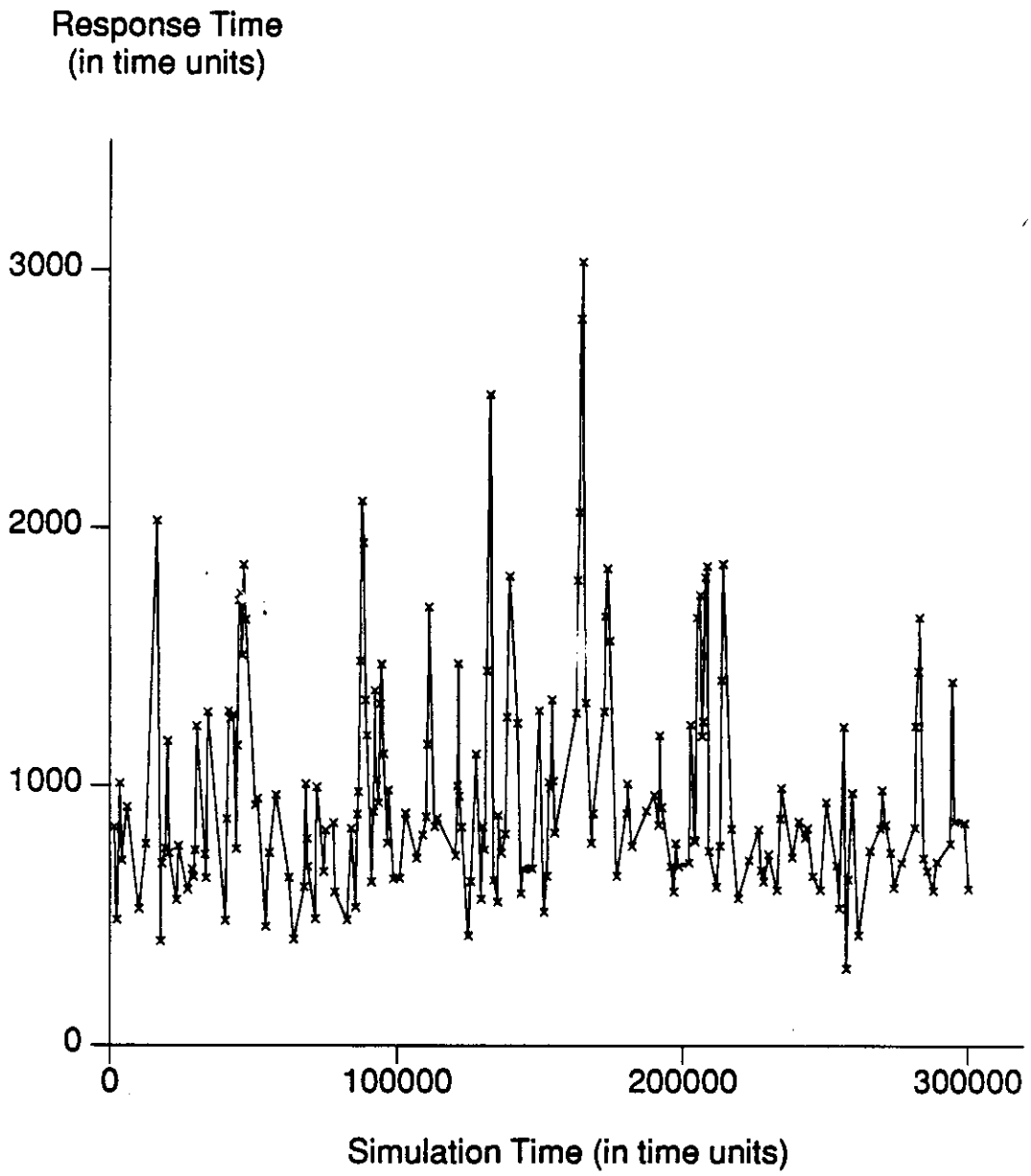


Figure 6.3: Response time of the 20 host/terminal model

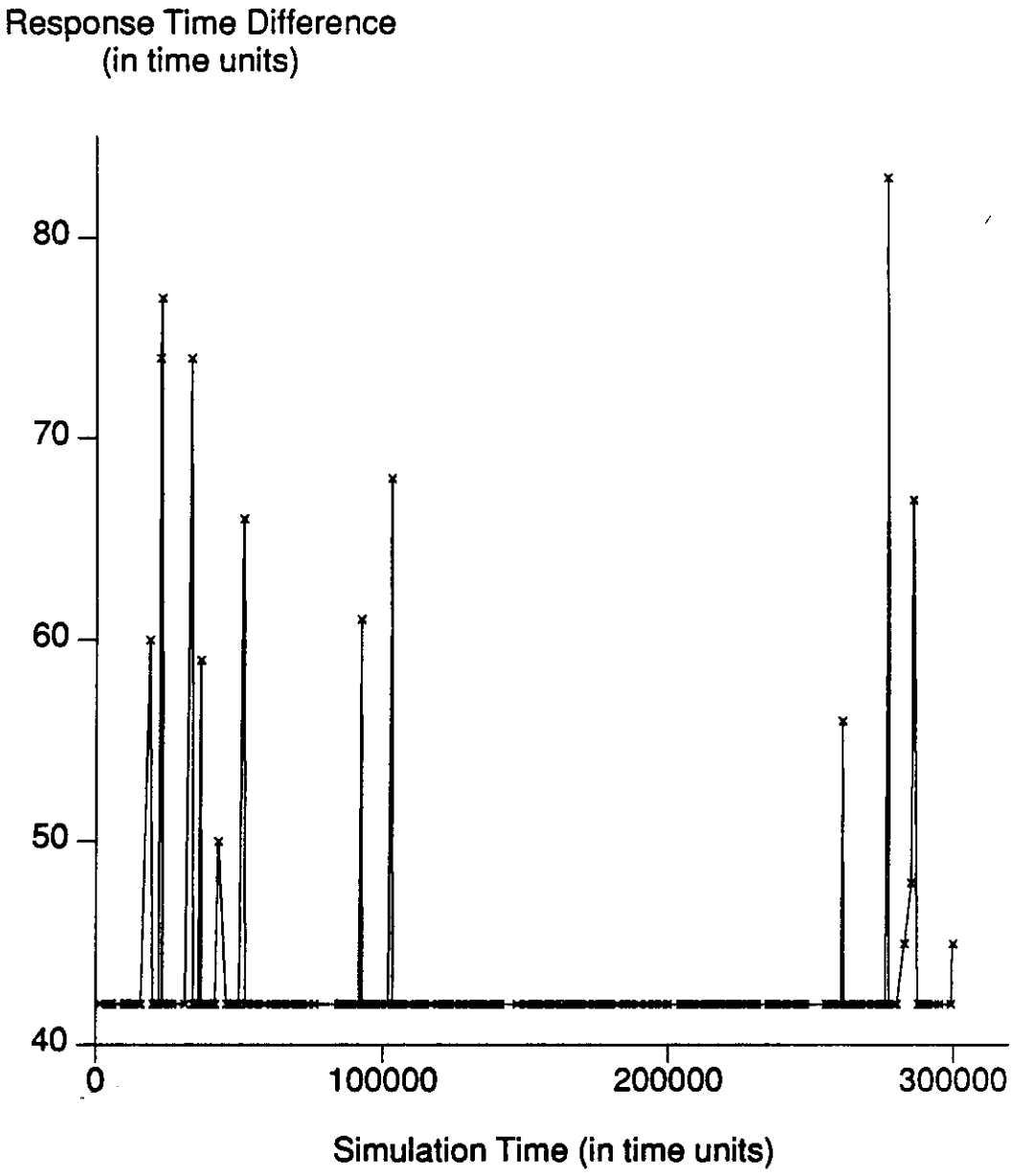


Figure 6.4: Response time difference of the 2 host/terminal model

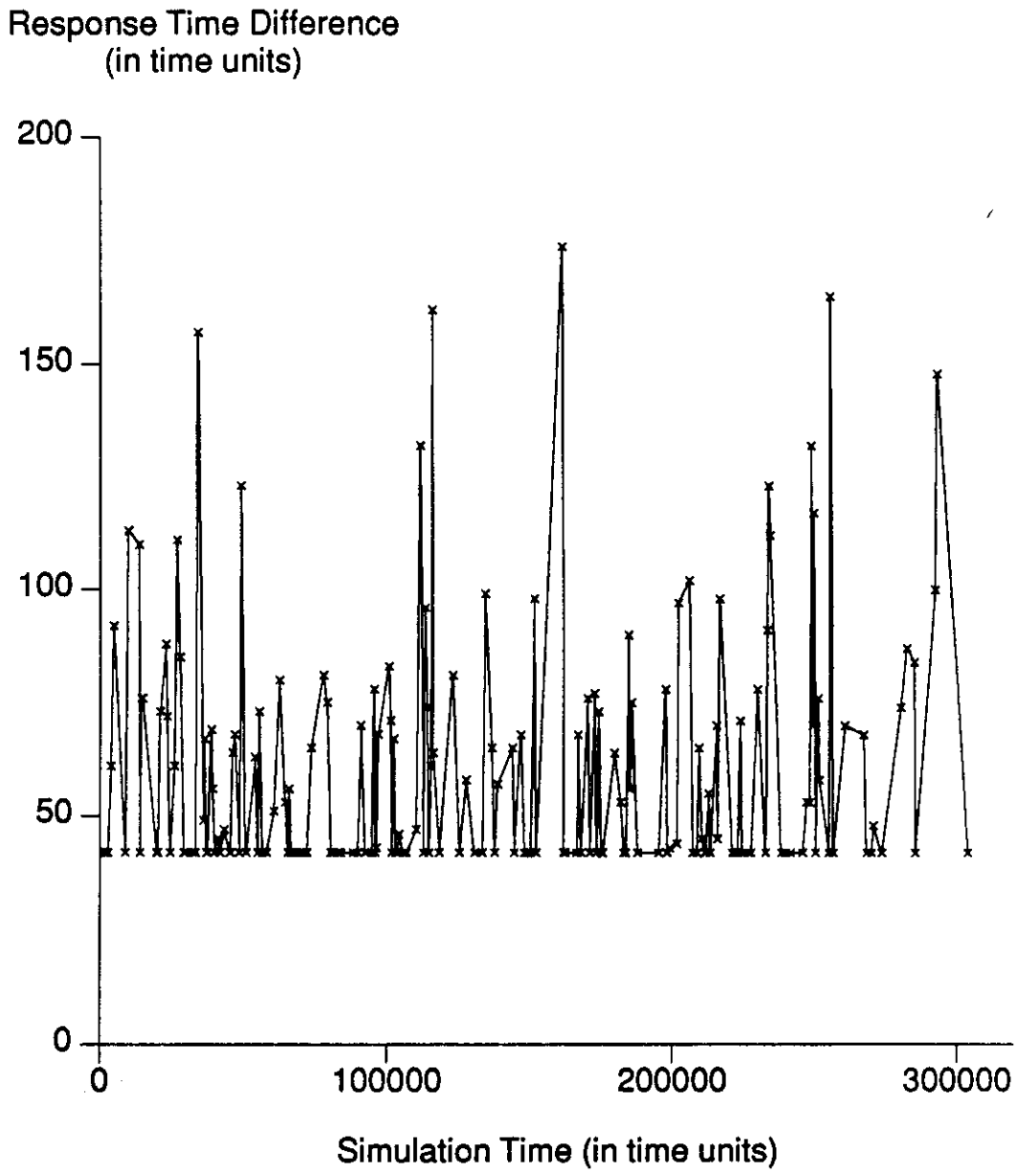


Figure 6.5: Response time difference of the 12 host/terminal model

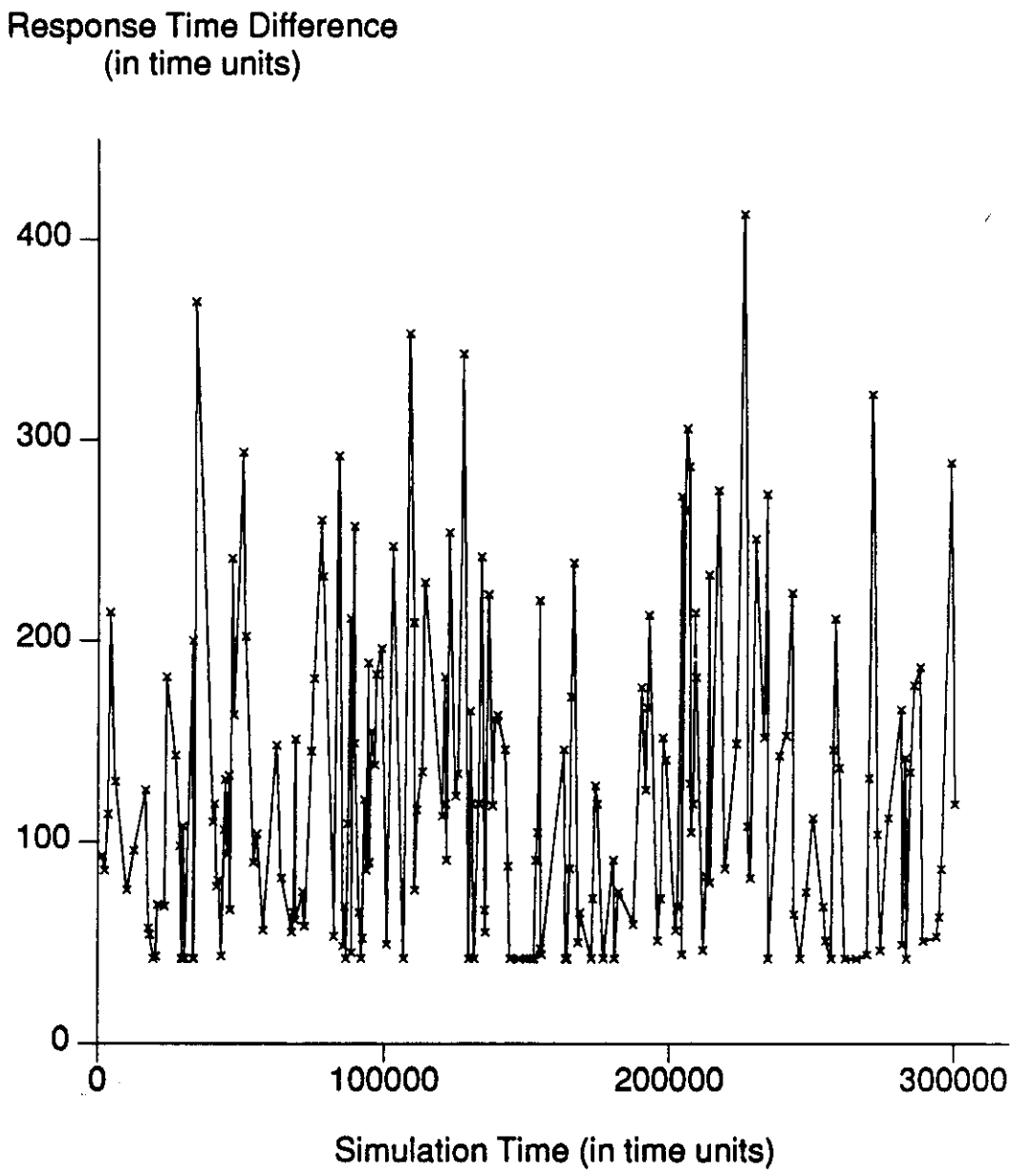


Figure 6.6: Response time difference of the 20 host/terminal model

The model with 12 host/terminal pairs has been simulated five times with different sets of random numbers. Result data from these runs are statistically independent. Therefore, we can obtain an "average" curve by averaging the first points, the second points, ... from each run. The result is plotted in Figure 6.7, in which some of the fluctuations in Figure 6.2 have been smoothed out.

6.4 Simulation Results Analysis

Frequently, a lot of result data are obtained from a simulation run (or several independent runs of the same model). The sample mean of a certain variable of interest may be calculated by taking the average of the available data. The information not provided by this average is its accuracy; that is, is it likely to be 1%, 10%, or 50% away from the actual mean? Simulation result analysis provides us a probabilistic answer to this question so that confidence intervals can be generated for the simulation result.

6.4.1 Estimating the Steady State Response Time

In discrete event simulation, when a simulation run begins, it will normally go through a *transient period* due to the influences of the initial conditions. The simulation results from this period are generally different from the long-term, or *steady state*, behaviors. For example, in the simulation of a queueing network, all of the queues are initially empty so that there will be no waiting for service. Hence the response time will tend to be short. As the simulation progresses, queues will start building up and the average waiting

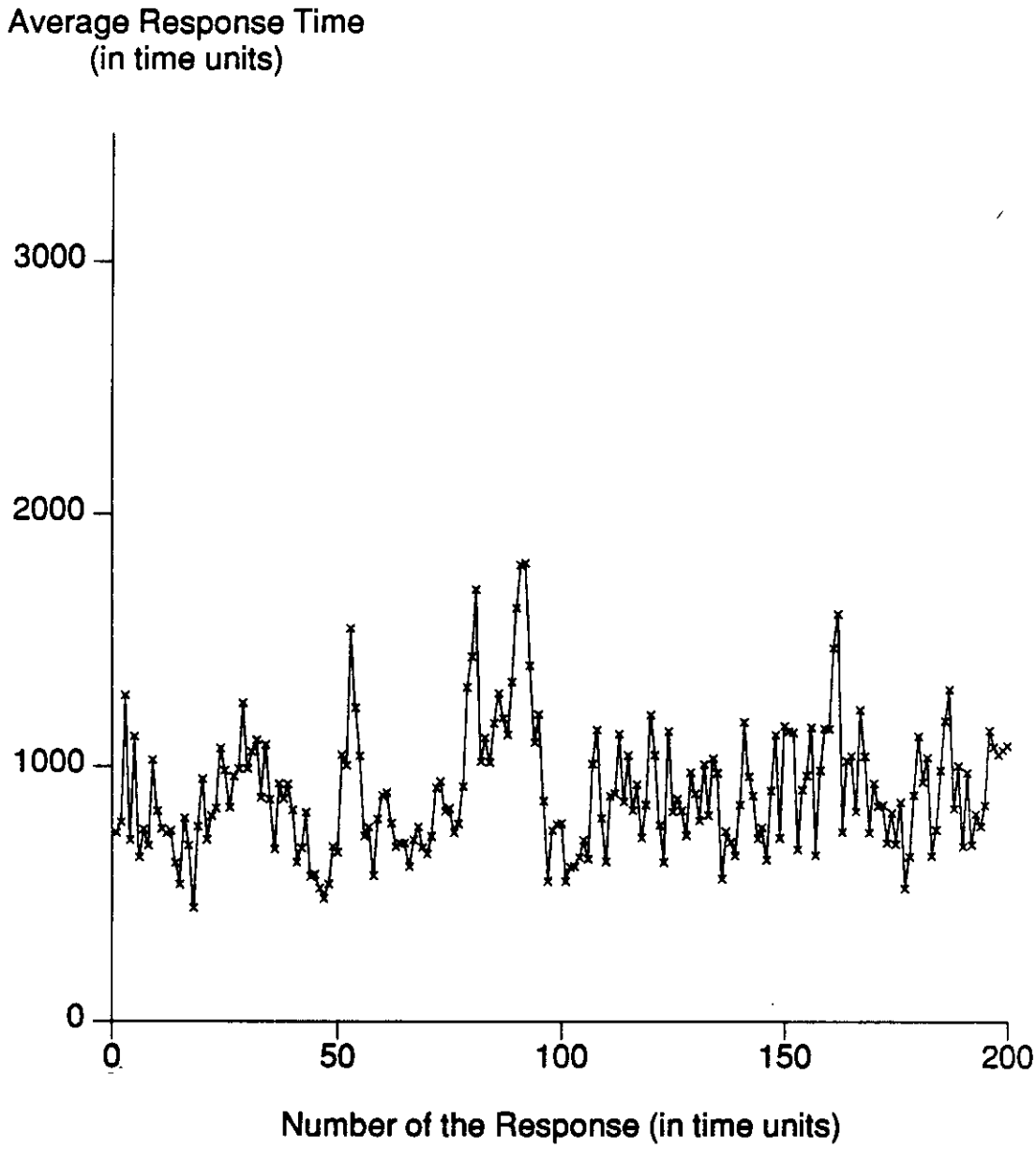


Figure 6.7: Average response time of five runs of the 12 host/terminal model

time will increase. This type of transient behavior does reflect the realistic characteristic of the actual system being simulated. After the influences of the initial conditions have subsided, in the steady state, although variables such as queue lengths, response times, etc. do vary from event to event, their respective probability distributions will remain the same (unless some outside influence changes in the mean time). Since an actual system operates in its steady state most of the time, the steady state behavior is usually the main factor to be measured in a performance evaluation, although various transient behaviors are often of interest as well.

To obtain the steady state behaviors, it is necessary to determine the duration of the transient period and remove the simulation result from this period so that the data from the steady state alone can be considered. As mentioned before, the influence of the initial conditions decays to a negligible level after a certain amount of time, but there is neither a clear definition nor any algorithm to determine where the border line between the two periods is. Usually, with the aid of some simulation result plots, one can make a reasonable judgment.

The estimated mean steady state response time is simply the average of all steady state response time data collected. If there are several independent simulation runs of the same model (i.e., with different series of random numbers), the estimate will be the overall average, provided that transient data are removed from each run.

6.4.2 Generating Confidence Intervals

Let $X_1, X_2, \ldots X_n$ be statistically independent random variables with the same distribution, mean m and standard deviation σ. The sample mean $\hat{m}$ is

$$\hat{m} = \frac{\sum_{i=1}^{n} X_i}{n}$$

and the sample standard deviation $\hat{\sigma}$ is

$$\hat{\sigma}^2 = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \hat{m})^2$$

According to the Central Limit Theorem [Fell68, Welc83], the random variable

$$\frac{(\hat{m}-m)\sqrt{n}}{\hat{\sigma}}$$

will have approximately a t-distribution with n−1 degrees of freedom (which is approximately the same as a normal distribution when $n \geq 25$). Hence

$$Prob. \left[t_{n-1}\left[\frac{\alpha}{2}\right] \leq \frac{(\hat{m}-m)\sqrt{n}}{\hat{\sigma}} \leq t_{n-1}\left[1-\frac{\alpha}{2}\right] \right] \approx 1-\alpha$$

$$Prob. \left[\hat{m} - t_{n-1}\left[\frac{\alpha}{2}\right]\frac{\hat{\sigma}}{\sqrt{n}} \leq m \leq \hat{m} + t_{n-1}\left[1-\frac{\alpha}{2}\right]\frac{\hat{\sigma}}{\sqrt{n}} \right] \approx 1-\alpha \qquad (6.1)$$

where α specifies the desired probability, or confidence interval. The values of the t-distribution have been tabulated [Law82]. For example, when n is large, $t_n(0.95) = 1.64$ (for 90% confidence intervals) and $t_n(0.975) = 1.96$ (for 95% confidence intervals). Hence once the sample mean $\hat{m}$ and standard deviation $\hat{\sigma}$ for the statically independent data are available, the confidence intervals can be

calculated.

Unfortunately, this method cannot be applied directly to the simulation result data because the response times are not statistically independent. For example, when there is a long queue of commands waiting to be processed by a host, it is very likely that several commands in a row will be affected and all have very long response times. Similarly, if there is no waiting at a host, consecutive commands will probably have short response times. Because of these dependencies, the sample standard deviation $\hat{\sigma}$ will be smaller than that for independent data and the calculated confidence intervals will be too optimistic (narrow). In order to apply the the t-distribution method, it is necessary to obtain statistically independent data. There are a number of available schemes such as *the Method of Independent Replications*, *the Method of Batch Means* and *the Regenerate Method* for this purpose [Welc83, Law82, Law83].

The Method of Independent Replications requires multiple simulation runs using the same model but different series of random numbers which are statistically independent. Hence the steady state result from different runs will have the same probability distribution and are also statistically independent. Therefore they may be analyzed using the t-distribution method. However, a certain amount of data has to be deleted from each run to remove the transient result. Moreover, this method cannot be used to control the length of a simulation run according to the accuracy of the simulation data (obtained so far from the current run).

The Regenerative Method uses regenerative states to separate groups of data which are independent from one another. A state of the model which appears frequently and can be detected easily in the simulation should be chosen as the regenerative state. For example, the state in which every queue in the model is empty is a possible choice. Because of the Markovian characteristics, the expected future behavior of the simulation will be the same every time it reaches the regenerative state and is independent of its past history. Hence the simulation result data collected in the period between every two consecutive regenerative states will have the same distribution but are statistically independent from data collected in other periods. The t-distribution method can therefore be applied to them. The main difficulty with the Regenerative Method is to select a suitable regenerative state and detect its occurrence. For asynchronous distributed simulation, since the objects (or processors) progress at different rates, there do not exist any snapshots of states relating to the simulation time. It therefore becomes even by far more difficult, if not impossible, to detect regenerative states. Hence we can conclude that this method is not applicable in this case.

The Method of Batch Means is similar to the Method of Independent Replications, but the statistically independent data sets come from different batchs in a very long simulation run rather than from several runs. The advantage is that only one set of transient data needs to be deleted. Since simulation result dependency is limited to neighboring data in a certain region, data which are very far apart may be considered to be independent. Hence the results from a long simulation run can be grouped into statistically independent

batches, and the t-distribution method may then be applied to data in different batches. The difficulty is to determine the range of data dependency. One can use plots of simulation results from pilots runs to make estimates.

When result data are available in batches which are statistically independent, the confidence intervals can be determined in the following manner [Welc83]. Assume that there are B sets of statistically independent data, and each set contains n data points. Let the sample mean of batch b be $\hat{m}_b$; these means are all independent for b = 1, ..., B. The unbiased estimate of the mean response time m is:

$$\hat{m} = \frac{1}{B}\sum_{i=1}^{B}\hat{m}_i$$

The sample variance of $\hat{m}_b$ is:

$$s^2(\hat{m}_b) = \frac{1}{B-1}\sum_{i=1}^{B}(\hat{m}_i-\hat{m})^2$$

Since the m_b's are independent,

$$\frac{(\hat{m}-m)\sqrt{B}}{s(\hat{m}_b)}$$

has approximately a t-distribution with B–1 degrees of freedom, and the confidence interval can be determined using (6.1).

From Figure 6.7 (and other figures for different cases), we realize that the transient period is very short for this particular model; the simulation enters the steady state almost immediately. Therefore, only a few result data need to be deleted to remove the transient behavior. The result data from the

simulations with the models listed in Section 6.3 were analyzed using the Batch Means method. According to Figures 6.1 to 6.3, we realize that the range of data dependency is fairly short; only a few neighboring data on each side are within the range of dependency. In each case, 600 to 800 data are generated; some of the initial data (about 50) are deleted to eliminate any possible transient effects. The data are then grouped into 10 batches. Each batch therefore contains 60 to 80 data, which is sufficient to fulfill the independence assumption.

The result from the analysis is shown in Table 6.2 (average response time of the last packet) and 6.3 (average response time difference between the first and the last packets). The average response times for the last packet is plotted against the number of host/terminal pairs in Figure 6.8. Although the response time does increase as the number of host/terminal pairs grows, the difference between the 2-pair case and the 22-pair case is relatively small. There are some fluctuations too. For example, according to the simulation result, the average response time for the 4-pair case is even smaller than that for the 2-pair case. These fluctuations are probably due to the fluctuation of the host processing time which affects the host waiting time and dominates the response time.

Table 6.2: Average Response Times (in time units)			
# of Host/Term Pairs	Ave. Resp. Time	90% Confidence Interval	
		Upper Limit	Lower Limit
2	897.7	967.8	827.6
4	841.2	877.1	805.3
8	954.5	1019.1	890.0
12	919.9	983.1	856.7
16	949.4	1023.2	875.5
20	1019.2	1120.3	918.2
22	1013.0	1086.9	939.1

Table 6.3: Average Differences between Response Times (in time units)			
# of Host/Term Pairs	Ave. Difference	90% Confidence Interval	
		Upper Limit	Lower Limit
2	43.8	44.2	43.3
4	46.9	47.8	46.1
8	56.3	58.4	54.1
12	66.0	69.0	63.0
16	89.7	93.8	85.6
20	125.4	133.3	117.6
22	161.8	181.2	142.3

To provide a better perspective of the relation between the amount of traffic and response time, we plotted the average response time difference between the last and first packets against different number of host/terminal pairs (i.e., the data in Table 6.3) in Figure 6.9. With the host processing time canceled, the response time difference is mainly due to network delay. It

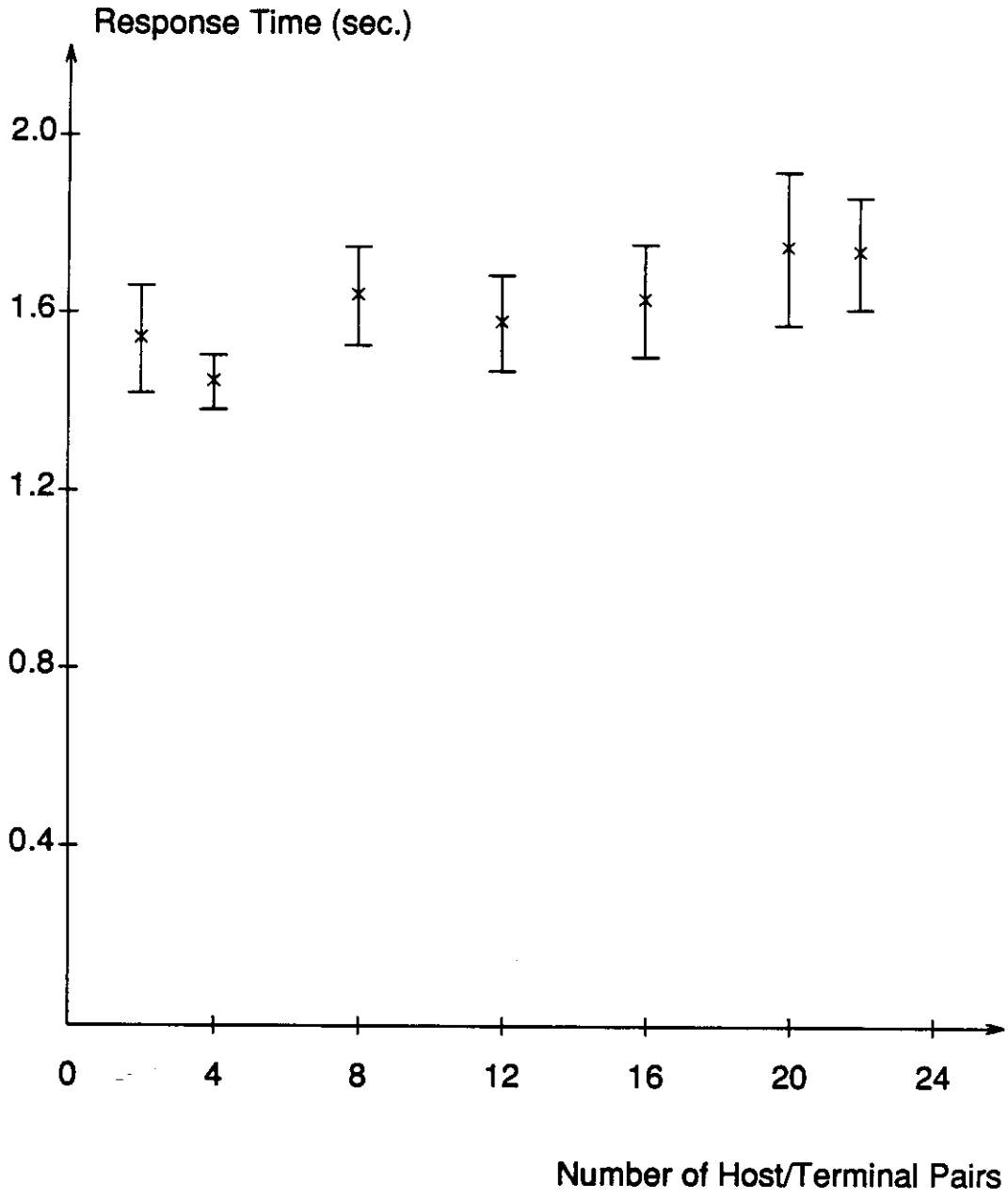


Figure 6.8: Average response times

increases exponentially as the load (number of host/terminal pairs) increases. According to this exponential growth, it is not surprising that the response time for the 24 host/terminal pair case grows to infinity in the simulation. The average response time difference in Table 6.3 provides information concerning the load. When two hosts are sharing the network, there is very little network delay. When there are 20 hosts, the average difference is three times the minimum. That is, on the average, it appears as if three hosts are continuously sharing the network. This factor grows to four when there are 22 hosts.

Simulation results from the five independent runs with the 12 host/terminal model are also analyzed using the Method of Independent Replications. The sample mean response time is 944.7 time units, and the mean response time difference is 66.8 time units, which are very close to the results in Tables 6.2 and 6.3.

6.4.3 A Simple Analytical Model

To verify the simulation results, we use the Pollaczek-Khinchin mean-value formula [Klei75] to calculate the average system time at the host of the 1 terminal/host case. Since there is no network sharing, the average system time at the host will be the same as the response time of the first packet in the response, assuming that the transmission time on the network is negligible. The formula for average system time is:

$$\frac{T}{\bar{x}} = 1 + \rho \frac{(1+C_b^2)}{2(1-\rho)}$$

where T is the average system time, $\bar{x}$ is the average service time, and C_b^2 is the

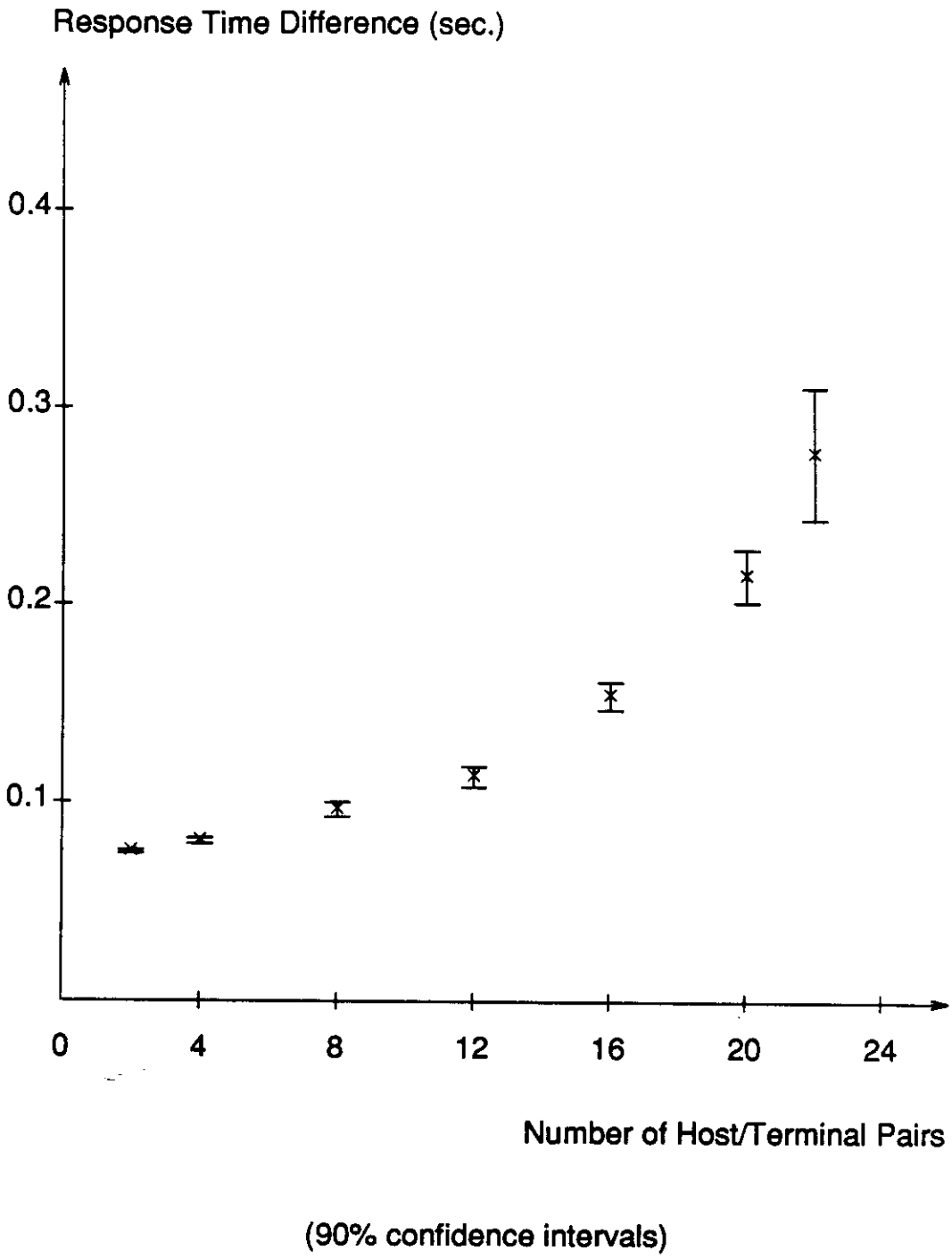


Figure 6.9: Response times difference between the first and last packets.

coefficient of variation of the service time.

In our model, $\bar{x} = 1$ sec., $\rho = 0.5$, and $C_b = 0.25$. Hence

$$T = \bar{x}\left[1 + \rho\,\frac{(1+C_b^2)}{2(1-\rho)}\right]$$

$$= (1)\left[1 + (0.5)\,\frac{(1+0.25^2)}{2(1-0.5)}\right]$$

$$= 1.53 \text{ sec. } (892.5 \text{ time units})$$

The response time for the last packet is therefore 892.5 + 42 = 934.5 time units. This result is reasonably close to the response time of the 2 host/terminal model (from the simulation), which should be approximately the same as the one host/terminal case.

6.5 Performance of the Distributed Simulator

The experiments listed in Section 6.3 were executed as distributed processes on one computer as well as multiple computers (Three computers were used in the experiments.). The simulator software is identical in both cases, except that the processes are migrated to different sites on the distributed operating system when multiple computers were used. The processing powers of these computers are combined so that the execution should be faster than on one computer, but it requires communication across machines, which introduces additional overheads. When the processes are executed on only one computer, they are time sharing it so that the simulation is actually carried out

sequentially. This is of course a very undesirable arrangement because it still involves the overheads in a distributed simulator (such as inter-process communication, synchronization, etc.) but cannot gain any advantage from distributed processing. However, for experimental purposes, the correctness of the simulator can be verified in either case and the efficiency of the two can be compared. When the processes are executed on one computer, a typical run with the 12 host/terminal model takes about eight hours on one VAX 11-750 to simulate the events up to one million time units (about ½ hour of simulation time), and it needs 12 hours for the 20 host/terminal model. When the simulation is distributed on three computers, the execution actually takes somewhat longer. This result suggests that the inter-computer communication overhead is very large and overshadows the advantage of multi-processing. This is not a very surprising result considering the distributed environment used.

We did not carry out extensive studies concerning the performance of the distributed simulator here for three reasons. (1) The computer system used is time shared by other users and jobs so that the execution time varies depending on the overall load. (2) The version of LOCUS on which the simulations were executed was an experimental version. The performance was not yet optimized. (3) In the simulator software itself, there are a lot of error checking statements to detect possible bugs. All of these problems affect the performance of the simulator. Moreover, according to the comparison between the one-computer and three-computer cases, the main problem with the performance is the communication overhead. Since neither LOCUS, the

distributed operating system the simulator is running on, nor the network of computers was developed for communication-intensive distributed processing, their support to some of the functions required by this application is limited. For example, they do not handle low-level inter-process communication efficiently and also provide no convenient functions to check whether there are messages available to be read at an input or an output has room for a leaving message. These functions are very important for the deadlock prevention mechanism. Currently, the simulator has to use some very inefficient methods to resolve these problems. Although LOCUS does not support enough features to carry out distributed simulation of data communication networks efficiently, it does provide a good environment to verify the feasibility of distributed simulation and the correctness of the methodologies developed.

= 1.53 sec. (892.5 time units)

The response time for the last packet is therefore 892.5 + 42 = 934.5 time units. This result is reasonably close to the response time of the 2 host/terminal model (from the simulation), which should be approximately the same as the one host/terminal case.

6.5 Performance of the Distributed Simulator

The experiments listed in Section 6.3 were executed as distributed processes on one computer as well as multiple computers (three computers were used in the experiments). The simulator software is identical in both cases, except that the processes are migrated to different sites on the distributed operating system when multiple computers were used. The processing powers of these computers are combined so that the execution would in principle be faster than on one computer, except for the obvious additional overhead for interprocess communication across machines. When the processes are executed on only one computer, they are in effect time sharing the machine so that the simulation is actually carried out sequentially. This is, of course, not an arrangement useful in practice because it still involves the overheads in a distributed simulator (such as inter-process communication, synchronization, etc.) but cannot gain any advantage from distributed processing. However, this configuration is useful for experimental purposes, since the correctness of the simulator can be verified in either case and the efficiency of the two can be compared. When the processes are executed on one computer, a typical run

with the 12 host/terminal model takes about eight hours on one VAX 11-750 to simulate the events up to one million time units (about ½ hour of simulation time), and it needs 12 hours for the 20 host/terminal model. When the simulation is distributed on three computers, the execution time is comparable (actually slightly longer). This result suggests that, as expected, the inter-computer communication overhead is very large and counteracts gains due to parallelism in processing. This is not a very surprising result considering that the distributed environment used is a general-purpose one, rather than one dedicated to a particular function such as simulation, and that our process synchronization methods are embedded in our application code rather than being provided by a dedicated special-purpose operating system. These were deliberate decisions in our project; as stated at the outset, the goal was to demonstrate the feasibility of utilizing such a general-purpose computing environment for our application, rather than seeking to tailor the environment to the problem. Under these circumstances, our observation of comparable running times for central and distributed execution is a positive result, since it suggests that, even in the absence of optimization or tailoring, the basic mechanisms implemented are adequate to trade off communication delay and parallel-processing power on a roughly comparable footing, a reasonable criterion in demonstrating feasibility.

We did not carry out extensive studies concerning the performance of the distributed simulator here for three reasons. (1) The computer system used is time shared by other users and jobs so that the execution time varies depending on the overall load and is quite difficult to instrument for repeatable

experimentation. (2) The version of LOCUS on which the simulations were executed was an experimental version; its performance was not yet optimized. (3) In the simulator software itself, many error checking statements have been incorporated, as is appropriate in an experimental development; this slows execution, of course. Each of these points reflects itself in the performance of the simulator. Moreover, according to the comparison between the one-computer and three-computer cases, the main problem with performance is the communication overhead, and neither the LOCUS software nor the network hardware was developed for support or measurement of communication-intensive distributed processing. For example, low-level inter-process communication is not necessarily efficiently handled, and also there is little provision for convenient functions to check whether there are messages available to be read at an input or an output has room for a leaving message. These functions are very important for implementing the deadlock prevention mechanism. Currently, our simulator has embedded within it some necessarily inefficient methods to resolve these problems, which would otherwise receive higher-level support through operating system calls. Although LOCUS does not currently provide enough features to carry out our distributed simulation efficiently, it does provide a good environment in which to verify the feasibility of distributed simulation and the correctness of the methodologies developed, and this in turn suggests that, with emerging enhancements in distributed processing support, general-purpose distributed operating systems can potentially be utilized for this purpose in the near future.

CHAPTER 7

CONCLUSIONS

7.1 Background and Scope

In view of the increasing demand on performance evaluation of data communication networks through digital computer simulation and the gradual popularity of networks of micro-computer workstations as a new computing environment, it is reasonable and necessary to consider carrying out simulation in this type of new environments. Problems such as model specification and representation, simulation load partitioning and allocation, synchronization among processors, and congestion control must be solved before distributed simulation can be realized. There is much work on these problems is general settings, and assuming that specialized architectures will be available. The research described in this dissertation is directed toward the development of a distributed simulator, accepting models represented as queueing networks, for a ring-type data communication network in particular, seeking an integrated solution which is implementable and extensible on general-purpose systems, and embedding needed mechanisms in the application code where the general-purpose system does not specifically provide high-level support.

7.2 Achievements and Contributions

Before a simulation run can begin, it is important to prepare an accurate model in the correct format for input to the simulator. To solve the model specification problem, we have implemented an interactive program which prompts a user for the necessary parameters. This program greatly reduces the probability of neglecting important information or using meaningless parameters in the simulation model.

Several synchronization methods for asynchronous distributed simulation are already available. We have developed a number of criteria to evaluate their respective merits and then carried out careful studies and comparisons among these methods. We concluded that the roll back scheme in the Time Warp method [Jeff85a] is the most promising approach because it (1) can exploit the maximum amount of inherent concurrency, (2) introduces relatively few extra message transfers, and (3) requires no tight interactions with a central controller. We have therefore adopted this synchronization method in our simulator. A significant portion of recent research in Time Warp is directed toward the implementation and future extension of a special-purpose distributed operating system for discrete-event simulation [Jeff85c]. Application programs are executed on top of this operating system on a special-purpose multi-processor system. In our work, the synchronization mechanism is built into the application program, and it runs on a general-purpose distributed operating system which provides low-level support for distributed processing. Hence we are able to consider special properties in the simulation models and achieve gains in simplicity by adapting the state save, roll back, and recovery schemes for simulation of communication networks. A

References

[Brya77] Bryant, R.E., "Simulation of Packet Communication Architecture Computer Systems," MIT, Cambridge, Massachusetts, Tech. Rep. MIT/LCS/TR-188, November, 1977.

[Chan79] Chandy, K.M. and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transaction on software Engineering*, Vol. SE-5, No. 5, September, 1979, pp. 440-452.

[Chan81] Chandy, K.M. and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Comm. ACM*, Vol. 24, No. 4, April, 1981, pp. 198-206.

[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.

[Fell68] Feller, W., *An Introduction to Probability Theory and Its Applications, third edition*: John Wiley & Sons, 1968.

[Gafn85] Gafni, A., "Space Management and Cancellation Mechanism for Time Warp," University of Southern California, Tech. Rep. TR-85-341, December 1985.

[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.

[Kirk80] Kirkpatrick, S. and E. Stoll, "A Very Fast Shift-Register Sequence Random Number Generator," IBM Thomas J. Watson Research Center, Yorktown Heights, New York, Tech. Rep. RC 8180 (#35055), January 21, 1980.

[Klei75] Kleinrock, L., *Queueing Systems, Volume I: Theory*, New York: John Wiley and Sons, 1975.

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

[Knut81] Knuth, D.E., *The Art of Computer Programming, volume two: Seminumerical Algorithms*: Addison-Wesley Publishing Company, 1981.

[Kris85] Krishnamurthi, M., U. Chandrasekaran, and S. Sheppard, "Two Approaches to the Implementation of a Distributed Simulation System," in *Proceedings of the 1985 Winter Simulation Conference*, San Francisco, California: December, 1985.

[Law82] Law, A.M. and W.D. Kelton, *Simulation Modeling and Analysis*: McGraw-Hill Book Company, 1982.

[Law83] Law, A.M., "Statistical Analysis of Simulation Output Data," *Operations Research*, Vol. 31, No. 6, November-December, 1983, pp. 983-1029.

[Misr86] Misra, J., "Distributed Discrete-Event Simulation," *ACM Computing Surveys*, Vol. 18, No. 1, March, 1986.

[Peac79] Peacock, J.K., J.W. Wong, and E.G. Manning, "Distributed Simulation Using a Network of Processors," *Computer Networks*, Vol. 3, No. 1, February, 1979, pp. 44-56.

[Pier72] Pierce, J.R., "Network for Block Switching of Data," *Bell System Technical Journal*, Vol. 51, No. 6, July-August 1972, pp. 1133-1175.

[Pope85] Popek, G. and B. Walker ed., *The LOCUS Distributed System Architecture*, Cambridge, Massachusetts: The MIT Press, 1985.

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

[Seet79] Seethalakshmi, M., *A Study and Analysis of Performance of Distributed Simulation*: Master thesis, University of Texas at Austin, May, 1979.

[Shep85] Sheppard, S., U. Chandrasekaran, and K. Murray, "Distributed Simulation Using Ada," in *Proceedings of the Conference on Distributed Simulation 1985*, San Diego, California: Society for Computer Simulation, January, 1985.

[Welc83] Welch, P.D., "The Statistical Analysis of Simulation Results," in *Computer Performance Modeling Handbook*, S.S. Lavenberg, Ed. New York: Academic Press, 1983.

[Wyat85] Wyatt, D.L., "Simulation Programming on a Distributed System: a Preprocessor Approach," in *Proceedings of the Conference on Distributed Simulation 1985*, San Diego, California: Society for Computer Simulation, January, 1985.