

AD-A151 287

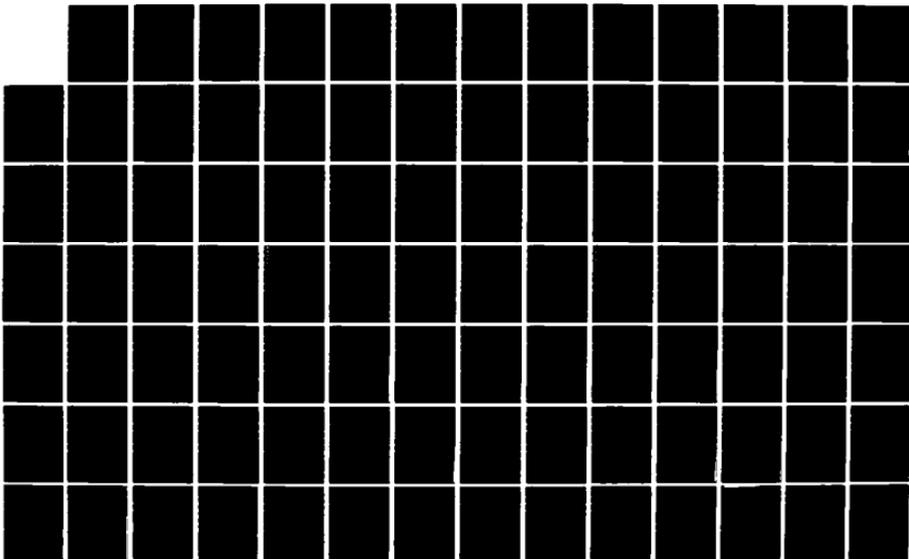
ANNUAL SCIENTIFIC REPORT GRANT AFOSR-81-0205(U) TEXAS
UNIV AT AUSTIN DEPT OF COMPUTER SCIENCES
K M CHANDY ET AL. DEC 84 AFOSR-TR-85-0200 AFOSR-81-0205

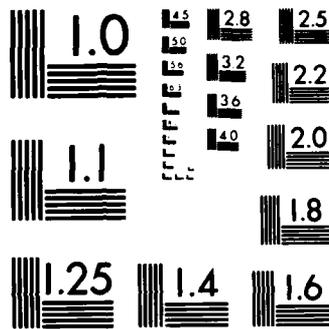
1/2

UNCLASSIFIED

F/G 12/1

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(4)

AD-A151 287

AFO SR ANNUAL REPORT
 AFOSR 81-0205
 6/15/83 - 6/14/84
 K. M. CHANDY AND J. MISRA



Approved for public release; distribution unlimited.

DEPARTMENT OF COMPUTER SCIENCES
THE UNIVERSITY OF TEXAS AT AUSTIN

AUSTIN, TEXAS 78712

ATAC FILE COPY

DIR
 MAR 12 1985
 A

85 02 27 047

(4)

AFOSR ANNUAL REPORT

AFOSR 81-0205

6/15/83 - 6/14/84

K. M. CHANDY AND J. MISRA

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFOSR)

NOTICE OF TRANSMITTAL TO DTIC

This technical report has been reviewed and is approved for public release under ESR 190-12.

Distribution is unlimited.

MATTHEW J. KERPER

Chief, Technical Information Division

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE		4. PERFORMING ORGANIZATION REPORT NUMBER(S)	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)		5. MONITORING ORGANIZATION REPORT NUMBER(S) AFOSR-TR- 85-0200	
6a. NAME OF PERFORMING ORGANIZATION University of Texas	6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Air Force Office of Scientific Research	
6c. ADDRESS (City, State and ZIP Code) Dept of Computer Sciences Austin TX 78712		7b. ADDRESS (City, State and ZIP Code) Directorate of Mathematical & Information Sciences, Bolling AFB DC 20332-6448	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFOSR	8b. OFFICE SYMBOL (If applicable) NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER AFOSR-81-0205	
8c. ADDRESS (City, State and ZIP Code) Bolling AFB DC 20332-6448		10. SOURCE OF FUNDING NOS.	
		PROGRAM ELEMENT NO. 61102F	PROJECT NO. 2304
		TASK NO. A2	WORK UNIT NO.
11. TITLE (Include Security Classification) ANNUAL . SCIENTIFIC REPORT, GRANT AFOSR-81-0205, 15 JUNE 1983 - 14 JUNE 1984			
12. PERSONAL AUTHOR(S) K.M. Chandy and J. Misra			
13a. TYPE OF REPORT ANNUAL	13b. TIME COVERED FROM 15/6/83 TO 14/6/84	14. DATE OF REPORT (Yr., Mo., Day) DEC 84	15. PAGE COUNT 145
16. SUPPLEMENTARY NOTATION			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB. GR.	
19. ABSTRACT (Continue on reverse if necessary and identify by block number) Work in the last year has continued in the direction of constructing unifying frameworks for the study of distributed systems and algorithms. The authors have previously reported work which unified all known detection algorithms (Distributed Snapshots: Determining Global States of Distributed Systems, by Chandy and Lamport) and unified treatment of conflict resolution (Drinking Philosophers Problem, by Chandy and Misra). They have recently developed a general theory for studying computability issues in asynchronous distributed systems so that they may be studied. The authors have obtained simple proofs for important results such as, "there is no robust commit protocol," or "common knowledge cannot be achieved in an asynchronous system." In a similar vein, the authors now have a paradigm for the development of a very efficient stability detection algorithm (Deriving Properties of Distributed Systems by Overlapping Monitoring, by Chandy and Misra).			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS <input type="checkbox"/>		21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL CPT John P. Thomas, Jr		22b. TELEPHONE NUMBER (Include Area Code) (202) 767- 5026	22c. OFFICE SYMBOL NM

DD FORM 1473, 83 APR

EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED

85 . 02 27 047

SECURITY CLASSIFICATION OF THIS PAGE

Annual Report for AFOSR 81-0205

1. Objectives

We have now been active for several years in the area of distributed systems. It has become apparent to us that this subarea of parallel programming or concurrent programming systems is tractable: a precise theory for distributed, message passing computations may be developed; important paradigms can be abstracted and applied in a number of practical situations and reasoning techniques can be developed for distributed programs which can also be effectively employed in their developments.

We have been active in all these areas. Our goal is to make distributed programming, conceptually as simple as sequential programming. The added burden of distribution could be handled if adequate general theories were available. We have had experience in developing specific algorithms (deadlock detection, knot detection, shortest path etc.); theories specific to a class of problems and reasoning techniques applicable to a class of properties of distributed programs.

Our thrust of research in the past year has been to move from specific to general, by abstracting the relevant concepts from specific problems and applying them to a general class. Our outstanding contributions in the past year are:

- development of a general theory for studying computability issues in distributed systems,
- a paradigm for development of very efficient stability detection algorithms,
- extension of our proof theory to encompass a wider variety of properties that can be proven.

We have continued our, very successful, work on modeling and distributed simulation.

Our work has attracted considerable international attention. Professor K. M. Chandy was invited to deliver the keynote address at the ACM Principles of Distributed Computing Conference, the premier conference for this area, in 1984; he was also invited to give talks at M.I.T., Stanford, Cornell, University of California at Berkeley, University of Minnesota, Pennsylvania State University, IBM Research at Yorktown Heights, and Computer Society of India. Professor J. Misra was made a member of the prestigious International Federation of Information Processing Working Group (IFIP WG 2.3), member of the Editorial Board of the Journal of the ACM; he delivered invited talks at the University of California at Berkeley, University of California at Los Angeles, Cal Tech, University of Washington, IBM Research at

San Jose, Xerox Palo Alto Research Center and at several workshops.

We sketch the technical aspects of our recent work in the following pages.

2. Work in 83-84

2.1. A Computability Theory for Asynchronous Distributed Systems

Sequential systems had a well-developed theory, founded in logic and developed by Church, Turing, and Godel and others [5], before any sequential program was ever executed on a computer. Existence of important theoretical results, such as the unsolvability of the *halting problem*, guided programming practitioners. The situation is entirely different in distributed systems; there is no well-founded theory of computability in distributed systems. Hence considerable effort has been expended on solving problems which have later been shown to be unsolvable; designing a distributed database commit protocol which is robust for process failures is only one instance [3].

In recent years a number of results of the following form have appeared in the literature: there is no asynchronous distributed algorithm for solving problem P , where P is a specific problem. For instance, it is now known that: it is impossible to elect a unique leader process from among a set of identical processes [1]; there is no symmetric algorithm for solving the dining philosophers problem [6]; it is impossible to implement a commit protocol for distributed databases in the presence of even one faulty process [3]; there is no solution to the Byzantine agreement problem in a fully asynchronous system [2]; and no protocol exists for achieving common knowledge in an asynchronous system [4]. In each case, ad hoc techniques have been used in proving these results. Absence of a common computability theory for distributed systems has hindered progress in this area. This is only natural, since the requisite concepts have not been developed for the foundation of such a theory. We contrast the situation with the well-developed computability theory in sequential systems. The powerful notion of *reducibility* has been applied in sequential systems in showing several problems unsolvable: if problem A is known to be unsolvable and problem A is reducible to problem B (i.e. if problem B can be solved then problem A can be solved), then problem B is unsolvable. Such an approach is attractive in that an entirely new proof of the unsolvability of B is now avoided. We have no notion of problem reducibility in distributed systems and therefore each unsolvability proof is entirely new. Secondly, there is no common, problem independent basis for showing that certain classes of problems are unsolvable; there is no commonly acceptable *halting problem* for distributed systems.

We have recently developed a theory to help solve some of these problems. The theory is based on a precise definition of distributed computations and a number of operators on these computations. The operators are projections -- study the computation of one or more process from the entire system computation -- prefix -- one computation precedes another -- union and intersection -- union being defined only for computations which are prefixes of a common computation -- and gluing -- merging of two computations. Gluing allows us to conclude properties of distributed systems in the following manner: if C_1, C_2 are system computations then $C = \text{glue}(C_1, C_2)$ is also a system computation, where the glue operation is suitably defined. If C is infeasible, viz. C allows more than one process to be in their one process to be in their critical sections simultaneously, or more than leader to be elected or a commit protocol to commit to two different values, then we may conclude that either C_1 or C_2 is not a system computation. We have applied this technique to prove a number of properties of any algorithm for mutual exclusion, electing leaders etc. We have also constructed very simple proofs of the impossibility of process failure detection, robust commit protocol, Byzantine agreement in asynchronous systems etc.

We have applied this theory to the study of knowledge and common knowledge [4]. We have derived very simple conditions for the number of messages required to establish or disestablish certain knowledge levels. For instance, if A knows that B knows that C knows fact f , where f is some fact local to c , then at least 2 message transmissions will be required in the system before c can falsify f . The bounds we have derived are tight; they can be used to prove the impossibility of achieving certain knowledge levels, such as common knowledge, because such computations can be shown to require an infinite number of messages.

There have been a large number of intuitive ideas and ad-hoc results in asynchronous distributed systems. The goal of any unifying theory is to abstract a certain kernel and provide rules for derivation of the different results. We believe that we are working towards an elegant theory with a small kernel and a small set of rules.

2.2. A Paradigm for Developing Efficient Algorithms for Stable Property Detection

It is often required to detect whether a system state has achieved stability, i.e. it is not going to change. Examples of such properties are termination, number of tokens equals zero (assuming no process creates tokens), deadlock in a subset of processes etc. In fact many important distributed algorithms can be best described in which termination is implicit. Last year, we developed an algorithm in "Distributed Snapshots: Determining Global States of Distributed Systems" by Chandy and Lamport, which allowed a process to take a snapshot or checkpoint of a distributed system during the evolution of its computation. This effectively

solved all stability detection problems. However, we recently discovered that stability detection does not require taking snapshots. There is a general paradigm for stability detection in which processes are merely observed over overlapping intervals of execution and each process reports presence or absence of activity over its interval. We show that if there is a single point common to all intervals then the system is stable if every process is inactive over the entire length of its interval.

This paradigm has led to the design of a new class of algorithms which are simpler to program and prove and which seem very efficient in their execution.

2.3. A Proof Theory Based on the Notion of Quiescence

Two general classes of properties in sequential or distributed systems are safety and liveness. In sequential programs safety properties are proven by postulating invariants and liveness properties -- one of which is program termination -- are proven by showing that a certain metric decreases in each step of program execution. The problem is much harder in distributed programs. Previously, we had developed a proof theory for verification of safety properties only. The difficulty with liveness is that termination is not a natural property for processes in a distributed system; normally, a distributed program -- such as an operating system -- never terminates. During the last year, we made some major progress in attacking the liveness problem. We identified a new property, *quiescence*, for a process in a distributed system which seems to be the natural generalization of termination in a sequential process. Roughly, a process is quiescent if it will send no messages provided it receives no messages. This is the most that can be derived from a process because a process by itself, cannot guarantee that it will receive no inputs. A network is quiescent if all component processes are quiescent. We introduced a novel technical idea which eliminated channels from quiescence consideration.

We have now developed the proof theory where we can (1) prove safety and liveness in a unified framework, (2) support hierarchical network structure, (3) develop modular process proofs and (4) construct proofs which directly map informal proofs into a formal proof in our logic. We are now experimenting with the applicability of these ideas in various difficult distributed algorithms.

2.4. Other Work

We have continued our work on distributed simulation. These ideas, first published in 79 and continuously being refined since then, have attracted wide international recognition. Professor Misra presented a 1 day tutorial on Distributed Computing at the IEEE Fourth International Conference on Distributed Computing Systems in San Francisco, California on May

18, 1984. This tutorial will probably appear in Computing Surveys.



References

1. Angluin, D., "Local and Global Properties in Networks of Processors," *Proceedings of the Symposium on Theory of Computation*, 1980.
2. Dolev, D., C. Dwork and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," *24th Annual Symposium on Foundations of Computer Science*, November 1983.
3. Fischer, M. J., N. A. Lynch and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," MIT/LCS/TR-282, Laboratory for Computer Science, M.I.T., September 1982.
4. Halpern, J. and Y. Moses, "Knowledge and Common Knowledge in a Distributed Environment," *Proceedings of the 3rd SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1984.
5. Manna, Z., *Mathematical Theory of Computation*, McGraw-Hill Publisher, 1974.
6. Rabin, M. and D. Lehmann, "On the Advantages of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem," *Proceedings of the 8th Symposium on Principles of Languages*, 1981.

List of Publications (since Annual Report of September 1983)

1. "Paradigms for Distributed Computing," Invited paper Third Conference on Foundations of Software Technology and Theoretical Computer Science, December 12-14, 1983, Bangalore, India.
(Mani Chandy)
2. "Distributed Simulation," Tutorial presented at the IEEE Computer Society 4th International Conference on Distributed Computing Systems, May 14-18, 1984, San Francisco, California. Also submitted to Computing Surveys.
(Jayadev Misra)
3. "Processor Queueing Disciplines in Distributed Systems," Proceedings of the 1984 ACM SIGMETRICS Conference on Measurement, and Modeling of Computer Systems, August 21-24, 1984, Cambridge, Massachusetts.
(Elizabeth Williams)
4. "The Effect of Queueing Disciplines on Response Times in Distributed Systems," Proceedings of the 1984 International Conference on Parallel Processing, August 22-24, 1984, Bellaire, Michigan.
(Elizabeth Williams)
5. "Quiescence: Specification of Termination for Nonterminating Processes," Springer Verlag Lecture Notes in Computer Science, (ed. K. R. Apt), to appear.
(K. M. Chandy and J. Misra)
6. "Deriving Properties of Distributed Systems by Overlapped Monitoring," in preparation.
7. "A Framework for Studying Capabilities of Distributed Systems," in preparation.

Update to Last Year's Annual Report (see attached - September 1983)

Item 3: Preserving Asymmetry by Symmetric Processes and
Distributed Fair Conflict Resolution

Status: New Title: Drinking Philosophers Problem
ACM Transactions on Programming Languages & Systems
October 1984

Item 4: A Distributed Procedure to Detect AND/OR Deadlock,

Status: Withdrawn from consideration

Item 5: Detecting Stability in Distributed Systems,

Status: New Title: Distributed Snapshots: Determining
Global States of Distributed Systems

ACM Transactions on Computing Systems
(to appear)

List of Publications (since Annual Report of August 1982)

1. "Finding Repeated Elements," Science of Computer Programming, No. 2, (1982), pp. 143-152, North-Holland Publishing Company. (Jayadev Misra and David Gries)
2. "Assigning Processes to Processors in Distributed Systems," Proceedings of the 1983 International Conference on Parallel Processing, Bellaire, Michigan, August 23-26, 1983, (Elizabeth Williams)
3. "Preserving Asymmetry by Symmetric Processes and Distributed Fair Conflict Resolution," submitted to ACM Transactions on Programming Languages and Systems, (K. Mani Chandy and Jayadev Misra).
4. "A Distributed Procedure to Detect AND/OR Deadlock," submitted to ACM Transactions on Distributed Systems, (K. Mani Chandy and Ted Herman).
5. "Detecting Stability in Distributed Systems," in preparation, (K. Mani Chandy and Leslie Lamport).
6. Design, Analysis and Implementation of Distributed Systems From a Performance Perspective, Ph.D Thesis, Department of Computer Sciences, University of Texas, Austin, Texas 78712, (Elizabeth Williams)
7. Efficient Distributed Simulation Schemes, Ph.D Thesis, (in preparation), Computer Sciences Department, University of Texas, Austin, Texas 78712, (Devendra Kumar).

List of Professional Personnel

Professor K. Mani Chandy, Co-Principal Investigator

Professor Jayadev Misra, Co-Principal Investigator

with

Faculty
Computer Sciences Department
University of Texas
Austin, Texas 78712

Uses of Travel Funds

Professor Misra delivered a paper entitled, "Detecting Termination of Distributed Computations Using Markers," at the Second ACM Conference on Principles of Distributed Computing which is the most important conference in this area. He visited Professor Tony Hoare at Oxford University to discuss trace theory for program verification, Professors Howard Barringer and Cliff Jones at the University of Manchester to discuss their verification method and Professor Peter Lockemann and his colleagues at the University of Karlsruhe to discuss implementations of distributed simulation. He visited Professors Eli Gafni and Richard Muntz at the University of California at Los Angeles and delivered a talk there. Professor Chandy visited Professor Nancy Lynch at M.I.T. and gave a talk on distributed snapshots.

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Paradigms for Distributed Computing		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Mani Chandy		8. CONTRACT OR GRANT NUMBER(s) AFOSR 81-0205B
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Sciences Department University of Texas at Austin Austin, Texas 78712		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Dr. Robert N. Buchal AFOSR/NM Bolling AFB, DC 20332		12. REPORT DATE
		13. NUMBER OF PAGES 9 pages
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES Invited paper presented at the Third Conference on Foundations of Software Technology and Theoretical Computer Science, December 12-14, 1983, Bangalore, India		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This paper describes a few fundamental paradigms for distributed computing systems. The key issue is that of determining the <u>state</u> of a distributed state. Algorithms to detect states are given.		

Paradigms for Distributed Computing

Invited Talk
Third Conference on
Foundations of Software
Technology and Theoretical
Computer Science,
Bangalore, India
December 12-14, 1983

by
Mani Chandy
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
(512)471-4353

September, 1983

This work was supported in part by a grant from the Air Force under grant AFOSR 81-0205B.

1. Goal

The goal of this paper is to discuss the area of distributed computing in an *informal* manner. I shall not present theory, algorithms or experimental results. Instead, I shall restrict attention to the following questions:

- What is distributed computing?
- Why should one study distributed computing?
- What are the fundamental questions in distributed computing?

The answers to these questions will be philosophical, rambling and subjective; but I think the answers have some merit.

2. What is Distributed Computing?

2.1. A Distributed System

We have to present our discussion in terms of a model of a system. The model chosen is not important in itself. We could have couched our discussion in terms of other models. We shall describe our model informally and only to the level of detail necessary to make our algorithms clear.

A distributed system D is defined by its set P of component processes and set C of directed channels, i.e. $D = (P, C)$. Let there be $N > 0$ processes in P and let them be indexed p_i , $1 \leq i \leq N$. A channel c in C , is directed from a (single) component process p_i to a (single) component process p_j , and the channel is defined by $c = (p_i, p_j)$. Each channel has an infinite buffer. (Bounds on buffer sizes are discussed later.) A process p_i can send a message along one of its outgoing channels (p_i, p_j) whenever it wishes to. Channels are loss-free, error-free and deliver messages in the order sent. The state of a channel (p_i, p_j) is a queue of messages; the queue represents the messages sent by p_i and not received by p_j .

A process p_i in P is specified by a set of *process states*, an initial process state and a set of *allowable events*. An event is (1) an autonomous state transition, (2) a send or (3) a receive. An autonomous state transition at p_i takes p_i from a process state s to a process state s' ; the autonomous state transition event is defined by the pair (s, s') , and this event can occur at p_i only if p_i is in process state s immediately before the event. An autonomous state transition at p_i does not change the state of any channel or the state of any process besides p_i 's.

A send event at p_i is the sending of a message by p_i coupled with a transition of p_i 's state. It is defined by the states s and s' before and after the event, respectively, the message M that is sent and the process p_j that it is sent to. This event can occur at p_i only if p_i is in state s immediately before the event. This event causes M to be inserted into the queue representing the state of channel (p_i, p_j) . The states of channels other than (p_i, p_j) and processes other than p_i are not changed by the occurrence of this event.

A receive event at p_i is the receipt of a message by p_i coupled with a state transition. It is defined by the states s and s' before and after the event, respectively, the message M that is received and the process p_j that the message is received from. This event can occur at p_i only if p_i is in state s immediately before the event and M is at the head of the queue of messages representing the state of channel (p_j, p_i) ; this event causes the deletion of the message at the head of this queue. The states of channels other than (p_j, p_i) and processes other than p_i are not changed by the occurrence of this event.

An event may occur at a process at any time provided the states of processes and channels are such that the event can occur. The process and channel states may be such that one of many events may occur; the selection among the potential events is non-deterministic.

2.2. A Distributed Computation

A process computation z_i of a process p_i is defined as a sequence of allowable events $\langle e_{i1}, e_{i2}, \dots \rangle$ at p_i such that the state of p_i before event e_{ik} , $k > 1$, is its state after the previous event $e_{i,k-1}$, and p_i 's state before the first event, e_{i1} , is its initial state. A process computation may be finite or infinite, empty or non-empty, and is prefix-closed, i.e. if $z_i = \langle e_{i1}, \dots, e_{ik}, \dots \rangle$ is a computation of p_i then every initial subsequence $\langle e_{i1}, \dots, e_{ik} \rangle$ of z_i is also a computation of p_i .

We define a system computation using Lamport's ideas of partially-ordered events. A system computation Z is a set of component process computations, $Z = \{z_i | 1 \leq i \leq N\}$, such that the channel rule and partially-ordered event rules (described below) are satisfied.

Channel Rule

The k -th message received along a channel in Z is the k -th message sent along the channel in Z , all k . Formally, let n_{ji} be the number of messages received by p_i from p_j in z_i . Let m_{ji} be the number of messages sent by p_j

to p_i in z_j . Then $m_{ji} \geq n_{ji}$, all i, j . Furthermore, the k -th message sent by p_j to p_i in z_j is the k -th message received by p_i from p_j in z_i , $1 \leq k \leq n_{ji}$, all i, j .

Partially Ordered Events Rule

The relation \rightarrow between events is a partial ordering of the events in process computations where \rightarrow is defined as follows:

$e \rightarrow e'$ if and only if

1. e and e' are events in the same process computation and e occurs before e' in the computation or
2. e' is the receipt of a message and e the corresponding send of that message or
3. there exists an event e^* such that $e \rightarrow e^* \rightarrow e'$.

Graphical Representations of a Computation

A set of events in a system computation may be represented by a directed graph whose vertices represent events. There is an edge from (the vertex representing) an event e to an event e' if and only if either (1) e and e' are events in the same process computation and e immediately precedes e' in that computation or (2) e represents the sending of a message and e' the receiving of it. Figure 1 shows an event graph for a system with 2 processes. The vertical lines represent process computation and the diagonal lines represent messages.

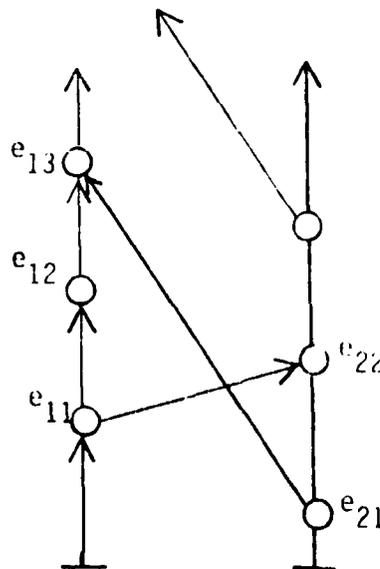


Figure 2-1: A Graph of a System Computation

Let $z_i = \langle e_{i1}, e_{i2}, \dots \rangle$ be a computation of p_i and let $|z_i|$ be the length of

z_i . A point in process computation z_i is an integer k , $0 \leq k \leq |z_i|$. The process computation $z_i(k)$ up to point k in z_i is defined as the empty sequence if $k=0$ and as the sequence $\langle e_{i1}, \dots, e_{ik} \rangle$ otherwise. A point K , in system computation z , is a set $K = \{k_i | k_i \text{ is a point in } z_i, 1 \leq i \leq N\}$, such that $\{z_i(k_i) | 1 \leq i \leq N\}$ is a system computation. For example, in Figure 1, $\{k_1 = 0, k_2 = 2\}$ is not a point in the system computation because $\{\langle \rangle, \langle e_{21}, e_{22} \rangle\}$ is not a system computation since the channel rule is violated - there is a receive event e_{22} in z_2 for which there is no corresponding send. We shall represent a system point $\{k_i | k \leq i \leq N\}$ by the vector $(k_1 \dots k_N)$. Examples of system points in Figure 1 are $(1,2), (2,2), (2,1)$.

The fundamental difference between concurrent computation and distributed computing is that a process in a distributed system can only access information stored in *its* memory; processes do not share variables or a clock. Time has no meaning in a distributed system; only causality (i.e. the relation \rightarrow between events) has significance. The focus of much of the research in distributed computing deals with the problem of limited information: How can a network of processes cooperate in achieving a global task when each process has only partial information about the task. For instance, how can the shortest path between two processes in a network be determined when each process only has information about its immediate neighbors?

The focus of research in the area of concurrent computations appears to be different. The fundamental problem is not limited information but speed. Synchronous solutions to problems (shortest path, detecting cycles ...) in which multiple processors access common memory, are usually quite different from distributed solutions, because even though the problems share the same name (for example, shortest path) the assumptions about the underlying architecture are so different that the problems are indeed different.

Confusion about the two goals, (1) problem-solving with limited information and (2) maximum speed should be dispelled before attacking a problem. Thus, to solve the shortest path problem as quickly as possible one would not use an architecture with one processor at each vertex of the graph. Then why study distributed computing?

Why Study Distributed Computing?

There are systems in which the time required for processes to communicate is significant compared to the time required for them to compute (carry out basic operations).

Examples include systems in which processing power is geographically distributed. Practical distributed systems include factory automation, transportation control (managing the flow of trains, car-traffic in a city, airplanes) and communication systems control. We must bear in mind that problems dealing with such systems are very different from the problems of super-computing.

What are the Fundamental Problems of Distributed Computing?

The problems that I consider to be fundamental may well be different from those that you consider fundamental. Identifying the essential issues is a subjective process. Nevertheless, I believe that it is worth our while to spend a great deal of time arguing about what is central, and what peripheral, before we begin attacking a problem.

I believe that the problem of distributed computation is the problem of partial information: many processes cooperating in achieving global ends though each process has limited information. My biased view of distributed computation leads me to identify the following questions as being fundamental.

(1) How can a process determine the state of a distributed system that it (the process) is part of? This is a natural question stemming from our viewpoint that the problem of distributed computation is the problem of local-information. A process has access to its local, process state, i.e. it has local information: how can it get global information, i.e. the state of the entire distributed system? Special cases of this question are practical questions such as: "How can a process detect whether a distributed computation has terminated? How can a process determine whether it is deadlocked?"

(2) How can one prove properties of a distributed system? This question is related to the question "How should a process be specified?"

A process may be specified by (a) sets of states and events, (b) a program or (c) its input/output relation. There are advantages to each approach.

(3) How can processes cooperate in sharing resources in a fair manner? This is also a problem of local information. If all processes had immediate access to global data, there are simple solutions to the problem of sharing. "How can sharing be achieved when no process has all the relevant information?" That is a much more difficult question.

(4) How can processes cooperate to achieve a global task when some of the processes may be faulty? This question leads to the Byzantine

General's problem and similar problems.

The purpose of this paper is to start a discussion, the object of which is to pose the right questions. However, we shall include one answer to show that the answers may be simple.

3. Determining Global States [Chandy and Lamport]

Observation 1

$(k_1..k_N)$ is a point in system computation Z if and only if for all i, j , $1 \leq i, j \leq N$, the number of messages received by p_j from p_i in $z_j(k_j)$ does not exceed the number of messages sent by p_i to p_j in $z_i(k_i)$.

Observation 2

$(k_1..k_N)$ is a point in system computation Z if and only if for all i, j , $1 \leq i, j \leq N$, there is no message sent by p_i to p_j after the k_i -th event in z_i which is received by p_j after the k_j -th event in z_j .

States

The state of a process p_i at system computation Z is its state after the last event on $Z[j_i]$ in Z . The state of a channel (p_i, p_j) is the sequence of messages sent by p_i to p_j in Z for which there are no receives by p_j from p_i in Z . The global system state at Z is the set of states of component processes and channels.

The state of a system at point K in Z is the state at computation $\{z_i(k_i) | 1 \leq i \leq N\}$.

Algorithm to Determine Global State

The processes collectively define a point $(k_1..k_N)$ as follows. For all i , p_i selects the k_i -th event. To ensure that the k_i selected correspond to a system point the processes send *signals* to one another where signals are special messages which have no effect on the underlying computation. Signals will ensure that the k_i meet the condition of observation 2.

Signal Sending Rule: p_i sends a signal along each outgoing channel after the k_i -th event at p_i and before the next (regular) message sent along the channel, all i .

Signal Receiving Rule: k_j must be such that the k_j -th event occurs before the first receive by p_j along a channel following a signal received along that channel.

The sending and receiving rules together ensure that no message sent by

p_i to p_j after the k_i -th event in Z_i is received by p_j after the k_j -th event in Z_j (observation 2).

The system state at point K is recorded as follows. Each process p_i records its own process state after the k_i -th event on p_i and before the k_i+1 -th event. Each process records the state of all incoming channels: the state of a channel (p_i, p_j) is the sequence of messages received by p_j after the k_j -th event on it and before p_j receives a signal from p_i .

References

1. K. R. Apt, N. Francez, and W. P. de Roever, "A proof system for communicating sequential processes," *TOPLAS*, vol. 2, July 1980.
2. M. Clint, "Program proving: Coroutines," *Acta Inform.*, vol. 2, 1973.
3. K. M. Chandy and J. Misra, "Deadlock absence proofs for networks of communicating processes," *Inform. Process. Lett.*, vol. 9, no. 4, 1974.
4. K. M. Chandy and J. Misra, "A simple model of distributed programs based on implementation hiding and process autonomy," *SIGPLAN Notices*, July 1980.
5. E. W. Dijkstra, "A correctness proof for communicating processes: A small exercise," EWD607, The Netherlands.
6. E. W. Dijkstra, "Two starvation free solutions of a general exclusion problem," EWD625, Plataanstraat 5, 5671 AL Nuenen, The Netherlands.
7. E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Operating Systems Techniques*, Academic Press, 1972.
8. N. Francez and M. Rodeh, "A distributed abstract data type implemented by a probabilistic communication scheme," IBM Israel Scientific Center, TR-080, April 1980 (presented at the 21st Annual Symposium on F.O.C.S., Syracuse, NY, October 1980).
9. D. I. Good, R. M. Cohen, and J. Keeton-Williams, "Principles of proving concurrent programs in GYPSY," in *Conf. Rec. 6th Annu. ACM Symp. on Principles of Programming Lang.*, San Antonio, TX, January 1979.
10. C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. Ass. Comput. Mach.*, vol. 12, 1969.
11. C. A. R. Hoare, "Communicating sequential processes," *Commun. Ass. Comput. Mach.*, vol. 21, no. 8, 1978.
12. C. A. R. Hoare, "A model for communicating sequential processes," *Comput. Lab.*, Oxford University, December 1978.
13. J. H. Howard, "Proving monitors," *Commun. Ass. Comput. Mach.*, vol. 19, no. 5, 1976.

14. R. M. Keller, "Formal verification of parallel programs," *Commun. Ass. Comput. Mach.*, vol. 19, no. 7, 1976.
15. L. Lamport, "Time, Clocks, and the Ordering of Events in Distributed System," *Commun. Ass. Comput. Mach.*, Vol 21, no. 7, July 1978.
16. Daniel Lehmann and Michael Rabin, "On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers' problem," *Proceedings of the Eighth Annual ACM Symposium on Principles of Programming Languages*, Williamsburgh, Virginia, January 26-28, 1981.
17. G. M. Levin, "A proof technique for communicating sequential processes (with an example)," TR 79-401, Dept. of Computer Science, Cornell University, Ithaca, NY, 1979.
18. Nancy Lynch, "Fast allocation of nearby resources in a distributed system," *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, Los Angeles, California, April 28-30, 1980.
19. S. Owicki and D. Gries, "A axiomatic proof technique for parallel programs," *Acta Inform.*, vol. 6, 1976.
20. Glenn Ricart and Ashok Agrawala, "An optimal algorithm for mutual exclusion in computer networks," *Commun. Ass. Comput. Mach.*, vol. 24, no. 1, January 1981.

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Distributed Simulation		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Jayadev Misra		8. CONTRACT OR GRANT NUMBER(s) AFOSR 81-0205B
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Sciences Department University of Texas at Austin Austin, Texas 78712		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Dr. Robert N. Buchal AFOSR/NM Bolling AFB, DC 20332		12. REPORT DATE
		13. NUMBER OF PAGES 56 pages
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION, DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES Tutorial presented at the IEEE Computer Society 4th International Conference on Distributed Computing Systems, May 14-18, 1984, San Francisco, California		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) I presented a one-day tutorial, at the invitation of IEEE Computer Society, at the 4th International Conference on Distributed Computing Systems. There were about 40 attendees, mainly from research and development departments of major corporations. Recent advances in computer and communication systems have resulted in demands for new tools for their analysis. Mathematical modelling techniques have so far proved inadequate in dealing with these systems. Only simulation seems to be a viable alternative. Unfortunately, simulation is proving to be inadequate, be-		

(20)

cause of the sheer magnitude of the problem. For instance, a telephone switch generates roughly around 100 messages in initiating and completing a local call made by a subscriber. Large telephone switches can handle around 100 calls per second. Thus simulation of a telephone switch for 15 minutes of real time requires the simulation of nearly 10 million messages. Detailed simulation of a telephone switch, even for a 15 minute interval, will require several hours on a very large uniprocessor.

An alternative is to exploit the cost benefits of cheap micro/mini computers and high bandwidth lines, by partitioning the simulation problem and executing the parts in parallel. Unfortunately however, the typical simulation algorithm does not easily partition for parallel execution. An entirely new approach to simulation, for multiprocessors, is required.

Recent advances in termination detection (Distributed Deadlock Detection, Chandy, Misra and Haas, TOCS, May 1983; a Distributed Graph Algorithm: Knot Detection, Misra and Chandy, TOPLAS, October 1982; Detecting Termination of Distributed Computations using markers, Misra, ACM Principles of Distributed Computing Conference, Montreal, August 1983) have made it possible to design efficient distributed simulation schemes. The tutorial explored various issues and alternatives in this area.

TUTORIAL NOTES

ICDCS

Tutorial 4

Distributed Simulation

Jay Misra



THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.



IEEE COMPUTER SOCIETY

Distributed Simulation

by

Jayadev Misra

**Department of Computer Science
University of Texas at Austin
Austin, Texas 78712**

Copyright © 1984 Jayadev Misra

Table of Contents

1. An Overview of Simulation	5
1.1. System Simulation Problem	5
1.2. Distributed Simulation	7
1.3. Scope Of This Monograph	7
1.4. History	8
2. Sequential Simulations of Systems	11
2.1. Physical Systems	11
2.2. What is Simulation	20
2.3. The Sequential Simulation Algorithm	21
3. Distributed Simulation: The Basic Scheme	27
3.1. A Model of Asynchronous Distributed Computation	27
3.2. Basic Scheme for Distributed Simulation	27
3.3. Partial Correctness of the Basic Distributed Simulation Scheme	32
3.4. Features of the Basic Distributed Simulation Scheme	33
4. Distributed Simulation: Deadlock Resolution	37
4.1. Overview of Deadlock Resolution	37
4.2. Deadlock Resolution Using NULL Messages	37
4.3. Correctness of the Simulation Algorithm	39
4.4. Demand Driven Null Message Transmission	43
4.5. Rollback and Recovery	43
4.6. Circulating Marker for Deadlock Detection and Recovery	44
4.7. Circulating Marker for Deadlock Avoidance	45
5. Summary and Conclusion	47
References	50
J. Misra - Short Biography	53
Index	54

List of Figures

Figure 2-1:	Schematics of Car Flow	12
Figure 2-2:	Schematics of Events in A Car Wash	13
Figure 2-3:	Schematics of Message-Flow in the Car Wash System	14
Figure 2-4:	A Merge Point pp	16
Figure 2-5:	Schematic Diagram of the Example Assembly Line	18
Figure 2-8:	Schematic Diagram of Job Flow in a Computer System Which Has a CPU and Two Peripheral Processors	19
Figure 3-1:	A Primitive Computer System	31
Figure 3-2:	A Distributed Simulation That Does Not Progress	33
Figure 3-3:	A Distributed Simulation That Deadlocks	34
Figure 4-1:	A Physical System with Loop	38

List of Tables

Table 2-1:	A Sequence of Events in the Car Wash	13
Table 2-2:	A Sequence of Message Transmissions in the Car Wash System	15
Table 2-3:	Job Generation Times and Servicing Times	18
Table 2-4:	Times at Which pp's Send Messages	19
Table 4-1:	Message Transmissions in the Simulation of Example 1	40

Preface

This monograph presents an entirely new approach to the problem of system simulation. System simulations are typically carried out in a sequential manner: a single processor fetches one item from a data structure, carries out one step of simulation, (possibly) updates the data structure and iterates this process. Such simulations are practical only when the number of events being simulated is modest.

Recent advances in computer and communication systems have resulted in demands for new tools for their analyses. Mathematical modelling techniques have so far proved inadequate in dealing with these systems. Only simulation seems to be a viable alternative. Unfortunately, simulation is proving to be inadequate, because of the sheer magnitude of the problem. For instance, a telephone switch generates roughly around 100 messages in initiating and completing a local call made by a subscriber. Large telephone switches can handle around 100 calls per second. Thus simulation of a telephone switch for 15 minutes of real time requires the simulation of nearly 10 million messages. Detailed simulation of a telephone switch, even for a 15 minute interval, will require several hours on a very large uniprocessor.

An alternative is to exploit the cost benefits of cheap micro/mini computers and high bandwidth lines, by partitioning the simulation problem and executing the parts in parallel. Unfortunately however, the typical simulation algorithm does not easily partition for parallel execution. An entirely new approach to simulation, for multiprocessors, is required. This monograph presents such an approach.

The text is organized in 5 chapters. Chapter 1 motivates the need for distributed simulation; it gives a quick survey of the system simulation problem, sequential simulation algorithm and its shortcomings. The scope of the monograph and a history of distributed simulation are also included in that chapter. Chapter 2 contains a detailed description of the sequential simulation scheme. It is shown why this scheme cannot be readily parallelized. Chapter 3 introduces the basic distributed simulation scheme. This scheme is shown to result in deadlock. Several different approaches for deadlock resolution are discussed in chapter 4. Chapter 5 contains a summary and assessment of the entire field.

We believe that distributed simulation offers the most promising approach to speeding up simulation. The basic theory has been developed; it remains to experiment with various alternative heuristics.

This text is mainly oriented toward, (i) machine designers, particularly for those

designing multiprocessors for application programs and (ii) application programmers and simulation practitioners. The material is largely self-contained. Some acquaintance with simulation and distributed systems is helpful though not necessary. Researchers in distributed software design will find this monograph to be useful in that general area. The reader will come away with an appreciation for (i) the nature of the simulation problem reduced to its barest minimum and (ii) how to approach a problem for distributed solution.

I apologize for lack of concrete empirical results. Some results, dealing particularly with queuing networks, exist but were found to be too problem specific for inclusion in this monograph.

Acknowledgements

I am deeply indebted to my friend and colleague, Professor K. M. Chandy of the University of Texas, Austin, for his help, advice and stimulating ideas. My students, particularly Rajive Bagrodia and Dev Kumar, have given me a new perspective on simulation. I am thankful to the Computer Systems Lab at Stanford University, and in particular to Professor Susan Owicki, for providing a proper environment for the preparation of this monograph. I am grateful to Dr. Doug DeGroot of IBM, Yorktown Heights, and Mr. Dev Kumar of the University of Texas, Austin, for their unusually thorough editing efforts and helpful comments on the first draft of this manuscript. Ms. Debra Davis has managed to produce a readable copy of this manuscript in spite of vague and contradictory instructions. I am thankful to the Air Force Office of Scientific Research for financial support over the years for my work on distributed systems.

Chapter 1

Introduction

1. An Overview of Simulation

This chapter motivates the need for distributed simulation. It gives a quick survey of the simulation problem, shortcomings of sequential simulation methods and an overview of distributed simulation. The scope of this monograph and the history of distributed simulation are also included.

1.1. System Simulation Problem

We consider the problem of simulating physical systems, also called *networks*, which consist of one or more *physical processes*. Each physical process operates autonomously except to interact with other physical processes in the system. The interaction is by *messages*. Contents of a message sent by a (physical) process depend upon the characteristics of the process (its initial state, its rules of operation) and the messages that the process has received so far.

We will describe the problem and the terminology more precisely in the next chapter. We note that many real systems can be modelled in terms of processes and messages as described above. For example, a computer system is one in which the CPU, disks, memory and job entry terminals may be thought of as processes; the CPU may interact with a disk by sending it messages requesting or releasing disk space; a job entry terminal may interact with the CPU by sending it messages, which are in fact jobs or tasks to be executed. Detailed examples are given in the next chapter.

Typical steps in simulation consist of,

1. starting with a real system and understanding its characteristics,
2. building a model from the real system in which aspects relevant to simulation are retained and irrelevant aspects are discarded,
3. constructing a simulation of the model which can be executed on a computer (simulations, other than computer programs will not be considered here), and
4. analyzing simulation outputs to understand and predict the behavior of the real system.

In addition, the model and the simulation must be verified and may be refined during steps (2) and (3), perhaps iteratively, if they do not meet the expectations. In this monograph, we look at only one step - step (3) of the entire simulation process.

What is typically called a model in step (2) is actually our physical system; we show how to go from a physical system to a computer program for simulation of this system, which in this case, is distributed and hence may be concurrently executed on several machines. We will not consider the problem of constructing a physical system description from the real system, nor do we consider how to analyze simulation outputs to predict the behavior of the real system. Stated another way, we show how to construct an asynchronous system (the simulator running on asynchronous machines) from a synchronous system (the physical system, running in real time). We will further restrict ourselves to discrete event simulations, we assume that *events* - in our case, message exchanges - happen at discrete points in time.

Traditional Approach to System Simulation

Traditionally, discrete-event system simulations are done in a sequential manner. A variable *clock* holds the time up to which the physical system has been simulated. A data structure, called the *event-list*, maintains some message transmissions with their associated times of transmissions, which are scheduled in the future. Each of these messages is guaranteed to be sent at the associated time in the physical system, provided the sender receives no message before this message transmission time. At each step, the message with the smallest associated future time is removed from the event-list, and the transmission of the corresponding message in the physical system is simulated. Sending this message may, in turn, cause other messages to be sent in the future (which then are added to the event-list) or cause previously scheduled messages to be cancelled (which are removed from the event-list). The clock is advanced to the time of the message transmission that was just simulated.

This form of simulation is called *event driven*, because events (i.e. message transmissions) in the physical system are simulated chronologically and the simulation clock is advanced after simulation of an event to the time of the next event. There is another important simulation scheme, time driven simulation, in which the clock advances by one tick in every step and all events scheduled at that time are simulated. We will not discuss time driven simulation in this monograph. We will, furthermore, assume that all events are discrete, which is certainly true for any system which can be modelled as a message passing system.

Drawbacks of Sequential Simulation

The nature of the event-list mechanism dictates a sequential simulation, since in each cycle of simulation, only one item is removed from the event-list, its effects simulated and the event-list, possibly, updated. This is unfortunate, because this algorithm cannot be readily adapted to concurrent execution on a number of processors, since the event-list cannot be effectively partitioned for such executions. We contend that a major bottleneck to the growth of widespread simulation is the sequentiality inherent in the event-list structure. Increasingly complex computer and communication systems of the future will be intractable mathematically and therefore

will have to resort to simulation for their performance evaluations. Current simulation techniques will prove to be inadequate for these systems because with current technology only a modest number of events can be simulated. A radically new approach to simulation must be taken which will utilize the power and cost benefits of small computers and high bandwidth communication lines.

1.2. Distributed Simulation

This monograph presents a radically different approach to simulation. Shared data objects such as the clock and the event-list are discarded. In fact, there are no shared variables in our algorithm. We suggest an algorithm in which one machine may simulate a single physical process; messages in the physical system are simulated by message transmissions among the machines. The synchronous nature of the physical system is captured by encoding time as part of each message transmitted between machines. We show that machines may operate concurrently as long as their physical counterparts operate autonomously; they must wait for message receptions to simulate interactions of the corresponding physical processes.

Distributed simulation offers many other advantages in addition to possible speed-up of the entire simulation process. It requires little additional memory compared to sequential simulation. There is little global control exercised by any machine. Simulation of a system can be adapted to the structure of the available hardware; for instance if only a few machines are available for simulation, several physical processes may be simulated (sequentially) on one machine.

Several distributed simulation algorithms have appeared in the literature. They all employ the same basic mechanism of encoding physical time as part of each message. The basic scheme they use, may cause deadlock. Different distributed simulation algorithms differ in the way they resolve the deadlock issue. Several new algorithms for distributed deadlock and termination detection have been discovered in the last few years. Combining these algorithms with the basic distributed simulation mechanism is expected to result in very efficient and practical simulation schemes. Empirical investigations are currently under way to assess the performance of different schemes.

1.3. Scope Of This Monograph

This monograph is a comprehensive survey of all known (to the author) distributed simulation schemes. In order to make the monograph self-contained, basic notions of sequential simulation are introduced and explained in Chapter 2. A proof that the sequential simulation algorithm works correctly is given in that chapter; surprisingly, the author could not find such a proof in any simulation book. Chapter 3 introduces a basic distributed simulation scheme and shows its partial correctness. It is shown why the basic scheme may be inadequate, i.e., may result in

deadlock. Different deadlock resolution schemes proposed in the literature are presented in chapter 4. A summary of current results and possible directions for future investigations are outlined in chapter 5.

This monograph does not introduce a new simulation language, because distributed simulations can be written using sequential simulation languages for simulating the physical processes and message communication languages for describing interactions among component machines. We also avoid a number of traditional issues in simulation: pseudo-random number generation, statistical analysis of the outputs etc. Methods developed in these areas for sequential simulation, see [14], still apply. Our goal in this monograph is to show how the body of actual simulation can be distributed among a set of interacting machines.

1.4. History

Sequential simulation has a long history; Franta [15] provides a discussion of a number of prominent simulation languages and their relative merits. Among the many simulation packages introduced recently, we mention DEMOS [5], SAMOA [21] and MAY [2]. DEMOS is a discrete event modelling package implemented in SIMULA [13]. It provides an extensive list of features for event scheduling, data collection and report generation. SAMOA uses Ada [1] as the base language. May is based on FORTRAN IV and provides the *minimum* number of constructs needed to carry out simulation; these features have been used to build an extensive library for data collection, output analysis and report generation. The minimality of MAY makes it possible for it to be implemented even on personal computers.

The idea of distributed simulation was first proposed by Chandy in 1977 in a series of lectures at the University of Waterloo; these ideas were later refined and published by Chandy and Misra [7] and Chandy, Holmes and Misra [8]. They observed that the basic scheme of time encoding may lead to deadlock and they proposed schemes for deadlock avoidance. Independently R.E. Bryant [6] discovered the basic simulation scheme. Peacock, Wong and Manning [24,25] and Holmes [17] proposed mechanisms for avoiding deadlock by periodic use of *probe messages*. Empirical work by Peacock, Wong and Manning has shown that the method is indeed viable: the time needed for simulation of a class of queueing networks steadily decreases when the number of processors available for simulation increases. Empirical investigations by Seethalakshmi [28] and Quinlivan [26] showed that the method is also efficient for acyclic physical systems and that performance can be substantially improved if there are many buffer spaces between machines for buffering messages.

Chandy and Misra [9] have subsequently suggested a scheme for deadlock detection and recovery. Reynolds [27] suggested using common memory among neighbors to avoid deadlock. A notable departure from these schemes is the one

proposed by Jefferson and Sowizral [18]. They suggest that a machine should wait to receive messages for a certain period of time; if it receives no messages from some machine in that period, it assumes that there will be no further messages from that machine, and it then continues simulation under this assumption. In case a message is received from some machine, which violates this machine's previous assumption, it rolls back to its previous state and sends "antimessages" cancelling the messages it may have sent under the false assumption. Empirical investigation of the behavior of this algorithm is continuing.

Bezivin and Imbert [3] propose an approach similar to Jefferson's. In their approach, each process in the simulator maintains a local time and an overall global time is maintained by a central process. Christopher et. al. [12] propose precomputing minimum wait time along all paths in a network so that, delay information may be propagated rapidly among non-neighboring processes. Practical simulation results, employing many processors, have been reported in [23,29].

Dev Kumar [20] has combined some recent work in deadlock and termination detection [22] with the basic simulation scheme. He has developed algorithms which are more hierarchically structured. His scheme has parameters to control the number of overhead messages. Behaviors of these algorithms on a wide class of practical simulation problems are currently being investigated.

Chapter 2
Sequential Simulations of Systems

2. Sequential Simulations of Systems

This chapter introduces the problem of system simulation. A precise definition of simulation is given. The sequential simulation algorithm using the event list structure is presented and proved. It is shown why the sequential simulation scheme cannot be readily adapted for parallel execution.

2.1. Physical Systems

We consider physical systems, also called *networks*, consisting of a finite number of *physical processes* (abbreviated as pp's). Each pp represents some component of the real system to be simulated. For instance, in a computer system, the CPU, each disk, each memory bank and each job entry terminal may be thought of as a pp. A pp usually interacts with other pp's from time to time. In traditional simulation terminology, *events* happen at a pp, and the occurrence of an event at one pp may cause other events to happen at various other pp's. We will use a slightly different terminology in this monograph which makes our description of the simulation algorithms considerably simpler.

Events that are local to a pp, i.e., those which do not directly affect the behaviors of other pp's directly may be simulated locally as part of the simulation of the pp. Any event that causes events to happen at other pp's may be modelled by transmitting a message whose reception causes the desired event to happen. For instance, if event e_1 at pp 1 causes event e_2 to happen at pp 2, we can model these event dependencies by (1) pp 1 sending a message to pp 2 and (2) pp 2 causing event e_2 to happen locally, at a proper time after the receipt of the message. Event e_2 may cause an event e_3 to happen at pp 3, in which case it must also be modelled as a message transmission between pp 2 and pp 3. An event at a pp which causes events to happen at several other pp's must be modelled as several message transmissions among a number of pp's.

We next give an example which clarifies the relationship between events and messages. The reader is urged to study this example because it shows explicit message transmissions between pp's which were not in the original description of the problem.

Example 2-1 (Car Wash)

The following example is a variation of one appearing in [4]. A car wash system consists of an attendant and two car washes, abbreviated CW1 and CW2. Cars arrive at random times at the attendant. The attendant directs cars to CW1 or CW2 according to the following rule: if both car washes are busy, i.e. washing cars, any arriving car is queued at the attendant; if exactly one car wash is idle, the car at the head of the queue, if any, is sent to that idle car wash; if both car washes are idle, the car at the head of the queue, if any, is sent to CW1. CW1 spends 8 minutes and CW2, 10 minutes in washing a car. Given some distribution of car arrivals, it is required to compute the average amount of time a car spends at the car wash (including the washing time) and the average length of the queue that builds up at the attendant. We will not compute the above statistics; we will simply show the sequence of events and message transmissions in two different views of the car wash problem.

The schematic diagram of the flow of cars is given below.

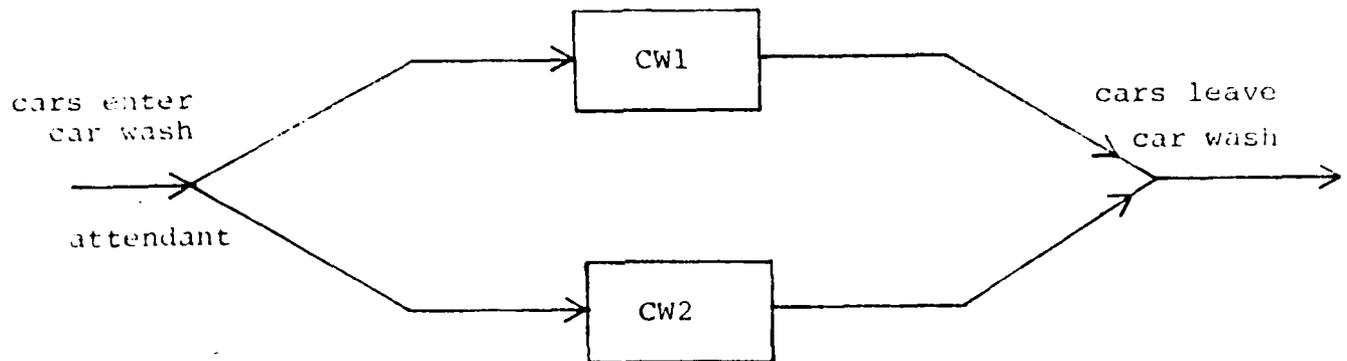


Figure 2-1: Schematics of Car Flow

Initially both CW1 and CW2 are idle. Assume that 6 cars, C1 through C6, arrive at the attendant at times 3,8,9,14,16,22. An event in this system is either a car arriving at some point, i.e., at the attendant, CW1 or CW2, or a car leaving the car wash. We assume that the driving time from the attendant to CW1 or CW2 is zero. Also, when a car arrives at CW1 or CW2, it starts getting service immediately. The chronological sequence of events is given in Table 2-1.

An event e_2 depends directly on an event e_1 , if

1. e_1, e_2 both happen at the same process and e_1 happens before e_2 , or
2. e_1, e_2 happen at different processes and e_1 is one of the causes of e_2 , i.e. if e_1 had not happened, e_2 would not have happened.

Event e is *dependent* on event e' , if e depends directly on e' , or e depends on e'' , which depends directly on e' . Two events are *independent* if they are not dependent on each other. It follows then that independent events can be simulated in parallel, while dependent events must be simulated in sequence.

<u>Event Number</u>	<u>Time</u>	<u>Event</u>
1	3	C1 arrives at the attendant
2	3	C1 arrives at CW1
3	8	C2 arrives at the attendant
4	8	C2 arrives at CW2
5	9	C3 arrives at the attendant
6	11	C1 leaves car wash
7	11	C3 arrives at CW1
8	14	C4 arrives at the attendant
9	16	C5 arrives at the attendant
10	18	C2 leaves car wash
11	18	C4 arrives at CW2
12	19	C3 leaves car wash
13	19	C5 arrives at CW1
14	22	C6 arrives at the attendant
15	27	C5 leaves car wash
16	27	C6 arrives at CW1
17	28	C4 leaves car wash
18	35	C6 leaves car wash

Table 2-1: A Sequence of Events in the Car Wash

Dependencies among events is shown in the directed graph of Figure 2-2; an edge from event e_1 to event e_2 denotes that event e_2 depends directly on event e_1 ; therefore e_1 must be simulated before e_2 .

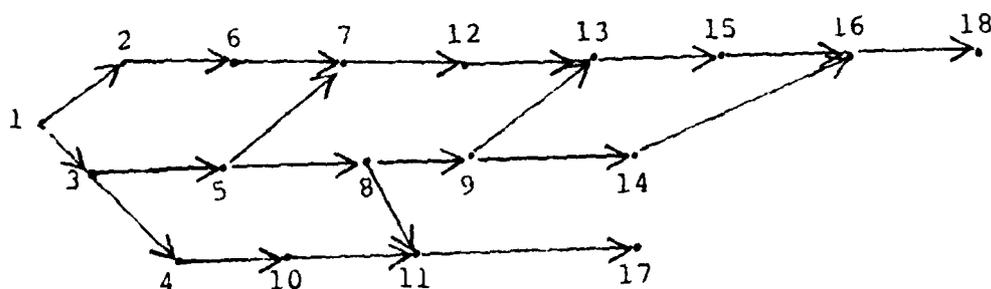


Figure 2-2: Schematics of Events in A Car Wash

Two independent events, such as event 8 (C4 arrives at the attendant) and

event 12 (C3 leaves car wash) are independent and hence can be simulated simultaneously.

We now present the car wash viewed as a message passing system. The car wash system has 5 processes: the source, which generates cars at the prescribed times, the attendant, CW1, CW2 and the sink (exit). The schematic diagram of message communications among these processes is given in Figure 2-3.

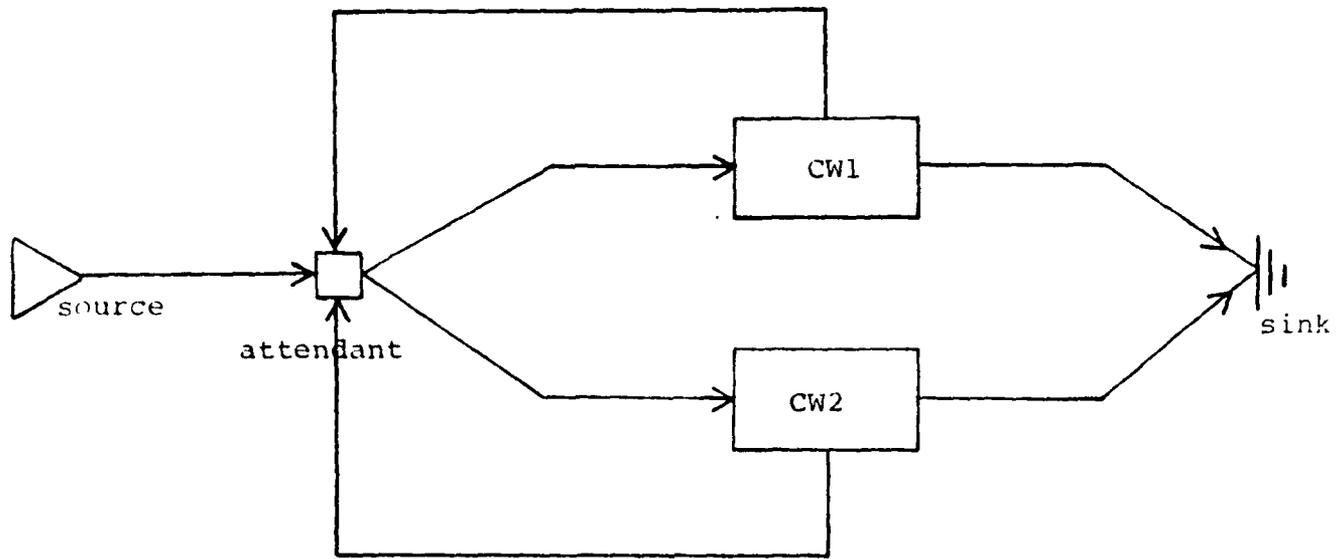


Figure 2-3: Schematics of Message-Flow in the Car Wash System

Note that we have possible message flow paths from CW1 and CW2 to the attendant. This is because the attendant must know when a car wash becomes idle. (In this particular problem, the attendant can keep track of the times at which he sent the last cars to CW1 and CW2, and since the washing times are fixed, he can deduce the times at which CW1 and CW2 will next become idle. This means that the attendant is simulating CW1 and CW2. In general it will not be possible nor preferable to do so). The attendant expects messages from CW1 and CW2 each time they become idle. A complete list of messages for this example is shown in Table 2-2 with corresponding event numbers from Table 2-1. Each message has a sender, a receiver and message content. In our case, the content is either a car number or the status (idle) of a car wash.

This example shows how to model event interactions by message transmissions. In particular, if an event at one pp causes events to happen at several other pp's, we will have to model such event dependencies by several message transmissions. Secondly, the chronological order of simulations of events in sequential simulation (described later) guarantees that every event simulation precedes the simulation of events that depend upon it. Our approach in distributed simulation will dispense

Message Number	Event Number	Time	Sender	Message Receiver	Content
1	-	0	CW1	attendant	idle
2	-	0	CW2	attendant	idle
3	1	3	source	attendant	C1
4	2	3	attendant	CW1	C1
5	3	8	source	attendant	C2
6	4	8	attendant	CW2	C2
7	5	9	source	attendant	C3
8	6	11	CW1	sink	C1
9	-	11	CW1	attendant	idle
10	7	11	attendant	CW1	C3
11	8	14	source	attendant	C4
12	9	16	source	attendant	C5
13	10	18	CW2	sink	C2
14	-	18	CW2	attendant	idle
15	11	18	attendant	CW2	C4
16	12	19	CW1	sink	C3
17	-	19	CW1	attendant	idle
18	13	19	attendant	CW1	C5
19	14	22	source	attendant	C6
20	15	27	CW1	sink	C5
21	-	27	CW1	attendant	idle
22	16	27	attendant	CW1	C6
23	17	28	CW2	sink	C4
24	-	28	CW2	attendant	idle
25	18	35	CW1	sink	C6
26	-	35	CW1	attendant	idle

Table 2-2: A Sequence of Message Transmissions in the Car Wash System with chronological simulations of events.

In summary, a pp may send messages to and receive messages from other pp's at discrete times. Message transmission delays are zero, i.e., any message sent at time t is received by the intended recipient at t . (Recall that we are describing a physical system, not the computer system on which the simulation is to run.) If it is necessary to model delays in the real world system (viz. driving time from attendant to a car wash in the last example), then either the sender of a message idles for some time before sending the message or the recipient of a message idles for some time after receiving the message; another possibility is to model the communication medium as a process, incorporating the delay.

There are two conditions which are met by every physical system imaginable: *realizability* and *predictability*, which are described next. We will assume that both

these conditions hold for all physical systems we consider.

Realizability

A message sent by a pp at time t may depend only upon the messages it has received up to and including t . Realizability says merely that a pp cannot guess any message it will receive in the future.

Note that we admit the possibility of a message that is received at t , affecting a message that is sent at t . An example of a pp in which this instantaneous cause-effect is seen is given below.

Example 2-2 (Instantaneous Message Transmission)

Consider a pp which acts as a merge point for several pp's. Schematically, such a pp, A, is shown in Figure 2-4.

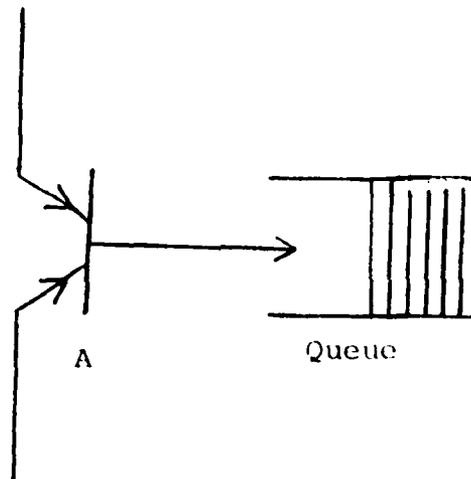


Figure 2-4: A Merge Point pp

Messages arriving at A, either from the top or from the bottom are instantaneously sent to the queue on the right. Therefore a message sent by A at t depends upon messages received at t . It may be argued that pp A cannot be physically constructed. However A might represent a real world entity where the interval between reception and transmission of a message is small enough to be ignored altogether in the modelling process; in fact pp A may not even exist in the real world system and is created, during modelling, to simplify description of the real system. Such merge points are often used in queuing network descriptions of systems.

Predictability

Suppose the physical system has cycles, i.e. it has a set of processes pp_0, \dots, pp_{n-1} where pp_i sends messages to pp_{i+1} (and perhaps to other pp 's) and receives messages from pp_{i-1} (and perhaps other pp 's)¹. Suppose that the message, if any, sent by pp_i at some time t depends upon what pp_i receives at t , for all i ; then we have a circular definition where the message received by every pp at t is a function of itself. Such definitions lead to "race conditions" in physical realizations. In order to avoid such situations, we require that *in every cycle and for every t , there is a pp whose outputs (messages it sends) along the edge of the cycle, can be determined beyond t , - up to $t+\epsilon$, for some fixed ϵ , $\epsilon > 0$ - given the set of input messages to it up to t .*

We next consider some typical simulation examples and show that they satisfy the realizability and predictability conditions.

Example 2-3 (Car Wash - Realizability and Predictability)

We consider the car wash problem introduced in Example 2-1. Each pp 's output at time t depends only upon the messages it has received up to t . Of particular interest is the behavior of the attendant. If it receives an "idle" message from either of the car washes at time t , and the queue is not empty at t , then it sends a message at t . Therefore the realizability condition is satisfied. The predictability condition is satisfied because each cycle contains one of CW1 or CW2, and given the input to CW1 (CW2) up to t , we can predict the output from it up to $t+8$ ($t+10$).

Example 2-4 (Assembly Line)

An assembly line consists of a series of n work stations. Jobs enter the assembly line at work station 1; when a job has been serviced at work station i it proceeds to work station $(i+1)$, $i=1,2,\dots,n-1$; a job leaves the system after being serviced at work station n . Service times at different work stations are random variables. There are queues at stations where the jobs awaiting to be serviced by a station may be queued. A work station takes one job from its input queue when it is free, services that job and then sends it to the queue of the following work station. All work stations service the jobs in a First-Come-First-Served (FCFS) basis. It is desired to find the expected number of jobs in the queue of each work station and the expected waiting time for jobs at each work station.

Specifically, consider an assembly line consisting of 3 work stations, A, B and C, which services 4 jobs identified as 1,2,3,4. Schematically, the assembly line is shown below.

The times at which the source generates jobs and the service time of each work

¹All arithmetic in pp subscripts is modulo n .

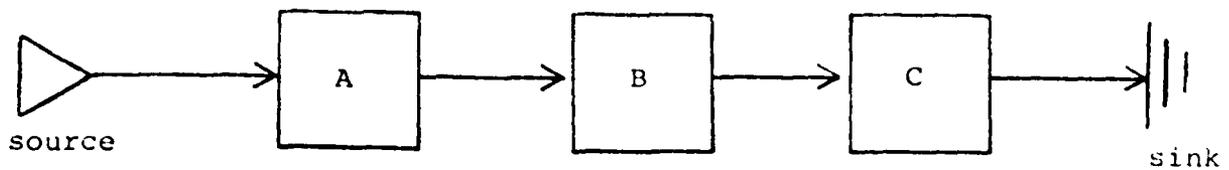


Figure 2-5: Schematic Diagram of the Example Assembly Line

		Jobs			
		1	2	3	4
Job Generation Times	Source	5	7	30	32
	A	4	10	1	5
	B	12	15	2	7
	C	2	3	1	4

Table 2-3: Job Generation Times and Servicing Times

station for each job is given in Table 2-3.

The source (call it work station 0), the sink (call it work station 4) and each work station is a pp. pp i sends messages to pp $(i+1)$, $i=0,1,\dots,3$. The source sends messages (which represent jobs) to work station 1 at times 5, 7, 30 and 32. If a job j , $j > 1$, arrives at a work station at time t , then its service at this work station begins either immediately (at t) if the work station is then idle or it begins immediately after the departure of the $(j-1)$ st job from the work station. Let A_j be the time of arrival of job j at some work station, let D_j be the departure time of job j from this work station, and let S_j be the service time required for job j at this work station. Then we have,

$$D_0 = 0$$

$$D_j = \max(A_j, D_{j-1}) + S_j, \quad j=1,2,\dots$$

Using the service times and generation times of jobs given in the previous table, we can construct the departure times from work stations, i.e., times at which messages are sent, as in the following table.

Each work station's output at time t depends only upon the jobs it has received up to t , and therefore the realizability condition is satisfied. The

Message pp	1	2	3	4
Source	5	7	30	32
A	9	19	31	37
B	21	38	38	45
C	23	39	40	49

Table 2-4: Times at Which pp's Send Messages

predictability condition is trivially satisfied since there is no cycle in the physical system.

Example 2-5 (A Computer Network)

Imagine a computer installation that consists of a CPU and 2 peripheral processors, proc1 and proc2. Jobs enter the CPU, spend some time there and then branch to one of the peripheral processors with some given probability. Upon completion of processing at the peripheral processor, the job may leave the system or return to the CPU with some probability. The schematic diagram of the system is shown Figure 2-6.

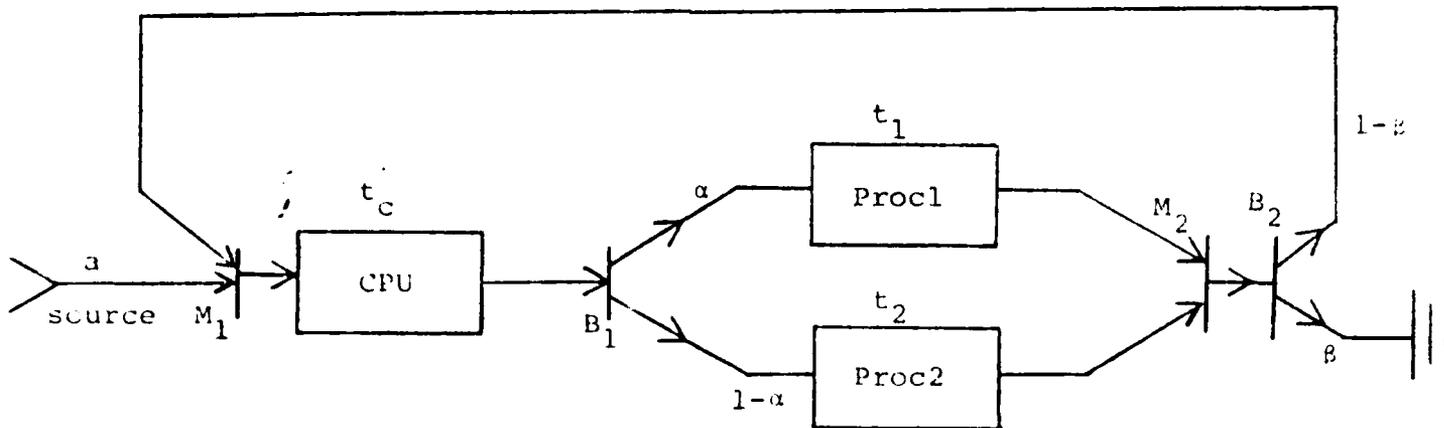


Figure 2-6: Schematic Diagram of Job Flow in a Computer System Which Has a CPU and Two Peripheral Processors

- a: mean time between arrival of jobs from the outside source, a random variable.
- t_c : mean time spent by a job at the CPU, a random variable

t_1 :	mean time spent by a job at the peripheral processor 1 (proc1), a random variable
t_2 :	mean time spent by a job at the peripheral processor 2 (proc2), a random variable
α :	probability of a job going to proc1
β :	probability of a job exiting the system
M_1, M_2 :	merge points
B_1, B_2 :	branch points

This system has pp's for the source, the sink, merge points M_1 and M_2 , branch points B_1 and B_2 , the CPU, proc1, and proc2. Each message represents the transfer of a job from one pp to another. The realizability property holds, because no pp bases its behavior on anticipation of the future. Probabilistic decisions at B_1, B_2 cause no difficulty because the inputs to B_1, B_2 up to time t determine their outputs up to time t (though the outputs may be different at different times due to the probabilistic nature). We can realistically assume that each processor spends non-zero time in processing a job. Therefore the system also has the predictability property.

This concludes our discussion of modelling real world systems by physical systems, i.e., a system consisting of message passing processes and, operating in real time. From now on, we will assume that this modelling has been performed and that we are dealing with physical systems with realizability and predictability properties. Now we define the meaning of simulation, precisely, for such physical systems.

2.2. What is Simulation

We wish to build a simulator or a *logical system*, consisting of *logical processes* (abbreviated lp), to simulate a physical system. We will use "simulation" in a rather strict sense: we say that a logical system correctly simulates a physical system if it is *possible* for the logical system to predict the exact sequence of message transmissions in the physical system. That is, if $t_1, t_2, \dots, t_i, \dots$ are the times at which the messages $m_1, m_2, \dots, m_i, \dots$ are transmitted in the physical system and $t_1 \leq t_2 \leq \dots \leq t_i \leq \dots$, then the logical system should be able to output the sequence $\langle (t_1, m_1), (t_2, m_2) \dots (t_i, m_i), \dots \rangle$.

We observe the following facts from the definition of simulation just stated.

1. The logical system *must* be able to determine the exact chronological sequence in which message transmissions take place in the physical

system; therefore it is not acceptable to predict (t,m) and then (t',m') , where $t' < t$.

2. The logical system may not actually print the sequence, $\langle \dots(t_i, m_i) \dots \rangle$. All that is desired is that it should be *possible* to do so from the logical system.

Clearly a physical system is a simulation of itself. We wish to construct logical systems which may *not* operate at the same speed as the physical system. Our goal is to construct a logical system out of a machine or machines where the speeds of processors and communication links (if any) are arbitrary. In other words, we wish to duplicate the behavior of a synchronous physical system using asynchronous logical components.

It should be observed that we can do the typical functions of a simulation - analyze data, predict performance or future behavior, generate reports etc. - by using the logical system. We do not address these issues in this monograph; we merely observe that since it is possible to create the sequence of physical message transmissions in the logical system, all interactions can be reconstructed and analyzed.

Example 2-6 (Message Transmission in the Assembly Line Example)

A simulation of the assembly line of Example 2-4 should be able to predict the following message sequence. This sequence is derived from Table 2-4. In the following, a message consists of (sender id, receiver id, message content). We will write a 4-tuple (t,s,r,m) to denote that at time t , process s sends a message to r with content m .

$\langle (5,\text{source},A,1), (7,\text{source},A,2), (9,A,B,1), (19,A,B,2), (21,B,C,1), (23,C,\text{sink},1),$
 $(30,\text{source},A,3), (31,A,B,3), (32,\text{source},A,4), (36,B,C,2), (37,A,B,4), (38,B,C,3),$
 $(39,C,\text{sink},2), (40,C,\text{sink},3), (45,B,C,4), (49,C,\text{sink},4) \rangle$

2.3. The Sequential Simulation Algorithm

Two major data objects used by the sequential simulation algorithm are *clock* and *event-list*. Their meanings are given below.

clock: is a real-valued variable. It gives the time up to which the corresponding physical system has been simulated.

event-list: is a set of tuples of the form (t_i, m_i) , where t_i is some time, $t_i \geq \text{clock}$ and m_i is a message. (We assume that the identities of the

sender and the receiver are parts of the message.) $A(t_i, m_i)$ in the event-list means, if the sender of m_i receives no further messages between the current time (given by clock) and t_i , then it sends no other message between clock and t_i , and sends m_i at t_i .

It is required that for every pp_i , there must be exactly one event-list entry (t_i, m_i) in which pp_i is the sender. If a pp sends no message in the future, unless it receives further messages, the corresponding event-list entry will be (∞, m) , where the message content in m is arbitrary. A similar entry, (∞, m) , will always be in the event-list for a pp that has terminated.

Example 2-7 (A Snapshot in Sequential Simulation of the Assembly Line)

In simulating the assembly line of Example 2-4, a possible value of *clock* and corresponding entries in the *event-list* are shown below.

clock 9

event-list $\{(19, A, B, 2), (21, B, C, 1), (\infty, C, \text{sink}, -), (30, \text{source}, A, 3)\}$

This snapshot of the simulation corresponds to the point in the physical system where the source has produced jobs 1 and 2 and job 1 has been processed at A and sent to B. The source has one more job scheduled for production; A has scheduled to send job 2 to B at time 19, provided A receives no more jobs between 9 and 19; B has scheduled to send job 1 to C at time 21 provided it receives no more jobs before then; C has scheduled no message because it has received no jobs.

It should be noted that each entry (t, m) in the event-list is *conditional*. (t, m) may not actually occur in the physical system because this message transmission *may* be cancelled if the sender of m receives a message prior to t . In fact, one can construct an example where nearly all entries in the event list are cancelled in each cycle. There is only one entry (t, m) , where t is the smallest time component of any entry in the event-list, which is guaranteed to occur. (We assume, for the moment, that there is a unique entry with the smallest value of t -component. The case of multiple entries with smallest t -components is discussed below.) The sequential simulation algorithm is based on this fact. We state and prove this result, somewhat rigorously, below.

Simulations of Simultaneous Events

Two message transmissions that happen simultaneously in the physical system, i.e., at the same time t , require that a sequential simulation simulate them in some order. This can lead to problems, as shown in the following example: pp A plans to send a message m to pp B at time t ; pp B is an alarm clock that is scheduled to go off, i.e., send a message m' to pp A, at time t , unless it receives a message from pp A *before or at* t . In the physical system, pp B will *not* send m' to pp A. However if

these message transmissions are simulated sequentially in arbitrary order, a possible simulation may result in pp B sending m' to pp A. As the reader will see later, this problem does not arise in distributed simulation. However sequential simulation requires an ordering of simultaneous events and different orderings may lead to different simulations. Certain sequential simulation languages, such as GPSS [15], provide the user with facilities for defining these orderings.

In proving theorem 1, below, which forms the basis for sequential simulation, we will skirt the issue: we will assume that, simultaneous events are independent, i.e., if (t,m) and (t,m') are both in the event list and t is the smallest time component, then both these message transmissions will take place in the physical system. Therefore these message transmissions may be simulated in either order.

The Simulation Algorithm

The theorem stated below forms the basis of the sequential simulation algorithm.

Theorem 1: Let (t,m) be an entry in the event-list such that $t \leq t'$, for every other entry (t',m') in the event-list. Then the message m is transmitted at time t in the physical system and no message is transmitted between the current clock value and t .

Proof: If message m is not transmitted at t , it must be because some other message is transmitted earlier than t (and after clock value) which causes the sender of m to cancel transmission of m . Consider the first message m' to be transmitted after the current clock value; it must be transmitted at t' where $\text{clock} \leq t' < t$. The sender of m' could not have received any message between clock and t' , because such a message would be the first message. (t',m') must be an entry in the event-list, because the sender of m' sends its message, without receiving any other message after the current clock value and before t' . $t' < t$ contradicts our choice of (t,m) . Hence the theorem.

The simulation algorithm, given below, works as follows. In each step, the message with the smallest associated time is removed from the event-list, its effects are simulated causing possible additions to and deletions from the event-list, and the clock is advanced to the time associated with this message transmission. This algorithm is given in a pseudo-programming notation below.

Algorithm for Sequential System Simulation

Initialize::

clock := 0; event-list := $\{(t_i, m_i) \mid \text{message } m_i \text{ will be sent at } t_i \text{ unless the sender of } m_i \text{ receives a message before } t_i; \text{ one such entry exists for each } p \text{ as the sender}\}$.

Iterate::

while termination criterion is not met **do**

remove any (t, m) from the event-list where t is the smallest time component;

simulate the effect of transmitting m at time t ;
 {This may cause changes in the event-list.
 Note however that any addition (t', m')
 to the event-list will have $t' \geq t$ and,
 any deletion (t', m') will have $t' > t$ }

clock := t

endwhile

The correctness of this algorithm should be obvious from our previous discussions. Note that the sequential simulation algorithm is capable of producing the sequence of message transmissions in the physical system; it simply prints (t, m) when it removes (t, m) from the event-list. Furthermore, this algorithm cannot, in general, choose to simulate more than one tuple in any step, because as we have noted earlier, none of the tuples except the one chosen by the algorithm may occur in the physical system.

Example 2-8 (A Sequence of Snapshots in the Simulation of the Assembly Line)

We consider the assembly line example and show a partial sequence of event-lists and clock values.

clock	event-list	message with smallest associated time
0	$\langle (5, \text{source}, A, 1), (\infty, A, -, -), (\infty, B, -, -), (\infty, C, -, -) \rangle$	$(5, \text{source}, A, 1)$
5	$\langle (7, \text{source}, A, 2), (0, A, B, 1), (\infty, B, -, -), (\infty, C, -, -) \rangle$	$(7, \text{source}, A, 2)$

7	$\langle (30, \text{source}, A, 3),$ $(9, A, B, 1),$ $(\infty, B, -, -),$ $(\infty, C, -, -) \rangle$	$(9, A, B, 1)$
9	$\langle (30, \text{source}, A, 3)$ $(19, A, B, 2),$ $(21, B, C, 1),$ $(\infty, C, -, -) \rangle$	$(19, A, B, 2)$

Notes on Parallel Execution

It should be obvious that, in general, we cannot do much better than processing one tuple from the event-list at a time. In order to process more than one tuple, say at once both (t, m) and (t', m') , we must be sure that these two events are independent, i.e., that execution of one will not in any way affect the execution of the other. This requires us to know more about the cause-effect relationship among messages. We consider these issues in the next chapter in developing a basic scheme for distributed simulation, the subject with which this monograph is concerned.

Chapter 3

Distributed Simulation: The Basic Scheme

3. Distributed Simulation: The Basic Scheme

In this chapter, we introduce a model of distributed computation and we show how a simulation may be carried out by a set of communicating processes. We limit our discussion here to a basic scheme, one which can result in deadlock. More sophisticated schemes which resolve deadlock are discussed in the next chapter.

3.1. A Model of Asynchronous Distributed Computation

A distributed system consists of a finite number of processes and *directed edges* connecting some pairs of processes. To distinguish these processes from physical processes, we call them *logical processes* or *lp's*. Each lp is a sequential process that executes both its sequential code and two special commands: *receive* and *send*. In a send command, an lp names an outgoing edge and a message that is to be sent along that edge. The execution of the send command results in the message being deposited on the named outgoing edge; the sender then proceeds with the execution of its code. Each message takes an arbitrary but finite time to reach its destination. Messages, sent along an edge, are delivered in the sequence in which they are sent. In a receive command, an lp names one or more incoming edges from any one of which it wishes to receive a message. An lp wishing to receive may have to wait until a message arrives along one of the edges that it is waiting for. Note that our communication protocol is extremely simple and can be implemented readily on many existing machine architectures.

A set of lp's D is *deadlocked* at some point in the computation if (1) every lp in D is either waiting to receive or is terminated, (2) at least one lp in D is waiting to receive, (3) if lp_i is in D and is waiting to receive from lp_j , then lp_j is also in D , and (4) there is no message in transit from lp_j to lp_i .

It follows then that none of the lp's in D will carry out any further computation as they will remain waiting for each other.

3.2. Basic Scheme for Distributed Simulation

To simulate any given physical system, we construct a distributed *logical system* as follows. We will associate one lp per pp_j ; lp_i will simulate the actions of pp_j . There is an *edge* from lp_i to lp_j , if lp_i can send messages to lp_j . Messages among lp's will be transmitted along the edges connecting them.

An lp can simulate the actions of a pp up to time t if the lp knows all messages that the corresponding pp receives up to time t . This is because, from the *realizability* property, no future message (message received by the pp after time t) can affect the pp's behavior at t . We note further that an lp *may* be able to simulate a pp even beyond time t by knowing its input messages up to time t , as shown in the following example.

Example 3-1 (An lp May Predict the Future)

Consider a typical non-preemptive First Come First Serve (FCFS) server which spends exactly 10 units of time servicing each job. Assume that a job arrives at time t when this server is idle. From this information about input messages up to time t , we can predict the behavior of the server up to time $t + 10$: it will produce no output between times t and $t + 10$, but it will output a message at $t + 10$, sending the job that has been serviced to its next destination.

From these observations, we can construct an algorithm for distributed simulation. We note that the times at which pp's send messages must be encoded into the message that the lp's send. *Thus if message m is sent by pp_i to pp_j at time t , message (t,m) will be sent by lp_i to lp_j at some point during simulation and vice versa.*

We make a *chronology* requirement: if an lp sends a sequence of messages $\langle \dots(t_i, m_i), (t_{i+1}, m_{i+1}) \dots \rangle$ to another lp, then $t_i < t_{i+1} \dots$. The implication of this requirement is, if lp_i receives (t, m) from lp_j , then it knows *all* messages that pp_j receives from pp_i up to and including time t , because any future message will have a higher t -component.

Define the *edge clock value* of an edge to be the t -component of the last message received along that edge; the edge clock value is 0 if no message has been received along that edge. Clearly, every lp_i knows all messages received by the corresponding pp_i up to time $T_i = \min_j \{t_j\}$, where t_j 's are the edge clock values of all incoming edges to that lp and the minimum is taken over all these incoming edges. lp_i can thus safely simulate pp_i up to T_i , i.e., it can deduce every message that pp_i sends up to time T_i . Also, lp_i *may* also be able to deduce pp_i 's message transmissions beyond T_i . In any case, lp_i will send messages, corresponding to all the messages it can deduce for pp_i . The basic simulation algorithm followed by lp_i is sketched next.

Algorithm A

Basic distributed simulation algorithm to be followed by lp_i .

Initialize: $T_i := 0$ (All messages received by pp_i up to T_i , are now known to lp_i)

while simulation completion criterion is not met **do**

{simulate pp_i up to T_i by doing the following}::

for each outgoing edge, compute the sequence of messages
 $\langle (t_1, m_1), (t_2, m_2) \dots (t_r, m_r) \rangle$, where $t_1 < t_2 < \dots < t_r$, and, pp_i sends
 m_j at time t_j along this edge;
 send each message in sequence along the appropriate edge;

{NOTE: all messages sent by pp_i up to T_i can be deduced
 by lp_i and sent; also some messages to be sent beyond
 T_i may be predicted by lp_i and sent. Only new messages
 that have not been sent before, are sent. Also note that
 some or all of these message sequences may be empty.}

{receive messages and update T_i until T_i changes value} ::

$T_i' := T_i$;

while $T_i' = T_i$ **do**

wait to receive messages along all incoming edges;

upon receipt of a message, update lp_i 's internal state and
 recompute T_i , the minimum over all incoming edge clock values

endwhile;

endwhile

Note: Those lp 's which have no incoming edges, will be called *source* lp 's. Each source lp also follows this algorithm except that it simply sends messages until the simulation completion criterion is met. A *sink* lp simply receives messages and otherwise does not affect the simulation.

Example 3-2 (Distributed Simulation of the Assembly Line)

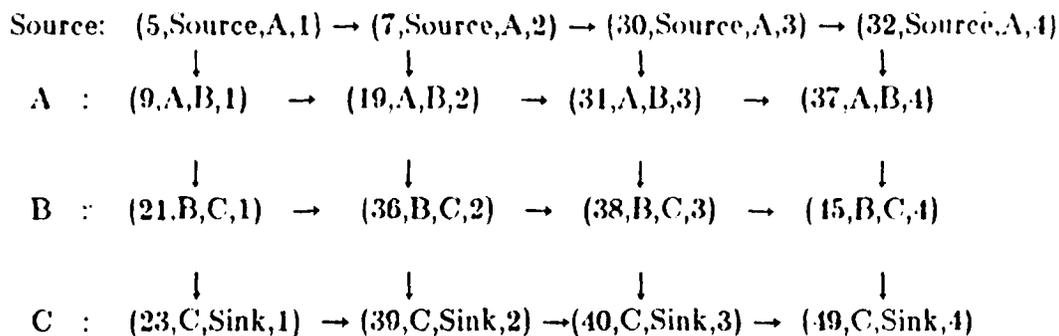
Let us review the assembly line example (Example 2-4). In the following, we have one lp each, for the source, the sink, work station A, work station B and work station C.

We reproduce Table 2-3 here, which shows the job generation and processing times.

		Jobs			
		1	2	3	4
Job Generation Times	work station				
	Source	5	7	30	32
Service Times	A	4	10	1	5
	B	12	15	2	7
	C	2	3	1	4

Job Generation Times and Servicing Times

The following diagram shows the messages sent by each lp; an arrow from (t,m) to (t',m') means that sending of (t,m) precedes sending of (t',m') .



Note in this example that the source can send its messages to A without waiting for any input; A can send the i -th message to B only after receiving the i -th message from the source, etc. Two messages on different lp's between which there is no sequence of arrows, are independent and hence may be transmitted simultaneously in the simulator. For instance, $(32, \text{Source}, \text{A}, 4)$, $(31, \text{A}, \text{B}, 3)$, $(36, \text{B}, \text{C}, 2)$, $(23, \text{C}, \text{Sink}, 1)$ can possibly be transmitted simultaneously. The five lp's form a pipeline through which each job passes. If the speeds of the lp's are approximately equal and the transmission delays between lp's are approximately equal then the pipeline should work at full efficiency; one job is input and one job is output per cycle after an initial delay of 3 cycles.

This is about the simplest simulation example one can think of. We study a harder example, next.

Example 3-3 (A Primitive Computer System)

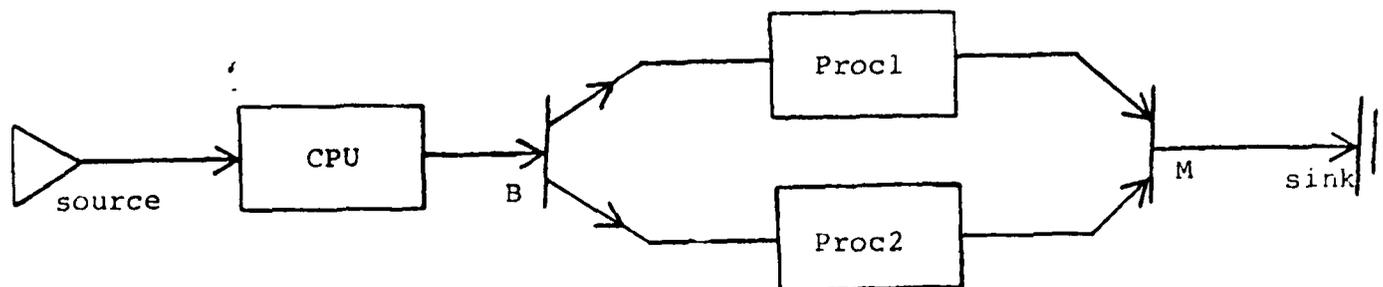
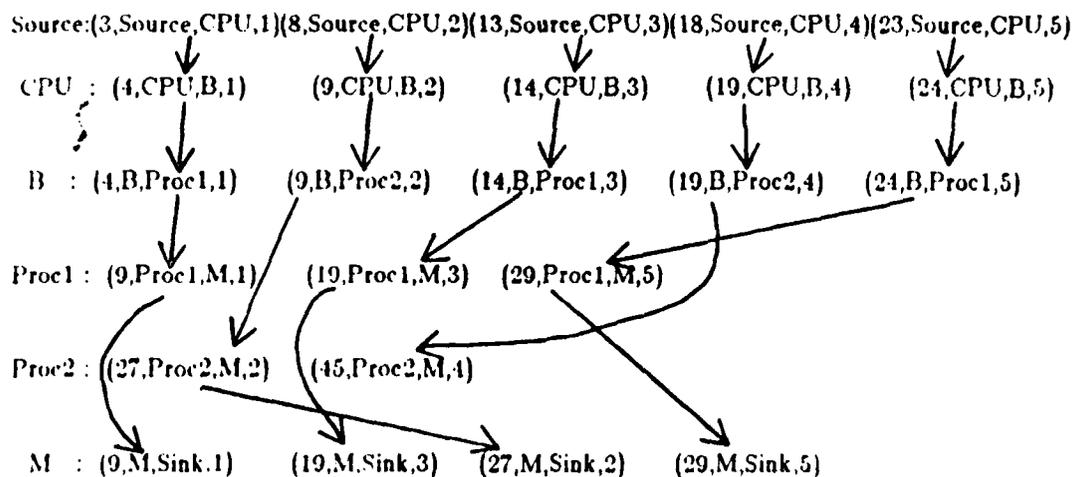


Figure 3-1: A Primitive Computer System

We have one lp each corresponding to the source, the CPU, Proc1, Proc2, M, B and the sink. For this example, assume that jobs arrive at the CPU from the source every 5 time units starting at time 3, that jobs spend 1 unit at the CPU, that jobs alternately go to Proc1 and Proc2 from B, and that a job spends 5 units at Proc1, 18 units at Proc2, and no time at B or M. We show the sequence of messages and their dependencies below. (To simplify the diagram, we have not shown the arrows between messages at a pp.)



Note the behavior of the lp corresponding to M. Assume that it first receives (27,Proc2,M,2) from the lp corresponding to Proc2. This could be entirely possible if, for instance, the lp corresponding to Proc2 were considerably faster than the one corresponding to Proc1. Then the lp for M can only infer that it won't receive any other message from the lp corresponding to Proc2 with time component smaller than

27. However, it cannot assert anything about messages from Proc1; it can thus simulate pp M only up to time 0. Suppose next it receives (45,Proc2,M,4); it must still wait. The next input is, say, (9,Proc1,M,1). Then the lp corresponding to M can assert that it knows all inputs that M receives up to time 9 and hence predict all of M's outputs, at least up to 9 and therefore, it can output (9,M,Sink,1), since jobs spend no time at M. The rest of the outputs of M are easy to see. Finally note that M cannot output (45,M,Sink,4) at the very end, because it does not know if it will receive a message with a t-component lower than 45, from the lp corresponding to Proc1. An extra message must be sent from Proc1 to M, with t-component exceeding 45, to "flush-out" this message. We will discuss this issue later.

3.3. Partial Correctness of the Basic Distributed Simulation Scheme

Correctness of a distributed simulation algorithm consists of two parts: (1) if a message m is transmitted in the physical system at time t, then (t,m) is transmitted in the simulator and, (2) if (t,m) is transmitted in the simulator, then message m was transmitted at time t in the physical system. These statements are not quite true in the basic distributed simulation scheme just presented. As we observed in the last example, job 4 is sent at time 45 from M to the sink in the physical system, although the corresponding message is never sent in the simulator. Therefore, we can prove only one part of the correctness condition stated above: whatever is transmitted in the simulator actually happens in the physical system. We will postpone discussion of the converse statement- if message m is transmitted at time t in the physical system, then (t,m) is transmitted in the simulator - to the next chapter.

Define a simulation to be *correct* at some point, (1) if message m is sent at time t along edge e in the physical system and t is less than or equal to the edge clock value of edge e, at this point in simulation, then (t,m) has been sent (along edge e) in the simulation, and (2) if (t,m) has been sent in the simulation, then message m is sent at time t in the physical system.

We note that in a simulation which is correct at some point, every lp must have received a *correct input sequence* along every incoming edge, i.e., every message on this edge that has been transmitted in the physical system up to this edge clock value, has been received along this edge in the simulation and vice versa. We will assume that every lp *correctly simulates* the corresponding pp, i.e., *if an lp receives correct input sequences along all incoming edges, then it sends correct output sequences along all outgoing edges*. Clearly a simulation is correct if and only if every lp has sent correct output sequences along every outgoing edge. The following theorem follows, by applying induction on the number of messages transmitted in the simulation.

Theorem 1: Simulation is correct at every point.

Proof: Simulation is obviously correct, from definition, when no message has been transmitted in the simulation. Assume that a simulation

is correct up to some point. The next message in the simulation is sent by some lp_i . Since simulation is correct prior to this message transmission, lp_i has received correct input sequences so far. From our assumption that lp_i correctly simulates pp_i , the output sequences of lp_i , including the last message sent, are correct. Every other lp has sent correct sequences so far, from the inductive hypothesis. Hence the simulation is correct following the last message transmission.

In a similar manner, we can derive the following result.

Theorem 2: All messages sent by one lp to another are chronological in their time components.

3.4. Features of the Basic Distributed Simulation Scheme

The Problem of Deadlock

Theorem 1 tells us only that whatever is transmitted in the simulator corresponds to a message in the physical system. As we have noted earlier, not all messages in the physical system do get transmitted in the simulator using the basic simulation scheme. In fact, the next example shows a system in which no message gets transmitted to a subsystem in the simulator.

Example 3-4 (A Deadlocked Subsystem in a Distributed Simulation)

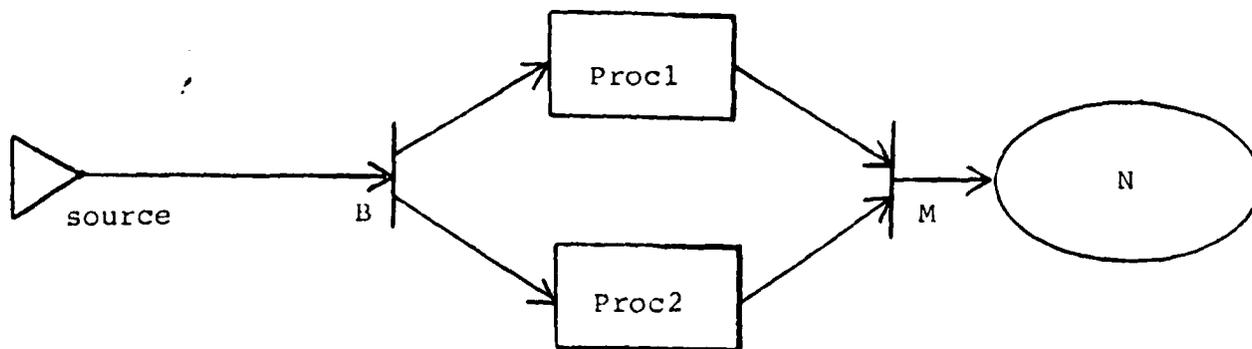


Figure 3-2: A Distributed Simulation That Does Not Progress

Consider a physical system in which the source sends messages to a branch point B, B routes the messages to Proc1 or Proc2 from where, after some finite time, each message is sent to a merge point M, after which it enters a network N (see Figure 3-2). Consider the case where B sends *every* message to Proc1. Then in the simulation, the lp corresponding to M will never receive a message from Proc2. Hence the edge clock value for the edge (Proc2,M) will remain at 0 and the lp for M will

never send a message. The subsystem N will thus never receive a message. If the simulation continues, certain parts of the system, viz. source and $proc1$ will keep on advancing their clocks; however neither M nor any lp within N will advance its clock. We can claim that the clock for no lp in N can progress beyond $t=0$.

We show another example in which the deadlock arises due to a circular pattern of waiting among the lp 's.

Example 3-5 (Cyclic Waiting in a Distributed Simulation)

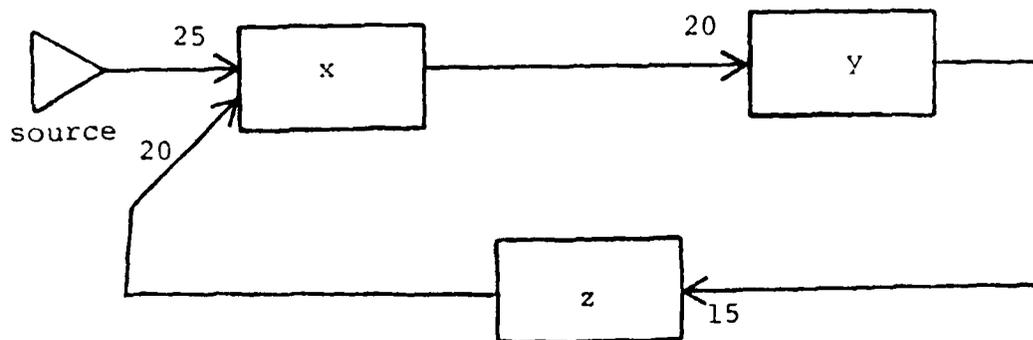


Figure 3-3: A Distributed Simulation That Deadlocks

Consider a network of 3 processes and a source, shown schematically above. The number on each edge is the edge clock value, i.e., the last message sent from x to y and received by y had a t -component of 20 and so on. Suppose that none of x, y, z will now send a message unless they receive a message, i.e., they can predict no future messages.

A global observer can see that z will not send a message unless x first sends a message to y . Hence x need not wait for z ; it can process the next message from the source. However none of the lp 's corresponding to x, y, z have this global knowledge; they only have local knowledge of the behavior of each individual pp . Therefore x cannot proceed unless it receives from z , z cannot proceed unless it receives from y and y cannot proceed unless it receives from x , leading to a deadlock.

Simulation Snapshot

In a sequential simulation, it is possible to assert that the simulator has completed simulation up to the time given by the *clock*: every pp must have been simulated up to this point in time. We cannot make a similar statement for distributed simulation, because each lp may have simulated the corresponding pp to a different point in time. For instance, in the example of the primitive computer system (Example 3-3), we can assert at the end that the lp 's have simulated the corresponding pp 's as follows: (Source : 23), (CPU : 23), (B : 24), (Proc1 : 24), (Proc2 : 19), (M : 20).

We define T_i , the *clock value* of lp_i , to be the point in time up to which pp_i has been simulated by lp_i . Thus lp_i has received messages along all incoming edges up to at least T_i and has sent messages at least up to T_i along every outgoing edge (i.e., all future messages it will send will have t -components exceeding T_i). T_i is the maximum value satisfying the above conditions. Define T , the *clock value of the simulator*, to be the minimum of all lp clock values. We can assert that at any point in simulation, the physical system has been simulated up to the simulator's clock value, even though some individual lp 's may have simulated the corresponding pp 's far beyond T .

Encapsulation of Physical Processes by Logical Processes

The radical departure in the proposed scheme from sequential simulation, however, is the lack of any global control. (We will show deadlock resolution without global control in the next chapter.) Since a pp is simulated entirely by one lp , various different simulations of a pp can be attempted by substituting different lp 's for it. Furthermore, the correctness of simulation can be checked one lp at a time - the proof of correctness is naturally partitioned among lp 's, i.e., we show that each lp correctly simulates the behavior of the corresponding pp . We have shown that if each lp behaves correctly, the ensemble as a whole behaves correctly. This observation will lead to major simplifications in designing complex simulations. In fact, distributed simulations can be implemented using existing sequential simulations; instead of reporting to a central event-list manager, an lp sends messages and otherwise the core of the simulation remains unchanged.

Chapter 4

Distributed Simulation: Deadlock Resolution

4. Distributed Simulation: Deadlock Resolution

We have seen in the last chapter that the basic distributed simulation scheme may lead to deadlock even in acyclic networks. In this chapter, we present several different approaches to resolution of deadlock. We comment on some of the most viable approaches for deadlock resolution.

4.1. Overview of Deadlock Resolution

In all the examples we have seen so far, the simulator clock value (recall that the simulator clock value is the minimum of all lp clock values) remains at some final value T forever. If T is smaller than the point up to which we need to run the simulation, we have to apply some other scheme to advance the simulation. Simulations stop (other than by conscious choice) when some lp has more than one input edge, it can be determined (by an external observer) that it will receive no more input messages along some particular edge and the lp cannot proceed further in its simulation unless it receives this information. For instance, in the example of the primitive computer system (Example 3-3), the lp corresponding to M cannot proceed any further unless it is told that $Proc1$ will never send it a message. Another example is Example 3-5, where process x must be told that it will never receive any input along zx until x first sends a message. The first scheme we describe, using null messages [7], is effectively an implementation of this idea. We will also discuss some other schemes which avoid deadlock by using different kinds of overhead messages.

4.2. Deadlock Resolution Using NULL Messages

We postulate a new kind of message to be used in the simulator. $(t, null)$ sent by lp_i to lp_j means that pp_i sends no message to pp_j between the current edge clock value of the edge from lp_i to lp_j , and t ; therefore any future message from lp_i to lp_j will have a t -component exceeding t . Clearly *null* messages have no counterpart in the physical system. A *null* message is used to announce absence of messages. Absence of messages in a physical system at time t is recognized by no message being transmitted at that time. Unfortunately, the basic scheme of the last chapter cannot guarantee absence of messages to an lp without sending it an actual (non-null) message having a higher t -component value.

We now propose modifications to the basic algorithm of chapter 3 to incorporate null messages. Let us first review the basic distributed simulation scheme

of the last chapter. T_i denotes the clock value of lp_i . Whenever an lp_i receives a message, it properly updates T_i , and if T_i changes in value, then lp_i advances the simulation of pp_i up to T_i . At this point, lp_i predicts for each outgoing edge, a sequence of messages that the pp_i would have sent. Thus lp_i typically generates $\langle (t_{j1}, m_{j1}), (t_{j2}, m_{j2}), \dots \rangle$ for transmission to lp_j , for every j to which it has outgoing edges. Some of these sequences may be empty, in which case no message is sent to the corresponding lp . Suppose that lp_i can further predict that after transmission of this message sequence pp_i will not send any more messages to pp_j , until time t_j . Then, in the new proposed scheme, lp_i sends (t_j, null) to lp_j after sending the genuine message sequence. Since lp_i knows the state of the corresponding pp up to time T_i , it can predict all messages (that are to be sent) and absence of messages, at least up to T_i . Therefore, every outgoing edge will have a last message on it with time component equal to or greater than T_i . Note that only the last message sent along an edge may be a null message, in any iteration.

Reception of a null message is treated in the same manner as the reception of any other message: it causes the lp to update its internal state including the clock value and (possibly) send messages.

Suppose it is required to simulate the physical system up to some time z . Then every source must send messages until the t -component of the last message equals z ; if no non-null message exists with this property then finally (z, null) should be sent.

Example 4-1

Consider the physical system shown schematically in Figure 4-1, below.

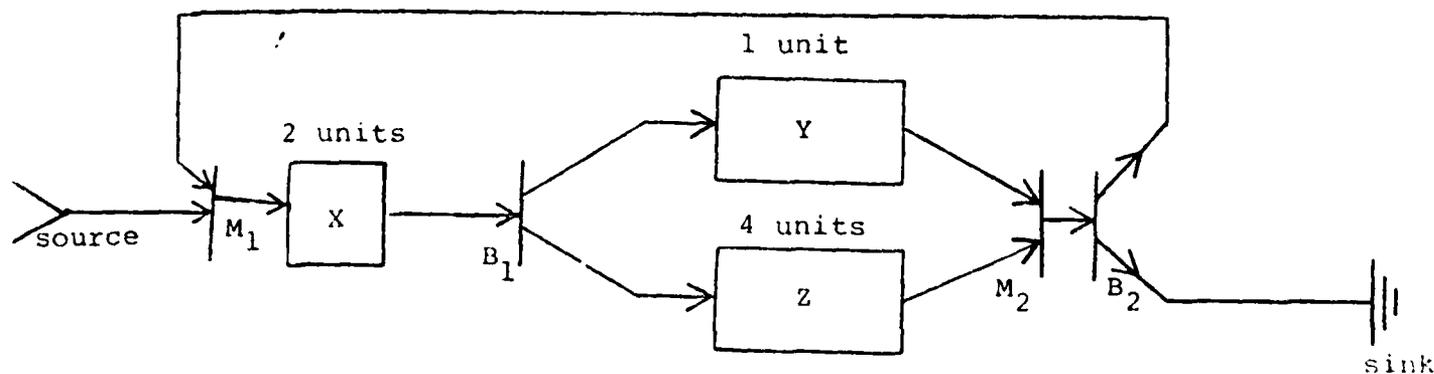


Figure 4-1: A Physical System with Loop

We will study the progress of one possible simulation run of this physical system. The source sends out jobs which are processed at X for 2 time units. Jobs are routed alternately to Y and Z from B_1 . Y processes a job for 1 unit and Z for 4 units.

Every job loops through the system twice, i.e., the first time a job arrives at B_2 , it is sent back to M_1 and on the second arrival at B_2 , it is sent to the sink.

Table 4-1 shows a succession of message transfers, where each horizontal row is a time slice and each column corresponds to a single activity of one of the processes. Concurrency is apparent because there are several activities happening at one time slice, i.e., in one row.

4.3. Correctness of the Simulation Algorithm

The partial correctness results of the last chapter still apply. The only difference now is the presence of null messages. We define the simulation to be *correct* at some point, if it is correct according to the definition of chapter 3 after ignoring null messages.

Theorem 1: Simulation is correct at every point

Proof: The proof is almost identical to the previous proof and hence omitted here.

The next theorem shows the power of adding null messages: we show that we have a deadlock-free system which can simulate a physical system up to time z .

Theorem 2: Assume that every source process sends messages until the t -component of a message equals z . Then every lp will simulate the corresponding pp, at least up to z .

Proof: Consider the point where the simulation terminates, i.e., where all messages that have been sent have been received and no lp has any outstanding message to send. The following observation is critical: for every lp (except a source lp) there exists an incoming edge to that lp whose edge clock value is less than or equal to the edge clock value of every outgoing edge from that lp. This observation follows because: (1) an lp that has received messages at least up to t along every input edge must have sent messages (t', m') , $t' \geq t$, along every outgoing edge, and (2) every message that has been sent has been received when simulation terminates. Note that (1) could not be asserted in the basic scheme because an lp need not send out messages with higher t -component values than the input messages.

We now claim that the edge clock value for every edge is at least z . If not, consider an edge e_1 for some lp, whose edge clock value is t_1 , with $t_1 < z$. According to the above observation, there exists an edge e_2 , which is an incoming edge to this lp, such that e_2 's edge clock value is t_2 , where $t_2 \leq t_1$. Continuing in this manner, we can construct a sequence of edges, $e_1, e_2, \dots, e_p, \dots$ such that for all i , e_{i+1} is a predecessor edge of e_i and $t_{i+1} \leq t_i$ and we have, $t_1 < z$. Since the physical network is finite, we will eventually either (i) get to a source lp, or (ii) we will have a cycle of edges. In the first case, since every source lp sends messages until the t -component

Table 4-1: Message Transmissions in the Simulation of Example 1

Step	Source Snds	M ₁ Rec	M ₁ Snds	X Rec	X Snds	B ₁ Rec	B ₁ Snds	E ₁ Snds	B ₁ Snds Z	Y Rec	Y Snds	Z Rec	Z Snds	M ₂ Rec	M ₂ Snds	B ₂ Rec	B ₂ Snds M ₁	B ₂ Snds Sink
1					(2, null)						(1, null)		(4, null)					
2						(2, null)												
3							(2, null)	(2, null)	(2, null)					(1, null)				
4										(2, null)		(2, null)		(4, null)				
5	(2, J ₁)									(3, null)			(6, null)		(1, null)			
6														(6, null)		(1, null)		
7	(3, J ₂)	(2, J ₁)												(3, null)			(1, null)	(1, null)
8	(6, J ₃)	(3, J ₂)																
9		(4, null)														(3, null)		
10		(3, null)	(1, null)															
11			(2, J ₁) (3, J ₂)	(1, null)														
12		(6, J ₃)		(2, J ₁)	(3, null)													
13					(4, J ₁)	(3, null)												
14				(3, J ₂)			(3, null)	(3, null)										
15					(6, J ₂)	(4, J ₁)				(3, null)			(7, null)					
16						(6, J ₂)	(4, J ₁)	(4, J ₁)	(4, null)		(4, null)							
17							(6, null)	(6, J ₂)	(6, J ₂)	(4, J ₁)				(4, null)				
18										(5, J ₁)	(6, J ₂)		(8, null)		(4, null)			
19												(10, J ₂)	(5, J ₁)					
20															(5, J ₁)	(4, null)		
21														(7, null)			(4, null)	(4, null)
22		(4, null)												(8, null)		(5, J ₁)		
23		(5, J ₁)												(10, J ₂)			(5, J ₁)	(5, null)
24																		
25			(5, J ₁)															
26					(6, null)													

of the last message sent is z , we can not have edge clock value of any outgoing edge of a source lp smaller than z . In the second case, all edge clock values in the cycle are equal to t_1 and $t_1 < z$. From the predictability property (chapter 2), for this cycle and this t_1 , there exists a pp , say pp_j , whose outputs can be determined beyond t_1 , given its inputs up to t_1 . Hence lp_j has some messages to send, which contradicts our assumption that the simulation has terminated. Therefore the edge clock value of every edge is at least z and hence the simulation clock value is at least z .

We have implicitly used the fact that for any finite z , only a finite number of messages may be transmitted in the logical system. This is derived from the predictability property, in which the parameter ϵ , $\epsilon > 0$, is a fixed quantity. A more rigorous proof of this boundedness property may be found in [7].

Discussion

It is interesting to note that the simulator never deadlocks: if the physical system deadlocks, the simulator continues computation by transmitting *null* messages with increasing t -values. This correctly simulates the corresponding physical situation, in that while time progresses, no messages are transmitted in the physical system. Ultimately, the simulator will terminate with every clock value at least at z . The simplicity of this scheme is one of its most attractive points. It requires small coding changes in existing distributed simulations to send out *null* messages. Furthermore, the requirement of unbounded buffers between two lp 's is not really necessary. The same results hold if there are only a finite number of buffer spaces between every lp_i and lp_j and lp_i has to wait to send if all buffer spaces are currently full. The proof that there is no deadlock in this situation is essentially contained in [7].

Empirical studies [28] show that this scheme is quite efficient for acyclic networks. Several factors seem to affect the efficiency:

(1) Degree of Branching in the Network

Consider a network with one source and one sink. The number of distinct paths between the source and the sink is a (rough) measure of the amount of branching in the network. Null messages tend to get created at branches and they may proliferate at all successive branches (if not subsumed). So one would expect that the fewer the number of branches, the better the performance. Empirical studies [28] seem to confirm this. Theoretically optimum efficiency is achieved for a tandem network (the assembly line example of chapter 2, Example 2-4), and excellent results are obtained for low-branching type networks. In general, acyclic networks exhibit reasonably good performance levels. Note that the metric of interest in performance calculations, is the *turnaround time*, i.e. the amount of time

it takes to complete the simulation, rather than processor utilization, i.e. the fraction of time the processors are utilized. In fact, one would expect the processors to be lightly utilized. The other parameter of interest, line bandwidth, has not received adequate attention.

Experiments were carried out by Peacock, Wong and Manning [24,25] on networks of various topologies. Their conclusions: "for some topologies of queueing networks models, this approach results in a speedup in the total time to complete a given simulation. However, for other topologies, especially those with loops, the speed-up may not be significant." They also investigated several different ways of partitioning the physical network so that more than one pp may be implemented on one lp.

(2) Time-Out Mechanisms to Prevent Null Message Transmission

A slight modification may save a considerable number of message transmissions. A null message (t,m) has no effect if it is followed by another message (t',m') , $t' > t$. Therefore it may be efficient to delay transmissions of null messages in the hope that future messages received by an lp would make it unnecessary to transmit them at all. Clearly the amount of time, τ , that an lp waits before transmitting a null message is of importance. If $\tau = 0$, we have the algorithm as stated in this chapter. If $\tau = \infty$, null messages are never transmitted and then we have the basic algorithm of chapter 3, which may lead to deadlock. Other values of τ are of potential interest, but no empirical studies have been performed to substantiate our claims.

(3) Amount of Buffering on Edges

The number of buffer spaces on edges seem to have substantial effects on performance [26,28]. When the number of buffer spaces was reduced to 0, senders had to wait until the receivers were ready to receive, and a considerable amount of time seemed to be spent in waiting. The number of buffer spaces was then increased and the following rule was used to annihilate null messages: any message put in the buffer *after* a null message (and therefore with a higher t -component) annihilates any null message ahead of it still in the buffer. The annihilation rule is somewhat similar to the time-out mechanism. It was found that in the simulation of a certain class of queueing networks the performance improved rapidly until the number of buffer spaces on an edge approached 10, increased less rapidly until about 20, and remained essentially unchanged thereafter. These numbers however cannot be applied directly for other problems; we expect these numbers to depend on the type of problem and the speeds of processors and lines.

We discuss various issues related to empirical investigations in the next chapter.

4.4. Demand Driven Null Message Transmission

Another variation with null message transmissions is not to transmit a null message until asked to do so. In this scheme, an lp_i may receive an *inquiry* from another lp_j where there is an edge from lp_i to lp_j . lp_j sends an inquiry to find out when lp_i will send it the next message. If lp_i has a genuine message to send or a null message, which will advance the edge clock value, it will do so in response to this inquiry. If it cannot send any such message, lp_i must itself be waiting for one or more of its incoming edge clock values to advance and hence it propagates this inquiry backward along those edges. The inquiry may be propagated along a sequence of edges. lp_i must remember to respond to the inquiry as soon as it can, i.e., as soon as its own clock value advances. An lp may receive several inquiries before responding to any of them. In this case, it will propagate at most one, wait for the reply and then reply to the others.

A particularly interesting part of this scheme is the detection of deadlock. In a situation as in Example 3-5, an inquiry initiated by x is propagated backward and arrives at x . lp_x can then detect deadlock. Resolution of deadlock requires finding the lp which has the smallest edge clock value t along some input edge, ignoring the set of deadlocked edges. This lp can then assert that it will receive no more input, up to t , along the deadlocked edge. Therefore it continues simulation assuming that it has received (t, null) along the deadlocked edge. In this example, x is the only process having edges outside the deadlocked set. Therefore x simply stops waiting to receive from z and advances its clock based on input from the source alone.

The claim that the inquiry propagation mechanism does indeed detect deadlock and that at most one inquiry by an lp is outstanding at any time, is not entirely trivial to prove; see [10,11] for discussions of a similar problem and its proof. A reasonable heuristic for an lp to initiate an inquiry may be based on time-outs.

4.5. Rollback and Recovery

A scheme suggested by Jefferson and Sowizral [18] allows an lp to proceed with its computation, with the belief that it will receive no further input along an incoming edge if it has not received any during a certain time period. Suppose that lp_i changes its state from s to s' and sends out messages M_1, M_2, \dots , as a result of this belief. Suppose that in the future, a message is received along an edge which contradicts this assumption. Then the state of the lp must be rolled back to s ; in addition, states of other lp 's which may have received M_1, M_2, \dots must also be rolled back. It is proposed to use a stack in which some of the recent states of an lp may be retained; the bottom of the stack is a guaranteed correct state at some point, and hence there will be no further rollback beyond that state. "Antimessages" M_1', M_2', \dots are sent to cancel the effects of the corresponding messages and roll back the states of the lp 's which previously received M_1, M_2, \dots .

If processor speeds, speed of simulating a pp by an lp, line delays, etc. can be accurately predicted, this method may turn out to be quite practical. In such cases, one would expect to have few rollbacks. However it seems that, in general, large amounts of memory would be required to stack the states and a large number of antimessages will have to be transmitted whenever a rollback is required.

4.6. Circulating Marker for Deadlock Detection and Recovery

A suggestion has been made in [9] to let the basic simulation scheme deadlock, detect deadlock and recover from it. Deadlock is infrequent, as has been suggested by Quinlivan [26] from a number of empirical studies on queueing networks. Therefore one would expect this to be a viable alternative if deadlock detection can be implemented efficiently. Dev Kumar [20] has used a recent deadlock detection scheme [22] to implement such an algorithm. We now discuss his method and several of its variations.

Consider a marker that continuously circulates in a network. It follows a cycle of edges such that it traverses every edge of the network sometime during a cycle - such a cycle exists if the network is strongly connected; new edges may be added to the network to make it strongly connected. The marker is merely a special type of message. It initially starts at some lp. If an lp receives the marker, its obligation is to send the marker (along its designated route) within a finite time of being idle (i.e., not having anything more to send). We let the marker carry some information for deadlock detection, as described below.

Each lp will have a one-bit flag to show whether the lp has received or sent a message since the last visit of the marker. We say that an lp is *white* if it has neither received nor sent a message since the last visit of the marker to that lp; the lp is *black* otherwise. Initially all lp's are black. The marker declares deadlock when it finds that the last N lp's that it has visited were all white (when the marker arrived at the lp), where N is the number of lp's in the network. This result holds if messages between two lp's, including the marker, are received in the order sent; see [22] for a precise description and proof of this result.

We can use this scheme to detect and recover from deadlock. The marker, in addition to keeping the number of white lp's it has seen since it last saw a black lp, carries the minimum of "next-event-times" for the white lp's it visits: each white lp can report the time of the next event, assuming it receives no further messages, to the marker and the marker merely keeps track of the smallest of these, and the corresponding lp. When the marker detects deadlock, it knows the next event time and the lp at which this next event occurs. Therefore, it can restart that lp. Alternately, a central process may broadcast (send messages to all lp's) to advance their clocks to the next event time in the system.

The overhead messages in this case, are for marker transmissions. If deadlocks are infrequent, the marker may be made to move slowly (and therefore the deadlock may be detected quite some time after its occurrence) and hence the proportion of overhead messages to genuine messages will be low.

4.7. Circulating Marker for Deadlock Avoidance

The marker scheme of the last section can also be used for deadlock avoidance. The idea is to let the marker carry messages. If lp_i is sending the marker to lp_j , it may send a message (t, null) , advancing that edge's clock value as much as possible. If lp_i cannot advance the clock value of the edge to lp_j , it still must send the marker, without a message, in finite time. The marker carries no further information. Using essentially the same arguments as in theorem 2 of this chapter, the system can be shown to be deadlock-free.

Overhead messages are for marker transmission; however, unlike null messages there is no proliferation of such messages. Another way to view this scheme is to consider the marker as a circulating packet which carries only null messages (or is empty) and delivers the messages to their destinations. The number of null message transfers is bounded by the marker's rate of traversal. By suitably adjusting the speed of the marker, i.e., the length of time for which an lp holds the marker before sending it, we can expect to reduce the number of overhead messages and still avoid long delays by the lp 's.

Dev Kumar is currently investigating the performance of these schemes and several variations of these, including the use of multiple markers.

Chapter 5

Summary and Conclusion

5. Summary and Conclusion

In this chapter we summarize the discussions about distributed simulation, its status, problems and future research directions. We hope to have demonstrated that distributed simulation may be applied in every situation where sequential discrete event simulation can be applied. Our examples have been predominantly from the area of computer systems, since a queuing network description of a computer is a physical system in our method. However, our physical systems encompass a large variety of real world applications; the only difference from sequential simulation modelling is to think in terms of pp's and messages rather than entities and events.

We have presented the methods of distributed simulation, but we have not shown how these may be implemented on existing or future machines. We first note that simulation of a pp by an lp can be realized in any simulation language - some particularly suitable ones are SIMULA [13], CSP [16,19], MAY [2], ADA [1], DEMOS [5], SAMOA [21]. All these languages provide enough abstraction mechanisms to describe the behaviors of elementary components and message communications among them. Hence, we contend that distributed simulation requires nothing more than a language for creating sequential processes and specifying their communications.

Implementation of distributed simulation therefore reduces to implementation of a message-communicating set of processes on some architecture. The logical system should then be partitioned among various processors in such a manner that the message traffic among various parts is as low as possible. Message communication may be accomplished either through a common memory (messages are deposited in a common memory by the sender and removed by the receiver) or by other interaction mechanisms among processors. The important criterion is how loosely coupled the processors are. If two processors are tightly coupled, i.e., the logical processes on these processors exchange a large number of messages, then the processors must also exchange at least that many messages and therefore the message traffic will be heavy. If processors are loosely coupled, they can operate autonomously, i.e. without communicating with other processors, for longer periods of time. It is also easier to avoid deadlock among a set of logical processes if they are simulated on one processor.

We have not yet explored the possibility of deadlock detection by a global processor which continuously observes message transmissions through the common memory. Unlike the manager of the event-list in the sequential simulation of Chapter

2, this global processor remains completely passive, i.e., in the background, until it detects deadlock. The global processor can resolve deadlock in an elegant manner: it transmits a null message (by depositing it in the proper memory location) which advances the edge clock value of an appropriate edge, such as z_x in Example 3-5. This technique seems to be a viable alternative when simulation is attempted on multiple processors sharing a common memory.

Static partitioning of the physical network among a fixed number of processors requires preprocessing prior to simulation. Preprocessing is useful for many other reasons. In the circulating marker algorithm, preprocessing is needed to determine a (static) cyclic path for the marker. Preprocessing could also be used to partition the lp 's such that the amount of branching is reduced and cycles are mostly contained within one processor. Preprocessing can determine other simulation parameters such as when to time-out, sizes of buffers on edges, etc. This is an area that has been extensively studied for sequential simulations. It needs to be studied again for distributed simulation since the problems are somewhat different in nature.

We have sketched several variations of the basic scheme for deadlock resolution. There is little evidence yet of the superiority of any one scheme. The large number of heuristics suggests that some combination may be appropriate for particular problem domains. For instance, if we use a set of uniform processors among which message communication is expected to be regular, we can expect that deadlock will rarely arise and therefore (a slowly) circulating marker scheme would be preferable. The circulating marker scheme also seems to be attractive in that it can be used (hopefully without much overhead) in more general cases. Also the marker can be used to collect statistical information about the simulation itself and hence the simulation parameters, such as time-outs, can be dynamically changed.

We have not discussed specific architectures that can support simulation. There is not enough experience with distributed simulation to know (1) where distributed simulation spends most of its time, and (2) whether any architectural improvement would be uniformly useful for all problems. At present, any architecture that supports (static) creation of processes and communications among them would be appropriate.

The circulating marker scheme seems attractive for hardware implementation. A hardware marker, analogous to the token in a token-ring, could cycle among the processes. Processes send genuine messages as before. Our requirement that messages, including the marker, be delivered in sequence sent, along an edge, can be met as follows: when the marker is sent from lp_i to lp_j , it is given the t -component of the last message sent by lp_i to lp_j ; when the marker arrives at lp_j , it stays there until lp_j has received a message with a t -component equal to or higher than the one that the marker has. Otherwise the marker algorithm operates as before. Advantage of a hardware marker is that the *simulation will spend no time in overhead message*

transmissions. The simplicity of the marker traversal scheme makes it feasible to implement it in hardware.

We next discuss some of the glaringly open problems in distributed simulation. The most important current problem is empirical investigations of various heuristics on a wide variety of problems to establish, (1) which heuristics work well for which problems and on which machine architectures, (2) how to partition the physical system among a fixed set of processors, and (3) how to set simulation parameters such as time outs and buffer sizes, etc. Some of the difficulties in empirical studies are listed below. First, it is useful to have a distributed architecture on which measurement capabilities exist, for implementation of the distributed simulation algorithm. The advantage of such a scheme is that processor and line speeds are realistic and that the implementation is quite straightforward. Another possibility is to first use a sequential simulator to simulate a distributed architecture and then implement the simulation algorithm on this (simulated) distributed architecture. One advantage is that the architecture can be continuously varied and its effect on simulation studied. This is the approach that is currently being taken at the University of Texas at Austin. MAY [2], a sequential process simulation language is being used to describe the distributed architecture and the distributed simulation algorithm. MAY is itself a simulation tool and hence its statistics-gathering mechanisms are used to collect and analyze the behaviors of various distributed simulation schemes.

A major disadvantage of this 2-tier approach is the actual CPU time required to run experiments. Not only does each experiment take longer, but the ease with which the parameters of the experiments can be changed has encouraged us to attempt many more experiments. A multiprocessor architecture, perhaps with a common memory, would provide an ideal simulation environment.

Traditional simulation issues have not been addressed in this monograph: what data to collect, how to collect it in a distributed manner, how to repeat experiments for statistical validity (a new experiment may be started even before an older one is completely over), etc. We feel that it is premature to address these issues without a firm understanding and resolution of the most basic issues.

References

1. Reference Manual for the ADA Programming Language, United States Department of Defense, 1982.
2. Bagrodia, Rajive, MAY: A Process Based Simulation Language, Master's Thesis, Computer Sciences Department, University of Texas at Austin. 78712, May, 1983.
3. Bezivin, J. and Imbert, H., "Adapting a Simulation Language to a Distributed Environment," *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pp. 596-603, Ft. Lauderdale, Florida, 1983.
4. Birtwistle, G. M., Dahl, O. J., Myhrhaug, B., and Nygaard, K., Simula Begin, Auerbach Publishers Inc., Philadelphia, Pennsylvania, 1973.
5. Birtwistle, G., DEMOS: A System for Discrete Event Simulation, Macmillan Press, 1979.
6. Bryant, R. E., "Simulation of Packet Communication Architecture Computer Systems," Technical Report, MIT, LCS, TR-188, Massachusetts Institute of Technology, November, 1977.
7. Chandy, K. M. and Misra, J., "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, SE-5, No. 5, September, 1979.
8. Chandy, K. M., Misra, J. and Holmes, V., "Distributed Simulation of Networks," *Computer Networks*, Vol. 3, pp. 105-113, January, 1979.
9. Chandy, K. M. and Misra, J. "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations," *Communications of the ACM*, Vol. 24, No. 4, pp. 198-205, April, 1981.
10. Chandy, K. M., Misra, J. and Haas, L., "Distributed Deadlock Detection," *ACM Transactions on Computing Systems*, Vol. 1, No. 2, pp. 144-156, May, 1983.
11. Chandy, K. M. and Misra, J., "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems," *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Ottawa, Canada, August, 1982.
12. Christopher, T., et. al., "Structure of a Distributed Simulation System," *Proceedings of the 3rd International Conference on Distributed*

Computing Systems, Ft. Lauderdale, Florida, 1983.

13. Dahl, O. J., Myhrhaug, B. and Nygaard, K., "Simula 67 Common Base Language," Norwegian Computing Centre, Oslo, Norway, 1970.
14. Fishman, G. S., *Principles of Discrete Event Simulation*, A Wiley-Interscience Publication, John Wiley and Sons, New York, New York, 1978.
15. Franta, *Process View of Simulation*, Elsevier Computer Science Library, Operating and Programming Systems Series, P. J. Denning (ed.), Elsevier North-Holland Publisher, 1977.
16. Hoare, C. A. R., "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 11, pp. 934-941, November, 1978.
17. Holmes, Victor, *Parallel Algorithms on Multiple Processor Architectures*, Ph.D Thesis, December 1978, Computer Sciences Department, University of Texas at Austin, 78712.
18. Jefferson, D. R. and Sowizral, H. A., "Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control," Technical Report, The Rand Corporation, Santa Monica, California, July, 1982.
19. Kaubish, W. H. and Hoare, C. A. R., "Discrete Event Simulation Based on Communicating Sequential Processes," Technical Report, Department of Computer Science, The Queen's University, Belfast, N. Ireland.
20. Kumar, Devendra, (Ph.D Thesis in Preparation), Computer Sciences Department, University of Texas at Austin, 78712.
21. Lonow, G. and Unger, B., "Process View of Simulation in Ada," *1982 Winter Simulation Conference*, pp. 77-86, 1982.
22. Misra, J., "Detecting Termination of Distributed Computations Using Markers," *Proceedings of the Second ACM Principles of Distributed Computing Conference*, Montreal, Canada, August 17-19, 1983.
23. Nakagawa, T., et. al., "A Multi-Microprocessor Approach to Discrete System Simulation," *Proceedings of CompCon*, pp. 350-355, San Francisco, Spring 1980.
24. Peacock, J. K., Wong, J. W. and Manning, E. G., "Distributed Simulation Using a Network of Processors," *Computer Networks*, Vol. 3, No. 1, pp. 44-56, February, 1979.
25. Peacock, J. K., Wong, J. W. and Manning, E., "A Distributed Approach to Queuing Network Simulation," *Proceedings of the 4th Berkeley Conference on Distributed Data Management and Computer Networks*,

Berkeley, California, August, 1979, pp. 237-259.

26. Quinlivan, Bill, Deadlock Resolution in Distributed Simulation, Master's Thesis, Computer Sciences Department, University of Texas at Austin, 78712, May, 1981.
27. Reynolds, Paul, "A Shared Resource Algorithm for Distributed Simulation," *Proceedings of the 9th International IEEE Architecture Conference*, Austin, Texas, April, 1982.
28. Seethalakshmi, M., "A Study and Analysis of Performance of Distributed Simulation," Master's Thesis, Computer Science Department, University of Texas, 78712, May, 1979.
29. Takenouchi, H., et. al., "Parallel Processing Simulator for Network Systems Using Multi-Microcomputers," *Proceedings of CompCon*, pp. 55-62, Washington, D.C., Fall 1980.

J. Misra - Short Biography

Jayadev Misra received his Ph.D from Johns Hopkins University in 1972. He worked for IBM, Federal Systems Division, from 1973 to 1974 and since then he has been at the Computer Sciences Department at The University of Texas at Austin. He is currently a Professor at the University of Texas and a Visiting Professor at Stanford University in the Computer Systems Lab. Dr. Misra has published widely in the areas of distributed computing systems, programming methodology and algorithms. He has consulted for several corporations including IBM, NCR, GTE, INTEL, Boole and Babbage, and Information Research Associates in the design of hardware and software systems.

Index

- Absence of messages 37
- Abstraction mechanisms 47
- Acyclic networks 37, 41, 42
- Acyclic physical systems 8
- Ada 8, 47
- Alarm clock 22
- Antimessages 9, 43, 44
- Asynchronous Distributed Computation 27
- Asynchronous logical components 21

- Basic Scheme for Distributed Simulation 27
- Bezivin, J. 9
- Boundedness property 41
- Branching 41
- Broadcast 45
- Bryant, R. E. 8
- Buffer spaces 8, 41, 42
- Buffering on Edges 42

- Central process 9
- Chandy, K. M. 8
- Christopher, T. 9
- Chronological 33
- Chronology 28
- Circular pattern of waiting 33
- Circulating Marker 44, 45, 48
- Clock 6, 7, 21, 22, 23
- Common memory 47, 49
- Communication protocol 27
- Computer Network 19
- Correct input sequences 32
- Correct output sequences 32
- Correctly simulates 32, 35
- Correctness of the Simulation Algorithm 39
- CSP 47

- Data collection 8
- Deadlock 27, 33, 35
- Deadlock Avoidance 45
- Deadlock detection 8
- Deadlock resolution 37
- Deadlock-free system 39
- Deadlocked 27, 33
- Demand Driven Null Message Transmission 43
- DEMOS 8, 47
- Departure times 18
- Dependent 13
- Depends directly 12, 13
- Directed edges 27
- Discrete event simulations 6

- Edge 27, 28, 29
- Edge clock value 28, 32, 33
- Encapsulation of Physical Processes by Logical Processes 35
- Event 11, 12
- Event scheduling 8
- Event-list 6, 21, 22, 23

- FCFS 17, 28

First-Come-First-Served 17
FORTRAN IV 8
Franta 8

Global control 35
Global knowledge 34
Global time 9
GPSS 23

Heuristics 39, 48, 49
Holmes, Victor 8

Imbert, H. 9
Incoming edge 29
Independent 13, 14, 23, 25
Induction 32
Inquiry 43
Inquiry propagation 43

Jefferson, D. R. 9, 43

Kumar, Dev 9, 44, 45

Line bandwidth 42
Local knowledge 34
Local time 9
Logical processes 20, 27
Logical system 20, 21
Lp 27, 28, 29, 30, 31, 32, 33

Manning, E. G. 8, 42
Marker 44, 45, 48, 49
MAY 8, 47, 49
Message 5, 6, 7, 8, 9, 11
Message transmission delays 15
Minimum wait time 9
Misra, J. 8
Model 5, 6

Network 5, 9
Next-event-times 44
Null message 38, 42, 43

Outgoing edge 34
Overhead messages 9, 37, 45

Parallel execution 11, 25
Partial Correctness 32, 39
Partitioning of the physical network 48
Peacock, J. K. 8, 42
Physical processes 5, 7, 11
Physical system 6, 7, 15, 17, 19, 20, 21, 22, 23
Pipeline 30
Pp 11, 14, 15, 16, 17
Predictability 15, 17, 19, 20
Predictability property 41
Preprocessing 48
Processor utilization 42
Pseudo-random number 8

Queue 12, 16, 17
Queuing network 42, 44, 47

Quinlivan, Bill 8, 44

Race conditions 17
Realizability 15, 16, 17, 18, 20, 28
Receive 27, 29, 31, 32
Reception of a null message 38
Recover from deadlock 44
Recovery 43, 44
Report generation 8
Reynolds, Paul 8
Rollback 43, 44

SAMOA 8, 47
Seethalakshmi, M. 8
Send 27, 28, 29, 30, 33, 34
Sequential simulation algorithm 11, 21, 22, 24
Service time 17, 18
SIMULA 8, 47
Simulation 11, 14, 15, 17, 21, 22, 23
Simulation algorithm 23
Simulation clock value 41
Simulation parameters 48, 49
Simulation Snapshot 34
Simulator 20
Simultaneous events 23
Sink 32, 41
Source 30, 33, 38, 39, 41, 43
Sowizral, H. A. 9, 43
Stack 43, 44
Strongly connected 44
Subsystem 33
Synchronous physical system 21

Tandem network 42
Terminated 27
Termination detection 7, 9
Time driven simulation 6
Time slice 39
Time-Out 42, 43, 48
Turnaround time 42

Unbounded buffers 41

Wong, J. W. 8, 42

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Processor Queueing Disciplines in Distributed Systems		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Elizabeth Williams		8. CONTRACT OR GRANT NUMBER(s) AROSR 81-0205B
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Sciences Department University of Texas at Austin Austin, Texas 78712		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Dr. Robert N. Buchal AFOSR/NM Bolling AFB, DC 20332		12. REPORT DATE
		13. NUMBER OF PAGES 7 pages
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES Proceedings of the 1984 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, August 21-24, 1984, Cambridge, Massachusetts		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) A distributed program consists of processes, many of which can execute concurrently on different processors in a distributed system of processors. When several processes from the same or different distributed programs have been assigned to a processor in a distributed system, the processor must select the next process to run. The following two questions are investigated: What is an appropriate method for selecting the next process to run? Under what conditions are substantial gains in performance achieved by an appropriate method		

of selection? Standard processor queueing disciplines, such as first-come-first-serve and round-robin-fixed-quantum, are studied. The results for four classes of queueing disciplines tested on three problems are presented. These problems were run on a testbed, consisting of a compiler and simulator used to run distributed programs on user-specified architectures.

PROCESSOR QUEUEING DISCIPLINES IN DISTRIBUTED SYSTEMS

Elizabeth Williams[†]

Computer Systems Group, C-8
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Abstract - A distributed program consists of processes, many of which can execute concurrently on different processors in a distributed system of processors. When several processes from the same or different distributed programs have been assigned to a processor in a distributed system, the processor must select the next process to run. The following two questions are investigated: What is an appropriate method for selecting the next process to run? Under what conditions are substantial gains in performance achieved by an appropriate method of selection? Standard processor queueing disciplines, such as first-come-first-serve and round-robin-fixed-quantum, are studied. The results for four classes of queueing disciplines tested on three problems are presented. These problems were run on a testbed, consisting of a compiler and simulator used to run distributed programs on user-specified architectures.

1. Introduction

When a problem has large computational demands and there is a network of processors available, a programmer can utilize the computational power of many processors. The programmer divides a problem so that pieces of the problem can be computed in parallel. It is common to see processors connected by local area networks. To effectively run a distributed program on a local area network as well as other interconnection networks, a good queueing discipline must take into account that its processor and other processors have pieces of the same program.

When several processes from the same or different distributed programs have been assigned to a processor in a distributed system, an important design question is how a processor selects the next process to run. This problem has not been considered in a distributed environment. An interesting question arises: How do the processes at other processors and communication delays in the system impact the selection of the next process to run? As a beginning study we have inves-

tigated the standard queueing disciplines - first-come-first-serve, round-robin-fixed-quantum, preemptive priority, and nonpreemptive priority - in a distributed environment. The study shows that the response time metric can differ by 50% with different choices of queueing disciplines for three problems.

Another important question is under what conditions are substantial gains in performance achieved by an appropriate method of selection. Communication delays are a factor; thus a graph of the response time metric was plotted as communication delays varied for each of the three problems. Trends are observed in these graphs. A rationale for the trends is given based on several factors.

The queueing disciplines were studied with three problems that differ functionally and have different behavioral characteristics. The partial differential equation solver is based on an iterative grid technique that is similar to those used in multidimensional applications such as weather prediction, structural mechanics, hydrodynamics, heat transport, and radiation transport. The centralized monitor has the typical tree structure of hierarchically designed applications. The producer-consumer pairs represent a multiprogramming environment in the distributed system and each pair is representative of a large class of problems. The different behavioral characteristics are described in Section 5.

In Section 2 a model of the distributed architecture and the distributed language are described. The metric for comparing the performance of the different queueing disciplines and a description of the testbed are given in Section 3. In Section 4 we give a heuristic for assigning priorities for the priority dependent queueing disciplines. Section 5 describes the distributed programs and architectures on which each problem executes. The results are given in Section 6. Section 7 describes the impact of queueing disciplines. In Appendix A a more detailed description of the simulator is given.

2. Model of Distributed Computing

2.1. Distributed Architecture

The distributed architecture is characterized by the number of processors, the speed of each processor, the queueing discipline at each processor, and the lines that connect the processors. The lines may have different capacities, lengths, and error rates. The processors have no shared memory and they communicate only by messages. We assume that any processor can communicate with any other processor by routing messages through intermediate

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1984 ACM 0-89791-141-5/84/008/0113 \$00.75

[†]This research was done at the Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712; arpaenet address ew@lslal

processors over fixed paths.

2.2. Distributed Language

A program in the distributed language consists of processes that communicate and share data by using messages. The language is similar to CSP, which is described in [2]. The language uses synchronous (blocking) communication primitives; the sending process cannot proceed until the receiving process is ready to receive the message. The two message passing constructs are the I/O statements, SEND(I/O variable) and RECEIVE(I/O variable). For each corresponding pair, SEND(I/O variable) and RECEIVE(I/O variable), that is executed, there are two nonblocking messages sent at the protocol level that implements the language. In this language there is a static number of processes. Dynamic creation of processes is simulated by a process beginning execution only after some other process sends it a message.

2.3. Terminology

We define virtual line time for a message between two processors connected directly by a line as the product of the actual time to move the message over the line and a constant derived from line reliability and the overhead of lower level protocols. The actual time to move the message over the line is the usual function of message length in message units (packets), number of bits per message unit, line capacity, and line length. Virtual line time does not include the time a message waits to use the communication subnet. Virtual line time for a message between two processors is the sum of the virtual line times for the lines on the route. Currently in local area networks, lower level protocols executing in the processors usually reduce the physical line capacity by at least a factor of 10 for any message [1]. Virtual line time reflects this effective line capacity.

The message delay of a process for a synchronous communication as in CSP is a function of virtual line time, queuing at the port queues on the route in a store and forward network, and the processing, waiting, and queuing time of the corresponding process at its processor. Message delays can be very large compared to a process's processing time between communications.

In the testbed 1 unit of time can be thought of as 1 μ s. For local area networks where processors are 1 km apart, transmission rates of 10 Mbit/s are common. For a packet of 256 bits it takes approximately 29 μ s to send a packet over the line. With the factor of 10 or more for lower level protocols, 300 time units is a reasonable number for virtual line time in this model of a local area network.

An important performance factor to consider is the tradeoff of processing time versus communication delay. To do a study on this tradeoff we must vary either processing or communication time. Since each process of a program has a fixed amount of processing, we have chosen to vary the communication time to study this tradeoff. Even though virtual line time varies beyond what may seem reasonable for our model, we stress that it is the ratio of processing time to communication delay that is actually changing and the ratio is a more meaningful factor to consider.

For each problem in this paper, we assume all the processors have the same speed, all lines are identical, and a message unit is 256 bits. We also assume that on any simulation run all processors have the same queuing discipline

These assumptions are made to isolate the effects of the choice of queuing discipline from other system variables

3. Testbed and Metric

The metric for comparing various queuing disciplines is defined as follows. All the processes of a distributed program are assumed to start at time zero. Each process terminates at some time, $t(i)$. The metric is the sum over N processes of the termination times $t(i)$ divided by N , and is termed the average of the process termination times (APTT). APTT reflects both the instruction processing requirements of processes and the message delays. Total time, defined as the maximum $t(i)$, is not always a good metric for comparing queuing disciplines, because when message delays are very small, total time is comparable for all queuing disciplines.

The testbed runs distributed programs coded in the distributed language mentioned above, which is similar to CSP. In addition to the distributed program, the testbed also requires a specification of the distributed architecture. The testbed consists of a compiler, interpreter, and simulator. The compiler produces pseudo-instructions for the hypothetical processors in the distributed system. The interpreter executes the pseudo-instructions. The simulator manages the interpreter, processor queues, and port queues and executes protocol routines. The simulator is based on the work presented in [4] and was validated extensively using commercial analytical and simulation packages [3,5]. A more detailed description of the simulator is given in Appendix A.

4. Queuing Disciplines

The queuing disciplines tested were first-come-first-serve (FCFS), round-robin-fixed-quantum (RRFQ), nonpreemptive-priority (NPP), and preemptive-priority (PP) [4]. The two priority disciplines NPP and PP must assign priorities to the processes. In a PP discipline if an expected message arrives for a blocked process of higher priority, the blocked process preempts the currently running process. In the following discussion we motivate and give a heuristic for assigning priorities.

Suppose that there are three processes on a processor ready to execute and that only one of these processes, process 1, must ever communicate across a line with another process on a different processor. Processes 2 and 3 communicate with each other and process 1. A good discipline will first let process 1 execute and block for communication across the line. While process 1 is blocked, processes 2 and 3 are executed. Hopefully, a message will arrive for process 1 and wake it up so that it is ready to execute before processes 2 and 3 block. A poor discipline will always execute processes 2 and 3 before process 1. Thus, all processes are blocked until a message arrives for process 1; the processor is idle for a longer period of time waiting on a message. The good discipline reduces the idle periods of the processor and thus, decreases the time when the processor finishes executing all processes. The good discipline must also determine which of process 2 or process 3 to execute first. A good selection depends on the characteristics of these processes.

Generally we have observed that scheduling a single processor in a distributed architecture must be analyzed considering both the single processor (local component) and the distributed environment (global component). Our heuristic for assigning priorities is given as follows:

- Processes that communicate across a line are assigned high priority (highest priority when message delays are

large since the global component is more important).

- A process on which several other processes may wait (a bottleneck process) is assigned high priority (highest priority when message delays are small since the local component is more important).
- Any other processes are assigned lower priorities to approximate shortest-remaining-time-first (SRTF) [4].

Thus a good priority discipline should generally give highest priority to those processes communicating across a line in order to minimize the processor idle periods and thus to finish executing all processes at the processor sooner. The discipline should be preemptive so that messages over the line can be received by the corresponding process as quickly as possible. Choosing priorities using this heuristic is demonstrated in the problems in the next section.

A priority discipline with priorities assigned as described above is denoted by PPg for preemptive priority and NPPg for nonpreemptive priority. A preemptive priority discipline with priorities assigned in such a way as not to follow the heuristic given above is denoted by Ppp; processes that communicate across lines and bottleneck processes are assigned lowest priority, and all the other processes are assigned highest priority. We have found that PPg usually does better than FCFS, RRFQ, Ppp, and NPPg; Ppp does the poorest.

5. Problems

The problems tested are a partial differential equation solver (PDE), a centralized monitor (MONITOR), and a system of five producer-consumer pairs (PC's). For each problem we present a brief description of the program and a figure that represents the distributed program, architecture, assignment of processes to processors, and priorities for both PPg and NPPg. Each process is represented by a circle with the process number in the circle; the total instruction processing time requirement per process is given below each circle. The priority for a process is given above each circle. The number and average size in message units of messages sent at the program level between two communicating processes is given above each line as the ordered pair (number,size). Values for communication and processing time are obtained by running the program on the testbed with any assignment and architecture; for these programs these quantities are independent of the architecture and assignment. Circles enclosed in a box mean that the enclosed processes are assigned to one processor. For each problem the processors are identical and the virtual line time for a message unit is the same between pairs of processes that must communicate over a line.

5.1. Partial Differential Equation

We solve Laplace's partial differential equation on a grid with the outer edges of the grid given as boundary conditions. The iterative method used is Gauss-Seidel. The grid is partitioned into subgrids where each subgrid is some number of contiguous rows. Each subgrid is solved by a process in the same way a sequential program would solve the entire grid. A grid value is computed as the average of its four adjacent neighbors; thus, to compute a row of values, the two adjacent rows are required. Hence, a process must request the two rows contiguous to its subgrid from its two neighboring processes. An important property of these processes is that they must remain closely synchronized. No process can compute very far ahead because it requires rows that cannot be computed unless the other processes execute.

Figure 1 shows the structure of the problem that runs on two processors. The two processors are connected by a line with virtual line time for a message unit set at 592 time units. In previous work we found that the assignment indicated in Figure 1 is best for this architecture [5].

All processes are comparable; there is no bottleneck process because each process is logically equivalent and computes an equal number of rows. Since each process must execute one time per Gauss-Seidel step over the same size subgrid, there is no need to assign priorities to approximate SRTF. The two processes that communicate over the line are given highest priority. For PPg and NPPg, processes 3 and 4 were assigned highest priority at 1.0; the others were assigned lower priority at 2.0. For Ppp, processes 3 and 4 were assigned lowest priority at 2.0 and the others were assigned highest priority at 1.0. PPg performed the best of the disciplines tested.

5.2. Centralized Monitor

The centralized monitor consists of a resource process and three groups; each group consists of a requester process and its three user processes. Each user process executes some given amount of time and then makes a request to use the resource through its requester process. The requester process passes the user request on to the resource process. This is repeated 20 times before a user terminates. The processing times per iteration were chosen so that (1) there is a small, medium, and large processing user process at each processor and (2) the sum of the processing time of the users at each processor is approximately the same at each processor. An important property of these processes is that a user process can compute to termination even when no other user process has executed. However, a user process must share the resource and a requester process with other user processes.

Figure 2 shows the structure of the centralized monitor that runs on four processors. Processor 4 is connected directly to processors 1, 2, and 3. Each line has a virtual line time of 58 time units for a message unit. In previous work we found that the assignment indicated in Figure 2 is best for this architecture [5].

The requester processes are 10, 11, and 12. A requester process has high priority because it is a bottleneck and also because it communicates over a line. The user processes - 1 through 9 - at each processor are not identical because of differing processing requirements. The user processes are assigned priority using the average processing time between I/O statements to estimate CPU bursts and thus to approximate SRTF. For PPg and NPPg, requester processes 10, 11, and 12 get priority 1.0; user processes 1, 4, and 7 get priority 2.0; user processes 2, 5, and 8 get priority 3.0, user processes 3, 6, and 9 get priority 4.0. For Ppp, processes 10, 11, and 12 get priority 2.0, while all user processes 1 - 9 get priority 1.0. SRTF is an important component of the priority discipline because a user process with a small burst time can finish earlier than the others and thus decrease APTT. Resource process 13 has priority 1 for each priority discipline. It is the only process on its processor; thus the choice is arbitrary for each priority discipline. PPg performed the best of the disciplines tested.

5.3. Producer-Consumer Pairs

There are five producer-consumer pairs. Figure 3 shows the structure of the problem that runs on two processors. The two processors are connected by a line with virtual line

time for a message unit set at 346 time units. Processes 1 to 5 are producers; processes 6 to 10 are consumers. Each pair - (1,6) (2,7) and (3,8) - has one-third the processing requirement of each pair - (4,9) and (5,10). Each producer sends 40 messages to its corresponding consumer. An important property of this problem is that each producer-consumer pair can execute to termination independently of the other pairs.

One pair of processes communicates over the line and both are given highest priority. There are no bottleneck processes in this example. The two pairs with the large processing requirements should get lower priority to approximate SRTF. Priorities for PPg are assigned as follows: processes 3 and 8 get priority 1.0; processes 1, 6, 2, and 7 get priority 2.0; processes 4, 9, 5, and 10 get priority 3.0. For Pp, processes 3 and 8 get priority 2.0; the other processes get priority 1.0. Since each pair can terminate independently of the other pairs, one process waiting on a line cannot cause all the processes on that processor to block as can happen in the other two problems. For this problem PPg performed the best of the disciplines tested.

6. Results

The results for each program and its architecture are given in Table 1. Of the disciplines tested, PPg is the best while Pp is the poorest. RRFQ always does better than FCFS; this is probably due to its preemptive characteristic. The nonpreemptive priority discipline, NPPg, is poorer than RRFQ for both the PDE and MONITOR problems. The percentage increase in APTT from PPg to Pp as computed by $(\max \text{APTT} - \min \text{APTT}) / (\min \text{APTT})$ is 32% for PDE, 49% for MONITOR, and 57% for PC's.

We have also experimented with varying the virtual line time and thus the message delays. The same assignment of processes to processors was maintained. The graph for each problem is given in Figures 4 - 6. These graphs show some conditions under which the choice of queuing discipline has an impact.

7. Impact of Processor Queuing Disciplines

The graphs for each problem show different trends. The graph for the PDE shows that at small virtual line times, the choice of queuing discipline has no impact. The graph for the MONITOR shows that at large virtual line times, the choice has no impact. The graph for the PC's shows that the choice has an impact for all the virtual line times tested. In order to explain some of the trends in the graphs generated by this experiment, a rationale was developed. It is used to explain the trends in the graph where virtual line time and thus message delays vary. The rationale is a partial solution.

7.1. Rationale

The rationale is based on two assumptions. (1) A good heuristic for minimizing APTT is to minimize the idle time at each processor. (2) There are two components of a discipline, the local and global components, and they vary in their contribution to the scheduling of a processor.

For a simplistic rationale one can say two things. (1) When virtual line time is very small, the local component of the discipline is more important. (2) As virtual line time increases, the global component of the discipline becomes more important. However, this is not enough. The processing time must be considered.

A more careful rationale compares the delay a process

can incur when waiting on a message to the remaining processing to be done at its processor until all processes are blocked. For a send or receive statement, the delay $D(j)$ for I/O variable j is the time that the process is in a wait state for the I/O variable j . It is at most the sum of the virtual line time to the corresponding processor, the virtual line time from the corresponding processor, queuing time at the appropriate port queues, and the processing, waiting, and queuing time of the corresponding process at its processor. When a process communicates with its corresponding process on the same processor, the delay does not include any virtual line time or port queuing time.

A function is described that measures the processing that can be overlapped when a process waits for a message. Busytime(k,s,t) is the amount of processing time remaining at time t until all processes are blocked at processor k , which is scheduled by discipline s . It is a function of the problem, the burst times between communication statements in processes, the processor's queuing discipline, and incoming messages from other processors. We are interested in this function only at those times when a process enters a wait state.

Suppose a process enters a wait state at time w for I/O variable j on processor k . If busytime(k,s,w) $\geq D(j)$, then there is no idle period for the processor for this communication. A good global discipline should always try to maintain this inequality for each message over a line at all processors to avoid idle periods. If busytime(k,s,w) $< D(j)$, then an idle period will result from this communication. Note that an idle period can not happen when two processes on the same processor are ready to communicate with each other; one process or the other can execute. Idle periods for a processor can only result from a communication across a line. Thus, for each communication across a line, the ratio $D(j)/\text{busytime}(k,s,w)$ is defined when a process enters a wait state at time w .

For the preemptive priority discipline, PPg, the average R of these ratios at a processor can give us a measure of how busy a processor is for a problem. $R \leq 1$ implies that on the average a processor is not idle. However, processor idle periods cannot be avoided when communication delays are very large compared to the largest amount of processing available at a processor under any queuing discipline.

If $R \ll 1$ then for any communication the processor was usually busy when a process was waiting on a message. If $R \gg 1$ then for any communication the processor became idle most of the time while a process was waiting on a message. We have assumed that for any discipline there is a local and global component. It seems reasonable that (1) when $R \ll 1$, local scheduling is a more important component since the processor is infrequently idle and (2) when $R \gg 1$, global scheduling is a more important component since global scheduling is responsible for minimizing the idle periods.

R was estimated but not computed; thus the following analysis is qualitative. At each processor, R changes as virtual line time is varied. There are many factors to consider but generally we can say that when virtual line time increases for all lines, each $D(j)$ increases while each busytime(k,pp,w) does not increase, where pp is the discipline PPg. $D(j)$ increases because of the larger virtual line time. Busytime(k,pp,w) cannot increase because incoming messages arrive later. Thus, R at each processor increases as virtual line time increases for all lines.

R has been defined for a single processor. We can extend the idea of how the size of R relates to global and local components of scheduling at a single processor to the entire set of processors since R increases for all processors as virtual line time increases for all lines. We define a measure R^* for the entire set of processors as the average of all R .

The graphs in Figures 4 - 6 plot APTT versus virtual line time. For the PDE and centralized monitor when virtual line time varies over a reasonable range of values, we have estimated that R^* varies from much smaller to much larger than 1. For the producer-consumer pairs program, R^* could not be forced to vary because of the disconnected communication structure. Each producer-consumer pair assigned to the same processor can execute until termination. Thus at a processor k , $\text{busytime}(k, pp, w)$ is very large when the one process that must communicate over the line enters a wait state at time w . There are always two producer-consumer pairs to execute until their termination.

To analyze the trends in these graphs, we look at d , which is defined as the difference of the maximum APTT and the minimum APTT at a given virtual line time. The minimum APTT should correspond to a good discipline, $APTT_g$, while the maximum APTT should correspond to a poor discipline, $APTT_p$. Thus, d should give us a bound on how queuing disciplines can impact APTT. If d is small, the choice of queuing discipline has no impact because all disciplines produce approximately the same metric value. If d is large, the choice has an impact on performance because the good discipline and poor discipline produce metric values that are not close.

Different trends are observed in Figures 4 - 6. A rationale to explain these trends is:

- For $R^* \ll 1$, d can be large or small. The local component of scheduling is more important. If all processes are comparable (no bottleneck processes and each process has the same approximate processing burst), then all disciplines are comparable and d is small. If the processes are different then the discipline can make a difference and d is large.
- For $R^* \approx 1$, both the local and global components are important. The size of d depends on how the problem responds to the components.
- For $R^* \gg 1$, d can be large or small, and $d \rightarrow$ constant. The global component of scheduling is more important. Each processor is mostly idle until a message arrives. If only one process is ready at a CPU queue at a time and the processes order themselves, then d is 0 for large enough delays. This is the case for the centralized monitor. If all the processes become ready at a CPU shortly after a message arrives for a process L , then running L is important because it communicates across a line. d is the time when L is ready to run but the other processes are scheduled ahead of L . This time is constant for large enough delays, and thus d is a constant and can be large. This is the case for the PDE.

7.2. Discussion of Graphs

Each graph plots APTT as a function of virtual line time, where virtual line time varies from 50 to at most 1200 time units. Experiments were conducted outside this domain but were not plotted because no additional information was provided. Trends established at the endpoints continued beyond the interval plotted.

For the PDE, each process is logically comparable and each process works on the same size subgrid. For $R^* \ll 1$, the order in which processes are executed is not very important; thus there is little difference in the queuing disciplines and d is small. For $R^* \gg 1$, all processes at a processor block because they are closely synchronized and cannot proceed until the process, waiting on a message across the line, receives the necessary row. In Figure 1 for processor 2, this is process 4. Process 4 computes the points in the middle of its subgrid and then communicates with its neighbor process 5 on the same processor. At this point (1) process 4 is ready to begin the next iteration step and communicate with its neighbor process 3 on the other processor again and (2) process 5 and in turn its neighbor 6 are ready to run. d is the difference due to executing process 4 first or last. d depends on whether or not the disciplines overlap waits and processing. For PDE, d is large.

For the MONITOR, the three requester processes and the resource process are bottleneck processes. The user processes have different processing bursts. For $R^* \ll 1$, the choice of discipline has an impact and d is large. For $R^* \gg 1$, the user processes are all blocked most of the time waiting on the resource to get and process their requests. When a message arrives for a user process U , only U is unblocked since all the user processes are independent. U executes and sends a message to the requester without interruption. Since only one process at a time is on the CPU queue, the queuing discipline never has to make a choice; thus d is small.

For the PC's, the choice of discipline has an impact over the virtual line times tested. R^* does not vary over a large range for the virtual line times tested because of the disconnected structure of the problem; there is always a process ready to run. This keeps busytime large relative to the virtual line times tested; thus $R^* \ll 1$. Since pairs differ in their processing bursts it is important to approximate SRTE; d is large and thus the choice of discipline has an impact.

8. Conclusion

We have presented the results for five queuing disciplines tested on three problems. The disciplines tested are first-come-first-serve, round-robin-fixed-quantum, nonpreemptive-priority, and preemptive-priority with two sets of priorities. A heuristic is given to assign priorities. We found that the preemptive priority discipline with priorities assigned according to our heuristic was the best discipline tested. We also found that the choice of queuing discipline varied in its impact on performance. A rationale is given to predict when the choice of discipline has the most impact.

9. Acknowledgments

The author wishes to thank Professor K. Mani Chandy for suggesting this problem and providing valuable guidance during this research. This research was supported in part by Air Force Office of Scientific Research under grant AFOSR 81-0205. This paper was prepared under the auspices of the U. S. Department of Energy.

Appendix A

The testbed consists of a compiler and a simulator. The simulator includes operating system routines, network protocol routines, and an interpreter. The compiler produces P-code instructions (instructions for the hypothetical processor) for each process. The simulator has two types of events

- Interpret Pcode instructions for processes on processor i until time for the next event or until processor i has no processes to execute. We refer to this event as run processor i .
- message arrival at processor i .

Initially, there are no message arrival events, and all processors i that have processes to execute are represented by the event, run processor i . If several messages arrive at a processor at the same time, the messages are handled FCFS depending on the simulator's event list. If all processors are the same speed and Pcode execution time is the same for most instructions, then a run processor event will be the execution of exactly one Pcode instruction at that processor.

The network architecture of the testbed is based on the conventional ISO OSI reference model. We simulated enough layers to give a detailed model of distributed computing without actually building a system. We simulated the language layer (application), transport, and a simplified network layer. Below the network layer, the testbed assumes error-free full-duplex lines. This assumption is not quite as strict as it seems. The actual line time can be increased by a random number to approximate the time for protocol execution and lower level messages in the data link and physical layers. We defined this in Section 2.3 as the virtual line time.

The language layer at a processor provides the buffers for the messages that arrive at and whose destination is that processor. These message arrivals are passed directly from the network layer to the language layer, where an uninteruptable language layer protocol routine is executed.

The testbed was validated using commercial analytical and simulation packages. The commercial simulation package was used to model several problems and architectures to validate detailed aspects of the simulator. The analytical package was used to model higher level aspects of the testbed.

The testbed provides confidence interval estimates at the 90% level with relative widths less than 0.05 for various performance measures. In this paper we have reported only the midpoint of the confidence interval for the measure, APTT [5].

References

- [1] E. E. Balkovich, Digital Equipment Corporation; David Wood, Mitre Corporation; Dieter Baum, Hahn-Meitner-Institute, Germany; private communications, 1983.
- [2] C. A. R. Hoare, "Communicating Sequential Processes," *Comm. ACM*, August 1978, pp. 666-677.
- [3] *PAWS User's Manual, CADS User's Manual*, Information Research Associates, Austin, TX, 1981.
- [4] C. H. Sauer and K. M. Chandy, *Computer Systems Performance Modeling*, Prentice-Hall, 1981, Chapter 7.
- [5] E. A. Williams, *Design, Analysis, and Implementation of Distributed Systems from a Performance Perspective*, Ph.D. Thesis, The University of Texas at Austin, 1983.

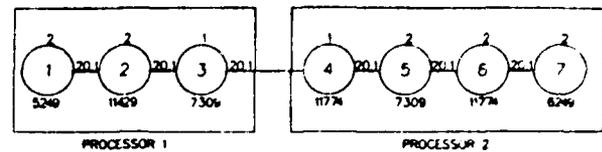


Figure 1. Structure of PDE Problem

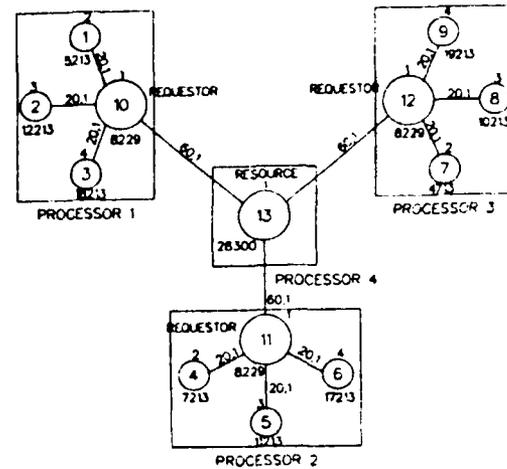


Figure 2. Structure of Centralized Monitor

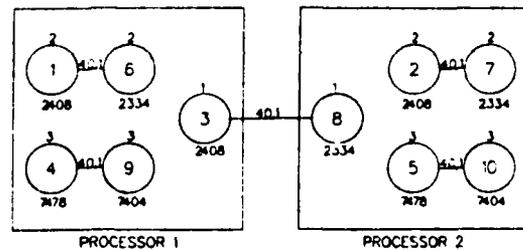
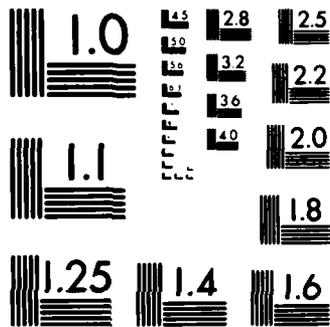


Figure 3. Structure for Producer - Consumer Pairs

APTT for each Problem and Discipline			
Queuing Discipline	PDE	Centralized Monitor	Producer-Consumers
FCFS	57071	57409	25182
RRFQ*	50831	49107	19320
NPPg	51625	57081	16268
PPg	45132	39910	14976
PPp	50638	59139	23461

* Quantum size PDE=100, MONITOR=75, PC's=80

Table 1



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

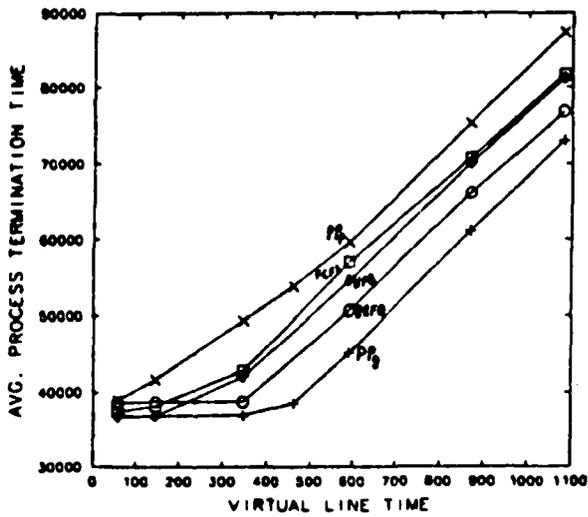


Figure 4. PDE: APTT versus Virtual Line Time

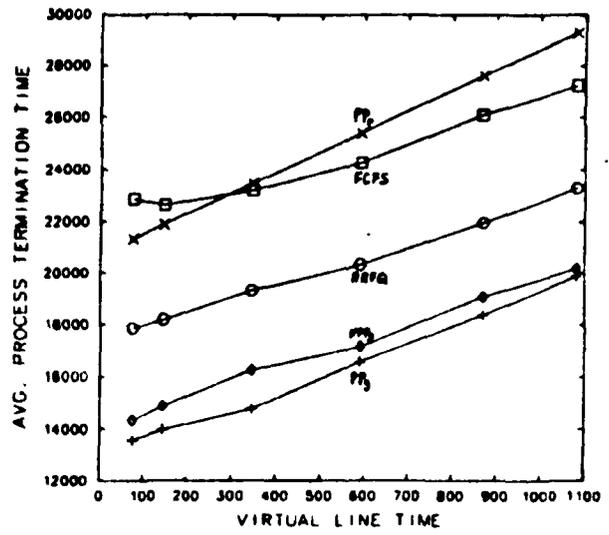


Figure 6. Producer-Consumer Pairs: APTT versus Virtual Line Time

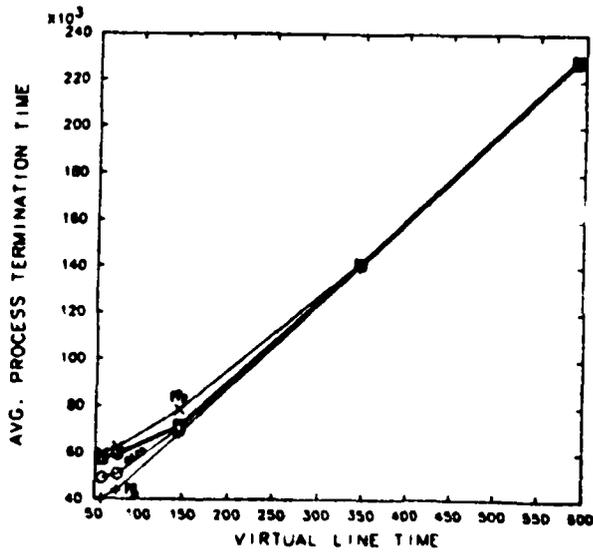


Figure 5. Centralized Monitor: APTT versus Virtual Line Time

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Effect of Queueing Disciplines on Response Times in Distributed Systems		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Elizabeth Williams		8. CONTRACT OR GRANT NUMBER(s) AFOSR 81-0205B
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Sciences Department University of Texas at Austin Austin, Texas 78712		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Dr. Robert N. Buchal AFOSR/NM Bolling AFB, DC 20332		12. REPORT DATE
		13. NUMBER OF PAGES 3 pages
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES Proceedings of: 1984 International Conference on Parallel Processing, August 22-24, 1984 Bellaire, Michigan		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) A distributed program consists of processes, many of which can execute concurrently on different processors in a distributed system of processors. When several processes from the same or different distributed programs have been assigned to a processor in a distributed system, the processor must select the next process to run. The question investigated is: What is an appropriate method for selecting the next process to run? Standard processor queueing disciplines, such as first-come-first-serve and round-robin-fixed-quantum, are		

studied. The results for four classes of queueing disciplines tested on three problems are presented. These problems were run on a testbed, consisting of a compiler and simulator used to run distributed programs on user-specified architectures.

THE EFFECT OF QUEUEING DISCIPLINES ON RESPONSE TIMES IN DISTRIBUTED SYSTEMS

Elizabeth Williams[†]
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Abstract - A distributed program consists of processes, many of which can execute concurrently on different processors in a distributed system of processors. When several processes from the same or different distributed programs have been assigned to a processor in a distributed system, the processor must select the next process to run. The question investigated is: What is an appropriate method for selecting the next process to run? Standard processor queueing disciplines, such as first-come-first-serve and round-robin-fixed-quantum, are studied. The results for four classes of queueing disciplines tested on three problems are presented. These problems were run on a testbed, consisting of a compiler and simulator used to run distributed programs on user-specified architectures.

1. Introduction

When several processes from the same or different distributed programs have been assigned to a processor in a distributed system, an important design question is how a processor selects the next process to run. This problem has not been considered in a distributed environment. An interesting question arises: How do the processes at other processors and communication delays in the system impact the selection of the next process to run? As a beginning study we have investigated the standard queueing disciplines - first-come-first-serve, round-robin-fixed-quantum, preemptive priority, and nonpreemptive priority - in a distributed environment. The study shows that the response time metric can differ by 50% with different choices of queueing disciplines for three problems.

The queueing disciplines were studied with several problems that represent three important classes of problems. The partial differential equation solver is based on an iterative grid technique that is similar to those used in multidimensional applications such as weather prediction, structural mechanics, hydrodynamics, heat transport, and radiation transport. The centralized monitor has the typical tree structure of hierarchically designed applications. The producer-consumer pairs represent a multiprogramming environment in the distributed system and are representative of a large class of problems.

In Section 2 a model of the distributed architecture and the distributed language are described. The metric for comparing the performance of the different queueing disciplines and a description of the testbed are given in Section 3. In Section 4 we give a heuristic for assigning priorities for the priority dependent queueing disciplines. Section 5 describes the distributed programs and architectures on which each problem executes. The results are given in Section 6.

2. Model of Distributed Computing

2.1. Distributed Architecture

The distributed architecture is characterized by the number of processors, the speed of each processor, the queueing discipline at each processor, and the lines that connect the processors. The lines may have different capacities, lengths, and error rates. The processors have no shared memory and they communicate only by messages. We assume that any processor can communicate with any other processor by routing messages through intermediate processors over fixed paths.

[†]Present address: Computer Systems Group, C-8, Los Alamos National Laboratory, Los Alamos, New Mexico 87545

2.2. Distributed Language

A program in the distributed language consists of processes that communicate and share data by using messages. The language is similar to CSP, which is described in [2]. The language uses asynchronous (blocking) communication primitives, the sending process cannot proceed until the receiving process is ready to receive the message. For each message sent at the program level, there are two messages sent at the protocol level that implements the language. In this language there is a static number of processes. Dynamic creation of processes is simulated by a process beginning execution only after some other process sends it a message.

2.3. Terminology

We define virtual line time for a message between two processors connected directly by a line as the product of the actual time to move the message over the line and a constant derived from line reliability and the overhead of lower level protocols. The actual time to move the message over the line is the usual function of message length in message units (packets), number of bits per message unit, line capacity, and line length. Virtual line time does not include the time a message waits to use the communication subnet. Virtual line time for a message between two processors is the sum of the virtual line times for the lines on the route. Currently in local area networks, lower level protocols executing in the processors usually reduce the physical line capacity by at least a factor of 10 for any message [1]. Virtual line time reflects this effective line capacity.

The message delay of a process for a synchronous communication as in CSP is a function of virtual line time, queueing at the port queues on the route in a store and forward network, and the processing, waiting, and queueing time of the corresponding process at its processor. Message delays can be very large compared to a process's processing time between communications.

In the testbed 1 unit of time can be thought of as 1 μ s. For local area networks where processors are 1 km apart, transmission rates of 10 Mbit/s are common. For a packet of 256 bits it takes approximately 29 μ s to send a packet over the line. With the factor of 10 or more for lower level protocols, 300 time units is a reasonable number for virtual line time in this model of a local area network.

For each problem in this paper, we assume all the processors have the same speed, all lines are identical, and a message unit is 256 bits. We also assume that on any simulation run all processors have the same queueing discipline. These assumptions are made to isolate the effects of the choice of queueing discipline from other system variables.

3. Testbed and Metric

The metric for comparing various queueing disciplines is defined as follows. All the processes of a distributed program are assumed to start at time zero. Each process terminates at some time, $t(i)$. The metric is the sum over N processes of the termination times $t(i)$ divided by N , and is termed the average of the process termination times (APTT). APTT reflects both the instruction processing requirements of processes and the message delays. Total time, defined as the maximum $t(i)$, is not always a good metric for comparing queueing disciplines, because when message

delays are very small, total time is comparable for all queueing disciplines.

The testbed runs distributed programs coded in the distributed language mentioned above, which is similar to CSP. In addition to the distributed program, the testbed also requires a specification of the distributed architecture. The testbed consists of a compiler, interpreter, and simulator. The compiler produces pseudo-instructions for the hypothetical processors in the distributed system. The interpreter executes the pseudo-instructions. The simulator manages the interpreter, processor queues, and port queues and executes protocol routines. The simulator is based on the work presented in [4] and was validated extensively using commercial analytical and simulation packages [3,5].

4. Queueing Disciplines

The queueing disciplines tested were first-come-first-serve (FCFS), round-robin-fixed-quantum (RRFQ), nonpreemptive-priority (NPP), and preemptive-priority (PP) [4]. The two priority disciplines NPP and PP must assign priorities to the processes. In a PP discipline if an expected message arrives for a blocked process of higher priority, the blocked process preempts the currently running process. In the following discussion we give a heuristic for assigning priorities.

Generally we have observed that scheduling a single processor in a distributed architecture must be analyzed considering both the single processor (local component) and the distributed environment (global component). Our heuristic for assigning priorities is given as follows:

- Processes that communicate across a line are assigned high priority (highest priority when message delays are large since the global component is more important).
- A process on which several other processes may wait (a bottleneck process) is assigned high priority (highest priority when message delays are small since the local component is more important).
- Any other processes are assigned lower priorities to approximate shortest-remaining-time-first (SRTF) [4].

Thus a good priority discipline should generally give highest priority to those processes communicating across a line in order to minimize the processor idle periods and thus to finish executing all processes at the processor sooner. The discipline should be preemptive so that messages over the line can be received by the corresponding process as quickly as possible. Choosing priorities using this heuristic is demonstrated in the problems in the next section.

A priority discipline with priorities assigned as described above is denoted by PPg for preemptive priority and NPPg for nonpreemptive priority. A preemptive priority discipline with priorities assigned in such a way as not to follow the heuristic given above is denoted by PPg: processes that communicate across lines and bottleneck processes are assigned lowest priority, and all the other processes are assigned highest priority. We have found that PPg usually does better than FCFS, RRFQ, PPg, and NPPg, PPg does the poorest.

5. Problems

The problems tested are a partial differential equation solver (PDE), a centralized monitor (MONITOR), and a system of five producer-consumer pairs (PC's). For each problem we present a brief description of the program and a figure that represents the distributed program, architecture, assignment of processes to processors, and priorities for both PPg and NPPg. Each process is represented by a circle with the process number in the circle; the total instruction processing time requirement per process is given below each circle. The priority for a process is given above each

circle. The number and average size in message units of messages sent at the program level between two communicating processes is given above each line as the ordered pair (number,size). Values for communication and processing time are obtained by running the program on the testbed with any assignment and architecture. For these programs these quantities are independent of the architecture and assignment. Circles enclosed in a box mean that the enclosed processes are assigned to one processor. For each problem the processors are identical and the virtual line time for a message unit is the same between pairs of processes that must communicate over a line.

5.1. Partial Differential Equation

We solve Laplace's partial differential equation (PDE) on a grid with the outer edges of the grid given as boundary conditions. The iterative method used is Gauss-Seidel. The grid is partitioned into subgrids where each subgrid is some number of contiguous rows. Each subgrid is solved by a process in the same way a sequential program would solve the entire grid. A grid value is computed as the average of its four adjacent neighbors, thus, to compute a row of values, the two adjacent rows are required. Hence, a process must request the two rows contiguous to its subgrid from its two neighboring processes.

Figure 1 shows the structure of the problem that runs on two processors. The two processors are connected by a line with virtual line time for a message unit set at 592 time units. In previous work we found that the assignment indicated in Figure 1 is best for this architecture [5].

All processes are comparable; there is no bottleneck process because each process is logically equivalent and computes an equal number of rows. Since each process must execute one time per Gauss-Seidel step over the same size subgrid, there is no need to assign priorities to approximate SRTF. The two processes that communicate over the line are given highest priority. For PPg and NPPg, processes 3 and 4 were assigned highest priority at 1.0, the others were assigned lower priority at 2.0. For PPg, processes 3 and 4 were assigned lowest priority at 2.0 and the others were assigned highest priority at 1.0.

5.2. Centralized Monitor

The centralized monitor consists of a resource process and three groups; each group consists of a requester process and its three user processes. Each user process executes some given amount of time and then makes a request to use the resource through its requester process. The requester process passes the user request on to the resource process. This is repeated 20 times before a user terminates. The processing times per iteration were chosen so that (1) there is a small, medium, and large processing user process at each processor and (2) the sum of the processing time of the users at each processor is approximately the same at each processor.

Figure 2 shows the structure of the centralized monitor that runs on four processors. Processor 4 is connected directly to processors 1, 2, and 3. Each line has a virtual line time of 58 time units for a message unit. In previous work we found that the assignment indicated in Figure 2 is best for this architecture [5].

The requester processes are 10, 11, and 12. A requester process has high priority because it is a bottleneck and also because it communicates over a line. The user processes - 1 through 9 - at each processor are not identical because of differing processing requirements. The user processes are assigned priority using the average processing time between I/O statements to estimate CPU bursts and thus to approximate SRTF. For PPg and NPPg, requester processes 10, 11, and 12 get priority 1.0; user processes 1, 4, and 7 get priority 2.0; user processes 2, 5, and 8 get priority 3.0; user processes 3, 6, and 9 get priority 4.0. For PPg, processes 10,

11, and 12 get priority 2.0, while all user processes 1 - 9 get priority 1.0. SRTF is an important component of the priority discipline because a user process with a small burst time can finish earlier than the others and thus decrease APTT.

5.3. Producer-Consumer Pairs

There are five producer-consumer pairs. Figure 3 shows the structure of the problem that runs on two processors. The two processors are connected by a line with virtual line time for a message unit set at 346 time units. Processes 1 to 5 are producers; processes 6 to 10 are consumers. Each pair - (1,6) (2,7) and (3,8) - has one-third the processing requirement of each pair - (4,9) and (5,10). Each producer sends 40 messages to its corresponding consumer.

One pair of processes communicates over the line and both are given highest priority. There are no bottleneck processes in this example. The two pairs with the large processing requirements should get lower priority to approximate SRTF. Priorities for PPg are assigned as follows: processes 3 and 8 get priority 1.0; processes 1, 6, 2, and 7 get priority 2.0; processes 4, 9, 5, and 10 get priority 3.0. For PPp, processes 3 and 8 get priority 2.0; the other processes get priority 1.0. The producer-consumer pairs that are not split across two processors are independent of each other. These pairs can terminate independently of the other pairs; one process waiting on a line cannot cause all the processes on that processor to block as can happen in the other two problems.

6. Results

The results for each program and its architecture are given in Table 1. Of the disciplines tested, PPg is the best while PPp is the poorest. RRFQ always does better than FCFS; this is probably due to its preemptive characteristic. The nonpreemptive priority discipline, NPPg, is poorer than RRFQ for both the PDE and MONITOR problems. The percentage increase in APTT from PPg to PPp as computed by $(\max \text{APTT} - \min \text{APTT}) / (\min \text{APTT})$ is 32% for PDE, 49% for MONITOR, and 57% for PC's.

7. Conclusion

We have presented the results for five queuing disciplines tested on three problems. The disciplines tested are first-come-first-serve, round-robin-fixed-quantum, nonpreemptive-priority, and preemptive-priority with two sets of priorities. A heuristic is given to assign priorities. We found that the preemptive priority discipline with priorities assigned according to our heuristic was the best discipline tested.

8. Acknowledgments

The author wishes to thank Professor K. Mani Chandy for suggesting this problem and providing valuable guidance during this research. This research was supported in part by Air Force Office of Scientific Research under grant AFOSR 81-0205. This paper was prepared under the auspices of the U. S. Department of Energy.

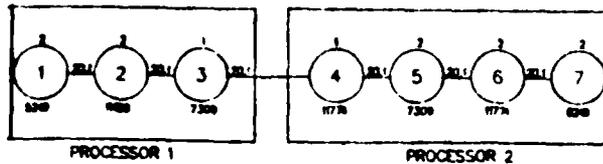


Figure 1. Structure of PDE Problem

References

- [1] E. E. Balkovich, Digital Equipment Corporation; David Wood, Mitre Corporation; Dieter Baum, Hahn-Meitner-Institute, Germany; Private Communications, 1983.
- [2] C. A. R. Hoare, "Communicating Sequential Processes," *Comm. ACM*, August 1978, pp. 666-677.
- [3] PAWS User's Manual, CADS User's Manual, Information Research Associates, Austin, TX, 1981.
- [4] C. H. Sauer and K. M. Chandy, *Computer Systems Performance Modeling*, Prentice-Hall, 1981, Chapter 7.
- [5] E. A. Williams, *Design, Analysis, and Implementation of Distributed Systems from a Performance Perspective*, Ph.D. Thesis, The University of Texas at Austin, 1983.

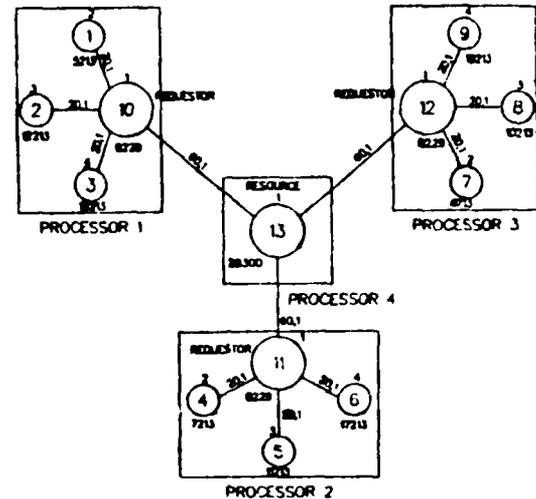


Figure 2. Structure of Centralized Monitor

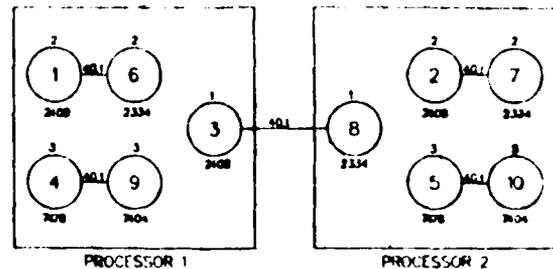


Figure 3. Structure for Producer - Consumer Pairs

APTT for each Problem and Discipline			
Queuing Discipline	PDE	Centralized Monitor	Producer-Consumers
FCFS	57071	57409	23182
RRFQ*	50631	49107	19320
NPPg	54625	57081	16268
PPg	45132	39910	14976
PPp	59638	59439	23461

* Quantum size: PDE-100; MONITOR-75; PC's-80

Table 1

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Drinking Philosophers Problem		5. TYPE OF REPORT & PERIOD COVERED interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) K. M. Chandy and J. Misra		8. CONTRACT OR GRANT NUMBER(s) AFOSR 81-0205B
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Sciences Department University of Texas at Austin Austin, Texas 78712		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Dr. Robert N. Buchal AFOSR/NM Bolling AFB, DC 20332		12. REPORT DATE
		13. NUMBER OF PAGES 24 pages
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES This paper will appear in the October 1984 issue of ACM Transactions on Programming Languages and Systems		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The problem of resolving conflicts between processes in distributed systems is of practical importance. A conflict between a set of processes must be resolved in favor of some (usually one) process and against the others: a favored process must have some property which distinguishes it from others. In order to guarantee fairness, the distinguishing property must be such that the process selected for favorable treatment is not always the same. We present a distributed implementation of an acyclic precedence graph in which the depth		

of a process (the longest chain of predecessors) is a distinguishing property. A simple conflict resolution rule coupled with the acyclic graph ensures fair resolution of all conflicts. To make the problem concrete, two paradigms are presented: the well known distributed dining philosophers problem, and a generalization of it: the distributed drinking philosophers problem.

The Drinking Philosophers Problem

K. M. CHANDY and J. MISRA

University of Texas at Austin

The problem of resolving conflicts between processes in distributed systems is of practical importance. A conflict between a set of processes must be resolved in favor of some (usually one) process and against the others: a favored process must have some property that distinguishes it from others. To guarantee fairness, the distinguishing property must be such that the process selected for favorable treatment is not always the same. A distributed implementation of an acyclic precedence graph, in which the depth of a process (the longest chain of predecessors) is a distinguishing property, is presented. A simple conflict resolution rule coupled with the acyclic graph ensures fair resolution of all conflicts. To make the problem concrete, two paradigms are presented: the well-known distributed dining philosophers problem and a generalization of it, the distributed drinking philosophers problem.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming; D.4.1 [Operating Systems]: Process Management—concurrency; mutual exclusion; synchronization; D.4.7 [Operating Systems]: Organization and Design—distributed systems

General Terms: Algorithms

Additional Key Words and Phrases: Asymmetry, dining philosophers problem

1. INTRODUCTION

We study the problem of fair conflict resolution in distributed systems. Conflicts can be resolved only if there is some property by which one process in every set of conflicting processes can be distinguished and selected for favorable treatment; that is, a conflict is resolved in favor of the distinguished process. In order to guarantee fairness, the distinguishing property must be such that the process selected for favorable treatment is not always the same. Traditional schemes for fair conflict resolution use priorities assigned to processes [2, 3, 7, 9, 10] or probabilistic selection [5, 8]. We propose a new approach by using the locations of shared resources as a distinguishing property. By introducing auxiliary resources, where needed, and by judiciously transferring resources among processes, we show that all conflicts can be resolved fairly. We propose a paradigm, the *drinking philosophers problem*, which captures the essence of conflict resolution problems in distributed systems. This problem is a generalization of the classical

This work was supported by the Air Force Office of Scientific Research under grant AFOSR 81-0205. Authors' present addresses: K.M. Chandy, Department of Computer Science, University of Texas at Austin, Austin, TX 78712; J. Misra, Computer Systems Laboratory, Stanford Electronics Laboratories, Department of Electrical Engineering, Stanford University, Stanford, CA 94305.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1984 ACM 0164-0925/84/1000-0632 \$00.75

ACM Transactions on Programming Languages and Systems, Vol. 6, No. 4, October 1984, Pages 632-646

dining philosophers problem [2, 3]. We present both problems formally in the following sections. This section presents an *informal* introduction to the problem of conflict resolution in distributed systems.

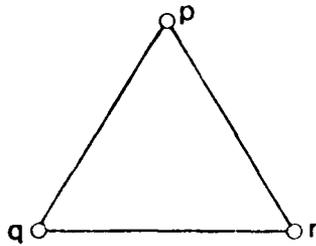
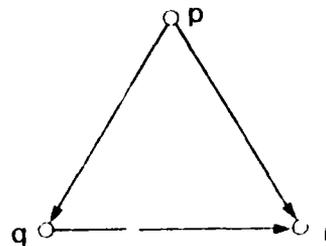
Two or more processes cannot execute certain actions simultaneously: for instance, two processes cannot hold "write locks" on the same data item at the same time. Conflicts arise when two or more processes attempt to carry out such actions simultaneously. The resolution of such a conflict requires that some processes be treated differently from others in the sense that the conflict be resolved *in favor* of some processes and *against* other conflicting processes. If all processes in a set of conflicting processes are indistinguishable (i.e., if every property that holds for one process also holds for the others), then it is impossible to resolve conflicts between them without resorting to random selection. This is because any deterministic algorithm that selects one of the processes for favorable treatment must carry out the selection on the basis of some property that holds for that process and not for the others. Therefore, if we do not wish to use probabilistic algorithms to resolve conflicts, we must ensure the following invariant:

Distinguishability. In every state of the system at least one process in every set of conflicting processes must be distinguishable from the other processes of the set.

An example of a distinguishing property is a process's identity number (provided that it is different from the identity numbers of all processes that it may conflict with).

Fairness. Usually we require not only that conflicts be resolved but also that they be resolved fairly, that is, conflicts should not always be resolved to the detriment of a particular process. If conflicts always occur in the same system state, a deterministic conflict resolution scheme will always resolve conflicts in the same way because the outcome of a deterministic scheme is a function of the system state. In this case conflict resolution will be unfair. Fairness requires that the states that obtain when conflicts occur not always be identical. An example of state information used to ensure that conflicts arise in different system states is *time*, where time may be determined by a centralized, global clock or by distributed logical clocks [7]: every request (which may result in a conflict) is timestamped, and a conflict between two requests is resolved in favor of the one with the smaller timestamp. However, conflicts between processes with equal timestamps must be resolved by using some other distinguishing property (such as process *IDs*). The state information used to ensure fairness may reside in a single process (the centralized solution) or it may be distributed. This paper is about distributed schemes to ensure (1) distinguishability and (2) fairness.

We describe our problem informally by using a graph model of conflict. A distributed system is represented by an undirected graph G with a one-to-one correspondence between vertices in G and processes in the system. Edge (u, v) exists in G if and only if there may be a conflict between (the processes corresponding to) vertices u and v . We assume that there is some mechanism that, in every state of the system, ascribes a precedence ordering to every pair of

Fig. 1. Graph G .Fig. 2. Graph H .

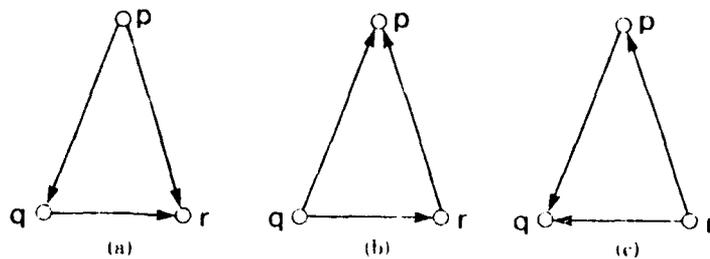
potentially conflicting processes so that one of the processes in the pair has precedence over the other. If there is a conflict between a pair of processes, the process with the lower precedence must yield to the process with greater precedence in finite time. We represent precedences between pairs of potentially conflicting processes by a *precedence graph* H , which is a graph identical to G except that each edge in G is given a direction in H as follows: An edge in H is directed away from the process with greater precedence toward the process with lesser precedence. For example, Figure 1 shows graph G for a system with 3 processes p , q , and r with the possibility of conflict between any pair of processes. Figure 2 shows graph H for a state of the system in which p has precedence over q and r , and q has precedence over r .

If H is acyclic, then the *depth* of a process in H is a distinguishing property by which a process can be distinguished from all processes that it may conflict with; *depth* of a process p in H is the maximum number of edges on any (directed) path to p from a process without any predecessors. Note that a process with no predecessor has depth 0. It follows that neighbors cannot have the same depth. For example, in Figure 2, the depth of p , q , and r are 0, 1, and 2, respectively.

If H is a cycle, the topology of H does not allow us to distinguish one process from another. We propose an algorithm that ensures that H is acyclic in every state of the system.

The acyclicity of H in every state of the system guarantees distinguishability but does not guarantee fairness. We wish to ensure that every process with conflicts has all its conflicts resolved in its favor in finite time; this requirement can be ensured by a guarantee that every process with conflicts rises to the top (i.e., to zero depth), in H in finite time. By the phrase, a "process p will rise to the top in H ," we mean that the state of the system will change, and hence H will change too, so that p will have no predecessor in the precedence graph H at some later state. If p is at depth 0, then any conflict that p has will be resolved in p 's favor in finite time because p takes precedence over all of its neighbors.

How should H change? The only way to change H is by changing the directions of the edges. We propose to implement H , and changes to H , by a distributed scheme, where each change in H is made locally at one process. Therefore our requirements are (1) H remains acyclic at all times, (2) H changes in such a manner that every conflicting process eventually rises to the top in H , and (3) each change to H be done locally at a process.

Fig. 3. Example illustrating rule for changing H .

To ensure acyclicity, we employ the following rule for changing H :

Acyclicity Rule. All edges incident on a process p may be simultaneously (i.e., in one atomic action) redirected toward p .

This transformation preserves acyclicity of H because no cycle containing p can be created by the transformation since there is no edge directed away from p after the transformation.

To ensure that every process in a conflict will rise to the top in H eventually we employ the following rule:

Fairness Rule. Within finite time after a conflict is resolved in favor of a process p at depth 0, p must yield precedence to all its neighbors.

This ensures that in the event that process at depth 0 is in conflict it will redirect all incident edges toward itself in finite time. This redirection of edges follows the acyclicity rule.

Example. Consider the precedence graph H shown in Figure 3a, where p , q , and r have depth 0, 1, and 2, respectively. If there are conflicts, then in finite time the directions of all edges incident on p will be reversed to give the precedence graph shown in Figure 3b, in which p , q , and r have depth 2, 0, and 1, respectively. If conflicts persist, in finite time the directions of all edges incident on q will be reversed to give the precedence graph in Figure 3c, in which p , q , and r have depth 1, 2, and 0, respectively.

The key issue is to devise a distributed implementation of H , as well as the acyclicity and fairness rules. The distributed aspect of the problem makes it nontrivial. The difficulty is that a process has to decide whether to yield or not to yield in a conflict, and the decision has to be made solely on the basis of the process's local state. It may not be possible to determine the direction of edges incident on a process only on the basis of the process's local state. Therefore we devise a distributed implementation of H and a scheme by which processes resolve conflicts by making local decisions based on partial information of H .

Our goal in this section was to discuss the concepts underlying distributed conflict resolution and the treatment has been informal. The following sections offer a more formal treatment of conflict resolution by defining and solving a specific problem: *The drinking philosopher problem*, which serves as a paradigm of conflict resolution problems.

2. THE DRINKING PHILOSOPHERS PROBLEM (DRINKERS PROBLEM)

The following problem is a generalization of the *dining philosophers problem* [2, 3], which has achieved the status of legend, since it captures the essence of many synchronization problems. Processes, called *philosophers*, are placed at the vertices of a finite undirected graph G with one philosopher at each vertex. A philosopher is in one of 3 states: (1) *tranquil*, (2) *thirsty*, or (3) *drinking*. Associated with each edge in G is a *bottle*.¹ A philosopher can *drink* only from bottles associated with his incident edges. A tranquil philosopher may become thirsty. A thirsty philosopher needs a nonempty set of bottles that he wishes to drink from. *He may need different sets of bottles for different drinking sessions. On holding all needed bottles, a thirsty philosopher starts drinking; a thirsty philosopher remains thirsty until he gets all bottles he needs to drink. On entering the drinking state a philosopher remains in that state for a finite period, after which he becomes tranquil. A philosopher may be in the tranquil state for an arbitrary period of time.*

Two philosophers are *neighbors* if and only if there is an edge between them in G . Neighbors may send messages to one another. Messages are delivered in arbitrary but finite time. Resources, such as bottles, are also encoded and transmitted as messages.

The problem is to devise a nonprobabilistic solution that satisfies the following constraints.

Fairness. No philosopher remains thirsty forever.

Symmetry. All philosophers obey precisely the same rules for acquiring and releasing bottles. There is no priority or any other form of externally specified static partial ordering among philosophers or bottles.

Economy. A philosopher sends and receives a finite number of messages between state transitions. In particular, permanently tranquil philosophers do not send or receive an infinite number of messages.

Concurrency. The solution does not deny the possibility of simultaneous drinking from different bottles by different philosophers.

Boundedness. The number of messages in transit, at any time, between any pair of philosophers is bounded. Furthermore, the size of each message is bounded.

The drinkers problem is a general paradigm for modeling conflicts between processes. Neighboring philosophers will be prevented from drinking simultaneously if they wish to drink from the same bottle—this situation models conflicts for exclusive access to a common file. Neighboring philosophers may drink simultaneously from different bottles—this situation models processes writing into different files.

We must prevent the system from entering states in which neighboring philosophers are indistinguishable. For example, consider philosophers arranged in a ring and a state in which each philosopher is drinking from his "left" bottle—philosophers cannot be distinguished in this state. If all philosophers are drinking from their left bottles and then require both bottles for their next drinking

¹The solution given in this paper also applies to multiple bottles on every edge. The assumption of one bottle per edge is made for brevity in exposition.

session, then the philosophers must remain thirsty forever because a deterministic algorithm cannot choose between indistinguishable philosophers. However, a system state is certainly possible in which all philosophers hold their left bottles. If we were to disallow this state, we would be disallowing a feasible state in which progress is being made, merely to solve our problem; disallowing feasible states violates our constraint of *concurrency*. We appear to be in a quandary because the constraints of symmetric processes, nonprobabilistic solutions, and concurrency are incompatible for this problem. The solution is to implement precedence graph H by using special "auxiliary" resources. The solution to the dining philosophers problem shows us how to implement H . Therefore we study the dining philosophers problem next. We then study the drinkers problem using the diners problem solution to implementing H .

3. THE DINING PHILOSOPHERS PROBLEM (DINERS PROBLEM)

The diners problem [2] is a special case of the drinkers problem in which every thirsty philosopher needs bottles associated with *all* its incident edges for *all* drinking sessions. We present a solution for this problem with the properties of fairness, symmetry, economy, concurrency, and boundedness. To distinguish between these two problems, we use the following terms for the diners problem, with the corresponding term for the drinkers problem in parentheses: *thinking* (*tranquil*), *hungry* (*thirsty*), *eating* (*drinking*), *fork* (*bottle*). The diners problem disallows neighbors from eating simultaneously. The drinkers problem allows neighbors to drink simultaneously provided that they are drinking from different bottles.

We first present an informal outline of the solution; the next section has a detailed formal description. A fork is either *clean* or *dirty*. A fork being used to eat with is dirty and remains dirty until it is cleaned. A clean fork remains clean until it is used for eating. A philosopher cleans a fork when mailing it (he is hygienic). A fork is cleaned only when it is mailed. An eating philosopher does not satisfy requests for forks until he has finished eating. The key issue is: which requests should a noneating philosopher defer? In our algorithm, a noneating philosopher defers requests for forks that are clean and satisfies requests for forks that are dirty.

This solution to the diners problem suggests an implementation of precedence graph H . The direction of an edge between two neighbors u and v is from u to v (i.e., u has precedence over v) if and only if (1) u holds the fork shared by u and v , and the fork is clean, or (2) v holds the fork, and the fork is dirty, or (3) the fork is in transit from v to u . Observe that the direction (from u to v) of the edge can change only when u starts eating. Furthermore, all edges incident on an eating philosopher are directed toward it. Hence we have an implementation of the acyclicity rule: The direction of edges incident on a process may be changed only in the following way—*all* edges incident on a process may be simultaneously directed toward it.

Initially all forks are dirty and are located at philosophers in such a way that H is acyclic. Hence the following is an invariant: H is acyclic.

Immediately upon completion of an eating session, a philosopher yields precedence to his neighbors. A hungry philosopher at depth 0 in H will commence

eating in finite time (because he has precedence over all his neighbors). By induction on depth, a hungry philosopher at depth k , $k \geq 0$, will eat in finite time because he has precedence over all philosophers at greater depth, and all philosophers at smaller depth will yield precedence to it in finite time.

A formal treatment of these arguments is found in the next section.

4. A HYGIENIC SOLUTION TO THE DINERS PROBLEM

4.1 Algorithm

We now give a precise description of the solution outlined in the last section. To simplify our description, we introduce a *request token* for each fork. Only the holder of the request token for fork f can request fork f . A request for a fork is made by sending the corresponding request token to the holder of the fork. It follows then that only one process—the holder of the request token for f —may request fork f and the requested process, after receiving the token, has the next chance to request the fork. Also, if a process holds a fork *and* the request token for the fork then his neighbor (with whom he shares the fork) has an outstanding request for the fork. We introduce the following Boolean variables:

$fork_u(f)$: philosopher u holds fork f ,
 $req_u(f)$: philosopher u holds the request token for fork f ,
 $dirty_u(f)$: fork f is at u and is dirty,
 $thinking_u/hungry_u/eating_u$: philosopher u is *thinking/hungry/eating*.

We drop the subscripts from the Boolean variables when the context is clear.

Each philosopher follows the rules given below for requesting and sending forks. In each case a rule is written as $g \rightarrow A$, where g is a condition and A is a sequence of actions. These rules constitute our solution to the diners problem. The set of rules forms a single guarded command [4].

- (R1) Requesting a fork f :
 $hungry, req(f), \sim fork(f) \rightarrow$
 send request token for fork f (to the philosopher with whom f is shared);
 $req(f) := false$
- (R2) Releasing a fork f :
 $\sim eating, req(f), dirty(f) \rightarrow$
 send fork f (to the philosopher with whom fork f is shared);
 $dirty(f) := false$;
 $fork(f) := false$
- (R3) Receiving a request token for f :
 upon receiving a request for fork $f \rightarrow$
 $req(f) := true$
- (R4) Receiving a fork f :
 upon receiving fork $f \rightarrow$
 $fork(f) := true$
 $\{ \sim dirty(f) \}$

We note that the statement of the diners problem defines transitions among states (*thinking, hungry, eating*) for a philosopher, and we furthermore have for any philosopher,

$$eating, fork(f) \Rightarrow dirty(f).$$

Initial Conditions

1. All forks are dirty. $\{\forall f, \text{dirty}_u(f) \text{ or } \text{dirty}_v(f) \text{ where } u, v \text{ are the philosophers who can use fork } f\}$.
2. Every fork f and request token for f are held by different philosophers. $\{\text{If fork } f \text{ is shared between philosophers } (u, v), \text{ then } u \text{ holds the fork and } v \text{ the token (i.e., } \text{fork}_u(f), \text{req}_v(f), \sim \text{fork}_v(f), \sim \text{req}_u(f)), \text{ or } v \text{ holds the fork and } u \text{ the token.}\}$
3. H is acyclic. $\{\text{The forks are initially placed in such a manner that } H \text{ is acyclic.}\}$

4.2 Proof of the Hygienic Solution for the Diners Problem

We show in this section that every hungry philosopher will eat. In addition to this fairness condition, we show that our solution has the properties of symmetry, economy, concurrency, and boundedness.

Fairness

LEMMA 1. H is always acyclic.

PROOF. Initially H is acyclic. The directions of edges in H may be affected only when a fork changes its status (dirty or clean) or its location. We will show that every change to H preserves acyclicity. Every transmission of a fork is accompanied by a change in its status from dirty to clean; this does not change the direction of any edge. A fork is dirtied when the philosopher u holding it, eats. In this case u must be holding all other forks associated with edges incident upon it, and they must all be dirty. u cannot then create a cycle in H because all edges upon u are directed toward it. \square

THEOREM 2. Every hungry philosopher eats.

The following proof is based on the fact that a hungry philosopher requesting a fork that is currently dirty will receive it (from rule R2), and since the fork is clean upon receipt the philosopher will hold it until he eats. A philosopher requesting a fork that is clean must make the request to a philosopher at a smaller depth and, by induction on depth, this philosopher will eat and then dirty the fork, in which case the first argument applies.

PROOF. Recall that the depth of a philosopher in H is the maximum number of edges along a path to that philosopher from one without predecessors. We prove the theorem by induction on depth of a hungry philosopher; the induction amounts to showing that hungry philosophers at depth k in every H eat, provided all hungry philosophers at depths smaller than k in every H eat, for all $k \geq 0$.

We will not do a special analysis for hungry philosophers at depth 0, because that case is subsumed by Case 1, below.

Let u, v be neighbors and u be hungry. We show that u holds or will hold the fork f corresponding to the edge (u, v) and will thereafter continue to hold it until u eats. If u holds the fork currently and holds it continuously until he eats, the result is trivial. Therefore assume that v holds the fork f sometime before u eats next. We do a case analysis on the status of f at the time that v holds the fork. At this time we have: $\text{hungry}_u, \sim \text{fork}_u(f), \text{fork}_v(f)$.

Case 1: f is dirty ($dirty_i(f) = true$). If $req_u(f)$ holds then u will request f (because precondition of rule R1 will hold) and subsequently $req_i(f)$ will hold; (otherwise $req_i(f)$ already holds. If $eating_i$ holds then at some later point (since eating is finite), $\sim eating_i$ holds, and all other predicates for rule R2 still hold. Therefore rule R2 will be applied by v , and u will eventually hold a clean fork. u will not release a clean fork until u eats.

Case 2: f is clean ($dirty_i(f) = false$). Every fork held by a nonhungry philosopher is dirty because

- (a) all forks are dirty initially,
- (b) only hungry philosophers receive clean forks, and
- (c) all forks held by eating philosophers are dirty.

Since f is clean, the philosopher v holding it must be hungry. Furthermore, because f is clean, (v, u) is an edge in H and hence $depth(v) < depth(u)$. According to the induction hypothesis, v eats and hence dirties f . Case 1 then applies. \square

Symmetry. It follows from the description of the algorithm that all philosophers follow the same rules.

Economy. The number of message sends and receives before a state transition is bounded as follows: if d is the number of neighbors of a philosopher, then no more than d requests or forks will be sent or received. More precisely, suppose a philosopher has e dirty forks when he transits to hungry state. Then he must send $d - e$ requests and receive a fork corresponding to each request. In addition, in the worst case, he may lose all e forks he had held initially and therefore have to request and receive them. Assume that a philosopher implements the latter situation by sending a fork and the request for it in one message. Then no more than $2d$ messages are needed before transiting to the eating state. The only messages received in the eating or thinking state are the requests for forks held by the philosopher and hence these do not exceed d . In the best case, a philosopher with permanently thinking philosophers as neighbors will receive no requests for forks and therefore may live a life (think and eat) free of interaction with others.

Concurrency. Our solution does not deny any feasible system state; that is, any state of the system in which neighboring philosophers are not eating is allowable in our solution. This is because the solution does not prevent a philosopher from entering the thinking or hungry state; the only restriction is in entering the eating state, and that is allowable when a hungry philosopher holds all forks, as required by the problem.

Boundedness. There are at most two messages - a fork and a request for a fork—in transit, between any two philosophers.

5. A SOLUTION TO THE DRINKERS PROBLEM

5.1 The Precedence Graph

Our solution to the drinkers problem uses precedence graphs discussed in Section 1. The solution to the diners problem demonstrates a distributed implementation of the precedence graph H . Fairness and the acyclicity of H are ensured by implementation of the fairness and acyclicity rules. It may appear that H provides

a simple resolution mechanism for any type of conflict, including conflict for bottles in the drinkers problem, since any conflict can be resolved in favor of the process with greatest precedence. However, there is a difficulty due to the *distributed* implementation of H . Given only the state of process u we can determine which of neighbors u or v has precedence *if u holds the fork*: If the fork is clean u has precedence, if it is dirty v has precedence. However, *if u does not hold the fork we cannot determine which of u or v has precedence from the state of u alone*. In this case u must make local decisions about holding on to or releasing bottles without using precedence graph H . This issue is discussed next in the context of the drinkers problem.

We use forks to implement H . Forks are *auxiliary resources* in the sense that their sole purpose is to implement precedence graph H . Forks are not part of the drinkers problem specification; they are part of the solution. The real resources in the drinkers problem are bottles. Our philosophers can eat and drink *simultaneously*, and we emphasize that *eating is an artifact of our solution*, used only to guarantee fair drinking. In our solution, the state of a philosopher is a pair (diner's state, drinker's state), where a diner's state is one of thinking, hungry, or eating and a drinker's state is one of tranquil, thirsty, or drinking. Our next step is to define the dining characteristics of our philosophers; the drinking characteristics are specified by the problem. We give rules that ensure that all thirsty philosophers drink in finite time.

Consider the state transitions of a dining philosopher. The only transitions that are decided by the philosopher are thinking-to-hungry and eating-to-thinking; the only transition completely specified by the diners problem is hungry-to-eating (which occurs when a philosopher holds all forks he needs). We now give rules for the dining philosopher to decide the point of the first two transitions.

(D1) *Thinking-to-Hungry Transition:*

A thinking, thirsty philosopher becomes hungry.

(D2) *Eating-to-Thinking Transition:*

An eating, nonthirsty philosopher starts thinking.

In the diners problem, a philosopher can think for arbitrary time though he must eat for finite time. Therefore our obligation, arising out of rules (D1) and (D2), is to ensure that each eating period is finite. This is accomplished by the rule (D3) given below.

(D3) *The Conflict Resolution Rule:*

Philosopher u sends a bottle to philosopher v , in response to a request from v , if and only if u does not need the bottle or [u is not drinking and does not hold the fork for the edge (u, v)].

Note that u 's decision to send or hold onto a bottle requested by v depends on whether u holds the fork associated with edge (u, v) , and *does not depend on whether u or v has precedence in H* . In particular, u must send the bottle to v if u has precedence over v , but u does not hold the fork associated with edge (u, v) . We must show that despite this fact, the algorithm is fair.

The basic idea is this: Suppose u has precedence over v (i.e., (u, v) is an edge in H), but v holds the fork (i.e., the fork is dirty), and suppose u requests a bottle

held by v . We require that u not only request the bottle held by v , but that u also request the fork. We show (from the solution to the diners problem) that in finite time v will yield the fork to u after which it must also yield the bottle to u . Thus, the algorithm ensures that if u has precedence over v in H then eventually the conflict resolution rule causes conflicts for bottles between u and v to be resolved in u 's favor.

5.2 Algorithm for the Drinkers Problem

Now, we state the algorithm formally. As before, we introduce a request token, req_b , for every bottle b . The following Boolean variables are used:

$bot_u(b)$: philosopher u holds bottle b
 $req_u(b)$: philosopher u holds request token for bottle b
 $need_u(b)$: philosopher u needs bottle b
 $tranquil_u/thirsty_u/drinking_u$: philosopher u is tranquil/thirsty/drinking

As before, we drop the subscript when the context is understood. From the problem statement we have,

$$tranquil \Rightarrow \forall b \{ \sim need(b) \}$$

State transitions for dining philosopher determined by drinking states are

- (D1) $thinking, thirsty \rightarrow hungry := true$
 (D2) $eating, \sim thirsty \rightarrow thinking := true$

Other actions of the dining philosopher remain unchanged.

Rules for bottle and request transmissions {Let f be the fork corresponding to bottle b , i.e., fork f and bottle b are shared by the same two processes}:

- (R1) Request a Bottle:
 $thirsty, need(b), req_b(b), \sim bot(b) \rightarrow$
 send request token for bottle b ;
 $req_b(b) := false$
- (R2) Send a Bottle:
 $req_b(b), bot(b), \sim [need(b) \text{ and } (drinking \text{ or } fork(f))] \rightarrow$
 send bottle b ;
 $bot(b) := false$
- (R3) Receive Request for a Bottle:
 upon receiving request for bottle $b \rightarrow$
 $req_b(b) := true$
- (R4) Receive a Bottle:
 upon receiving bottle $b \rightarrow$
 $bot(b) := true$

Initial Conditions

For Dining Philosophers: As before.

For Drinking Philosophers: A bottle and the request token for it are held by different philosophers; that is, if u, v share bottle b , then u holds the bottle and v the token ($bot_u(b), req_b(b), \sim bot_v(b), \sim req_b_v(b)$), or v holds the bottle and u the token.

5.3 Proof of Correctness of the Solution to Drinkers Problem

We show that the solution has the desired properties of fairness, symmetry, economy, concurrency and boundedness.

Fairness

LEMMA 3. *Every eating period is finite.*

PROOF. If an eating philosopher is nonthirsty, he completes eating (D2). If philosopher u is eating, he is holding all forks. If he holds a bottle that he needs, he will not release it until he completes drinking, from the precondition of (R2). If he needs and does not hold a bottle that he shares with v , then he holds or will hold the request token for the bottle (same proof as in Case 1 of Theorem 2). He will request the bottle, from (R1), and v will have to send the bottle in finite time (R2) since v does not hold the fork and v can be in drinking state only for finite duration. Therefore u will hold all bottles he needs in finite time. Since u drinks for finite time, u will become tranquil in finite time and, from (D2), u will stop eating in finite time. \square

Since every eating period is finite, Theorem 2 applies and we have

COROLLARY 4. *Every hungry philosopher starts eating in finite time.*

THEOREM 5. *Every thirsty philosopher drinks in finite time.*

PROOF. When a philosopher becomes thirsty he is either thinking, hungry, or eating. A (thirsty, thinking) philosopher becomes hungry in finite time (from D1); a hungry philosopher starts eating in finite time (from Corollary 4). Therefore every philosopher who remains thirsty will eat in finite time. The theorem follows from Lemma 3 and the fact (D2) that eating can be terminated only by drinking. \square

Symmetry. Follows from the description of the algorithm.

Economy. We first show that a bottle b can travel at most twice between neighbors, u, v , before one of them drinks from b . A bottle is sent in response to a request from a thirsty philosopher. Let (u, v) be a directed edge in H ; the bottle will travel at most once from u to v and will then be held by v until v drinks. This is because (1) either v holds a clean fork, which will not be released until after eating (and hence drinking), and therefore the bottle b , which is needed by v will not be released, or (2) u holds a dirty fork, which must have been requested by v (when v became thirsty and hence hungry) and will be mailed, after being cleaned, along with the bottle to v , and then case (1) applies. Hence a bottle can travel at most twice between neighbors before one of them drinks.

LEMMA 6. *There are at most $4qd$ message transmissions for q drinking sessions among all philosophers, where d is the maximum degree (i.e., the maximum number of neighbors) of any philosopher.*

PROOF. There is at most one request (for fork and/or bottle), one transmission of a fork, and two transmissions of a bottle between neighbors before one of them

drinks. Therefore, when a philosopher drinks, there must have been no more than 4 messages per each of its neighbors and hence the result. □

Concurrency. The argument for concurrency is similar to that for the diners problem. We note that no feasible state of the drinkers problem is being eliminated in our solution.

Boundedness. There are at most three messages (request for a bottle and/or fork, a bottle, or a fork) in transit from one philosopher to another at any point.

6. SUMMARY

We have described a distributed implementation of a precedence graph. The changes to the graph are such that the graph is always acyclic. The depth of a process in the graph is the process's distinguishing characteristic. The graph is implemented by the "forks" of the diners problem. Two processes share a fork if they may conflict with one another. The conflict-resolution rule is: A process u yields in a conflict with a process v if and only if u does not hold the fork shared with v . The algorithm ensures that if processes u and v are in conflict, and u has precedence over v in the precedence graph, then the conflict resolution rule will, eventually, cause conflicts to be resolved in u 's favor.

Many types of conflict can be resolved by using the conflict-resolution rule coupled with our distributed implementation of the precedence graph. For instance, consider the multiple concurrent mutual exclusion problem described next. A critical section in a process has an arbitrary number of colors associated with it (where a color is some attribute of the critical section). The problem is to devise a scheme by which, for each color c , there is at most one process executing a critical section with associated color c . For example, a color may correspond to the privilege of exclusive access to a specific file, and associated with each critical section is the set of files accessed within that section. If all critical sections have the same set of colors, the problem reduces to the classical mutual exclusion problem.

We use our solution to the drinkers problem to solve the concurrent mutual exclusion problem. We use a variant of the drinkers problem in which a pair of philosophers may share an arbitrary number of bottles. The bottles are colored, each bottle having precisely one color. A pair of philosophers share at most one bottle of a given color. A bottle is specified by the edge it is on (i.e., by the pair of philosophers who share it) and by its color. The set of bottles a thirsty philosopher needs to drink is arbitrary — it may include any bottle he shares. For instance, when philosopher i becomes thirsty, he may need to hold the red bottle shared with j and the red bottle shared with k and the blue bottle shared with k . If there is precisely one bottle on each edge the problem reduces to the one discussed earlier. We leave it to the reader to show that the algorithm given earlier also applies to the extension to colored bottles.

Given a concurrent mutual exclusion problem, we construct a drinkers problem as follows. Philosophers (processes) i and j share a bottle with color c if and only if both philosophers have critical sections with color c . A process i may enter a critical section with a set of colors c if and only if, for all colors c in c , and for all edges e incident on process i , the bottle of color c on edge e is held by philosopher

t. In this case it is obvious that no neighboring philosopher can enter a critical section with a color *c* in ζ .

7. PREVIOUS WORK

The distributed dining philosophers problem (philosophers at the vertices of a graph) and the dining philosophers problem (five philosophers arranged in a ring) appear in [2, 3]. Dijkstra's solutions to the former problem are based on instantaneous atomic transmissions of messages to all neighbors or static fork orderings. Lynch [9] has carried out an extensive analysis of static resource ordering algorithms.

The problem of mutual exclusion among a group of processes in executing their critical sections is a special case of the diners problem: Every process is a neighbor of every other process and execution of a critical section corresponds to eating. Distributed solutions to mutual exclusion using timestamps and process IDs to break ties, appear in Lamport [7] and in Ricart and Agrawala [10]. Shared counter variables have been used in [1], for solving the dining philosophers problem.

A symmetric distributed solution to the diners problem appears in Francez and Rodeh [5]. They use an extended form of CSP [6], in which both input and output commands are used in guards.

Lehmann and Rabin [8] give a perfectly symmetric probabilistic algorithm and show that there is no perfectly symmetric nonprobabilistic solution to the diners problem.

ACKNOWLEDGMENT

We thank W.H.J. Feijen and A.J.M. Van Gasteren of Eindhoven University of Technology and Greg Andrews of the University of Arizona for their detailed comments. We are grateful to three unknown referees and to Susan Graham for detailed comments. Conversations with E.W. Dijkstra on this problem were most helpful.

REFERENCES

1. DEVILLERS, R.E., AND LAUER, P.E. A general mechanism for avoiding starvation with distributed control. *Inf. Process. Lett.* 7, 3 (Apr. 1978), 156-158.
2. DIJKSTRA, E.W. Two starvation free solutions to a general exclusion problem. EWD 625, Plataanstraat 5, 5671 AI Nuenen, The Netherlands.
3. DIJKSTRA, E.W. Hierarchical Ordering of Sequential Processes. In *Operating Systems Techniques*, C.A.R. Hoare and R.H. Perrott, Eds., Academic Press, New York, 1972.
4. DIJKSTRA, E.W. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (Aug. 1975), 453-457.
5. FRANCEZ, N., AND RODEH, M. A distributed abstract data type implemented by a probabilistic communication scheme. Tech. Rep. TR 080, IBM Israel Scientific Center, Haifa, Israel, Apr. 1980.
6. HOARE, C.A.R. Communicating sequential processes. *Commun. ACM* 21, 8 (Aug. 1978), 666-677.
7. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558-565.

8. LEHMAN, D., AND RABIN, M. On the advantages of free choice. A symmetric and fully distributed solution of the dining philosophers problem. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages* (Williamsburg, Va., Jan. 26-28), ACM, New York, 1981, pp. 133-138.
9. LYNCH, N.A. Fast allocation of nearby resources in a distributed system. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing* (Los Angeles, Calif., Apr. 28-30), ACM, New York, 1980, pp. 70-81.
10. RICART, G., AND AGRAWALA, A. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM* 24, 1 (Jan. 1981), 9-17.

Received May 1983; revised February 1984; accepted May 1984

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Distributed Snapshots: Determining Global States of Distributed Systems		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Dr. K. Mani Chandy Dr. Leslie Lamport		8. CONTRACT OR GRANT NUMBER(s) AFOSR 81-0205B
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Sciences Department University of Texas Austin, Texas 78712		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Dr. Robert N. Buchal AFOSR/NM Bolling AFB, DC 20332		12. REPORT DATE
		13. NUMBER OF PAGES 19 pages
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES this paper is to appear in ACM Transactions on Computing Systems		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This paper presents algorithms by which a process in a distributed system can determine a global state of the system during a computation of the system. Processes in a distributed system communicate by sending and receiving messages. A process can record its own state and the messages it sends and receives; there is nothing else that it can record. To determine a global system state, a process p must enlist the cooperation of other processes who must record their own local states and send the recorded local states to p.		

All processes cannot record their local states at precisely the same instant unless they have access to a common clock. We assume that processes do not share clocks or memory. The problem is to devise algorithms by which processes record their own states and the states of communication channels so that the set of process and channel states recorded form a global system state. The global state detection algorithm is to be superimposed on the underlying computation: it must run concurrently with, but not alter, the underlying computation.

Distributed Snapshots: Determining Global States of Distributed Systems

K. Mani Chandy*

Leslie Lamport**

*Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

**Stanford Research Institute
Menlo Park, California 94025

August 1983

This work was supported in part by a grant from the Air Force Office of Scientific Research under grant AFOSR 81-0205 and by a grant from the National Science Foundation under grant MCS81-04459.

Table of Contents

1. Introduction	2
2. Model Of A Distributed System	4
3. The Algorithm	8
3.1. Motivation for the Steps of the Algorithm	8
3.2. Global State Detection Algorithm Outline	11
3.3. Termination of the Algorithm	11
4. Properties of the Recorded Global State	12
5. Stability Detection	16

List of Figures

Figure 2-1:	A Distributed System With Processes p, q, r and Channels c_1, c_2, c_3 and c_4	4
Figure 2-2:	The Simple Distributed System of Example 2.1 and 2.2	6
Figure 2-3:	State-Transition Diagram of a Process in Example 2.1	6
Figure 2-4:	Global States and Transitions of the Single-Token Conservation System	7
Figure 2-5:	State-Transition Diagram for Process p in Example 2.2	7
Figure 2-6:	State Transition Diagram for Process q in Example 2.2	8
Figure 2-7:	A Computation for Example 2.2	9
Figure 4-1:	A Recorded Global State for Example 2.2	13

1. Introduction

This paper presents algorithms by which a process in a distributed system can determine a global state of the system during a computation. Processes in a distributed system communicate by sending and receiving messages. A process can record its own state and the messages it sends and receives; *it can record nothing else*. To determine a global system state, a process p must enlist the cooperation of other processes who must record their own local states and send the recorded local states to p . All processes cannot record their local states at precisely the same instant unless they have access to a common clock. We assume that processes do not share clocks or memory. The problem is to devise algorithms by which processes record their own states and the states of communication channels so that the set of process and channel states recorded form a global system state. The global state detection algorithm is to be superimposed on the underlying computation: it must run concurrently with, but not alter, this underlying computation.

The state-detection algorithm plays the role of a group of photographers observing a panoramic, dynamic scene, such as a sky filled with migrating birds - a scene so vast that it cannot be captured by a single photograph. The photographers must take several snapshots and piece the snapshots together to form a picture of the overall scene. The snapshots cannot all be taken at precisely the same instant because of synchronization problems. Furthermore, the photographers should not disturb the process that is being photographed; for instance they cannot get all the birds in the heavens to remain motionless while the photographs are taken. Yet, the composite picture should be meaningful. The problem before us is to define "meaningful" and then to determine how the photographs should be taken.

We now describe an important class of problems that can be solved with the global state detection algorithm. Let y be a predicate function defined on the global states of a distributed system D : i.e. $y(S)$ is true or false for a global state S of D . The predicate y is said to be a *stable property* of D if $y(S)$ implies $y(S')$ for all global states S' of D reachable from global state S of D . In other words, if y is a stable property and y is true at a point in a computation of D , then y is true at all later points in that computation. Examples of stable properties are "computation has terminated," "the system is deadlocked" and "all tokens in a token ring have disappeared."

Several distributed system problems can be formulated as the general problem of devising an algorithm by which a process in a distributed system can determine whether a stable property y of the system holds. Deadlock detection [1-5] and termination detection [6-8] are special cases of the stable property detection problem. Details of the algorithm are presented later. The basic idea of the algorithm is that a global state S of the system is determined and $y(S)$ is computed to

see if the stable property y holds.

Several algorithms for solving deadlock and termination problems by determining the global states of distributed systems have been published. Gligor and Shattuck [1] state that many of the published algorithms are incorrect and impractical. A reason for the incorrect or impractical algorithms may be that the relationships between local process states, global system states and points in a distributed computation are not well understood. One of the contributions of this paper is to define these relationships.

Many distributed algorithms are structured as a sequence of phases, where each phase consists of a transient part in which useful work is done, followed by a stable part in which the system cycles endlessly and uselessly. The presence of stable behavior indicates the end of a phase. A phase is similar to a series of iterations in a sequential program, which are repeated until successive iterations produce no change, i.e. stability is attained. Stability must be detected so that one phase can be terminated and the next phase initiated [7]. The termination of a computational phase is not identical to the termination of a computation. When a computation terminates, all activities cease - messages are not sent and process states do not change. There may be activity during the stable behavior which indicates the end of a computational phase - messages may be sent and received, and processes may change state, but this activity serves no purpose other than to signal the end of a phase. In this paper, we are concerned with the detection of stable system properties; the cessation of activity is only one example of a stable property.

Strictly speaking properties such as "the system is deadlocked" are not stable if the deadlock is "broken" and computation is reinitiated. However, to keep exposition simple, we shall partition the overall problem into the problems of (1) detection of the termination of one phase (and informing all processes that a phase has ended) and (2) initiating a new phase. The following is a stable property: "the k -th computational phase has terminated", $k = 1, 2, \dots$. Hence, the methods presented in this paper are applicable to detecting the termination of the k -th phase for a given k .

In this paper we restrict attention to the problem of detecting stable properties. The problem of initiating the next phase of computation is not considered here because the solution to the problem varies significantly depending on the application, being different for database deadlock detection than for detecting the termination of a diffusing computation.

We have to present our algorithms in terms of a model of a system. The model chosen is not important in itself, we could have couched our discussion in terms of other models. We shall

describe our model informally and only to the level of detail necessary to make the algorithms clear.

2. Model Of A Distributed System

A distributed system consists of a finite set of processes and a finite set of channels. It is described by a labeled, directed graph in which the vertices represent processes and the edges represent channels. Figure 2.1 is an example.

Figure 2-1: A Distributed System With Processes p,q,r and Channels c1,c2,c3 and c4

Channels are assumed to have infinite buffers, to be error-free and to deliver messages in the order sent. (The infinite buffer assumption is made for ease of exposition: bounded buffers may be assumed provided there exists a proof that no process attempts to add a message to a full buffer.) The delay experienced by a message in a channel is arbitrary, but finite. The sequence of messages received along a channel is an initial subsequence of the sequence of messages sent along the channel. The *state* of a channel is the sequence of messages sent along the channel excluding the messages received along the channel.

A process is defined by a set of states, an initial state (from this set), and a set of events. An event e in a process p is an atomic action which may change the state of p itself and the state of *at most one* channel c incident on p : the state of c may be changed by the sending of a message along c (if c is directed away from p) or the receipt of a message along c (if c is directed towards p). An event e is defined by (1) the process p in which the event occurs, (2) the state s of p immediately before the event, (3) the state s' of p immediately after the event, (4) the channel c (if any) whose state is altered by the event, and (5) the message M , if any, sent along c (if c is a

channel directed away from p), or received along c (if c is directed towards p). We define e by the 5-tuple $\langle p, s, s', M, c \rangle$ where M and c are a special symbol, *null*, if the occurrence of e does not change the state of any channel.

A global state of a distributed system is a set of component process and channel states: the *initial* global state is one in which the state of each process is its initial state and the state of each channel is the empty sequence. The occurrence of an event may change the global state. Let $e = \langle p, s, s', M, c \rangle$; we say e can occur in global state S if and only if (1) the state of process p in global state S is s and (2) if c is a channel directed towards p , then the state of c in global state S is a sequence of messages with M at its head. We define a function *next*, where $next(S, e)$ is the global state immediately after the occurrence of event e in global state S . The value of $next(S, e)$ is defined only if event e can occur in global state S , in which case $next(S, e)$ is the global state identical to S except that: (1) the state of p in $next(S, e)$ is s' , (2) if e is a channel directed towards p then the state of c in $next(S, e)$ is c 's state in S with message M deleted from its head, and (3) if c is a channel directed away from p then the state of c in $next(S, e)$ is the same as c 's state in S with message M added to the tail.

Let $seq = (e_i; 0 \leq i \leq n)$ be a sequence of events in component processes of a distributed system. We say that seq is a *computation of the system* if and only if event e_i can occur in global state S_i , $0 \leq i \leq n$, where S_0 is the initial global state and

$$S_{i+1} = next(S_i, e_i), \text{ for } 0 \leq i \leq n$$

An alternate model, based on Lamport [9], which views computations as partially ordered sets of events is given in [10].

Example 2.1

To illustrate the definition of a distributed system consider a simple system consisting of 2 processes p and q , and 2 channels c and c' as shown in figure 2.2.

The system contains one *token* which is passed from one process to another, and hence we call this system the "single-token conservation" system. Each process has 2 states: s_0 and s_1 , where s_0 is the state in which the process does not possess the token and s_1 is the state in which it does. The initial state of p is s_1 and of q is s_0 . Each process has 2 events: (1) a transition from s_1 to s_0 with the sending of the token and (2) a transition from s_0 to s_1 with the receipt of the token. The state transition diagram for a process is shown in figure 2.3.

The global states and transitions are shown in figure 2.4.

Figure 2-2: The Simple Distributed System of Example 2.1 and 2.2

Figure 2-3: State-Transition Diagram of a Process in Example 2.1

A system computation corresponds to a path in the global state transition diagram (figure 2.4) starting at the initial global state. Examples of system computations are: (1) the empty sequence and (2) $\langle p \text{ sends token, } q \text{ receives token, } q \text{ sends token} \rangle$. The following sequence is not a computation of the system: $\langle p \text{ sends token, } q \text{ sends token} \rangle$, because the event "q sends token" cannot occur while q is in the state s_0 .

For brevity, the four global states, in order of transition (see figure 2.4), will be called: (1) in-p, (2) in-c, (3) in-q and (4) in-c' to denote the location of the token. This example will be used later to motivate the algorithm.

Example 2.2

This example illustrates non-deterministic computations. Non-determinism plays an interesting role in the snapshot algorithm.

In example 2.1 there is exactly one event possible in each global state. Consider a system with the same topology as example 2.1 (see figure 2.2) but where the processes p and q are

Figure 2-4: Global States and Transitions of the Single-Token Conservation System

Figure 2-5: State-Transition Diagram for Process p in Example 2.2 defined by the state transition diagrams of figures 2.5 and 2.6.

Figure 2-6: State Transition Diagram for Process q in Example 2.2

An example of a computation is shown in figure 2.7. The reader should observe that there may be more than one transition allowable from a global state: for instance events " p sends M " and " q sends M' " may occur in the initial global state, and the next states after these events are different.

3. The Algorithm

3.1. Motivation for the Steps of the Algorithm

The global-state recording algorithm works as follows: each process records its own state, and the 2 processes that a channel is incident on cooperate in recording the channel state. We cannot ensure that the states of all processes and channels will be recorded at the same instant because there is no global clock; however, we require that the recorded process and channel states form a "meaningful" global system state.

The global-state recording algorithm is to be superimposed on the underlying computation, i.e. it must run concurrently with, but not alter, the underlying computation. The algorithm may send messages and require processes to carry out computations; however, the messages and computation required to record the global state must not interfere with the underlying computation.

We now consider an example to motivate the steps of the algorithm. In the example we shall assume that we can record the state of a channel instantaneously; we postpone discussion of how the channel state is recorded. Let c be a channel from p to q . The purpose of the example is to gain an intuitive understanding of the relationship between the instant at which the state of

Figure 2-7: A Computation for Example 2.2

channel c is to be recorded and the instants at which the states of processes p and q are to be recorded.

Example 3.1

Consider the single-token conservation system. Assume that the state of process p is recorded in global state $in-p$. Then the state recorded for p shows the token in p . Now assume that the global state transits to $in-c$ (because p sends the token). Suppose the states of channels c and c' , and process q were recorded in global state $in-c$, so the state recorded for channel c shows it with the token and the states recorded for channel c' and process q show them not in possession of the token. The composite global state recorded in this fashion would show 2 tokens in the system, one in p and the other in c . But a global state with 2 tokens is unreachable from the initial global state in a *single-token* conservation system! The inconsistency arises because the state of p is recorded *before* p sent a message along c and the state of c is recorded *after* p sent the message. Let n be the number of messages sent along c before p 's state is recorded, and let n' be the number of messages sent along c before c 's state is recorded. Our example suggests that the recorded global state may be inconsistent if $n < n'$.

Now consider an alternate scenario. Suppose the state of c is recorded in global state $in-p$, the system then transits to global state $in-c$, and the states of c' , p and q are recorded in global state $in-c$. The recorded global state shows *no* tokens in the system. This example suggests that the recorded global state may be inconsistent if the state of c is recorded *before* p sends a message along c and the state of p is recorded *after* p sends a message along c , i.e. if $n > n'$.

We learn from these examples that (in general) a consistent global state requires

$$n = n' \tag{1}$$

Let m be the number of messages received along c before q 's state is recorded. Let m' be the number of messages received along c before c 's state is recorded. We leave it up to the reader to extend the example to show that consistency requires

$$m = m' \tag{2}$$

In every state, the number of messages received along a channel cannot exceed the number of messages sent along that channel, i.e.

$$n' \geq m' \tag{3}$$

From the above equations:

$$n \geq m \tag{4}$$

The state of channel c that is recorded must be the sequence of messages sent along the

channel before the sender's state is recorded excluding the sequence of messages received along the channel before the receiver's state is recorded, i.e. if $n' = m'$, the recorded state of c must be the empty sequence and if $n' > m'$, the recorded state of c must be the $(m'+1)$ st, ..., n' -th messages sent by p along c . This fact and eqns 1 - 4 suggest a simple algorithm by which q can record the state of channel c . Process p sends a special message, called a *marker*, after the n -th message it sends along c (and before sending further messages along c). The marker has no effect on the underlying computation. The state of c is the sequence of messages received by q after q records q 's state and before q receives the marker along c . To ensure eqn(4), q must record its state, if it hasn't done so already, after receiving a marker along c and before q receives further messages along c .

Our example suggests the following outline for a global state detection algorithm.

3.2. Global State Detection Algorithm Outline

Marker Sending Rule for a Process p

For each channel c , incident on, and directed away from p :

p sends one marker along c after p records its state
and before p sends further messages along c .

Marker Receiving Rule For a Process q

On receiving a marker along a channel c :

if q has not recorded its state then
begin q records its state;
 q records the state c as the empty sequence
end

else q records the state of c as the sequence of messages
 received along c after q 's state was recorded and
 before q received the marker along c .

3.3. Termination of the Algorithm

The marker receiving and sending rules guarantee that if a marker is received along every channel then each process will record its state and the states of all incoming channels. To ensure that the global-state recording algorithm terminates in finite time, each process must ensure that (L1) no marker remains forever in an incident input channel and (L2) it records its state within

finite time of initiation of the algorithm.

The algorithm can be initiated by one or more processes, each of which records its state spontaneously, without receiving markers from other processes; we postpone discussion of what may cause a process to record its state spontaneously. If process p records its state and there is a channel from p to a process q , then q will record its state in finite time because p will send a marker along the channel and q will receive the marker in finite time (L1). Hence if p records its state and there is a path (in the graph representing the system) from p to a process q , then q will record its state in finite time because, by induction, every process along the path will record its state in finite time. Termination in finite time is ensured if for every process q : q spontaneously records its state or there is a path from a process p , which spontaneously records its state, to q .

In particular, if the graph is strongly connected and at least one process spontaneously records its state, then all processes will record their states in finite time (provided L1 is ensured).

The algorithm described so far allows each process to record its state and the states of incoming channels. The recorded process and channel states must be collected and assembled to form the recorded global state. We shall not describe algorithms for collecting the recorded information because such algorithms have been described elsewhere [6,7]. A simple algorithm for collecting information in a system whose topology is strongly-connected is for each process to send the information it records along all outgoing channels, and for each process receiving information for the first time to copy it and propagate it along all of its outgoing channels. All the recorded information will then get to all the processes in finite time, allowing all processes to determine the recorded global state.

4. Properties of the Recorded Global State

To gain an intuitive understanding of the properties of the global state recorded by the algorithm, we shall study example 2.2. Assume that the state of p is recorded in global state S_0 (Figure 2.7) so the state recorded for p is A . After recording its state, p sends a marker along channel c . Now assume that the system goes to global state S_1 , then S_2 and then S_3 while the marker is still in transit, and the marker is received by q when the system is in global state S_3 . On receiving the marker, q records its state, which is D , and records the state of c to be the empty sequence. After recording its state, q sends a marker along channel c' . On receiving the marker, p records the state of c' as the sequence consisting of the single message M' . The recorded global state S^* is shown in figure 4.1. The recording algorithm was initiated in global state S_0 and terminated in global state S_3 .

Figure 4-1: A Recorded Global State for Example 2.2

Observe that the global state S^* recorded by the algorithm is not identical to any of the global states S_0, S_1, S_2, S_3 that occurred in the computation. Of what use is the algorithm if the recorded global state never occurred? We shall now answer this question.

Let $seq = (e_i, 0 \leq i)$ be a distributed computation and let S_i be the global state of the system immediately before event $e_i, 0 \leq i$, in seq . Let the algorithm be initiated in global state S_i and let it terminate in global state $S_\phi, 0 \leq i \leq \phi$; in other words, the algorithm is initiated after e_{i-1} if $i > 0$, and before e_i , and it terminates after $e_{\phi-1}$ if $\phi > 0$, and before e_ϕ . We observed in example 2.2 that the recorded global state S^* may be different from all global states $S_k, i \leq k \leq \phi$.

We shall show that:

1. S^* is reachable from S_i and
2. S_ϕ is reachable from S^* .

Specifically, we shall show that there exists a computation seq' where

1. seq' is a permutation of seq , such that S_i, S^* and S_ϕ occur as global states in seq' , and
2. $S_i = S^*$ or S_i occurs earlier than S^* , and
3. $S_\phi = S^*$ or S^* occurs earlier than S_ϕ in seq' .

Theorem 1: There exists a computation $seq' = (e'_i, 0 \leq i)$ where

1. For all i , where $i < i$ or $i \geq \phi$: $e'_i = e_i$, and
2. The subsequence $(e'_i, i \leq i < \phi)$ is a permutation of the subsequence $(e_i, i \leq i < \phi)$, and
3. For all i where $i \leq i$ or $i \geq \phi$: $S'_i = S_i$, and
4. There exists some $k, i \leq k \leq \phi$, such that $S^* = S'_k$.

Proof: Event e_i in seq is called a *pre-recording* event if and only if e_i is on a process p and p records its state *after* e_i in seq . Event e_i in seq is called a *post-recording* event if and only if it is not a pre-recording event -- i.e. if e_i is on a process p and p records its state *before* e_i in seq . All events e_i , $i < \iota$, are pre-recording events and all events e_i , $i \geq \phi$, are post-recording events in seq . There *may be* a post-recording event e_{j-1} before a pre-recording event e_j for some j , $\iota < j < \phi$; this can occur only if e_{j-1} and e_j are in different processes (because if e_{j-1} and e_j are on the same process and e_{j-1} is a post-recording event then so is e_j).

We shall derive a computation seq' by permuting seq , where all pre-recording events occur before all post-recording events in seq' . We shall show that S^* is the global state in seq' after all pre-recording events and before all post-recording events.

Assume that there is a post-recording event e_{j-1} before a pre-recording event e_j in seq . We shall show that the sequence obtained by interchanging e_{j-1} and e_j must also be a computation. Events e_{j-1} and e_j must be on different processes. Let p be the process in which e_{j-1} occurs and let q be the process in which e_j occurs. There cannot be a message sent at e_{j-1} which is received at e_j because (1) if a message is sent along a channel c when event e_{j-1} occurs, then a marker must have been sent along c *before* e_{j-1} , since e_{j-1} is a post-recording event and (2) if the message is received along channel c when e_j occurs, then the marker must have been received along c *before* e_j occurs (since channels are first-in-first-out), in which case (by the marker-receiving rule) e_j would be a post-recording event too.

The state of process q is not altered by the occurrence of event e_{j-1} because e_{j-1} is on a different process p . If e_j is an event in which q receives a message M along a channel c then M must have been the message at the head of c before event e_{j-1} , since a message sent at e_{j-1} cannot be received at e_j . Hence event e_j can occur in global state S_{j-1} .

The state of process p is not altered by the occurrence of e_j . Hence e_{j-1} can occur after e_j . Hence the sequence of events $e_1, \dots, e_{j-2}, e_j, e_{j-1}$ is a computation. From the arguments in the last paragraph it follows that the global state after computation e_1, \dots, e_j is the same as the global state after computation $e_1, \dots, e_{j-2}, e_j, e_{j-1}$.

Let seq^* be a permutation of seq which is identical to seq except that e_j and e_{j-1} are interchanged. Then seq^* must also be a computation. Let S_i be the global state immediately before the i -th event in seq^* . From the arguments of the previous paragraph

$$S_i = S_i \text{ for all } i \text{ where } i \neq j$$

By repeatedly swapping post-recording events which immediately following pre-recording events, we see that there exists a permutation seq' of seq in which

1. all pre-recording events precede all post-recording events, and

2. seq' is a computation, and
3. for all i where $i < \phi$ or $i \geq \phi$: $e'_i = e_i$, and
4. for all i where $i \leq \phi$ or $i \geq \phi$: $S'_i = S_i$.

Now we shall show that the global state after all pre-recording events and before all post-recording events in seq' is S^* . To do this we need to show that:

1. the state of each process p in S^* is the same as its state after the process computation consisting of the sequence of pre-recorded events on p and
2. the state of each channel c in S^* is: (sequence of messages corresponding to pre-recorded sends on c) - (sequence of messages corresponding to pre-recorded receives on c).

The proof of the first part is trivial. Now we prove part (2). Let c be a channel from process p to process q . The state of channel c recorded in S^* is the sequence of messages received on c by q *after* q records its state *and before* q receives a marker on c . The sequence of messages sent by p along c before p sends a marker along c is the sequence corresponding to pre-recorded sends on c . Part (2) now follows.

Example 4.1: The purpose of this example is to show how the computation seq' is derived from the computation seq . Consider example 2.2. The sequence of events shown in the computation of Figure 2.7 is:

- e_0 : p sends M and changes state to B
 (a post-recording event)
- e_1 : q sends M' and changes state to D
 (a pre-recording event)
- e_2 : p receives M' and changes state to A
 (a post-recording event)

Since e_0 , a post-recording event immediately precedes e_1 , a pre-recording event, we interchange them, to get the permuted sequence seq' :

- e'_0 : q sends M' and changes state to D
 (a pre-recording event)
- e'_1 : p sends M and changes state to B
 (a post-recording event)
- e'_2 : p receives M' and changes state to A
 (a post-recording event)

In *seq'*, all pre-recording events precede all post-recording events. We leave it up to the reader to show that the global state after e'_0 is the recorded global state.

5. Stability Detection

We now solve the stability detection problem described in section 1. We study the stability detection problem because it is a paradigm for many practical problems such as distributed deadlock detection.

A stability-detection algorithm is defined as follows:

Input: A stable property y

Output: A Boolean value *definite* with the property:
 $(y(S_i) \rightarrow \textit{definite})$ and
 $(\textit{definite} \rightarrow y(S_\phi))$
 where S_i and S_ϕ are the global states of the system when the algorithm is initiated and when it terminates, respectively. (The symbol \rightarrow denotes logical implication.)

The input to the algorithm is (the definition of) function y . During the execution of the algorithm the value $y(S)$ for some global state S may be determined by a process in the system by applying the *externally defined* function y to global state S . By the output of the algorithm being a Boolean value *definite* we mean that (1) some specially designated process (say p) enters and thereafter remains in some special state to symbolize an output of *definite = true*, and (2) p enters and remains in some other special state to symbolize an output of *definite = false*.

Definite = true implies that the stable property holds when the algorithm *terminates*. However, *definite = false* implies that the stable property does not hold when the algorithm is *initiated*. We emphasize that *definite = true* gives us information about the state of the system at the *termination* of the algorithm whereas *definite = false* gives us information about the system state at the *initiation* of the algorithm. In particular, we cannot deduce from *definite = false* that the stable property does not hold at termination of the algorithm.

Solution to the stability detection problem:

```
begin
  record a global state S*;
  definite := y(S*);
end.
```

The correctness of the stability detection algorithm follows from the following facts:

S^* is reachable from S_i and

S_ϕ is reachable from S^* (Theorem 1) and

$y(S) \rightarrow y(S')$ for all S' reachable from S
(definition of a stable property)

Acknowledgements

J. Misra's contributions in defining the problem of global state detection are gratefully acknowledged. We are grateful to E. W. Dijkstra and C. S. Scholten for their comments particularly regarding the proof of theorem 1. The outline of the current version of the proof was suggested by them. Dijkstra's note [11] on the subject provides colourful insight into the problem of stability detection. Thanks are due to C. A. R. Hoare, F. Schneider and G. Andrews who helped us with detailed comments. We are grateful to Anita Jones and anonymous referees for suggestions.

References

1. Gligor, V. D. and Shattuck, S. H., "Deadlock Detection in Distributed Systems," *IEEE Transactions on Software Engineering*, SE-6, 5, (Sept. 1980), 435-440.
2. Menasce, D. and Muntz, R., "Locking and Deadlock Detection in Distributed Data Bases," *IEEE Transactions on Software Engineering*, SE-5, 3 (May 1979), 195-202.
3. Obermarck, R., "Distributed Deadlock Detection Algorithm," *ACM Transactions on Database Systems*, 7, 2 (June 1982), 187-208.
4. Mahoud, S. A. and Riordan, J. S., "Software Controlled Access to Distributed Databases," *INFOR*, 15, 1 (Feb. 1977), 22-36.
5. Chandy, K. M., Misra, J. and Haas, L., "Distributed Deadlock Detection," *ACM Transactions on Computer Systems*, Vol. 1, No. 2 (May 1983), 144-156.
6. Dijkstra, E. W. and Scholten, C. S., "Termination Detection for Diffusing Computations," *Information Processing Letters*, 11, 1 (Aug. 1980), 1-4.
7. Misra, J. and Chandy, K. M., "Termination Detection of Diffusing Computations in Communicating Sequential Processes," *ACM Transactions on Programming Languages and Systems*, 4, 1 (Jan. 1982), 37-43.
8. Chandy, K. M. and Misra, J., "Distributed Computation on Graphs: Shortest Path Algorithms," *Communications of the ACM*, Vol. 25, No. 11 (Nov. 1982), 833-837.
9. Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, 21, 7 (July 1978), 558-565.
10. Lamport, L. and Chandy, K. M., "On Partially-Ordered Event Models of Distributed Computations," to appear.
11. Dijkstra, E. W., "The Distributed Snapshot of K. M. Chandy and L. Lamport," EWD 864a, Plataanstraat 5, 5671 AL NUENEN, The Netherlands.

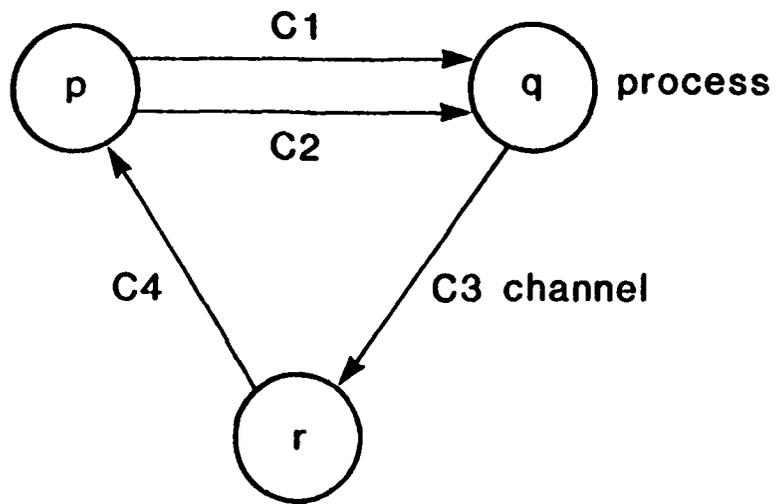


Figure 2-1

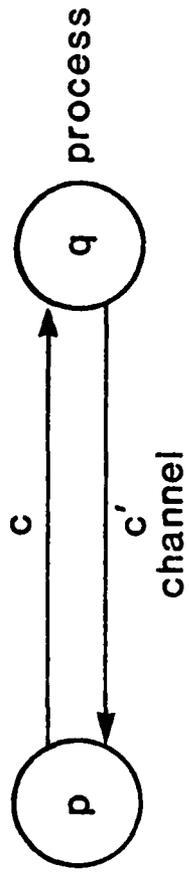


Figure 2-2

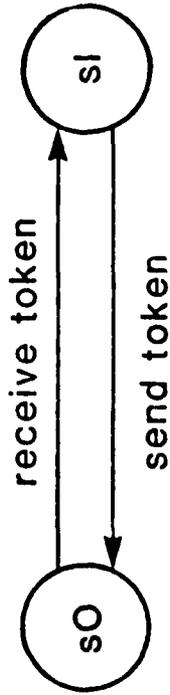


Figure 2-3

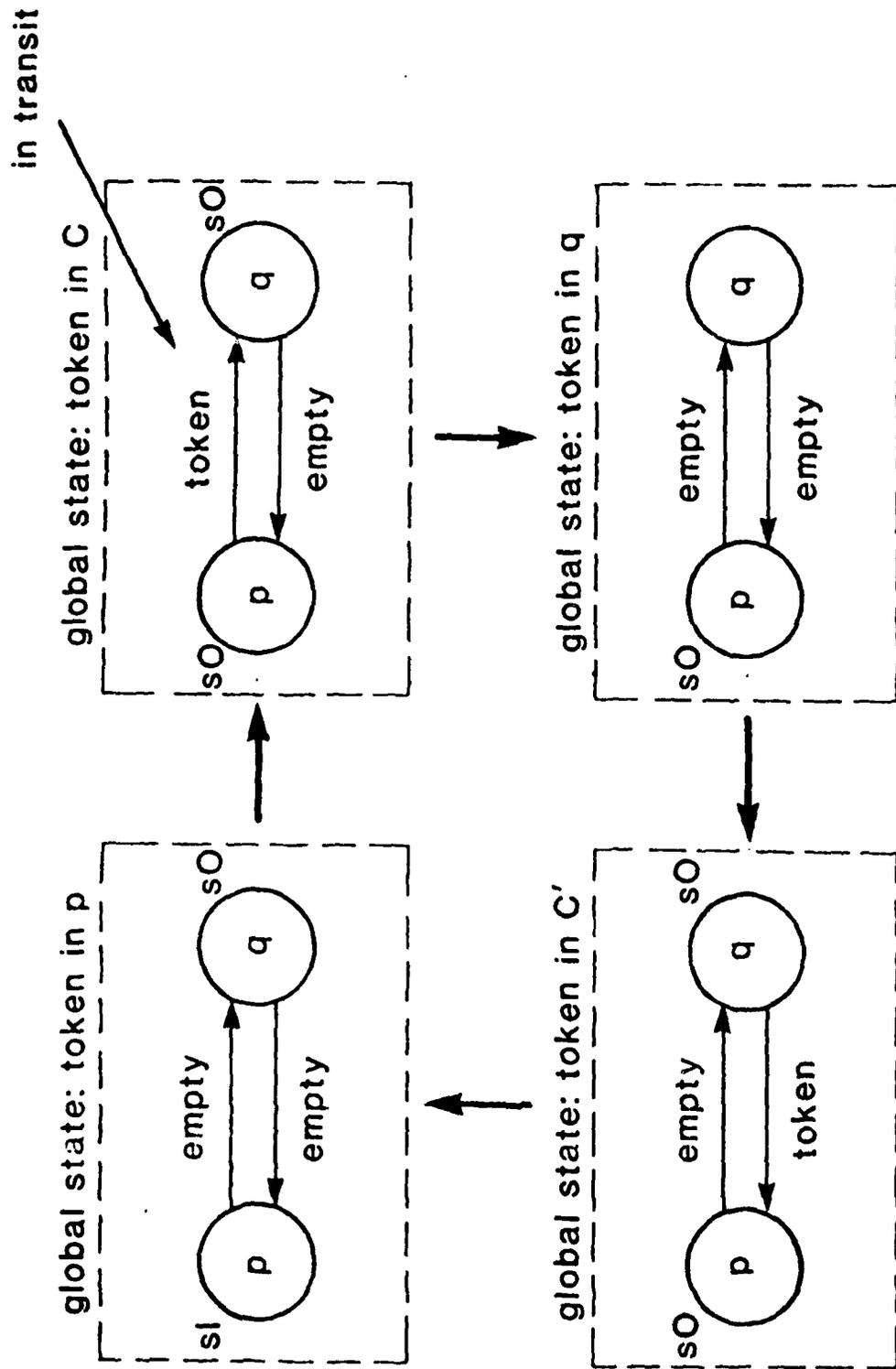


Figure 2-4

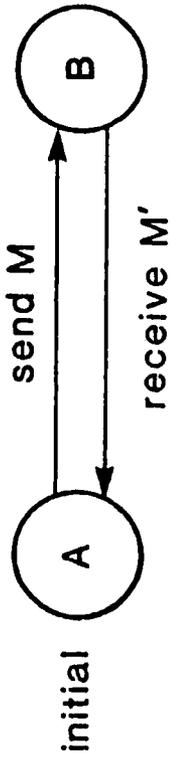


Figure 2-5

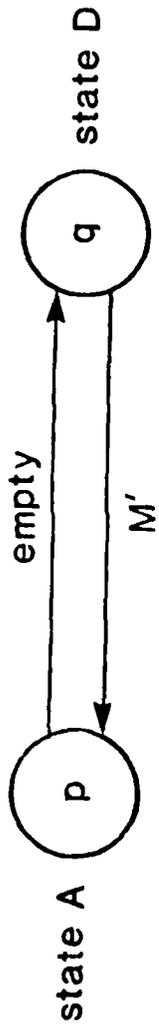


Figure 1-1

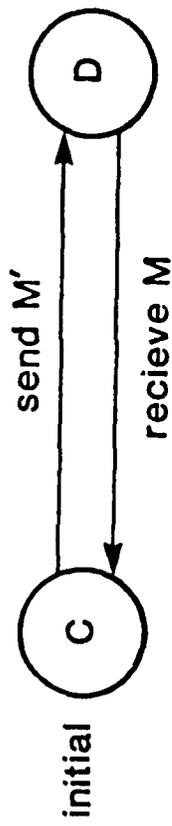


Figure 2-6

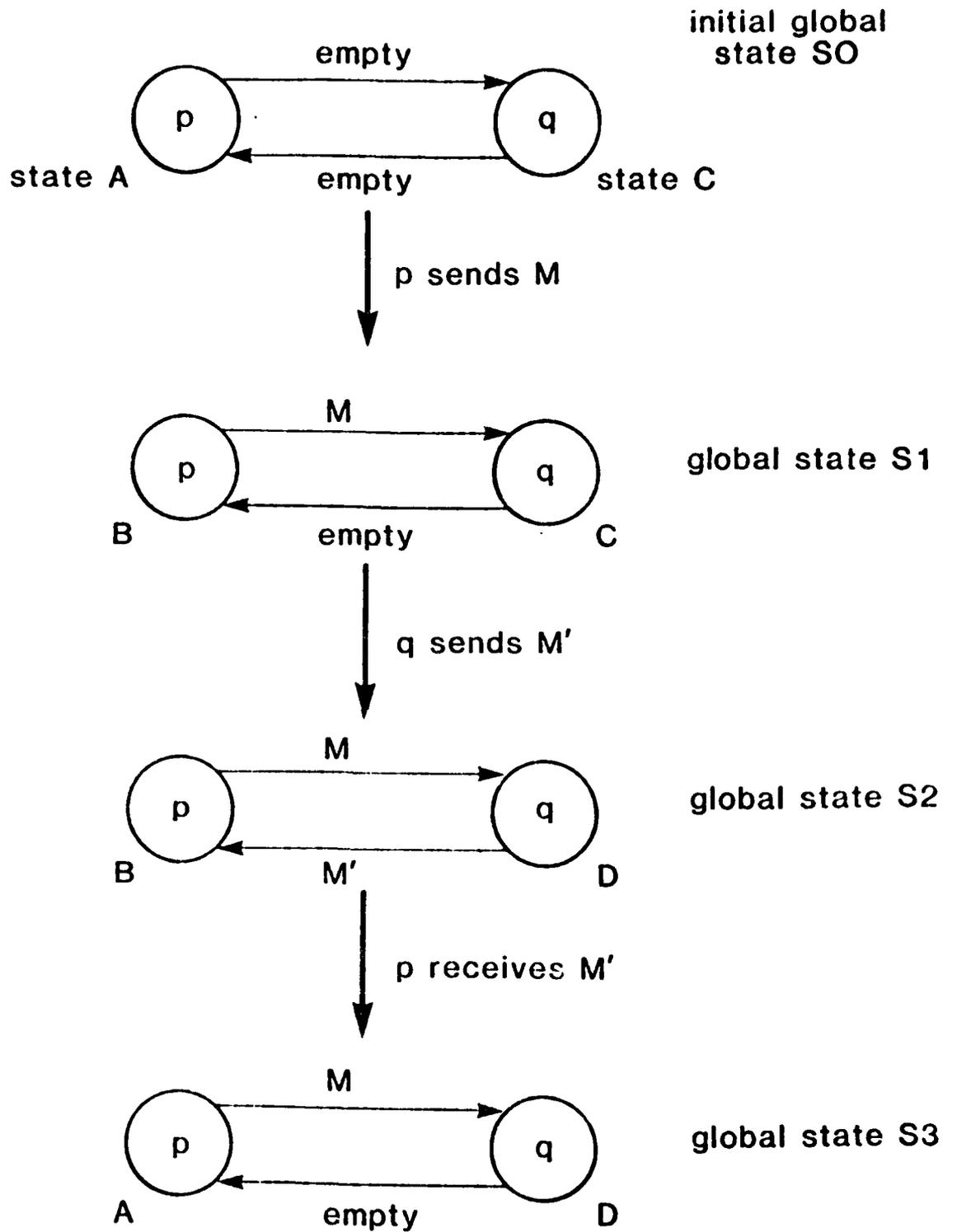


Figure 2-7

END

FILMED

4-85

DTIC