National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

THE UNIVERSITY OF ALBERTA

Distributed Simulation of Queueing Networks

by

David Alejandro Velasco-Gallegos

A thesis

submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree

of Master of Science

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

SPRING 1990

National Library of Canada

Bibliothèque nationale du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

ISBN 0-315-60183-3

NL-339 (r. 88/04) c

Canada

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: David Alejandro Velasco-Gallegos

TITLE OF THESIS: Distributed Simulation of Queueing Networks

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: Spring 1990

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only. The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

Signed: David A. Velasco.

Permanent Address:
Bachilleres # 111 Colonia Tecnológico
Monterrey, N.L. 64700
México

Date: December 4, 1989

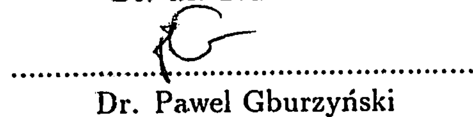
THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

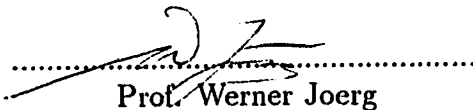
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Distributed Simulation of Queueing Networks** submitted by **David Alejandro Velasco-Gallegos** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dr. Piotr Rudnicki

Dr. M. Tamer Özsu



Dr. Pawel Gburzyński



Prof. Werner Joerg

Date: DECEMBER 4, 1989

Dedicated to my parents

Abstract

This thesis is concerned with the design and implementation of a distributed simulation testbed for queueing networks. A distributed simulator was implemented using the simulation package LANSF. Two implementations were made. The first implementation corresponds to the Chandy and Misra paradigm for distributed simulation. The second implementation presents a new distributed algorithm that introduces the novel concept of a two-colored NULL message.

In the experiments, 18 different network topologies were investigated and simulation experiments were carried out on a MIPS M/1000 machine running UMIPS-BSD. Different varying factors were taken into consideration for the experiments: inter-arrival time, processing delay, communication delay and number of messages transmitted.

The results of both implementations were compared with other studies using the following performance metrics: speedup, degree of parallelism achieved and null/real messages transmission ratio.

The simulation experiments demonstrated that the performance of the considered distributed simulation algorithm is sensitive to the structure of the topology being simulated and some factors characterizing the traffic intensity. A characterization of this sensivity is presented.

Acknowledgements

I would like to thank my supervisor Dr. Piotr Rudnicki for his help, patience, and valuable criticisms and comments made during the preparation of this research. Special thanks to the members of my examining committee Dr. M. Tamer Özsu, Dr. Pawel Gburzyński, and Prof. Werner Joerg for their thorough reviews of this work. In particular, the assistance provided by Dr. P. Gburzyński made possible the implementation part of this thesis.

This work would not been possible to accomplish without the financial support provided by the Government of Canada (World University Service of Canada), the National Council of Science and Technology of México (CONACYT), and the University of Alberta.

I am indebted to my friend Dave Straube for the proofreading and the valuable comments he made during the preparation of this work.

Finally, I want to thank all my friends at the Computing Science Department of the University of Alberta, who made my graduate studies and my stay in Canada an invaluable experience.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis Objectives and Outline 4

2 Distributed Simulation **6**

2.1 Introduction 6

2.2 What is Simulation ? 7

2.2.1 System Modeling and Analysis 7

2.2.2 Simulation Methodology 8

2.2.3 Classification of Simulations 9

2.3 Sequential Simulation 10

2.4 Distributed Simulation 11

2.4.1 The Causality Principle 11

2.4.2 A Basic Scheme for Distributed Simulation 12

2.4.3 Global Clocks vs Distributed Clocks 16

2.5 Overview of Distributed Simulation Strategies 21

3 Avoiding Deadlock in Distributed Simulation **25**

3.1 Introduction 25

3.2 Assumptions 26

3.3 Notation 29

3.4 Phases of the Algorithm 31

3.4.1 The Selection Phase 32

3.4.2 The Computation Phase 32

3.4.3 The I/O Phase 33

3.4.4 Initialization 33

3.4.5 Termination 33

3.5 Notations for Distributed Programs 34

3.5.1 SEND Command 34

3.5.2 RECEIVE Command 35

3.5.3 Parallel Command 35

3.6 Types of Processes 36

3.6.1 Delay Process 36

3.6.2 Branch Process 41

3.6.3 Merge Process 43

3.7 Interconnection of Logical Processes 45

4 A New Algorithm: Two-Colored NULL-Messages **47**

4.1 Introduction 47
4.2 Disadvantage of NULL Messages 48
4.3 Using Two-Colored NULL-Messages 49
4.4 Merge Process 53
4.5 Delay Process 57
4.6 Correctness of the Algorithm 58

5 Implementation **66**

5.1 Introduction 66
5.2 LANSF: A Simulation Package 67
5.3 Delay Process 69
5.4 Branch Process 71
5.5 Merge Process 73
5.6 Buffer Process 74

6 Experimental Results **76**

6.1 Introduction 76
6.2 The Experiments 77
6.2.1 Network Topologies 77
6.2.2 IART and d/bm Ratio 78
6.2.3 Performance Metrics 79
6.3 The Results 83
6.3.1 Tandem Networks 84

6.3.2 Feedforward Networks 86

6.3.3 General Feedback Networks 88

6.3.4 Cluster Network 90

6.3.5 Central Server Network 91

6.4 Remarks on the Operation of the 2cNULL Algorithm 93

7 Conclusions **95**

7.1 Summary of Results 95

7.2 Future Research 98

Bibliography **100**

Appendix A: Network Topologies **107**

Appendix B: Performance Charts **112**

List of Tables

4.1 Cases to consider for L_0 and L_1 60

6.1 Mix of IART and d/bm ratios 78

6.2 Ranges for the confidence intervals of feedforward networks in % of the mean value (Chandy and Misra algorithm) 81

6.3 Ranges for the confidence intervals of feedback networks in % of the mean value (Chandy and Misra algorithm) 82

6.4 Ranges for the confidence intervals of feedback networks in % of the mean value (2cNULL algorithm) 82

6.5 Description of notation for DP charts 83

List of Figures

1.1 Components of distributed simulation 2

2.1 A physical system 11

2.2 Basic logical processes 14

2.3 A physical system mapped into a logical system 15

2.4 Basic algorithm for a single process in a distributed simulation 16

2.5 A distributed system that deadlocks 20

2.6 A tandem network 23

3.1 A distributed system without deadlock 27

3.2 A delay process 37

3.3 Algorithm for delay process 39

3.4 Algorithm for INPUT operation 40

3.5 Algorithm for OUTPUT operation 41

3.6 A branch process 41

3.7 Algorithm for branch process 42

3.8 A merge process 43

3.9 Algorithm for merge process 44

3.10 An assembly line 45

3.11 A logical assembly line 46

4.1 A logical system 48

4.2 Message propagation 49

4.3 Merge process sending a 2cNULL message 51

4.4 Delay process sending a 2cNULL message 52

4.5 Branch process sending a 2cNULL message 52

4.6 New algorithm for INPUT operation 58

4.7 A feedback topology 59

4.8 A feedback topology with multiple cycles 64

5.1 A set of LANSF processes 68

5.2 Local processes of delay process 70

5.3 Local processes of branch process 71

5.4 Local processes of merge process 73

5.5 Local processes of buffer process 74

6.1 Relation between black-time and gray-time 93

6.2 Increasing merge's clock value 94

A.1 Tandem Networks 107

A.2 Network Topology Feedforward1 108
A.3 Network Topology Feedforward2 108
A.4 Network Topology Feedforward3 108
A.5 Network Topology Feedforward4 108
A.6 Network Topology Feedforward5 109
A.7 Network Topology Feedforward6 109
A.8 Network Topology Feedforward7 109
A.9 Network Topology Feedforward8 109
A.10 Network Topology Feedforward9 110
A.11 Network Topology Feedback1 110
A.12 Network Topology Feedback2 110
A.13 Network Topology Cluster 111
A.14 Network Topology Central Server 111

B.1 Speedup Tandem Networks (6/2) 113
B.2 Speedup Tandem Networks (20/2) 113
B.3 Speedup Tandem Networks (600/2) 114
B.4 Degree of Parallelism Tandem1 115
B.5 Degree of Parallelism Tandem2 115
B.6 Degree of Parallelism Tandem3 116
B.7 Degree of Parallelism Tandem4 116
B.8 Degree of Parallelism Tandem5 117

B.9 Speedup Topology Feedforward1 118
B.10 Degree of Parallelism Topology Feedforward1 118
B.11 Speedup Topology Feedforward2 119
B.12 Degree of Parallelism Topology Feedforward2 119
B.13 Speedup Topology Feedforward3 120
B.14 Degree of Parallelism Topology Feedforward3 120
B.15 Speedup Topology Feedforward4 121
B.16 Degree of Parallelism Topology Feedforward4 121
B.17 Speedup Topology Feedforward5 122
B.18 Degree of Parallelism Topology Feedforward5 122
B.19 Speedup Topology Feedforward6 123
B.20 Degree of Parallelism Topology Feedforward6 123
B.21 Speedup Topology Feedforward7 124
B.22 Degree of Parallelism Topology Feedforward7 124
B.23 Speedup Topology Feedforward8 125
B.24 Degree of Parallelism Topology Feedforward8 125
B.25 Speedup Topology Feedforward9 126
B.26 Degree of Parallelism Topology Feedforward9 126
B.27 Speedup Topology Feedback1 127
B.28 Degree of Parallelism Topology Feedback1 127
B.29 Speedup Topology Feedback2 128
B.30 Degree of Parallelism Topology Feedback2 128

B.31 Speedup Topology Cluster 129

B.32 Degree of Parallelism Topology Cluster 129

B.33 Speedup Topology Central Server 130

B.34 Degree of Parallelism Topology Central Server 130

B.35 Speedup Topology Feedback1 (2cNULL) 131

B.36 Degree of Parallelism Topology Feedback1 (2cNULL) 131

B.37 Speedup Topology Feedback2 (2cNULL) 132

B.38 Degree of Parallelism Topology Feedback2 (2cNULL) 132

B.39 Speedup Topology Cluster (2cNULL) 133

B.40 Degree of Parallelism Topology Cluster (2cNULL) 133

B.41 Speedup Topology Central Server (2cNULL) 134

B.42 Degree of Parallelism Topology Central Server (2cNULL) . . . 134

B.43 Null/Real Transmissions Topology Feedback1 135

B.44 Null/Real Transmissions Topology Feedback1 (2cNULL) . . . 135

B.45 Null/Real Transmissions Topology Feedback2 136

B.46 Null/Real Transmissions Topology Feedback2 (2cNULL) . . . 136

B.47 Null/Real Transmissions Topology Central Server 137

B.48 Null/Real Transmissions Topology Central Server (2cNULL) . 137

B.49 Null/Real Transmissions Topology Cluster 138

B.50 Null/Real Transmissions Topology Cluster (2cNULL) 138

Chapter 1

Introduction

1.1 Motivation

Simulation plays a strategic role in many fields today. Most of the products we use every day, have been designed with the help of simulation techniques. The understanding of complex phenomena such as weather forecasting or oil reservoir modeling is possible through the use of simulation programs.

Analytical modeling can be used to study the behavior of physical systems. However, a model constructed and analyzed with analytical methods is usually a simplified one in order to make it tractable. The other system modeling technique is called *simulation.* Simulation is more flexible than analytical modeling. The majority of interesting problems today are tractable by simulation only.

Simulation can take two approaches: the traditional approach is called *sequential simulation* and the newer approach is known as *distributed simulation.* The idea of distributed simulation was first proposed in 1977 by K. M. Chandy at the University of Waterloo, and was also proposed independently by R. E. Bryant [Bry77].

What is distributed simulation? It is a technique used to simulate a physical system and consists of three components (see Figure 1.1).

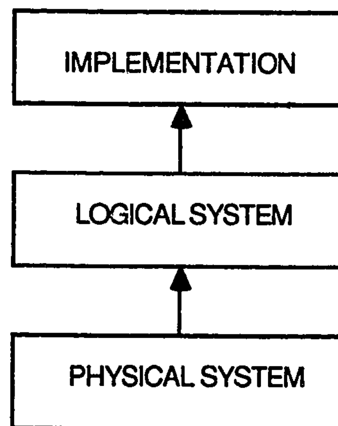


Figure 1.1: Components of distributed simulation

A *physical system* is defined by a set of *components.* For example, in a car wash, the arriving cars, and the car wash machine are the components. These components are known as *physical processes* (PPs).

Once the PPs have been identified in the physical system, a *mapping* operation has to be performed. We map the physical system into a *logical*

system. The logical system decomposes the physical system into autonomous processes that communicate via message passing. These autonomous processes are called *logical processes* (LPs). The idea is that each LP will be simulated by a different processor. There are different types of LPs as we will define in Chapter 2.

The *implementation* component is a simulation program that, based on the logical system, reflects the essential features of the physical system. The simulation program can run in a uniprocessor or in a parallel computer as long as the language allows inter-process communication [Mis86].

The main problems with distributed simulation are avoiding deadlock and insuring that causality constraints are not violated. Chandy and Misra [CM79] proposed an algorithm to solve these two problems. The algorithm uses NULL messages and is described in detail in Chapter 3.

The use of NULL messages causes overhead in the system being simulated. In Chapter 4 we introduce a new algorithm that requires additional information to be encoded in a message. The goal of the new algorithm is to reduce the number of NULL messages transmitted in the network. The main questions this thesis investigates are: What kind of networks are suitable for simulation in a distributed environment within the deadlock avoidance paradigm? Can we improve the speed of the sequential simulation algorithm by distributing tasks among different processors?

1.2 Thesis Objectives and Outline

This work is about the design and implementation of a distributed simulation testbed of queueing networks. We are interested in decomposing a given system into subsystems and simulating each subsystem in a single processor. There are very few empirical studies related to distributed simulation [Lak87, RF87, Sam85, Fuj88, Won88].

In this thesis, we report on experiments in which different factors that affect the performance of the distributed simulation approach varied:

- Topologies of the simulated system
- Inter-arrival time of transactions
- Processing delay
- Communication delay

The simulation techniques are compared with respect to the following measures:

- Speedup
- Degree of Parallelism
- Number of NULL messages transmitted

A distributed simulator was implemented using the simulation package LANSF [GR88c] to find out about the relationship among the factors mentioned above.

The outline of this thesis is as follows. In Chapter 2 we present a survey of simulation techniques. We discuss the sequential simulation algorithm and a basic distributed scheme. A brief survey of distributed simulation strategies is also included.

Chapter 3 contains a description of the deadlock avoidance paradigm for distributed simulation proposed by Chandy and Misra.

Chapter 4 introduces a new distributed algorithm: two-colored NULL messages. We define what changes have to be made to the Chandy and Misra algorithm presented in Chapter 3.

Chapter 5 describes the implementation details considered for the distributed simulator. We present a brief introduction to LANSF and provide an operational description of the different LPs implemented under LANSF.

Chapter 6 presents the results obtained for the two distributed simulation methods of Chapter 3 and Chapter 4. Eighteen different topologies were studied. To make the comparisons easier, we took many topologies from existing literature [Lak87, RMM88].

The last chapter summarizes the results obtained from the simulation experiments and presents some directions of future research for distributed simulation.

Chapter 2

Distributed Simulation

2.1 Introduction

This chapter presents a survey of simulation techniques. We also discuss how a sequential simulation differs from a distributed one. The general information is given in section 2.2. Section 2.3 describes the algorithm used in sequential simulation. Section 2.4 presents a basic distributed scheme found in most distributed simulation strategies. Section 2.5 provides a brief survey of distributed simulation techniques.

2.2 What is Simulation ?

A *simulation program* is a model of a physical system. A *model* has to reflect the essential features of interest of the *physical system.* An example of such a physical system could be an assembly line in a manufacturing plant or a computer communications network.

A simulation model can be used to predict how the physical system behaves under different working conditions.

2.2.1 System Modeling and Analysis

The methods widely used for system modeling and analysis are *analytic modeling* and *simulation.* When the physical system to be investigated is not very complex, analytic modeling can be used effectively. However, this approach is limited, because a number of restrictions have to be satisfied to keep the model mathematically tractable. On the other hand, in principle, simula[illegible]del a physical system as closely as required.

The [illegible] very detailed simulation models is that they are computational[illegible]nsive. For example, simulation of complex VLSI digital circuits for logic verification and fault analysis consume months of machine time [FWW84].

Models of many interesting physical systems are analytically intractable and simulation seems to be the only viable alternative to study these systems.

2.2.2 Simulation Methodology

A *simulation methodology* is needed to study the behaviour of a system. In a majority of cases, the user of a simulation program is also the simulation designer and often also the implementer. The current trend is that the user of a simulation program is a professional with functional expertise in the area being studied [MP86].

The simulation methodology described in [MP86] and [Nan84] consists of the following phases :

1. *Problem definition*

   Analysis of the system to be simulated and a statement of the objectives of the model.

2. *Construction of a simulation model*

   This model should reflect the principal features of the system.

3. *Data collection*

   This is needed to validate the model and to use the model for experimentation.

4. *Implementation of a simulation program*

   This phase not only includes the coding of the program, but also the design and implementation of the user interface, the output reports, and the file or data base structure to be used.

5. *Integration and test*

   This phase combines the software units and verifies that they work as specified in the design.

6. *Validation*

   The last phase verifies that the behavior of the constructed simulator mimics, as closely as required, the system being modeled.

2.2.3 Classification of Simulations

Simulations can be classified into two different groups: *discrete event simulations* and *continuous event simulation* [Kum86]. These approaches differ in their definition of how state transitions occur in time. A state transition is a change in the state of the model. In a discrete event simulation transitions occur at discrete points in time. Conversely, in a continuous event simulation model transitions occur "all of the time" and usually the system is described by a set of differential equations. In both cases multiple transitions may occur at the same instant in time. We will only consider discrete models in the sequel.

2.3 Sequential Simulation

Traditional simulation has an inherently sequential nature. A variable *clock* determines the time up to which the physical system has been simulated [Mis86]. An event list is maintained; it contains the events to be processed by the simulator. The events are ordered according to their non-decreasing time of occurrence. The sequential simulation algorithm works as follows: the head of the event list is removed, the event simulated and the simulation clock is updated, increasing the simulated time. This may cause new events to be added to the event list. Only the first item from the event list is removed in each iteration of the algorithm. It may be seen as a disadvantage because it is not possible to partition the event list and perform a concurrent execution on a multicomputer network. We will see in the next sections how the event list can be "distributed" in order to exploit the benefits of parallel processing.

The implementation of the event list affects the performance of the sequential simulation approach. McCormac and Sargent [MS81] performed an empirical study of 12 different algorithms for managing the event queue in discrete event simulations. The algorithms showed performance to be dependent on the physical system being modeled. For example, in models of telecommunication systems, the runtime ratio between different algorithms can differ as much as 5:1 [Hen83]. Empirical evidence suggests that *splay*

trees are one of the best approaches for implementing an event list [Jon86].

A survey of sequential simulation languages can be found in [Fra77] and [Mis86].

2.4 Distributed Simulation

2.4.1 The Causality Principle

Consider a physical system such as the car wash in Figure 2.1.

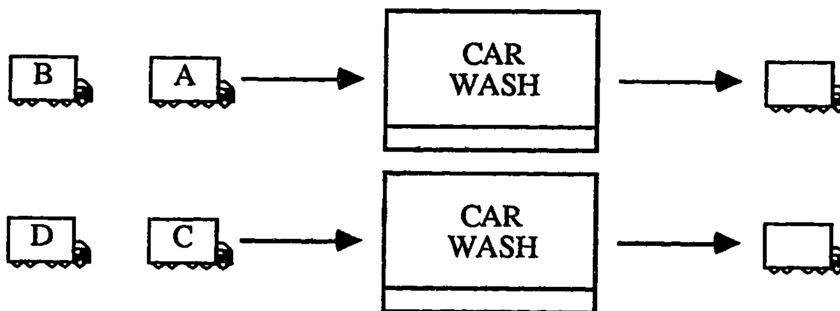


Figure 2.1: A physical system

When a car arrives, it waits in line until the car ahead has been washed. If the car wash is not servicing any car, then the car that arrives is serviced immediately.

Physical systems always obey the *causality principle* : the future can not affect the past. For example, in Figure 2.1 car B can not be serviced before car A. Car B depends on car A, but car A does not depend on car

B. Causality imposes a partial ordering of events in the physical system. In other words, the cause must always precede the effect. Events that have no dependencies do not have to occur at any specific order in time. Then, we have a partial ordering of events caused by the causality principle.

In sequential simulation, causality is accomplished by maintaining a priority queue of events. The event with the smallest timestamp is removed from the queue, simulated, and as a result, new events may have to be scheduled. These events are inserted in the event queue according to their time of occurrence. However, if we would like to run a simulation on multiple processors, insuring that causality constraints are not violated is a more difficult task. Many events are scheduled and executed in parallel. This is one of the core problems in distributed simulation.

2.4.2 A Basic Scheme for Distributed Simulation

The sequential nature of traditional simulation imposes a limitation on the study of complex systems: computational requirements are too costly. How can we improve the speed of the sequential simulation algorithm? Several techniques have been proposed: vectorization techniques [CB83], dedicated processors to perform specific simulation functions [Com84], execution of parallel independent replications on separate processors [Bil85, Hei88], and the development of parallel algorithms on a multicomputer network. To

benefit from the parallel algorithms, the physical system has to be:

1. Decomposed into autonomous processes that communicate only via message passing[1].

2. Simulate each subsystem on a different processor.

This technique is called *distributed simulation* [Mis86] and the set of communicating processes performing the simulation is called a *distributed simulator.*

A natural way of mapping a physical system—or queueing network—into a distributed simulator is to have a 1–1 correspondence between a process and a single physical server—the physical server includes the associated input queue. Each physical process is modeled by a logical process. The logical processes are interconnected according to the topology of the physical system being simulated.

The following kinds of logical processes are commonly used: source, delay, branch, merge, and sink processes (Figure 2.2).

The function performed by each of these process types is as follows:

- The *source* process generates jobs with an arrival rate and distribution which reflects the properties of the physical system.

[1]The communication could be performed through the use of shared variables, too.

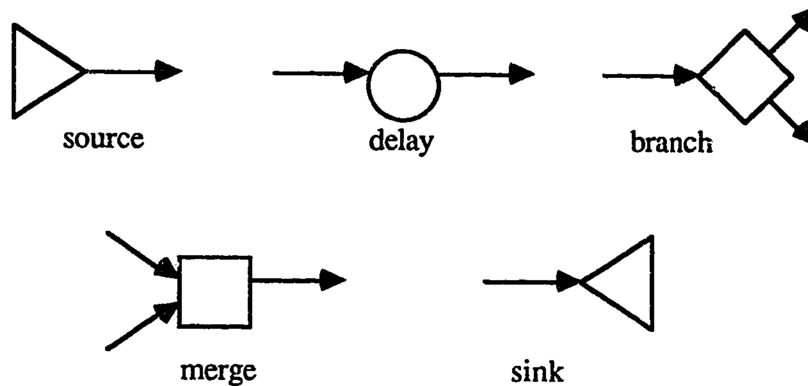


Figure 2.2: Basic logical processes

- The *delay* process receives a job, simulates a finite service time, and outputs a new job. The distribution of the service time is dependent on the physical server the delay node is modeling.

- The *branch* process selects one of the several output links to transmit the job.

- The *merge* process receives several input jobs, selects the one with the smallest timestamp, and outputs a new job. This process waits until all the input links have received jobs before performing the selection.

- The *sink* process receives jobs only.

The events of the physical system are modeled as transmissions of messages in the logical system. For example, jobs will enter the system from a

source process according to certain distribution of arrivals. These jobs (messages) will be received by other servers, get processed, and transmitted to other servers in the system. Finally, a job will leave the system when it is sent to a sink process (see Figure 2.3).

Each message is a pair of the form (t, m). The first element, t, is a timestamp indicating the time when a message was sent. The second element, m, codes the information carried by the message. In Figure 2.3 (Tb, Mb) represents the arrival of car b and (Ta, Ma) represents the departure of car a from the car wash. Propagation delays can always be modeled by adding more delay nodes, or by changing the job duration value.

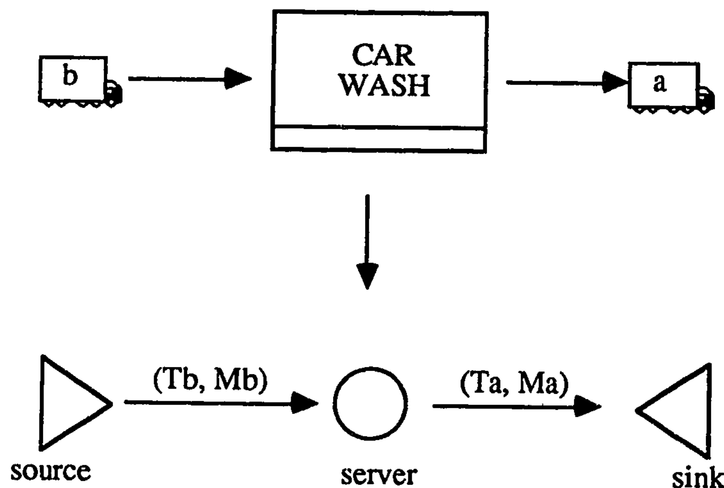


Figure 2.3: A physical system mapped into a logical system

Now, we define a basic scheme for distributed simulation that underlies most of the simulation schemes proposed in the literature [Lak87]. The pro-

cessors in the distributed system repeat the cycle defined in Figure 2.4.

```
---------------------------------------------------------
    WHILE termination criterion is not met DO
          WAIT until a message is received on all
               input lines
          SIMULATE the service for the earliest job
          COLLECT statistics
          OUTPUT message(s) in output line(s)
    ENDWHILE
---------------------------------------------------------
```

Figure 2.4: Basic algorithm for a single process in a distributed simulation

2.4.3 Global Clocks vs Distributed Clocks

It is natural to partition the system and map each physical process into a logical process. However, mapping the physical time to logical timestamps is more difficult.

In sequential simulation, time is simulated by a global clock. Causality constraints are not violated since events are processed according to their timestamp. The next event to be processed is the one with the smallest timestamp. This serialization has to be eliminated if we want to take advantage of parallel processing. Events that do not depend on each other can be executed concurrently.

How can we replace the global clock by a distributed one? There are two approaches to this problem. The first approach is called *time driven*

simulation. A global clock is used to advance one tick in every step, and all events scheduled at that time are simulated. This approach is practical in situations where the number of events per unit of simulated time is high.

The second approach is the so-called *event driven* method. Each logical process has its own local clock. The local clocks are advanced to simulate events in the physical system. Clocks in different processes may differ at any point of simulation. The fundamental problem with this approach is to ensure that causality constraints are not violated. We can consider each process as a subsystem that has local causality constraints. Each logical process inputs timestamped messages from other neighboring processes. If messages arriving from these processes are processed in increasing timestamp order, then the local causality constraint will never be violated.

Communication between processes is done exclusively through timestamped messages. Processes do not share any variables. The causality constraint problem is solved if each process guarantees not to violate the local causality constraint [RF87]. When a process has a message to be processed, it has to decide if it needs to wait for other messages, on the other input links. It compares the timestamp from all the messages and picks the smallest one. This is needed to ensure that messages are processed in the correct order.

The problem with the basic scheme of Figure 2.4 is that it may deadlock, as we show in the following modified example taken from [CHM79]. Consider

the distributed system depicted in Figure 2.5.

Suppose that messages from the source are transmitted every 5 time units and that they follow the ABD path. When a delay node receives a message, it simulates a job of 1 second duration. We will assume the propagation delay to be 0. The communication protocol used by the processes works as follows: a message is transmitted only if the sender is ready to transmit and the receiver is ready to receive a message. We assume that channels do not have buffers.

- **Step a.** The source process sends the first message: (5,m1).
- **Step b.** Process A receives (5,m1); this message is transmitted to process B.
- **Step c.** The source process generates a second message: (10,m2); process B simulates one unit of time and outputs (6,m1) to process D.
- **Step d.** (10,m2) is transmitted to process B. We notice that process D can not process (6,m1), since it has to have messages on both input links.
- **Step e.** The source process generates a third message: (15,m3). However, process A can not send (15,m3) to process B. Process B is not ready to receive a message. Because messages will never flow through process C, process D will be waiting forever. Process B will be waiting

for D to be ready to receive, and process A will be waiting for B to be ready to receive. The source is unable to send messages to process A, because process A will never be ready to receive. The simulator will not be able to advance and deadlock results. If we change the protocol when the incomming messages are buffered, there will be no deadlock but the memory requirement will be very high.

Several distributed simulation strategies have been proposed to solve the core problems of causality constraints and deadlock resolution as we will explain in the following section.

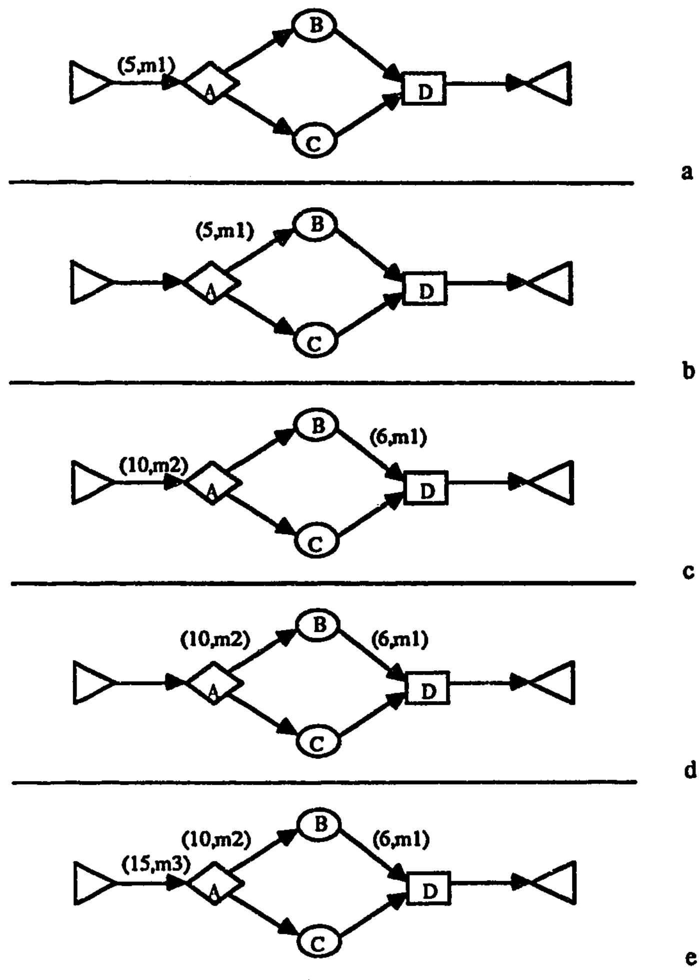


Figure 2.5: A distributed system that deadlocks

2.5 Overview of Distributed Simulation Strategies

The idea of distributed simulation was initially proposed by Chandy in 1977 in a series of lectures at the University of Waterloo, and was also proposed independently by R. E. Bryant [Bry77]. Bryant's work assumes an infinite buffer simulation model.

Chandy and Misra [CM79, CHM79] proposed a deadlock avoidance approach based on *NULL messages* to prevent deadlock. The major drawback of this technique is the overhead produced by the transmission of NULL messages. This scheme will be described in detail in the following chapter.

Chandy and Misra's ideas were significantly developed and extended by the Computer and Communication Networks Group at the University of Waterloo: Peacock, Manning and Wong [PWM79, PMW80] proposed algorithms for event-driven simulation with a network of processors. Examples of these algorithms are the virtual ring algorithm and the link-time algorithm. These algorithms use a central controller to broadcast a signal to every other component, in order to indicate the end of a simulation interval.

Holmes [Hol78] designed parallel algorithms for discrete event simulations of feedforward networks using *probe messages.* Probe messages were sent out to collect status information of processes and detect a deadlock situation. He suggests a method for allocating processors to servers, in order to optimize

processor utilization.

Seetha Lakshmi's [Lak87] performance study uses Chandy and Misra's deadlock avoidance scheme with the introduction of limited in size buffers on each communication line between processors. We will make a comparison with this study in Chapter 6.

In 1981, Chandy and Misra [CM81] proposed another distributed scheme for deadlock detection and recovery. A distributed algorithm proposed by Dijkstra and Scholten [DS80] is used as the basis for deadlock detection. The distributed simulator is allowed to deadlock using this technique. Separate mechanisms are needed to identify and break the deadlock situation. The strategy is composed of a sequence of parallel steps : 1) Simulate until deadlock. 2) Detect the deadlock. 3) Resolve the deadlock and return to step 1. There is a central controller process who manages the transition from one phase to the next.

Jefferson and Sowizral [Jef85, JS85] proposed the *Time Warp* strategy. This scheme is more liberal than the previous conservative approaches. Processes are allowed to advance local clocks as rapidly as possible. When the causality principle is violated, i.e. an error does happen, the processes *roll back* their computations to a point in simulated time before the error. Processes continue computations until an error is detected again. Lavenberg [LMS83], Samadi [Sam85], and Wong [Won88] reported performance studies of this method.

Kumar [Kum86] suggests a new scheme to simulate feedforward networks and offers efficient algorithms for the related problems of termination detection and sequential simulation, based on Chandy and Misra's work. The emphasis in Kumar's work is on reducing the overhead produced by NULL messages. An analytic method to predict the performance of the algorithm is presented and verified via simulation.

Reed, Malony and McCreedie [Ree85, RMM88] provide an empirical study of data, measuring the performance of specific implementations of Chandy and Misra's deadlock avoidance and deadlock detection and recovery schemes. The study was performed in a shared memory environment and concluded that the conservative approaches performed poorly for different network topologies, with the exception of tandem networks. Tandem networks are a set of processors connected in a pipeline fashion (see Figure 2.6). These results however are not surprising, accessing the shared memory was a bottleneck.

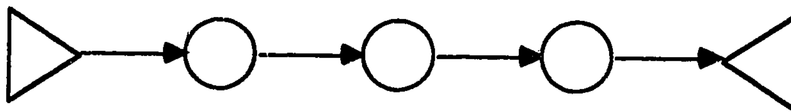


Figure 2.6: A tandem network

Lubachevsky [Lub88] presents a new distributed algorithm to simulate asynchronous multiple loop networks. The algorithm achieves substantial

speed-ups compared to the sequential simulation algorithm in simulating a toroidal network: 16 times faster using 25 processing elements (PEs) on a shared memory MIMD computer, and 1900 times faster using 2^{14} PEs on a SIMD computer.

A good survey of distributed simulation, can be found in [Mis86] and [Kau88].

Chapter 3

Avoiding Deadlock in Distributed Simulation

3.1 Introduction

This chapter presents a detailed descrip[illegible] of the deadlock avoidance paradigm for distributed simulation. The algorithm was first proposed by Chandy and Misra [CM79].

This chapter is organized in 7 sections. Section 3.2 contains the assumptions to consider for the behavior of the logical processes and the parallel computer used. Section 3.3 introduces the notation that will be used in describing the algorithm. In section 3.4 we present the operational details of the three phases that compose the algorithm: selection phase, computation

phase and the input/output phase. Section 3.5 defines the notation used in parallel programs. In section 3.6 we use this notation along with pseudocode to specify the operation of the basic logical processes: delay, branch and merge. Section 3.7 explains how the interconnection between processes takes place in the logical system and presents Hoare's synchronous protocol as the communication protocol used by the logical processes.

3.2 Assumptions

We start by giving an informal description of the algorithm. A more detailed description follows in the next sections.

In the previous chapter we presented an example that deadlocked (see Figure 2.5). How can process D continue processing messages if it has to have messages on all incoming input lines and one input line is starved? The solution is to send a NULL message from process A to process C. When process A—a branch process— receives a message, the message is serviced, and a new message is output to process B. Now, we require that, besides this message, another message is sent to process C, a NULL message. This NULL message contains the same timestamp as the real message sent to process B. NULL messages are needed to avoid deadlock. Process D will be able to

process messages without having to wait forever on both of its input links.

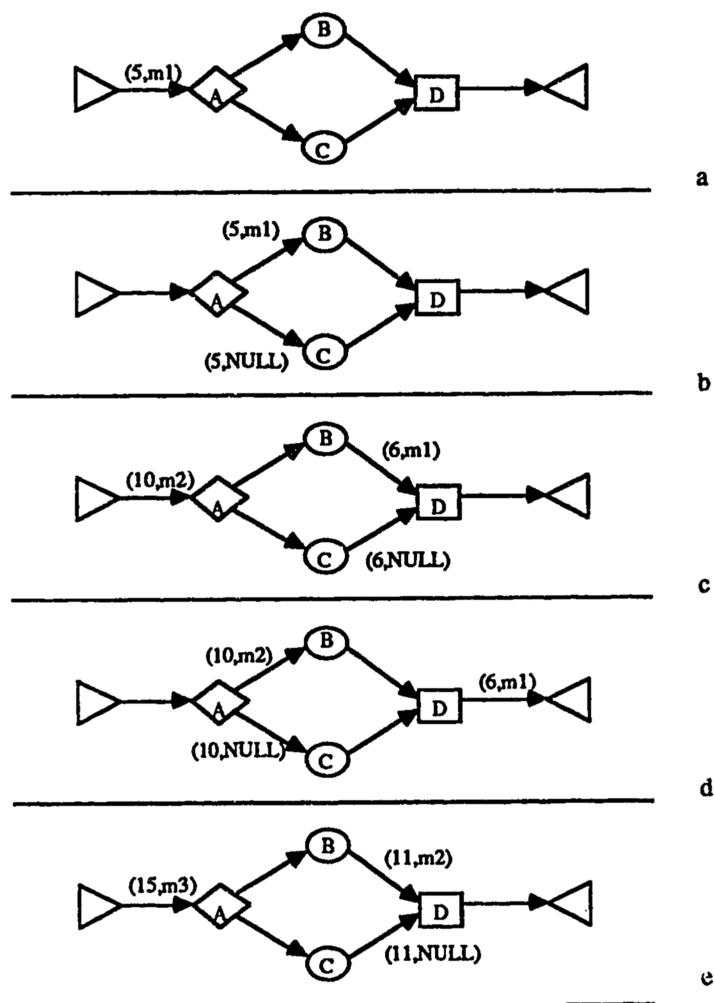


Figure 3.1: A distributed system without deadlock

NULL messages are not a part of the physical system, but they are needed to keep the logical processes working. Figure 3.1 shows the steps required to send NULL messages in the distributed system.

In the discussion that follows, we consider the following assumptions:

- Messages sent from one process to another arrive in the same order as they were sent.
- Messages are transmitted through the network without error. In other words, retransmissions are never required.
- A process sends timestamped messages to neighboring processes. These timestamps form a nondecreasing sequence.

  $$0 < t_A < t_B < \ldots < t_K \leq Z$$

  We are simulating the system for a period of time [0, Z], where Z is referred to as the *termination time.*
- All computation occurs as the result of processing a received message. There are no "magic" processes which spontaneously generate messages —aside from the source process.
- Some time must elapse after a process sends one message before it sends another one on the same line.

  $$t_k - t_{k-1} > \epsilon \quad \text{for} \quad k = 2, \ldots, K$$

- The output of a process at time t depends solely upon the messages received by the process at or before t.

3.3 Notation

We present the notation used to describe the algorithm:

- We are interested in simulating a physical system composed of processes. Such processes are called *physical processes* (PP).

- The physical system is mapped into a logical system. Each PP is mapped to a logical process LP.

- Interactions between processes in the physical system are modeled as the transmissions of messages in the logical system. LPs transmit messages that are tuples of the form (t, m). The first element, t, is the timestamp of the message; the second element is the actual message or it is a special message denoted as the NULL message.

- TIN_{ki} is defined as the *channel clock value* for the last message received from LP_k by LP_i. $TOUT_{ij}$ is the channel clock value for the last message transmitted by LP_i to LP_j. The channel clock value indicates the time up to which the logical processes have simulated interactions between the corresponding physical processes.

- T_i is the clock value of LP_i and is defined as

$$T_i = \min_{j,k} \quad (TIN_{ki}, \ \ TOUT_{ij}).$$

  T_i reflects the time up to which LP_i has simulated PP_i.

- $L_{ij}(t)$ is the *lookahead function* for communications between LP_i and LP_j. When LP_i receives a message, it has to predict when this message will be output. The process has to determine the timestamp of the next message to be sent to LP_j, assuming it has received all incoming messages with timestamp of t or less.

- $LINKCLOCK_{ij}$ is a variable associated with the link between LP_i and LP_j; its value indicates a lower bound on the timestamp of any future message that can be transmitted over the link. $LINCLOCK_{ij}$ is defined as

$$LINKCLOCK_{ij} = \min_{k} \quad TIN_{ki} \ + \ L_{ij}(\min_{k} \quad TIN_{ki}).$$

  LP_i can determine all messages transmitted to LP_j up to time $LINKCLOCK_{ij}$. $TOUT_{ij}$ is updated whenever a message is transmitted from LP_i to LP_j. $LINKCLOCK_{ij}$ is affected when a message is received by LP_i.

The processors in the logical system repeat the following cycle, composed of three phases described in detail below, until the termination criterion is met :

1. **Selection Phase.** Selects input and output links on which I/O operations will be carried out in parallel. $NEXT_i$ is defined as the set of processors on which LP_i is to perform the I/O operations. $NEXT_i$ may contain input lines and/or output lines.

2. **Computation Phase.** For each output link in $NEXT_i$, determine the next message to be sent on that link. The next message can be a real message modeling an interaction in the physical system or a NULL message, required to avoid deadlock.

3. **I/O Phase.** Perform the input/output operations for all links in $NEXT_i$.

3.4 Phases of the Algorithm

The Chandy and Misra algorithm is correct, deadlock free, and terminates properly [CM79]. Each of the phases is explained now in detail.

3.4.1 The Selection Phase

LP_i selects the set of input and output links in $NEXT_i$ on which the I/O operations will be performed next.

$$NEXT_i = \{k \mid TIN_{ki} = T_i\} \cup \{j \mid TOUT_{ij} = T_i \wedge LINKCLOCK_{ij} > TOUTij\}.$$

$NEXT_i$ contains all input and output links of LP_i whose clock value equals the clock value of LP_i. $NEXT_i$ is defined in such a way that after the I/O operations are computed, the clock value of LP_i increases to the minimum value between TIN_{ki} and $TOUT_{ij}$.

3.4.2 The Computation Phase

The computation phase decides what messages will be sent on each output link in $NEXT_i$. If $NEXT_i$ contains only input links, then LP_i does not perform computations in this phase. The selection phase specifies that all messages with timestamp $LINKCLOCK_{ij}$ or less can be computed without any additional input. Two possible cases exist :

1. If PP_i sends a message to PP_j in the time interval $(TOUT_{ij}, LINKCLOCK_{ij}]$, then send a message from LP_i to LP_j.

2. If no messages are sent from PP_i to PP_j in the interval $(TOUT_{ij}, LINKCLOCK_{ij}]$, send a NULL message with timestamp

$LINKCLOCK_{ij}$.

3.4.3 The I/O Phase

Input/Output operations can be performed concurrently on all links in $NEXT_i$. The channel clock value for the links is updated for each message received or transmitted. The clock value for LP_i is updated to the minimum channel clock value for LP_i.

3.4.4 Initialization

At the beginning, the following variables are set to 0: The channel clock value (TIN_{ki} and $TOUT_{ij}$), the process clock (T_i) and the variable associated with each link, $LINKCLOCK_{ij}$. The distributed simulator starts with the selection phase. The source process begins generating messages immediately. Also, delay processes are required to send a NULL message at the beginning. This is needed to avoid deadlock if the topology contains feedback or cycles. Logical processes do not have knowledge about the topology they belong to, and it is necessary to send this NULL message.

3.4.5 Termination

The LPs repeat the three phases, selection, computation and I/O operations as long as $T_i < Z$. Z is defined as the termination time. If a process generates

a message with a timestamp greater than Z, the message is replaced by the tuple (Z, NULL).

3.5 Notations for Distributed Programs

In this section we present the notation used to describe the operation of the logical processes. A *distributed program* is a collection of LPs that communicate only through messages and there are no global or shared variables. There is no central process to control or synchronize other processes. We present the SEND and RECEIVE commands based on Lakshmi's work [Lak87] and Hoare's CSP input and output commands [Hoa78]. By using these two commands we define a parallel operation for a distributed program. The internal operation of a LP is defined by a code composed of SEND, RECEIVE, parallel and sequential commands.

3.5.1 SEND Command

Interactions in the physical system are modeled as the transmission of messages in the logical system. For example, consider when a car arrives at the car wash of Figure 2.1. This interaction is modeled by the source process as the transmission of a message containing the time of car's arrival. When a process needs to transmit a message, the command SEND—defined in the LP's internal code—will be used to accomplish this action. To specify that

LP_i needs to send a message to LP_j we use:

$$SEND\ (j,\ message)$$

The SEND command has two arguments. The first argument is the name of the process that will receive the message (process j); the second argument — message — is a tuple of the form (t, m) as described earlier.

3.5.2 RECEIVE Command

On the other side of the communication channel we have the RECEIVE command:

$$RECEIVE\ (i,\ message)$$

If LP_j wants to receive a message from LP_i, it will have to issue the previous command. We require that the execution of the RECEIVE command by LP_j will be successful only if LP_i issues the corresponding SEND command. The same applies for the SEND command: the respective RECEIVE command has to be issued. More details of the communication protocol will be given in section 3.7.

3.5.3 Parallel Command

A basic operation in a distributed program is a *parallel command.* A parallel command is a collection of commands, which may be executed in any order.

We consider only RECEIVE or SEND commands as the constituents of the parallel command we will use for the description of our distributed programs. Merge processes have to input messages from two or more sources; this input operation can be performed by the use of a parallel command. Consider that LP_k has to input messages from LP_i and LP_j. This operation is described by:

$$[RECEIVE\ (i,\ message)\ ||\ RECEIVE\ (j,\ message)]$$

The execution of the parallel command will be completed when all the constituent commands are completed.

3.6 Types of Processes

Based on the parallel command and the SEND and RECEIVE commands, we specify the operation for the basic processes : delay, branch and merge. These basic processes are sufficient to model any physical system by a distributed simulation scheme.

3.6.1 Delay Process

A *delay process* has an input and output link. It models a single server in the physical system (see Figure 3.2).

The following variables are associated with the delay process:

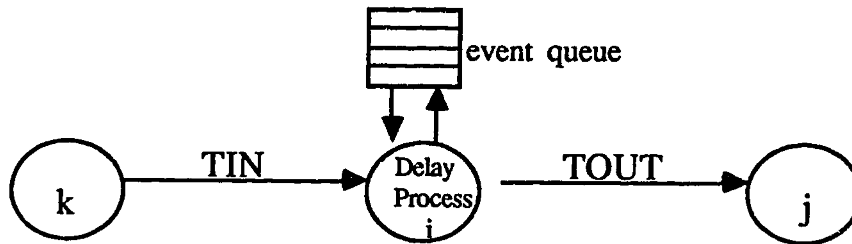


Figure 3.2: A delay process

- **T** is the process clock value. It indicates the time up to which the logical process has simulated the physical process.

- **TIN** is the timestamp of the last message received by the process. The last message can be a real message or a NULL message.

- **TOUT** is the timestamp of the last message transmitted by the process.

- **event queue.** Each delay process has a local event queue. Messages are stored in this local list and then transmitted at a later time. The original Chandy and Misra algorithm does not require an event queue in a delay process. The implementation by Lakshmi [Lak87] uses a counter to keep track of the number of real messages received; this is the reason why the algorithm requires a bounded amount of memory. However, there are physical systems in which the jobs contain information required to make a decision at a certain point in time; the only

way of having this information is to store the job in the event queue.

- **service time.** It is the simulated time the process spends with each message.

The operation of the algorithm for the delay process is as follows:

1. During the **initialization** step, all the variables maintained by the process are set to 0. A NULL message is initially sent at the beginning by LP_i to prevent a deadlock if the topology contains cycles. This NULL message is not needed if the topology is classified as a feedforward topology[1]. We assume that processes do not have knowledge of the topology they belong to. Also, TOUT is updated to the value of service time. If service time is 5 time units, then the message (5, NULL) is sent at the beginning. We notice that this message is not sent in the physical system, but it is required to avoid deadlock. Message (5, NULL) announces the absence of messages to neighbor LP_j up to time 5.

2. After the initialization procedure is completed, the process **loops** in a cycle until the termination criterion is met. During this loop, three different conditions are tested (see Figure 3.3):

[1] A feedforward topology is a network that contains no directed cycles.

```
DELAY ::
 {
'Initialization
     T := 0;
     TIN := 0;
     TOUT := 0;
     predicted_time := 0;
     event_queue := empty;
 'Send first NULL message
     SEND(service_time, NULL);
     TOUT := service_time;

 'Perform loop until termination condition is met
      while T < Z do
               in case of :
                  (TIN < TOUT) : INPUT
                  (TIN > TOUT) : OUTPUT
                  (TIN = TOUT) : [INPUT || OUTPUT]
               endcase;
               T := min (TIN, TOUT);
      endwhile;
}
```

Figure 3.3: Algorithm for delay process

- $TIN < TOUT$. This condition means that an *input operation* is performed. When LP_i receives a message from LP_k, the message can be a NULL message or a real message. If the message is a NULL message, the timestamp is updated with the value of TIN + service-time, and the message is inserted in the local event queue. If the message arriving is a real message, the predicted time for

this message will depend on its timestamp and the value of the predicted time for the last message (Figure 3.4). This is required to model the physical system correctly.

```
INPUT ::
 {
    RECEIVE(k, (TIN, m));
    if (m = NULL) then insert(TIN + service_time, NULL);
     else
      if (predicted_time < TIN) {
           insert(TIN + service_time , m);
           predicted_time := TIN + service_time;
      } else {
           insert(predicted_time + service_time, m);
           predicted_time := predicted_time + service_time;
      }
 }
```

Figure 3.4: Algorithm for INPUT operation

- $TIN > TOUT$. When this condition is true an *output operation* is executed. When LP_i has to output a message, it searches through the event queue for a real message. If there are no real messages, the last message in the event queue is sent and the previous messages are ignored (see Figure 3.5).
- $TIN = TOUT$. This condition states an *input and output operation.* When the process has to perform input and output operations, the two previous operations are executed.

```
OUTPUT ::
 {
      repeat
        next_message := delete(event_queue);
      until (next_message = last) or (next_message = real);

      SEND(j, next_message);
 }
```

Figure 3.5: Algorithm for OUTPUT operation

3.6.2 Branch Process

A *branch process* has one input link and two or more output links. We will explain the algorithm for the case of two output links, but it is straightforward to generalize the approach for more links.

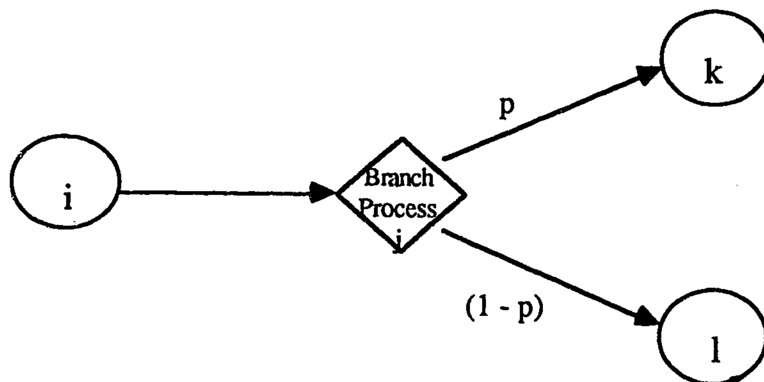


Figure 3.6: A branch process

In Figure 3.6 LP_i is the process from where branch process j can receive messages. LP_j can output messages to LP_k and LP_l. When a message arrives,

the branch process generates a random number $0 \leq p \leq 1$. The output links have associated probabilities, p and $(1-p)$, respectively. Depending on the random number generated, one of the two output links is selected as the one that will transmit the original message received. The original message received can be a real message or a NULL message. When a message is sent on one link, the other output link will transmit a NULL message with the same timestamp as the original message. The timestamp is not modified because there is no processing performed by the branch process. Figure 3.7 shows the algorithm for the branch process. Notice that in general, the routing need not be based on a random number, it can depend on the information contained by the received message.

```
BRANCH ::
{
 'Perform loop forever

 repeat
  RECEIVE(i, (TIN, m));
  output_link := random(0,1);
  if (output_link <= p) [SEND(k, (TIN, m)) || SEND(l, (TIN, NULL)];
      else              [SEND(l, (TIN, m)) || SEND(k, (TIN, NULL)];
 forever;
}
```

Figure 3.7: Algorithm for branch process

3.6.3 Merge Process

A *merge process* receives messages on two or more input links and transmits messages on a single output link (see Figure 3.8).

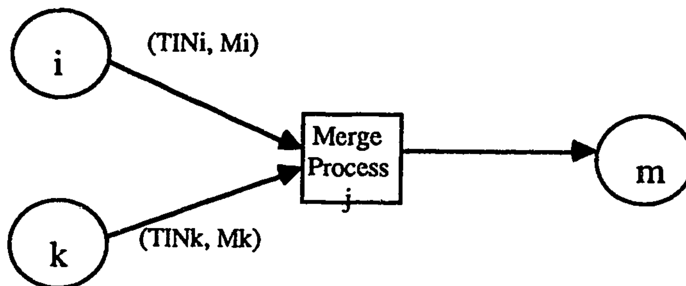


Figure 3.8: A merge process

Initially, merge process j issues a parallel command to receive messages on both input ports. When the parallel command is completed, that is, when messages have arrived on both input ports, their timestamps are compared. The message with the smallest timestamp is transmitted (SEND) and a request to input another message (RECEIVE) is issued for the respective link. When the timestamps of both messages are equal, we have to know if the messages are real or NULL messages. If one or both of the messages are NULL messages, then only one is sent. When both are real messages, then one message is transmitted after the other, and a request to RECEIVE messages on the input links is performed. As in the case for the branch process, it is straightforward to adapt the algorithm for more than two input links (see Figure 3.9).

In the previous chapter we mentioned that ensuring that the causality constraints are not violated is one of the core problems in distributed simulation. The merge process has to compare both timestamps on incoming messages to ensure that the sequence of transmitted messages matches the chronological order of the corresponding interactions in the physical system.

```
MERGE ::
{
 [RECEIVE(i, (TINi, Mi)) || RECEIVE(k, (TINk, Mk))];
'Perform loop forever
 repeat
  in case of :
   (TINi < TINk) :  SEND(l, (TINi, Mi));
                    RECEIVE(i, (TINi, Mi));
   (TINi > TINk) :  SEND(k, (TINk, Mk));
                    RECEIVE(k, (TINk, Mk));
   (TINi = TINk) :  if (Mi = NULL) SEND(l, (TINk, Mk));
                     else
                      if (Mk = NULL) SEND(l, (TINi, Mi));
                       else {
                         SEND(l, (TINi, Mi));
                         SEND(l, (TINk, Mk));
                       }
                    [RECEIVE(i, (TINi, Mi)) || RECEIVE(k, (TINk, Mk))];
  endcase;
 forever;
}
```

Figure 3.9: Algorithm for merge process

3.7 Interconnection of Logical Processes

There is no processing performed by merge and branch processes. The function of these processes is to provide routing functions only. The interconnection of logical processes is dependent on the physical topology that is being simulated. For example, consider the assembly line depicted in Figure 3.10.

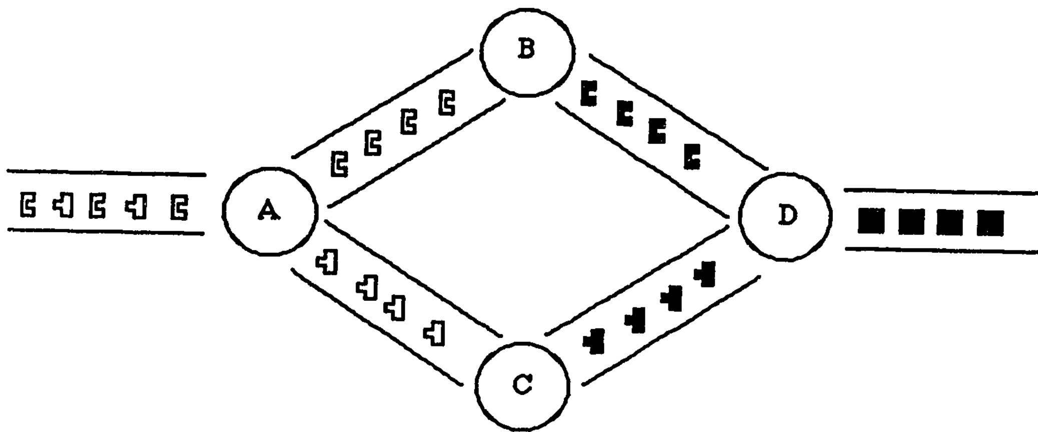


Figure 3.10: An assembly line

Each of the servers (A, B, C and D) in the assembly line will be simulated by a delay process in the logical system. When one of the servers outputs jobs to more than one server, then we will have to introduce a branch process. Also, when one of the servers has multiple inputs, a merge process will be needed. The corresponding logical system is shown in Figure 3.11.

The communication protocol used by the logical processes is based on

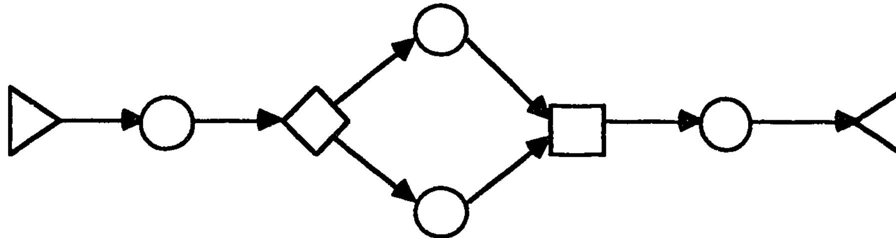


Figure 3.11: A logical assembly line

a very simple protocol proposed by Hoare [Hoa78]: *a message is sent from process i to process j if and only if process i is ready to send the message and process j is ready to receive it.*

When a pair of processes want to communicate, they have to issue a SEND or RECEIVE command. However, it may happen that one of the processes issued the SEND command, and the other process is busy performing a computation. This is an example of a blocked process. Also, we can have the situation in which a process is ready to RECEIVE, but the other process has not issued the SEND command. In this case, we say that the process is starved.

Chapter 4

A New Algorithm: Two-Colored NULL-Messages

4.1 Introduction

We introduce a new distributed algorithm for simulation of queueing systems in this chapter. It is based on a method proposed by Gburzyński and Rudnicki [GR88a] and Chandy and Misra's conservative approach [CM79]. The new algorithm has been simplified from its original version [GR88a] and extended to handle multiple source processes. This chapter is organized as follows. Section 4.2 describes the main disadvantage in Chandy and Misra's approach by means of an example. In section 4.3 we present the operation of the new algorithm based on a two-colored NULL-message. Sections 4.4 and

4.5 present the changes required by merge and delay processes in the new algorithm.

4.2 Disadvantage of NULL Messages

We have seen that NULL messages are required to avoid deadlock. However, the processing of NULL messages causes overhead in the simulation system. Consider the logical system in Figure 4.1.

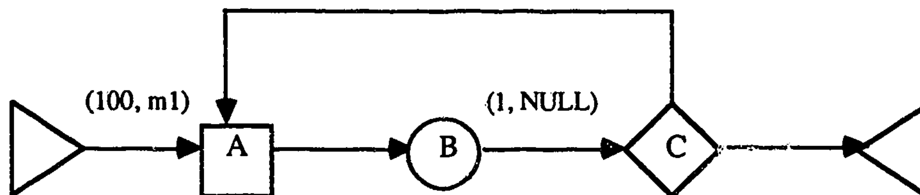


Figure 4.1: A logical system

Assume that the source sends messages every 100 time units, and that process B spends 1 time unit with every message received. During the initialization step, process B sends the message (1, NULL) to process C. The source process sends (100, m1). Process C receives the message (1, NULL) which in turn is transmitted on both of the output links. Messages are propagated as we can see in Figure 4.2.

Process A has to compare the timestamps of incoming messages as mentioned in the previous chapter. Message (1, NULL) is selected and transmit-

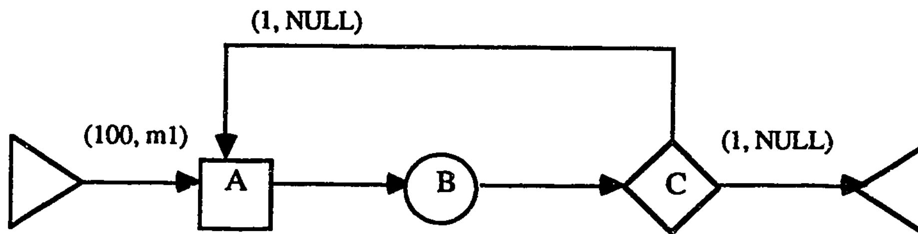


Figure 4.2: Message propagation

ted due to its lesser timestamp. By looking at this specific topology, we can see that 100 NULL messages will be required just to process (100, m1). The logical system remains idle for a long time, waiting for specific conditions to happen. In this case, merge process A has to process the message with the smallest timestamp. The flow of global time is simulated through the use of NULL messages. Process A has no means of knowing that there are no real messages in the system, and that (100, m1) could be processed safely, without violating causality constraints. How can we advance the local clock in a LP without violating causality constraints?

4.3 Using Two-Colored NULL-Messages

We propose the use of *Two-Colored NULL-Messages* as the solution to the problem mentioned above. We will use the Chandy and Misra algorithm with NULL messages, and additionally introduce a special kind of NULL message

called *Two-Colored NULL-Message*[1]. A 2cNULL message has the following components:

(black-time, 2cNULL, gray-time, owner)

- **black-time.** It is the value of the t component in the tuple (t, m) used in the original algorithm. That is, the time up to which the logical system has simulated the physical channel.
- **2cNULL.** Identifies the special message.
- **gray-time.** It is a predicted time, that according to certain conditions, will allow a merge process in a cycle, to increase the local clock time.
- **owner.** It is the identity of the process that originally sends the 2cNULL message. Only merge processes are allowed to create and be owners of 2cNULL messages.

The difference between the new distributed algorithm and the Chandy and Misra strategy resides in encoding the last two items mentioned above— gray-time and owner— as additional information in a message. In the last example, we required 100 NULL messages to process one real message. In Figure 4.3, consider that process A instead of sending (1, NULL) sends the following 2cNULL message: (1, 2cNULL, 100, A).

[1] We will use the term 2cNULL for short.

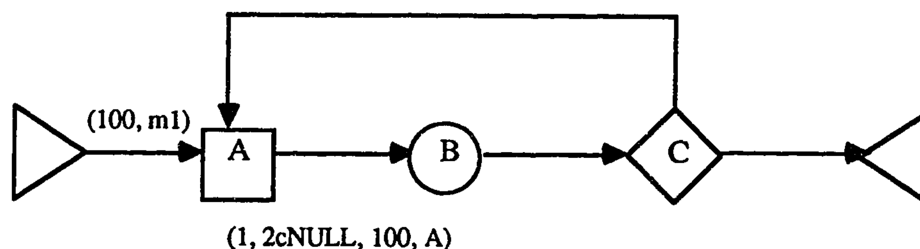


Figure 4.3: Merge process sending a 2cNULL message

The additional information sent is the gray-time and the owner. The gray-time is a forecasted time that will allow a merge process, under certain conditions, to select from the two input messages the one with the greatest timestamp, transmit this message, and increase its local clock safely.

In Figure 4.3 the input link for process B is open since TIN = 0 (no message has been received by process B) and the output link is closed: TIN = 1 (the last message transmitted by process B had a timestamp of 1). Process B receives the 2cNULL and TIN is updated to a value of 1. Because TIN = TOUT, the output link for process B is open and process B outputs a 2cNULL message to process C, increasing the timestamp to two units of time (see Figure 4.4).

Process C accepts (2, 2cNULL, 100, A) and sends this 2cNULL message to process A and the sink (see Figure 4.5).

Process A can safely transmit (100, m1) now without having to process 100 NULL messages. It knows that the 2cNULL has returned and that there

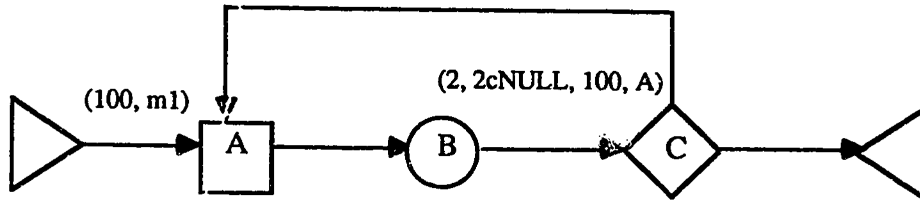


Figure 4.4: Delay process sending a 2cNULL message

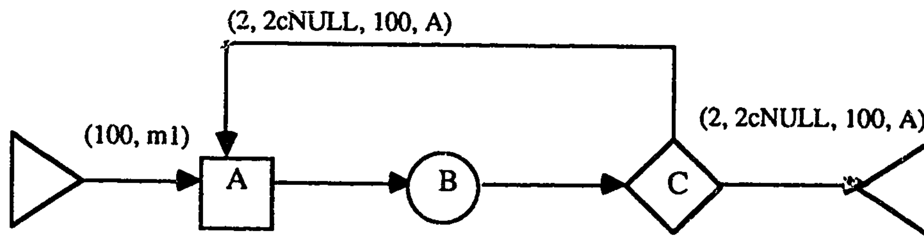


Figure 4.5: Branch process sending a 2cNULL message

are no messages with *timestamps* ≤ 100. It is a required condition that the LP has to be the father of the 2cNULL message, in order to send (100, m1). The algorithm takes advantage of knowledge of global time by encoding the gray-time in the message. Gray-time notifies the merge process, that only NULL messages are traveling in a cycle, and that it can safely process the message on the other input line, without having to wait for additional inputs.

The new algorithm modifies the behavior of merge and delay processes only. Branch, source, and sink processes behave as in the original Chandy and Misra approach. The purpose of the colored messages is to reduce the

number of NULL messages transmitted in topologies that contain cycles. By reducing the number of NULL messages, it is expected to improve the performance of the distributed simulation method.

4.4 Merge Process

The modified algorithm for the merge process follows. The notation is as in Figure 3.8.

```
MERGE ::
{
 [RECEIVE(i, (TINi, m)) || RECEIVE(k, (TINk, m))];
 repeat
  in case of :
   (TINi < TINk) :
            ------------------------------
             create a 2cNULL message
            ------------------------------
             IF (Mi = NULL) {
                 gray_time := TINk;
                 SEND(1, (TINi, 2cNULL, gray_time, MERGEj));
                 RECEIVE(i, (TINi, Mi));
              } else {
            ------------------------------
              advance local clock
            ------------------------------
               if (Mi=2cNULL) & (owner=MERGEj) & (TINk=GRAY_TIMEi) {
                   SEND(1, (TINk, m));
                   [RECEIVE(i, (TINi, Mi)) || RECEIVE(k, (TINk, Mk))];
             } else {
            ------------------------------
              send smallest
            ------------------------------
```

```
             if (Mi=2cNULL) & (GRAY_TIMEi > TINk) & (owner <> MERGEj)
                SEND(1, (TINi, 2cNULL, TINk, MERGEj));
               else
               SEND(1, (TINi, Mi));
             RECEIVE(i, (TINi, Mi);
            }

(TINi > TINk) :  { symmetric to  TINi < TINk }

(TINi = TINk) :

          ------------------------------
            send both real messages
          ------------------------------
            if (Mi = real) & (Mk = real) {
                  SEND(1, (TINi, Mi));
                  SEND(1, (TINk, Mk));
            } else
          ------------------------------
            send Mi
          ------------------------------
              if (Mi = real) & (Mk = NULL or Mk = 2cNULL)
                  SEND(1, (TINi, Mi));
              else
          ------------------------------
            send Mk
          ------------------------------
               if (Mk = real) & (Mi = NULL or Mi = 2cNULL)
                   SEND(1, (TINk, Mk));
               else
          ------------------------------
            send the one that is 2cNULL
          ------------------------------
                if (Mi = 2cNULL) SEND(1, (TINi, Mi);
                 else
                  if (Mk = 2cNULL) SEND(1, (TINk,Mk));
                   else
```

```
              ------------------------------
                send anyone, both are NULL
              ------------------------------
                        SEND(1, (TINi, Mi));

               [RECEIVE(i, (TINi, Mi)) || RECEIVE(k, (TINk, Mk))];
           endcase;
           repeat
             if ((Mi = NULL) or (Mi = 2cNULL)) and (TINi <= TOUT)
               RECEIVE(i, (TINi, Mi)
              else
             if ((Mk = NULL) or (Mk = 2cNULL)) and (TINk <= TOUT)
               RECEIVE(k, (TINk, Mk)
              else
             if (Mi = real) and (Ti = TOUT) SEND(1, (TINi, Mi))
               RECEIVE(i, (TINi, Mi);
              else
             if (Mk = real) and (Tk = TOUT) SEND(1, (TINk, Mk))
               RECEIVE(k, (TINk, Mk);
           until (TIN of message received > TOUT)
 forever;
}
```

If the LP is in a cycle, then it receives messages in both input links and then tests for three conditions[2]:

1. $TIN_i < TIN_k$. A merge process is the only process capable of creating a 2cNULL message. If the message received from LP_i is a NULL message, then a 2cNULL is sent. The encoded gray-time is equal to the timestamp of the message received from LP_k. This is a forecasted

[2]See Figure 3.8 for the description of notation

time, whose purpose is to advance the LP's local clock. A request to RECEIVE a message from LP_i is issued.

When a 2cNULL message arrives from LP_i, if merge process j owns the message and the gray-time is equal to the timestamp of M_k, that is TIN_k, then the merge process is able to advance the flow of global physical time by sending message M_k without further delay. The merge process requests a RECEIVE on both input ports.

If message M_i does not belong to any of the above cases, then the message with the smallest timestamp is transmitted and a request to RECEIVE from LP_i is scheduled.

2. $TIN_i > TIN_k$. Operates in a symmetric fashion to $TIN_i < TIN_k$.

3. $TIN_i = TIN_k$. When the timestamps of both messages are the same and they are real messages, one message is sent after the other.

   Real messages have precedence over NULL and 2cNULL messages. Because the timestamps are the same, the real message is sent.

   2cNULL messages have higher priority over NULL messages. This is due to the fact that a 2cNULL message will help to reduce the transmission of NULL messages. When there is a 2cNULL message and a NULL message, the 2cNULL is transmitted.

   When both input messages are NULL messages, then only one of them

is transmitted. For all of these cases a request to RECEIVE on both input ports is scheduled.

After the first two messages are received and one or both of them are sent, the algorithm compares the timestamps of incoming messages against TOUT, the timestamp of the last message sent. If the incoming message is 2cNULL and its timestamp is less than or equal to the last message transmitted, then we can receive a new message. But, if the received message is real and it has the same timestamp as the last message transmitted, then this message is sent and a RECEIVE for another message is scheduled.

4.5 Delay Process

The only change in the delay process is in the INPUT operation (see Figure 4.6).

When the delay process receives a 2cNULL message a new black time is computed. If this time is greater or equal than the gray time, the 2cNULL message is no longer needed and a NULL message is inserted in the local event list. If the new black time is less than the gray time, then the 2cNULL message is inserted instead.

```
INPUT ::
 {
  RECEIVE(k, (TIN, m));
  if (m = NULL) then insert(TIN + service_time, NULL)
   else
  if (m = 2cNULL) then {
      new_black_time := black_time + service_time;
      if (new_black_time >= gray_time) insert(new_black_time, NULL)
       else insert(new_black_time, m, gray_time, owner);
   } else
    if (predicted_time < TIN) {
       insert(TIN + service_time , m);
       predicted_time := TIN + service_time;
    } else {
       insert(predicted_time + service_time, m);
       predicted_time := predicted_time + service_time;
    }
 }
```

Figure 4.6: New algorithm for INPUT operation

4.6 Correctness of the Algorithm

We present an informal discussion of the correctness of the 2cNULL algorithm. We will discuss the changes required in a merge process since this is the only process that advances its clock in a different way than in Chandy and Misra's algorithm.

For non-feedback topologies the 2cNULL algorithm can be used. If the non-feedback topology contains a merge process, then this merge process will send a 2cNULL message at a certain time during the simulation. However,

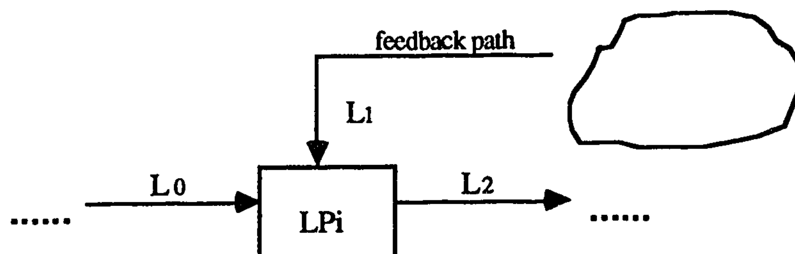


Figure 4.7: A feedback topology

because the topology does not have feedback, the 2cNULL will never return to its owner. The overhead produced by the 2cNULL is negligible, since we only add, as additional information, the owner and the gray-time. There is basically no difference, if the 2cNULL algorithm or Chandy and Misra's algorithm is used for non-feedback topologies.

Suppose we have a feedback topology with one cycle, one merge process, and a single source process. LP_i is the merge process connected to the feedback path. Consider a certain point in time up to which LP_i has not sent a 2cNULL message. We assume that messages received from links L_0 and L_1 arrive with monotonically increasing timestamps (see Figure 4.7).

One of the input ports receives a message, (τ_1, M_1), from L_0 and the other input port receives a message (t_1, m_1) from L_1. Notice that the contents of M_1 and m_1 may correspond to a real message or to a NULL message. Depending on the types of messages received on L_0 and L_1 we have to consider the cases in Table 4.1.

Case	L_0	L_1
1	Real	Real
2	Real	NULL
3	NULL	Real
4	NULL	NULL

Table 4.1: Cases to consider for L_0 and L_1

Case 1. $L_0 = real$ and $L_1 = real.$

In this case the 2cNULL algorithm operates as the original Chandy and Misra algorithm.

Case 2. $L_0 = real$ and $L_1 = NULL.$

Depending on the values of τ_1 and t_1 we have three different cases.

1. Case 2.1 ($t_1 > \tau_1$).

   If $t_1 > \tau_1$ then message (τ_1, M_1) is transmitted. LP_i requests another message from L_0. The algorithm operates as the Chandy and Misra algorithm.

2. Case 2.2. ($t_1 = \tau_1$).

   Because both timestamps have the same value, the real message is transmitted and the NULL message discarded. Two new messages are RECEIVED on links L_0 and L_1—the algorithm operates like Chandy and Misra's algorithm.

3. Case 2.3 ($t_1 < \tau_1$).

In this case, we send a 2cNULL message with gray-time = τ_1 and owner = LP_i, that is, (t_1, 2cNULL, τ_1, LP_i). Up to this point in time, we are sure that histories on all links are chronological according to [CM79], since thus far we have followed their algorithm. We want to prove that if the 2cNULL message $(t_1, 2cNULL, \tau_1, LP_i)$ returns to LP_i on link L_1 and the message on link L_0 is still (τ_1, M_1), then we can advance LP_i's clock value to τ_1, and LP_i will not receive a non-2cNULL message with t-value $< \tau_1$. If LP_i receives messages with t-value $< \tau_1$, these messages will be non-real messages only (NULLs or 2cNULLs). The conditions required to advance LP_i's clock value to τ_1 are specified as:

$$(\tau_1, M_1) \ \ on \ \ L_0 \tag{4.1}$$

$$(t_1', 2cNULL, \tau_1, LP_i) \ \ on \ \ L_1 \tag{4.2}$$

In (4.2) $t_1' > t_1$. t_1 is the black-time of the first 2cNULL sent by LP_i. The 2cNULL has passed through one or more delay processes and it has increased its black-time to a value of t_1'.

Consider what happens after (t_1, 2cNULL, τ_1, LP_i) is sent, link L_0 is closed and messages can only arrive on link L_1. LP_i can receive on link L_1 three kinds of messages: a real message, a NULL

message or a 2cNULL message.

(a) A Real message.

This situation is covered by cases 1 and 3.

(b) A NULL message.

This situation is covered by cases 2 and 4.

(c) A 2cNULL message.

We have two situations for this case depending on the conditions (4.1) and (4.2).

i. Conditions (4.1) and (4.2) are true.

When conditions (4.1) and (4.2) hold, we can advance LP_i's clock value to τ_1. (t_1, 2cNULL, τ_1, LP_i) is the first 2cNULL sent and has returned to LP_i. LP_i can safely transmit (τ_1, M_1) and two new messages are RECEIVED on links L_0 and L_1. If the first 2cNULL message returns to LP_i it is impossible that there exists a non-2cNULL message with t-value $< \tau_1$ in the cycle—since we assume there is only one merge process and one source process.

ii. Conditions (4.1) or (4.2) is false.

During the time (t_1, 2cNULL, LP_i, τ_1) has been travelling in the cycle, L_0 has been closed and LP_i had to process NULL or real messages on link L_1. If NULL mes-

sages were processed, LP_i transmitted them as 2cNULL messages. If (τ_1, M_1) was transmitted, then any 2cNULL messages with black-time $= \tau_1$ arriving on link L_1 will not satisfy condition (4.1). We have to flush out these messages by issuing RECEIVE commands on input link L_1. If the 2cNULL messages increase their black-time up to or beyond their gray-time, these 2cNULL messages will be converted into NULL messages. For these "new" NULL messages, their timestamps are $\geq \tau_1$, and the condition that LP_1 will not receive a non-2cNULL message with t-value $< \tau_1$ still holds.

Case 3. $L_0 = NULL$ and $L_1 = real$.

This case is different from case 2. We have (τ_1, NULL) on link L_0 and (t_1, m_1) on link L_1. We will send the following 2cNULL message: (τ_1, 2cNULL, t_1, LP_i). Link L_1 is closed and messages will arrive on link L_0 only. Because we have only one cycle in the topology, all 2cNULL messages transmitted will have their timestamp beyond t_1 when arriving at link L_1. In this type of situation, the 2cNULL message never returns to LP_i.

Case 4. $L_0 = NULL$ and $L_1 = NULL$.

Depending on the values of τ_1 and t_1 we have to consider three cases.

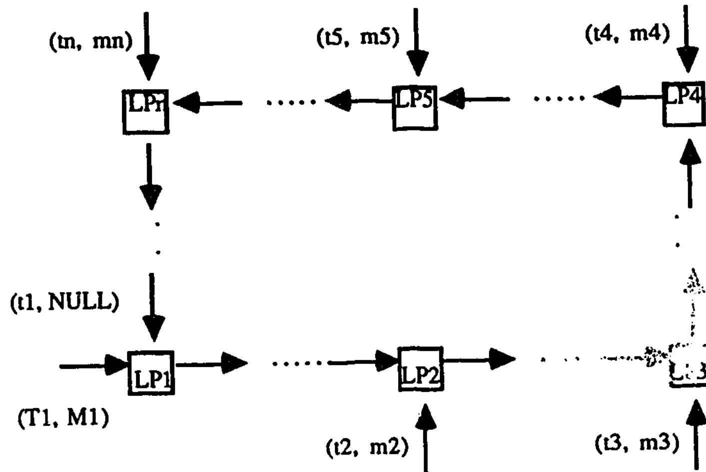


Figure 4.8: A feedback topology with multiple cycles

1. $\tau_1 < t_1$.

   We have the same situation as for case 3 explained before with the exception that the message on link L_1 is NULL.

2. $\tau_1 = t_1$.

   Since the timestamps are equal, only one of the NULL messages is sent. We operate as in the Chandy and Misra algorithm.

3. $\tau_1 > t_1$.

   The operation performed for this case is like in 2.3 except that we have a NULL message on link L_0.

What does happen if there are more cycles in the topology? Consider the topology in Figure 4.8.

LP_1 has two input messages: (t_1, NULL) and (τ_1, M_1). Suppose that $t_1 < \tau_1$. LP_1 sends a 2cNULL message: (t_1, 2cNULL, τ_1, LP_1). When the 2cNULL is passing through the cycles, at some point in time, it will have to enter a merge process LP_i ($i = 2, 3, \ldots, n$). Before relaying the 2cNULL, LP_i has to compare it to the message sitting on the other input link.

$$t'_1 < t_i \tag{4.3}$$

$$\tau_1 < t_i \tag{4.4}$$

If conditions (4.3) and (4.4) are true then, the 2cNULL can be relayed by LP_i. t_i is the timestamp of the other input message that the merge process has received. Condition (4.3) states that we are sending the message with the smallest timestamp. In addition, condition (4.4) forces the algorithm not to violate causality constraints. If condition (4.4) is false, it means that LP_i will propose a new 2cNULL message: (t'_1, 2cNULL, t_i, LP_i). If the first 2cNULL sent by LP_1 returns, and conditions (4.1) and (4.2) hold, then we can advance LP_1's clock value to τ_1. No LP_i in any cycle had a better proposal for the black-time.

Chapter 5

Implementation

5.1 Introduction

This chapter presents the implementation details of the distributed simulator built to measure the performance of the algorithms presented in chapters 3 and 4. Section 5.2 introduces LANSF, a simulation package used for the implementation of the distributed simulator. Sections 5.3 to 5.6 explain the detailed operation of each of the logical processes implemented as LANSF processes.

5.2 LANSF: A Simulation Package

The simulation package LANSF (Local Area Network Simulation Facility) was developed at the University of Alberta by Gburzyński and Rudnicki [GR88c]. The original idea behind LANSF was to have a tool for investigating MAC-level protocols for local area networks. LANSF can be used to "model most physical systems in which communication is the most critical issue" [GR88c]. Furthermore, LANSF has evolved and now can be used to describe the behavior of different systems: long-haul networks, distributed computer architectures, bus-type networks, protocols and distributed data bases [GR88d, GR87, GR88b].

In LANSF, a physical system is described as a collection of *processes.* These processes communicate with each other through the use of *messages* and *signals.* A logical process (LP) is described by a set of *local processes.* Local processes can communicate with each other and with external processes. The local processes perform different functions: reception of a message, transmission of a message, computations, etc. For example, the logical system for the car wash in Figure 2.3 can be described by the set of processes depicted in Figure 5.1.

Process A performs the function of the source process; it has a local process called *transmitter* that generates messages (cars) according to a certain distribution. Process B simulates the server process (car wash) and has two

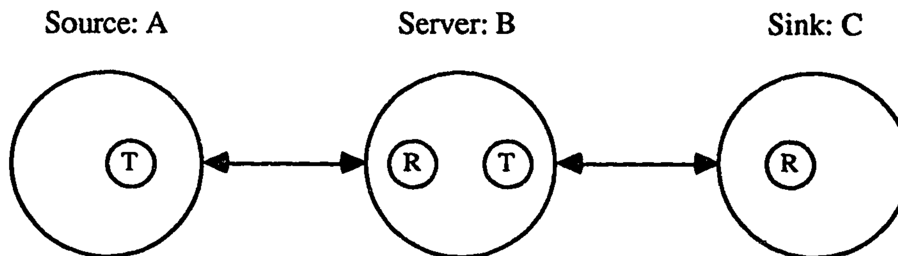


Figure 5.1: A set of LANSF processes

local processes, a *transmitter* (T) and a *receiver* (R). The [illegible] process in B receives the messages (cars) sent by process A, these messages represent events that are simulated, and then the transmitter sends the appropriate messages to process C. Process C simulates the sink process. It only has a *receiver* process that inputs messages from the transmitter in process B.

Communication between receivers and transmitters is synchronized by the use of signals. When a transmitter needs to send a message to a receiver it issues a signal. The receiver gets this signal and replies back with another signal to the transmitter. The transmitter accepts this last signal and then transmits the message safely. Our goal is to model a logical system as a set of cooperating LANSF processes. This is the way Hoare's synchronous protocol [Hoa78] is implemented under LANSF.

An *input data file* is used to define the network configuration of a logical system. A program in C language has to be implemented to describe the operation of the processes. LANSF processes can accurately model the logical

processes with as much detail as required.

A number of tools is provided to express a protocol and test different conditions in the communication environment. The system reads the input data file definition and creates the required processes. The processes start generating messages and simulating the behavior defined in their code. A global clock keeps track of the simulated time until the termination conditions are met.

A report containing statistics of the simulated topology is output at the end of the simulation run.

An extensive description of the LANSF system can be found in [GR88b].

5.3 Delay Process

A delay process is composed of three local processes: a *receiver* process, a *controller* process and a *transmitter* process (see Figure 5.2).

The controller process performs a loop verifying which one of the three conditions defined in section 3.6.1 of Chapter 3 is true:

- $TIN < TOUT$. When the delay process has to perform an input operation, the controller process notifies the receiver process with a signal. The receiver process requests a message from the neighbor[1] process by

[1] A neighbor process is the process from where a receiver process receives messages from, or a transmitter process sends messages to.

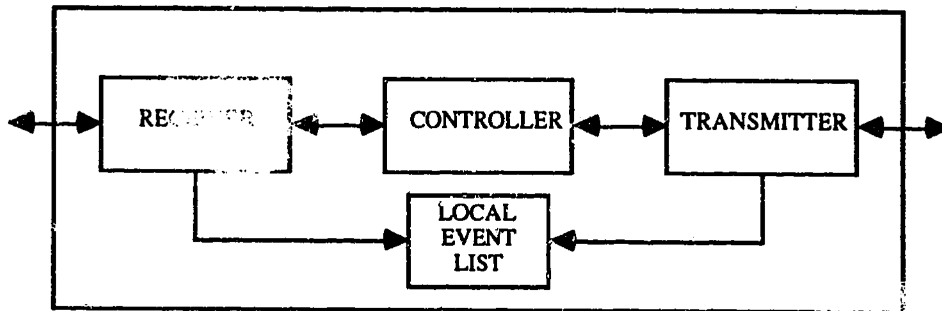


Figure 5.2: Local processes of delay process

issuing a signal, too. When the message arrives, the local event list is updated and a signal is sent to the controller. Upon reception of t signal, the controller checks again which condition is true and signals the respective process(es).

- $TIN > TOUT$. This condition states that an output operation is required. The controller process wakes up the transmitter process by sending a signal. The transmitter process updates the local event list according to the algorithm of Figure 3.5 in Chapter 3, and signals the next neighbor process that a message is ready to be sent. The transmitter waits until receiving a signal from a neighbor process before sending a message. If the transmitter already received a signal from a neighbor process, the message is sent immediately. After the message

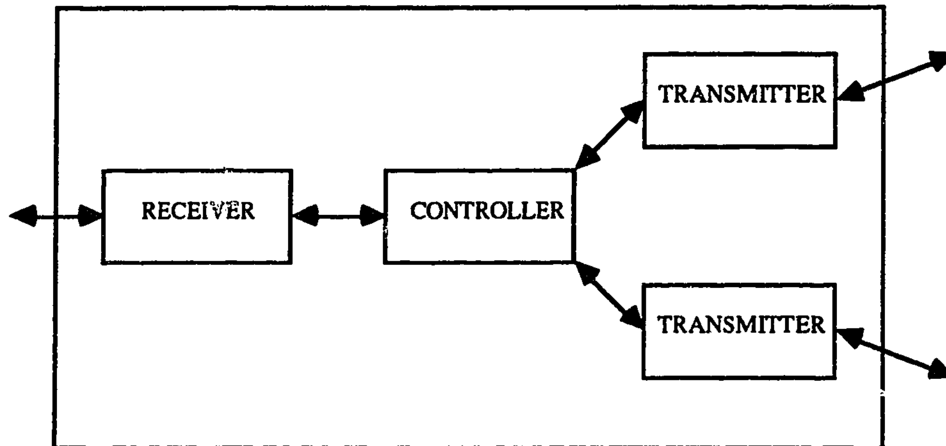


Figure 5.3: Local processes of branch process

is sent, the transmitter issues a signal to the controller.

- $TIN = TOUT$. When input and output operations are needed, the two scenarios described above occur. The controller waits until the two signals from the receiver and transmitter arrive, in order to decide which condition is true, and sends a signal to the respective process(es).

5.4 Branch Process

A branch process has four internal processes: a *receiver*, a *controller* and two *transmitters* (see Figure 5.3).

- **RECEIVER.** In order to receive a message, the receiver sends a signal to a neighbor process. When the message is received, the receiver issues a signal to the controller to notify that a message is ready to be transmitted. The receiver waits until the controller acknowledges with a signal that the message has been sent and the receiver can request to RECEIVE another message.

- **CONTROLLER.** The controller randomly selects one of the transmitters. The selected transmitter is notified via a signal, that should send the message. The other transmitter(s) receives a signal requesting to send a NULL message(s). The controller waits for n number of signals, where n is the total number of transmitters. When all signals arrive, the controller signals the receiver to continue receiving messages.

- **TRANSMITTERS.** When the transmitter receives a signal from the controller, it sends a signal to a neighbor process to indicate that a message is ready to SEND. When the transmitter receives a signal acknowledging that the neighbor process is ready, it sends the message. After this operation, it notifies the controller with a signal.

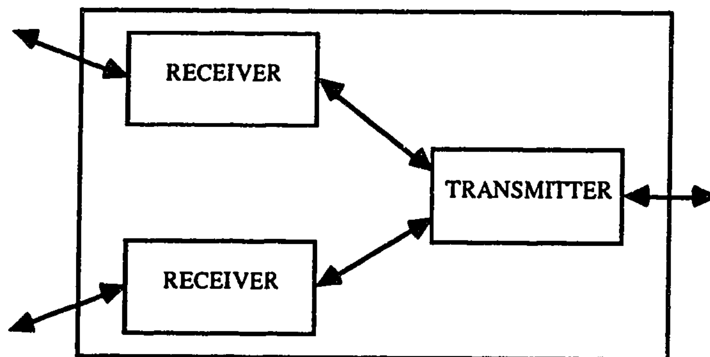


Figure 5.4: Local processes of merge process

5.5 Merge Process

A merge process is composed of two *receivers* and one *transmitter* (see Figure 5.4).

Once a receiver has accepted a message, it signals the transmitter that a new message has arrived. The transmitter maintains a count of the number of signals that it has received from the receivers. This is needed because the merge process has to compare the timestamps of all input messages. When the number of signals received by the transmitter matches the number of receivers, the transmitter follows the algorithm described in Figure 3.9 to decide which message to SEND. Depending on the situation, the transmitter sends one of the messages and then issues a signal to one or more of the receivers. The receivers wait for this signal. When the signal arrives, they

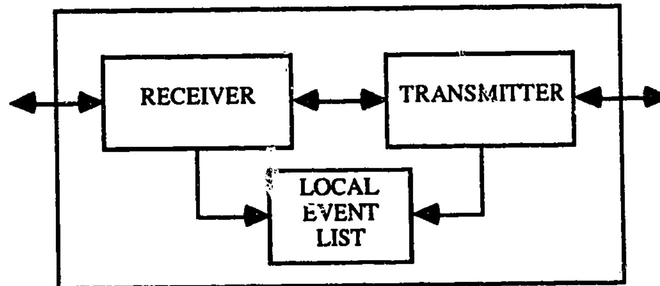


Figure 5.5: Local processes of buffer process

can request to receive more messages.

5.6 Buffer Process

A buffer process contains a receiver process, a transmitter process and a local event list (see Figure 5.5).

Buffer processes are optional and they are located between two basic LPs to help improve the operation of the algorithm. The operation of the buffer process is similar to the classical consumer/producer buffer [BS88].

Every time a message arrives at the receiver, the new message is added to the local event list. The receiver will request to RECEIVE a message only if there are buffers available, otherwise the receiver will be blocked until one entry in the local event list becomes available. Also, the receiver will signal

the transmitter when a message arrives, so the transmitter would be able to SEND the message. On the other hand, the transmitter may receive a request from a neighbor process to SEND a message. If the local event list is empty, then the transmitter will have to wait until a message arrives. Every time the transmitter deletes a message from the local event list, a signal is sent to the receiver because the receiver may be blocked.

Chapter 6

Experimental Results

6.1 Introduction

In this chapter we present the results obtained from the evaluation of the Chandy and Misra approach and the 2cNULL distributed algorithm. Section 6.2 describes the experimental methodology used for the experiments. Section 6.3 explains the results generated from the simulations by showing different performance metrics such as speedup, degree of parallelism achieved, and null/real messages transmission ratio.

6.2 The Experiments

In this section we describe the set of experiments carried out to evaluate the performance of the distributed simulation strategies explained in Chapters 3 and 4. We took into consideration different factors: network topology, inter-arrival time of messages, and processing/communication cost ratio.

6.2.1 Network Topologies

The set of network topologies[1] consisted of 18 different cases:

- **Tandem Networks.** Five topologies were considered with different numbers of delay nodes: 1 up to five nodes (see Figure A.1 in Appendix A).

- **Feedforward Networks.** A feedforward network is a generalization of a tandem network; besides having delay nodes, it also has branch and merge nodes, but not directed cycles. We studied nine different topologies (see Figures A.2 to A.10 in Appendix A).

- **Feedback Networks.** In this network category four different topologies were considered: two general feedback topologies, a central server network and a cluster network (see Figures A.11 to A.14 in Appendix A).

[1]The examples were taken from [Lak87] and [RMM88].

6.2.2 IART and d/bm Ratio

We consider the following mix of inter-arrival times (IART) of messages from source processes, the processing time in a delay node (d), and the processing time in branch (b) and merge (m) nodes. We will call this quantity the d/bm *ratio*.

al times
50
150
3000

Ta l/bm ratios

In the first row of Table 6.1 6 corresponds to *physical delay.* For example, if $d = 6$ and a delay process receives its first real message, (5, m_1) at simulation time 100, it will output the message (11, m_1) at simulation time 106. We have a 1–1 correspondence between the delay time units and the simulator time units. Merge and branch nodes are needed to perform routing operations, but they incur a nominal delay when doing so. A delay of 2 units of time will be used for b and m; this delay does not have a counterpart in the physical system, it is the time a branch or a merge process requires to execute its internal code. We are assuming that the communication delay between processes is 0. Since some time must elapse before one message is sent after another, an inter-packet spacing of 2 units of time was used in

all the experiments. We will keep fixed the communication delay (c) in the simulator and vary the processing delay (d) in the physical system. We have three cases to consider:

$$d \approx c$$

$$d > c$$

$$d >> c$$

Previous studies [Lak87, RMM88] have not taken into consideration that one of these three scenarios is likely to happen in any distributed system.

6.2.3 Performance Metrics

The experiments were carried out on a MIPS M/1000 machine running UMIPS-BSD, and the algorithms were implemented using the simulation package LANSF [GR88c]. Each experiment simulated the transmission of 1000 messages.

For each of the topologies several trial runs were performed for each of the combinations of d/bm ratio and IART. Different seeds for the random number generator (for the branch processes) were used and the results were obtained by averaging. To give accurate results, 95% confidence intervals [MS88] were computed for each one of the performance metrics: mean speedup ($\bar{S}$), mean

degree of parallelism ($\bar{D}$) and mean NULL/real transmission ratio ($\bar{N}$):

$$\bar{S} \pm 1.96\sqrt{\frac{s^2}{n}}$$

$$\bar{D} \pm 1.96\sqrt{\frac{s^2}{n}}$$

$$\bar{N} \pm 1.96\sqrt{\frac{s^2}{n}}$$

where s^2 is the sample variance obtained from the simulations for the respective performance metric and n is the total number of independent simulation runs. Because the data generated by the experiments is large, we are including only the worst and best cases for the confidence intervals. By inspecting tables 6.2, 6.3, and 6.4, we can verify that most of the experimental data is in the 95% accuracy range.

Now, we define each of the performance metrics of interest.

1. **Speedup.** It is defined as,

$$SP = \frac{T_1}{T_N}$$

where T_1 and T_N are the execution times for one and N processors. The execution time for one processor corresponds to the execution time of the sequential simulation algorithm. We used the method proposed by Lakshmi [Lak87] to compute T_1. Suppose we know the number of messages transmitted by each node at the end of the simulation. If we

	$\bar{S}$		$\bar{N}$		$\bar{D}$	
Topology	*Best*	*Worst*	*Best*	*Worst*	*Best*	*Worst*
Feedforward1	0.47	1.8	0.01	0.19	1.4	4.8
Feedforward2	0.38	3.56	0.05	0.42	1.68	5.52
Feedforward3	0.42	2.56	0.20	1.40	0.79	4.42
Feedforward4	0.01	0.15	0.01	0.07	0.04	2.45
Feedforward5	0.01	0.59	0.02	0.35	0.35	5.74
Feedforward6	0.0002	0.13	0.04	0.27	0.12	5.71
Feedforward7	0.06	2.25	0.09	8.14	0.09	6.42
Feedforward8	0.01	0.53	0.01	0.18	0.23	6.49
Feedforward9	0.002	1.13	0.15	0.54	0.13	5.50

Table 6.2: Ranges for the confidence intervals of feedforward networks in % of the mean value (Chandy and Misra algorithm)

multiply the number of real messages by the node's processing delay, for every node in the topology, and add these numbers, we will compute the value for T_1. $\bar{S}$ is the mean SP computed for three simulation runs.

2. **Degree of Parallelism.** It is defined as the fraction of time n processors are busy.

$$DP = \frac{time\ n\ processors\ are\ busy}{simulation\ time}$$

This metric has a direct relationship with speedup. If the speedup is good, then we will observe a good degree of parallelism too. $\bar{D}$ is the mean DP computed for three simulation runs.

3. **NULL/Real Transmission Ratio.** It is defined as,

	$\bar{S}$		$\bar{N}$		$\bar{D}$	
Topology	*Best*	*Worst*	*Best*	*Worst*	*Best*	*Worst*
Feedback1	0.24	2.4	0.20	6.42	0.30	6.97
Feedback2	0.06	3.94	0.007	0.82	0.07	5.77
Cluster	0.49	6.40	0.03	6.49	0.29	5.97
Central Server	0.18	2.36	0.07	6.19	0.18	7.38

Table 6.3: Ranges for the confidence intervals of feedback networks in % of the mean value (Chandy and Misra algorithm)

	$\bar{S}$		$\bar{N}$		$\bar{D}$	
Topology	*Best*	*Worst*	*Best*	*Worst*	*Best*	*Worst*
Feedback1	0.53	3.71	0.29	3.7	2.09	6.12
Feedback2	0.02	3.94	0.10	0.69	0.7	5.7
Cluster	0.64	2.48	0.2	4.7	0.76	7.3
Central Server	0.05	3.87	0.27	2.49	0.20	5.06

Table 6.4: Ranges for the confidence intervals of feedback networks in % of the mean value (2cNULL algorithm)

$$NRTR = \frac{number\ of\ NULL\ messages\ transmitted}{number\ of\ real\ messages\ transmitted}$$

and measures the overhead caused by NULL messages in the distributed algorithms. $\bar{N}$ is the mean $NRTR$ computed for three simulation runs.

6.3 The Results

The results obtained by running the distributed simulator with the mix of IART and d/bm ratios of Table 6.1 are presented here. We start with the results for tandem networks, followed by feedforward topologies and conclude with the feedback topologies.

The reader is referred to Appendix B that contains the performance charts for the topologies mentioned above. The performance charts showing the degree of parallelism (DP) contain different inter-arrival times and d/bm ratios. The following table explains the meaning of the notation found in the x axis of the DP charts.

symbol	IART	d/bm
Δa	1	
Δb	7	6/2
Δc	50	
Δd	7	
Δe	21	20/2
Δf	150	
Δg	200	
Δh	601	600/2
Δi	3000	

Table 6.5: Description of notation for DP charts

6.3.1 Tandem Networks

The results for the speedup of tandem networks are shown in Figures B.1, B.2, and B.3. The three Figures show a linear speedup except for the lowest inter-arrival time. When the inter-arrival time has a minimum value, in this case 1, 7 and 200, the speedup has a sublinear behavior. Looking at the three d/bm ratios in Figures B.1 to B.3 for the case of $\Delta = 1$,7 and 200, we observe that the speedup gained is about the same for the 6/2 and 20/2 ratios. The next ratio, 600/2, increases the speedup very little. As the IART increases, the speedup obtained by the topology is better, too. The ideal value for the speedup is equal to the number of delay nodes in the topology, but this value is achievable only for the 600/2 ratio (for $\Delta = 601$ and 3000).

Figures B.4 to B.8 show the degree of parallelism obtained for each tandem topology. The charts show that as the IART increases in each d/bm ratio, the parallelism achieved increases, yielding a better speedup. For example, in Figure B.8 when the d/bm ratio is 200/2 (Δg), the topology is only able to have 40% of the time four processors active. But, when the d/bm ratio increases to 600/2 or 3000/2, then the network operates all the five processors in parallel almost 100% of the time. The reader is encouraged to inspect the other performance charts and find that similar results are obtained for the other tandem networks. In all topologies, there is a lower bound at which, the degree of parallelism has a certain value, which can be

increased by increasing the value of the d/bm ratio.

The communication protocol has to follow the inter-packet spacing rules. That is, when a message is sent, two units of time have to elapse before sending the next message. In Figures B.4 to B.8 the percentage of idle time decreases as the IART increases in each of the d/bm ratios. Consider the performance chart of Figure B.6. When the d/bm ratio is 6/2 and the IART = 1 (Δa), 7% of the time the topology is idle. If we increase the IART to 7 (Δb), the topology is no longer idle; 45% of the time two processors are active and 55% of the time three processors work concurrently. A similar scenario happens for the other two ratios: by increasing the IART the topology is idle lesser time and is able to perform more operations in parallel. The inter-packet spacing time accounts for a big percentage of the time for this topology. As the number of nodes increases (Figures B.5 to B.8), the inter-packet spacing is distributed over more delay nodes and the topology is able to perform more concurrent operations.

Reed [RMM88] states that "in a tandem network all nodes are always active", but the DP charts show that sometimes the nodes are idle as much as 30% of the time (see Figure B.4). By inspecting the DP charts for tandem networks we observe that if we increase the IART, then more nodes become active during the simulation.

Lakshmi [Lak87] found that "an ideal speedup factor of N seems to be achievable", but charts B.1 to B.3 show that the speedup is dependent on the

IART and the d/bm ratio. The only case when we obtained a speedup factor of N was for the 600/2 ratio. In the experiments performed by Lakshmi [Lak87] the value used for the d/bm ratio was 3/1. We used a multiple of 3/1, that is, 6/2, and found that a factor of $\approx$N/2 is possible for an IART of 1, and a factor of $\approx$0.9N for the IARTs of 7 and 50. The speedups obtained for 6/2 and Δ=1 were: 0.92, 1.16, 1.66, 2.16 and 2.66.; for 6/2 and $\Delta = 7$ or 50 were: 1.14, 1.97, 2.81, 3.70 and 4.55.

NULL messages produce a negligible overhead, since only one NULL message is sent at the beginning by each delay node during the operation of the algorithm. The 2cNULL algorithm could be used to simulate feedforward topologies. However, because there are no merge nodes in a tandem network, a 2cNULL message would never be sent. In this case, both algorithms have the same performance.

6.3.2 Feedforward Networks

In the topologies depicted in Figures A.2 to A.10 the probability associated with a branch process is 0.5 for each output link (all branch processes in feedforward networks have only 2 output links). Also, there is a buffer process between every basic LP with capacity for 10 messages. Nine different topologies were studied under this category and Figures B.9 to B.26 summarize the results.

The speedups obtained for the feedforward topologies are sublinear in the number of processors. The DP charts show that delay nodes never achieve full parallelism. For example, in some topologies (Figures B.18, B.20, B.24, B.26) all delay nodes never work in parallel for all the combinations of IART and d/bm ratios. The DP chart for topology Feedforward5 in Figure B.18 shows that a maximum of four processors—the topology has five processors— are active only 5% of the time; the results for topology Feedforward6 in Figure B.20 show that only 5 out of 6 processors are active less than 5% of the time. The maximum degree of parallelism achieved is 85% for the topology Feedforward1 (Figure B.10). In general, feedforward topologies have a better speedup as the inter-arrival time increases in each of the three d/bm ratios: 6/2, 20/2 and 600/2.

Feedforward networks have branch nodes and require the sending of NULL messages to avoid deadlock. Merge, branch and delay nodes have to process NULL messages with the result that the processing of real messages is delayed. Also, merge nodes have to wait for all input messages, before sending a message to the next neighboring process. This affects the performance of the algorithm and as a result, the speedup degrades.

Lakshmi [Lak87] found that "changes in the service rate does not have any effect on the turnaround time ". Unfortunately, she does not show the experiments performed to support this conclusion. In all the feedforward topologies simulated —except for the topology Feedforward3 —we found that

the d/bm ratio has a direct effect on the speedup: as the d/bm ratio goes up, the speedup goes up too. Also, because the speedup improves as the d/bm ratio increases, we are effectively reducing the overhead of the simulation algorithm.

The results from [illegible] [RMM88] are very close to what we found in our simulations. Reed a[illegible] performed experiments to see the effect of clustering several nodes in a processor: the simulation experiments concluded that the speedup decreases as more nodes were clustered in a single processor.

6.3.3 General Feedback Networks

The two general feedback topologies have very close performance. In the three d/bm ratios, the topologies behave different from the feedforward networks: the speedup decreases as the IART increases (see Figures B.27 and B.29). For some IARTs (50 and 150) the speedup decreases up to 80%. The reason for this decrease is that the number of NULL messages processed increases to huge numbers (see Figures B.43 and B.45). In Figure B.43 we notice that for Δc, the number of NULL messages was 108,436; in Figure B.45 the number of NULL messages increases even more for Δc: 206,807.

When the IART is high, we have a lot of NULL messages travelling in the network. Nodes keep processing NULL messages in order to advance their clocks. The nodes in the topology do not know that only NULL messages

are present and that merge processes could advance their clocks.

The 2cNULL algorithm was evaluated to verify that it reduces the overhead produced by the processing of NULL messages. Figures B.35 and B.37 show the increase in speedup gained using the 2cNULL algorithm. The number of NULL messages processed is reduced for the three d/bm ratios, especially when the IART is high: 50 (Δc), 150 (Δf), and 3000 (Δi). See Figures B.44 and B.46. In Figure B.44 Δc decreases the number of NULL messages transmitted from 108,436 to 17,047; Δf reduces the number from 31,844 to 17,047; and Δi reduces the number of NULL messages from 34,464 to 17,409.

The reduction in NULL messages yields a better degree of parallelism. Comparing Figures B.28 versus B.36 and B.30 versus B.38 we observe that the idle time spent during the simulation is reduced by 20% for some IARTs.

Even though the 2cNULL algorithm increases the speedup for some IARTs, the speedup gained is not spectacular. Figures B.36 and B.38 show that all delay nodes never work concurrently and that most of the time 1 or 2 nodes work in parallel during the simulation (topology Feedback1 has 4 delay nodes and topology feedback2 has 5 delay nodes).

The study performed by Reed [RMM88] concludes that the speedup obtained by the distributed algorithm is dependent on the population of real messages in the topology. As the IART increases, we have less real messages present in the system and the number of NULL messages increases

tremendously. The speedup is reduced because the network has to process a big number of NULL messages. This is congruent with the results we have obtained.

6.3.4 Cluster Network

The cluster network performed badly as the IART increased for all the d/bm ratios (see Figure B.31). As in the general feedback topologies, the number of NULL messages processed increased to a big number, more than 200,000 (see Figure B.49) for a simulation run of only 1000 messages. In fact, for the IART of 50, the network is idle 90% of the simulation time. The topology has 8 delay LPs, but the DP chart (see Figure B.32) shows that only a maximum of 5 LPs achieve full parallelism only 5% of the simulation time.

Looking at Figure B.32 we notice that as the IART increases, less processors are able to work concurrently. The same situation occurs with the 2cNULL algorithm in Figure B.40. However, the 2cNULL algorithm improves the speedup for all d/bm ratios when the IART is high. The speedup goes from 0.18 to 0.26 for Δc, from 0.36 to 0.44 for Δf, and from 0.96 to 1.04 for Δi. Also, the number of NULL messages is reduced for the following cases: Δc (from 247,690 to 160,675), Δf (from 219,571 to 166,232), and Δi (from 135,443 to 125,442).

In the three d/bm ratios, the 2cNULL algorithm performed the same as

the Chandy and Misra approach, for the first two IARTs. The reason for this behavior is explained by the actions performed by the delay process (see Figure 4.6). When a delay node receives a 2cNULL message, the black-time is increased. If the black-time $\geq$ gray-time, then the 2cNULL message becomes a NULL message. When the 2cNULL message has to pass through many nodes or the IART is low, then 2cNULL messages will be converted into NULL messages and the speedup obtained will be like in the Chandy and Misra technique.

Reed [RF87] found very similar results with this topology; his study shows that as the IART decreases, the number of NULL messages decreases significantly (see Figures B.49 and B.50). For example, for the 6/2 ratio in Figure B.49, the number of NULL messages goes from 247,690 (Δc) to 13,962 (Δa); for the 20/2 ratio the number of NULLs ranges between 219,571 (Δf) and 22,861 (Δd); and for the 600/2 ratio the number of NULL messages varies between 135,443 (Δi) and 13,153 (Δg). In addition, a big overhead is produced by NULL messages in order to avoid deadlock, causing the speedup to drop considerably (see Figures B.31 and B.39).

6.3.5 Central Server Network

This network category had a behavior close to that of the other feedback topologies. As the IART increases, the speedup obtained decreases rapidly

(see Figure B.33).

In Figure B.34 we can observe that the number of processors working in parallel decreases, as the IART increases. Most of the time one processor is active. When we have 2 processors working in parallel, the topology is idle between 50% and 60% for the 6/2 ratio, and 20% to 70% for the 20/2 ratio. For the 600/2 ratio the percentage of idle time is low, less than 5%, but only one processor is active most of the time ($\Delta = 21$).

The 2cNULL algorithm was effective in improving the speedup for Δc (from 0.18 to 0.38) and Δf (from 0.39 to 0.6). (See Figures B.33 and B.41). For these two cases, the 2cNULL algorithm reduced the percentage of idle time: from 90% to 79% for Δc, and from 68% to 51% for Δf. (see Figures B.34 and B.42) The number of NULL messages transmitted was also reduced for Δc (from 92,379 to 26139) and for Δf (from 80,425 to 26,493). (see Figures B.47 and B.48).

For the 2cNULL algorithm, this topology showed the problem we described before: for low IARTs, 2cNULL messages are converted into NULL messages, and the topology can not take advantage of the algorithm. Figure B.41 shows the speedup obtained with the 2cNULL algorithm. For low IARTs, the speedup obtained is the same as the speedup gained with the Chandy and Misra approach (see Figures B.33 and B.41).

In [RF87] Reed reports that as the population in the network increases, the speedup decreases. This result parallels with our findings. We have

shown that as the IART increases, the number of NULL messages increases too, and as a consequence, the speedup degrades. The same applies for the 2cNULL algorithm, except that we can have some improvements in the speedup gained.

6.4 Remarks on the Operation of the 2cNULL Algorithm

The following remarks apply to the feedback topologies we considered in our simulation experiments: general feedback networks, cluster network and central server network. There are two situations in which the 2cNULL algorithm switches to operate as the Chandy and Misra algorithm.

1. black-time > gray-time.

   We will use the diagram depicted in Figure 6.1.

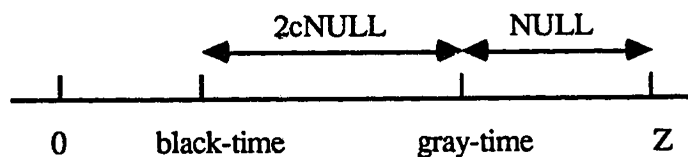


Figure 6.1: Relation between black-time and gray-time

   When we have a 2cNULL message, its t-value or black-time, increases as the message traverses the network. However, if the black-time increases

up to or beyond the gray-time value, the 2cNULL is converted into a NULL message. When this situation occurs, the 2cNULL algorithm does not help in reducing the number of NULL messages transmitted.

2. The other situation occurs when the algorithm tries to increase the LP_i's clock to the channel clock value of the link connected to the feedback path. Suppose we have a feedback topology (see Figure 6.2).

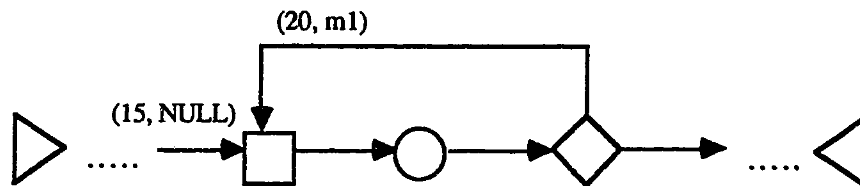


Figure 6.2: Increasing merge's clock value

The merge process in Figure 6.2 will send a 2cNULL with a gray-time of 20 because it wants to increase the merge's clock value up to 20. We notice that the channel clock value on the feedback path is 20; this means that any 2cNULL sent by the merge process will increase its black-time beyond 20. Because the gray-time is 20, the 2cNULL messages will be transformed into NULL messages.

Chapter 7

Conclusions

7.1 Summary of Results

In this thesis we have studied the design and implementation of a distributed simulation method for queueing networks. The method is known as the Chandy and Misra paradigm for distributed simulation. It consists of transforming a system of physical processes (PPs) into a network of logical processes (LPs). LPs perform different functions. There are five different types of processes that can model any physical system: source, delay, branch, merge and sink.

The core problems in distributed simulation are avoiding deadlock and ensuring that causality constraints are not violated. The Chandy and Misra approach uses NULL messages to avoid deadlock. LPs perform specific func-

tions to simulate correctly the physical system given, i.e. waiting until all input messages have arrived and selecting the message with the smallest timestamp. The network composed of LPs repeats three phases—selection, computation and I/O operations— until a termination time Z is reached.

The major drawback of the Chandy and Misra technique is the overhead produced by NULL messages. A new distributed algorithm is presented to overcome this problem. It introduces the novel concept of a two-colored NULL message. The new algorithm takes advantage of knowledge of global time by encoding additional information in a message. The two methods were implemented using the simulation package LANSF [GR88c]. Each of the five basic LPs was implemented as a LANSF process, which in turn, is composed of several local processes. The 2cNULL method changes the operation of only two basic LPs: merge and delay.

The performance evaluation study of the two implementations was carried out on a MIPS m/1000 machine running UMIPS-BSD. The set of test cases consisted of 18 network topologies. The performance metrics examined included: speedup, degree of parallelism achieved and null/real messages transmission ratio.

The simulations showed that the performance of the distributed simulation algorithms is sensitive to the structure of of the topology being simulated. This is consistent with the results reported by Wong [Won88], Reed [RMM88], Fujimoto [Fuj88] and Lakshmi [Lak87].

Tandem networks showed the best performance of all topologies considered. As the inter-arrival time (IART) increases, the speedup obtained by the topology is better. As a consequence, the degree of parallelism improves when the number of delay nodes increases in the topology. Also, the percentage of idle time decreases when the physical delay (d) increases in the d/bm ratio.

The speedup found in feedforward networks was not satisfactory. In all the experiments performed, all delay nodes in the topology never achieved full parallelism. A general behavior in these networks is that the speedup increases as the IART increases for the three d/bm ratios. The overhead produced by the processing of NULL messages is noticeable since the speedups obtained were sublinear in the number of processors.

In all feedback topologies (general feedback networks, cluster network and central server network) the speedup was poor. Feedback topologies have to process a tremendous number of NULL messages. When the IART increases, the speedup decreases precipitously (up to 80% or 90%), because the network is saturated with NULL messages. The 2cNULL algorithm was effective in improving the speedup for some cases. The new algorithm performs as well or better than the original Chandy and Misra approach. The improvement was considerably better for two cases: the cluster network improved the speedup 44% (from 0.18 to 0.26 for Δc) and the central server network improved the speedup in 111% (from 0.18 to 0.38 for Δc), and 65% (from 0.39 to 0.6 for

Δf).

The studies performed by Wong [Won88], Reed [RMM88], Fujimoto [Fuj88] and Lakshmi [Lak87] found that certain topologies are suitable for distributed simulation, with the exception of feedback topologies.

To conclude, we have demonstrated that the IART, processing delay and network topology affect the performance of the distributed simulation method and we took into account factors not presented in other works. Furthermore, we have presented a new algorithm that improves the performance of the distributed simulation method when the topology contains cycles.

7.2 Future Research

The future research directions of distributed simulation can be extended in a number of areas. Our work has considered a good number of test cases, but it is not exhaustive. There is a need to consider a wider variety of topologies.

The *mapping* operation described in the Introduction (see Figure 1.1) has to be considered with more detail in future work. We considered a 1–1 mapping between PPs and LPs. However, we could *cluster* more than one PP in a single LP. Heuristics that perform clustering decisions are needed and simulation is required to verify if the heuristics show improvements in the performance of the distributed simulation method.

Finally, we considered that LPs do not have a knowledge of the underlying

queueing domain. If a LP knows more about the topology it belongs to, it could make decisions that could improve the performance of the distributed simulation strategy used.

Bibliography

[Bil85] W. Biles. Statistical Considerations in Simulation on a Network of Microcomputers. In *Proceedings 1985 Winter Simulation Conference*, pages 388–393, 1985.

[Bry77] R. E. Bryant. Simulation of packet communication architecture computer systems. Technical Report TR-188, MIT, 1977.

[BS88] L. Bic and A. C. Shaw. *The Logical Design of Operating Systems.* Prentice Hall, 2nd edition, 1988.

[CB83] A. Chandak and J. C. Browne. Vectorization of Discrete Event Simulation. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 359–361, 1983.

[CHM79] K. M. Chandy, V. Holmes, and J. Misra. Distributed Simulation of Networks. *Computer Networks*, 3(2):105–113, Mar 1979.

[CM79] K. M. Chandy and J. Misra. Distributed Simulation : A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, Sep 1979.

[CM81] K. M. Chandy and J. Misra. Asynchronous Distributed Simulation Via a Sequence of Parallel Computations. *Communications of the ACM*, 24(11):198–206, Apr 1981.

[Com84] J. C. Comfort. The Simulation of a Master-Slave Event Set Processor. *Simulation*, 3(42):117–124, Mar 1984.

[DS80] E. W. Dijkstra and C. S. Scholten. Termination Detection for Diffusing Computations. *Information Processing Letters*, 11(1):1–4, Aug 1980.

[Fra77] W. R. Franta. *Process View of Simulation.* Operating and Programming Systems Series. Editorial Elsevier North-Holland, 1977.

[Fuj88] R. M. Fujimoto. Performance Measurments of Distributed Simulation Strategies. In *Distributed Simulation 1988*, pages 14–20. The Society for Computer Simulation, 1988.

[FWW84] M. A. Franklin, D. F. Wann, and K. F. Wong. Parallel Machines and Algorithms for Discrete-Event Simulation. In *Proceedings*

1984 International Conference on Parallel Processing, pages 449–458, Aug 1984.

[GR87] P. Gburzyński and P. Rudnicki. A Better-than-Token Protocol with Bounded Packet Delay Time for Ethernet-type LAN's. In *Symposium on the Simulation of Computer Networks.* ACM/IEEE, 1987. Colorado, Springs.

[GR88a] P. Gburzyński and P. Rudnicki. Asynchronous Distributed Simulation : Deadlock Avoidance with Two-Colored Null-Messages, 1988. Unpublished report.

[GR88b] P. Gburzyński and P. Rudnicki. Bounded Packet Delay Time Protocols on the CSMA/CD bus: Modeling in LANSF. In *19-th Annual Modeling and Simulation Conference*, 1988. Pittsburgh, PA.

[GR88c] P. Gburzyński and P. Rudnicki. LANSF - A Configurable System for Modeling Communication Networks. Technical Report TR 88-17, The University of Alberta, 1988.

[GR88d] P. Gburzyński and P. Rudnicki. Using Time to Synchronize a Token Ethernet. In *CIPS*, Nov 1988. Edmonton, Alberta.

[Hei88] P. Heidelberger. Discrete Event Simulations and Parallel Processing: Statistical Properties. *SIAM Journal of Statistical Computing*, 9(6):1114–1132, Nov 1988.

[Hen83] J. O. Henriksen. Event List Management – A Tutorial. In *Winter Simulation Conference Proceedings*, pages 543–551, 1983.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug 1978.

[Hol78] V. Holmes. *Parallel Algorithms on Multiple Processor Architectures*. PhD thesis, University of Texas, Austin, Texas, 1978.

[Jef85] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 1(1):404–425, Apr 1985.

[Jon86] D. W. Jones. An Empirical Comparison of Priority-Queue and Event-Set Implementations. *Communications of the ACM*, 29(4):300–311, Apr 1986.

[JS85] D. R. Jefferson and H. A. Sowizral. Fast Concurrent Simulation Using the Time Warp Mechanism. In *Proceedings of the SCS Distributed Simulation Conference*, 1985.

[Kau88] F. J. Kaudel. A Literature Survey on Distributed Discrete Event Simulation. *Simuletter*, 18(2):11–21, Jun 1988.

[Kum86] D. Kumar. Simulating Feedforward Systems Using a Network of Processors. In *Proceedings 19th Annual Simulation Symposium*, pages 127–144, 1986.

[Lak87] M. S. Lakshmi. A Study and Analysis of Performance of Distributed Simulation. Technical Report TR-87-32, The University of Texas at Austin, Aug 1987.

[LMS83] S. Lavenberg, R. Muntz, and B. Samadi. Performance Analysis of Distributed Simulation Using Roll Back. In *Proceedings of the Fifth Annual Conference on Performance*, 1983.

[Lub88] B. D. Lubachevsky. Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks. In *Proceedings 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, volume 16, pages 12–21, May 1988.

[Mis86] J. Misra. Distributed Discrete-Event Simulation. *Communications of the ACM*, 18(1):39–65, Mar 1986.

[MP86] A. Miller and J. Pare. Managing Simulation Projects. In *Proceedings 19th Annual Simulation Symposium*, pages 43–57, 1986.

[illegible] W. M. McCormack and R. G. Sargent. Analysis of Future Event Set Algorithms for Discrete Event Simulation. *Communications of the ACM*, 24(12):801–812, Dec 1981.

[MS88] W. Mendenhall and T. Sincich. *Statistics for the Engineering and Computer Sciences.* Dellen Publishing Co., 2nd edition, 1988.

[Nan84] R. E. Nance. Model Development Revisited. In *Proceedings 17th Winter Simulation Confrence*, pages 75–80, 1984.

[PMW80] J. K. Peacock, E. Manning, and J. W. Wong. Synchronization of Distributed Simulation Using Broadcast Algorithms. *Computer Networks*, 4(1):3–10, 1980.

[PWM79] J. K. Peacock, J. W. Wong, and E. G. Manning. Distributed Simulation Using a Network of Processors. *Computer Networks*, 3(1):44–56, 1979.

[Ree85] D. A. Reed. Parallel Discrete Event Simulation: a Case Study. In *Proceedings 18th Annual Simulation Symposium*, 1985.

[RF87] D. A. Reed and R.M. Fujimoto. *Multicomputer Networks: Message-Based Parallel Processing.* Series in Scientific Computation. The MIT Press, 1987.

[RMM88] D. A. Reed, A. D. Malony, and B. D. McCredie. Parallel Discrete Event Simulation Using Shared Memory. *IEEE Transactions on Software Engineering*, 14(4):541–553, Apr 1988.

[Sam85] B. Samadi. *Distributed Simulation: Performance and Analysis.* PhD thesis, University of California, Los Angeles, California, 1985.

[Won88] K. L. Wong. Distributed Simulation of Performance Petri Nets. Master's thesis, University of Alberta, Edmonton, Aberta, Canada, Nov 1988.

Appendix A

Network Topologies

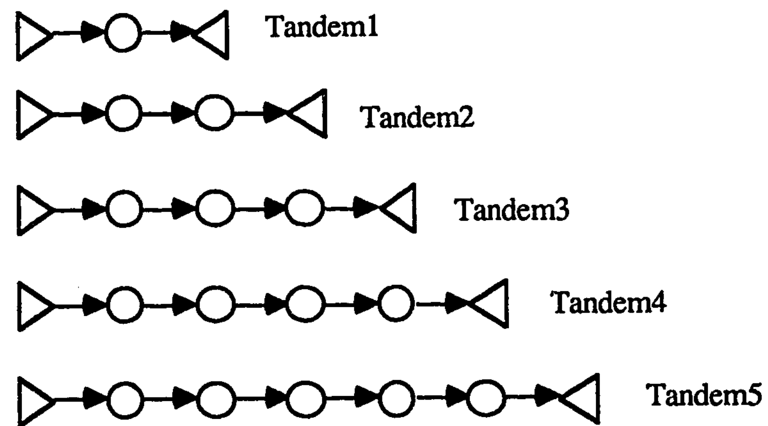


Figure A.1: Tandem Networks

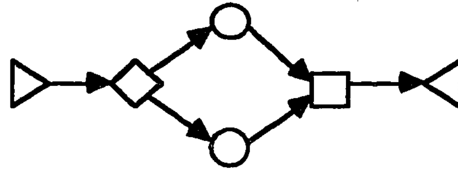


Figure A.2: Network Topology Feedforward1

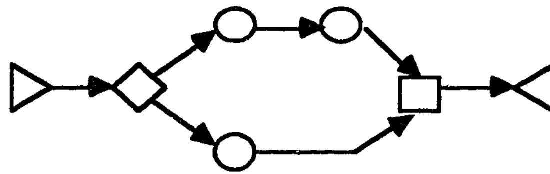


Figure A.3: Network Topology Feedforward2

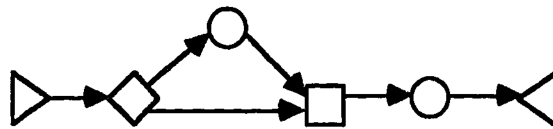


Figure A.4: Network Topology Feedforward3

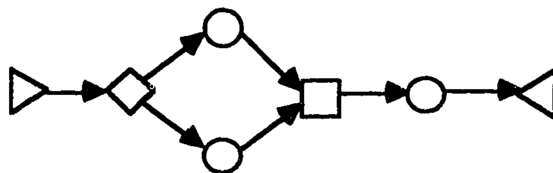


Figure A.5: Network Topology Feedforward4

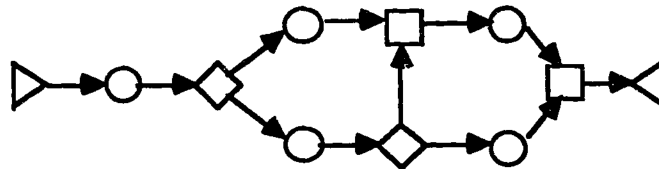


Figure A.6: Network Topology Feedforward5

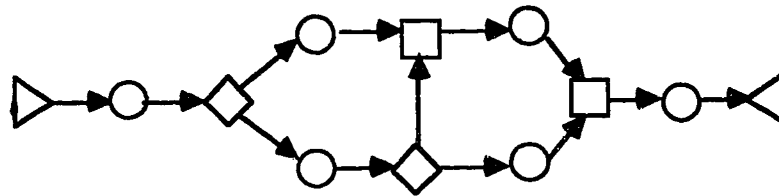


Figure A.7: Network Topology Feedforward6

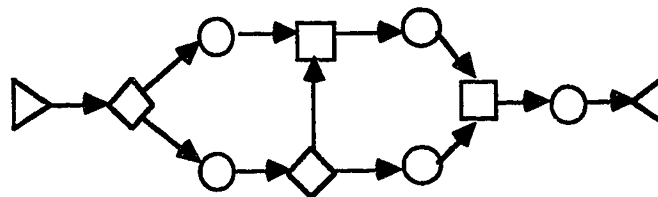


Figure A.8: Network Topology Feedforward7

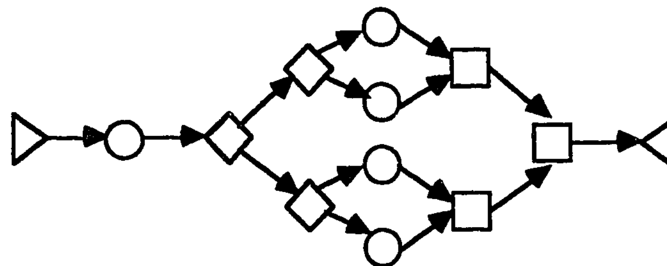


Figure A.9: Network Topology Feedforward8

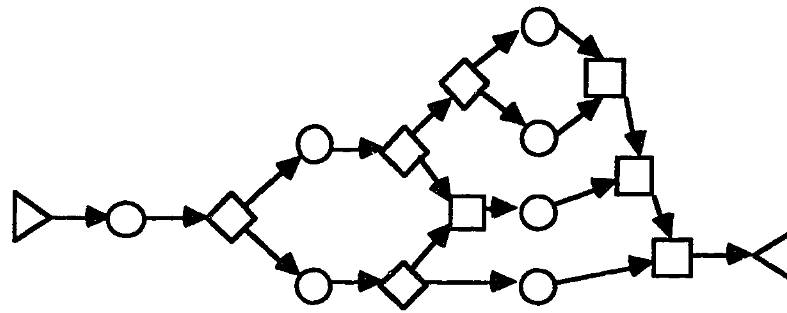


Figure A.10: Network Topology Feedforward9

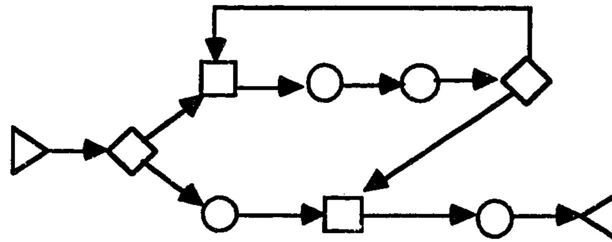


Figure A.11: Network Topology Feedback1

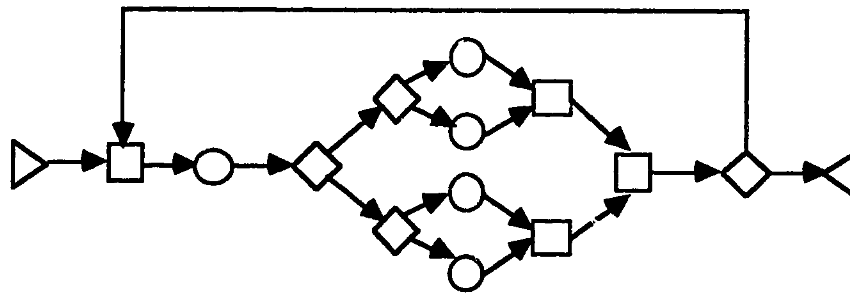


Figure A.12: Network Topology Feedback2

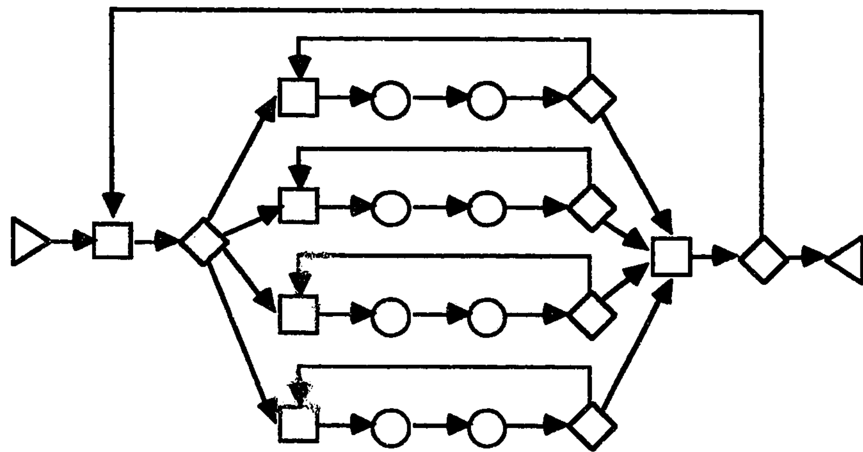


Figure A.13: Network Topology Cluster

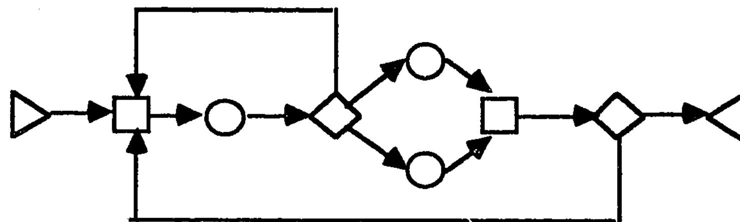


Figure A.14: Network Topology Central Server

Appendix B

Performance Charts

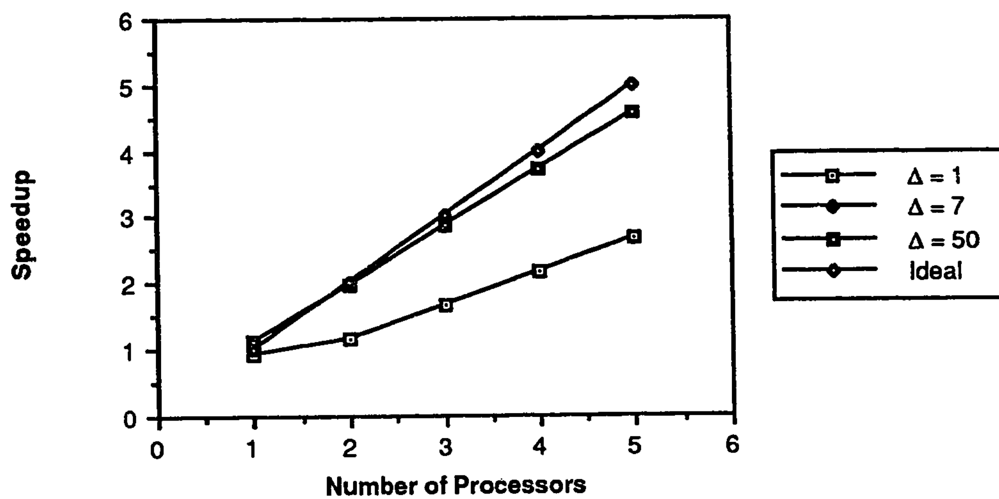


Figure B.1: Speedup Tandem Networks (6/2)

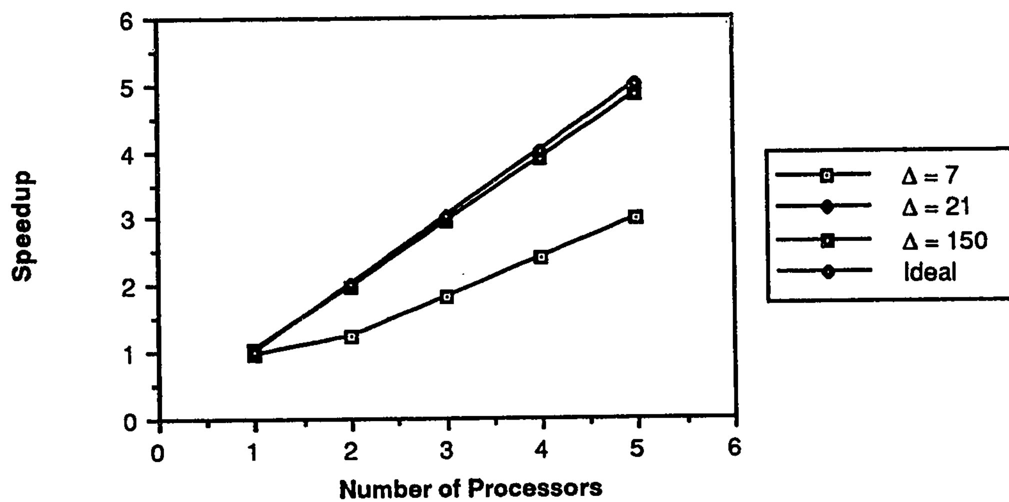


Figure B.2: Speedup Tandem Networks (20/2)

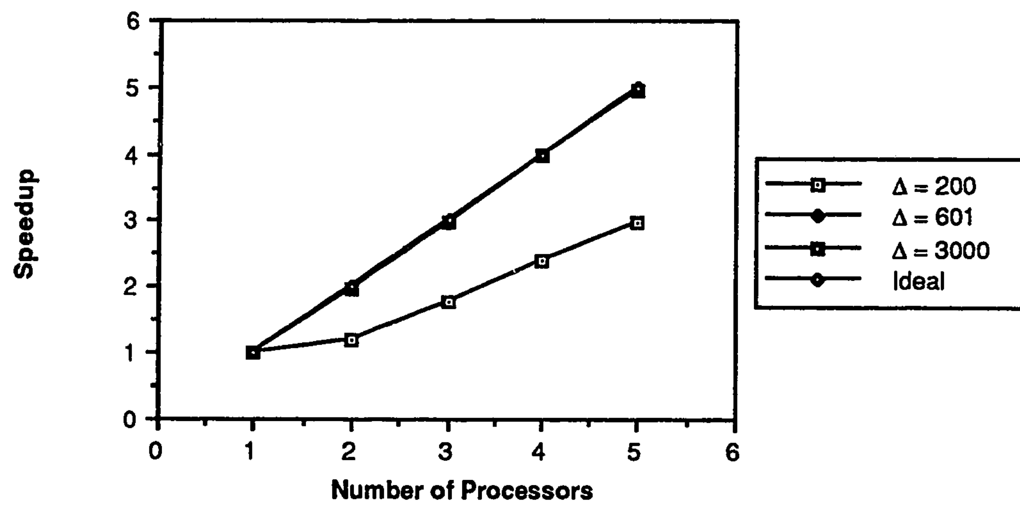


Figure B.3: Speedup Tandem Networks (600/2)

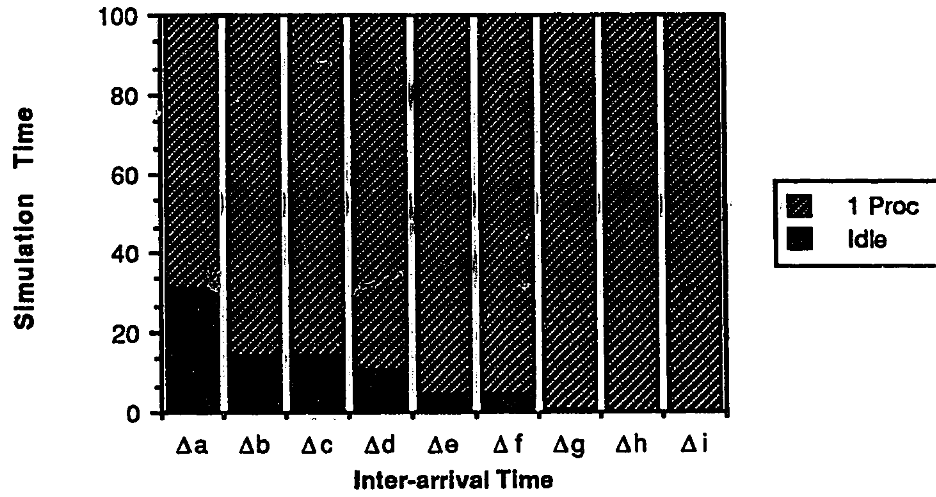


Figure B.4: Degree of Parallelism Tandem1

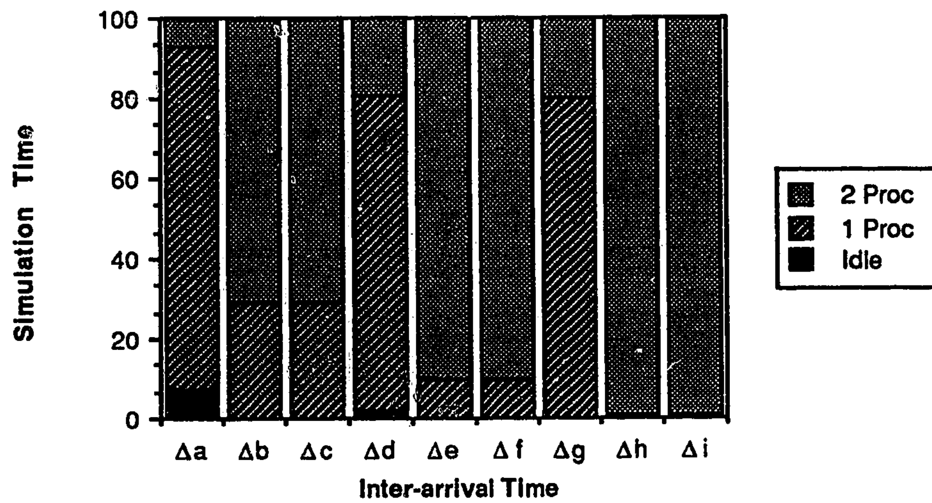


Figure B.5: Degree of Parallelism Tandem2

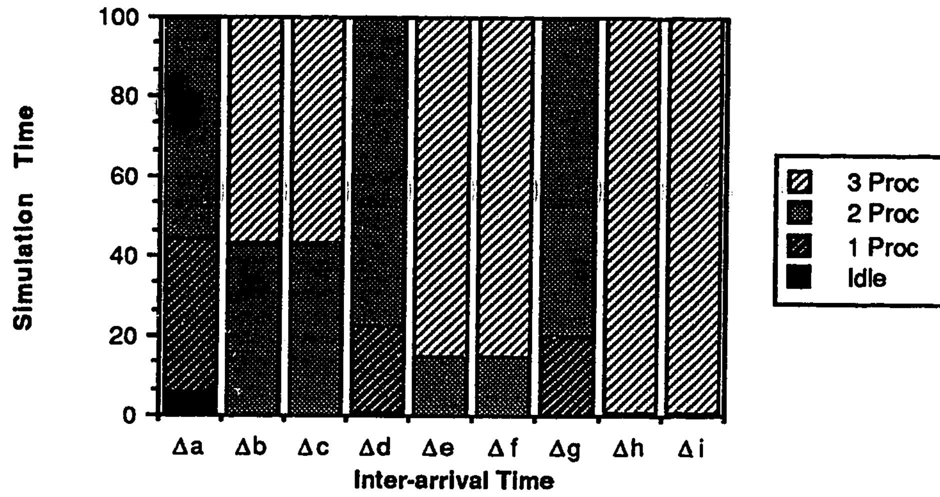


Figure B.6: Degree of Parallelism Tandem3

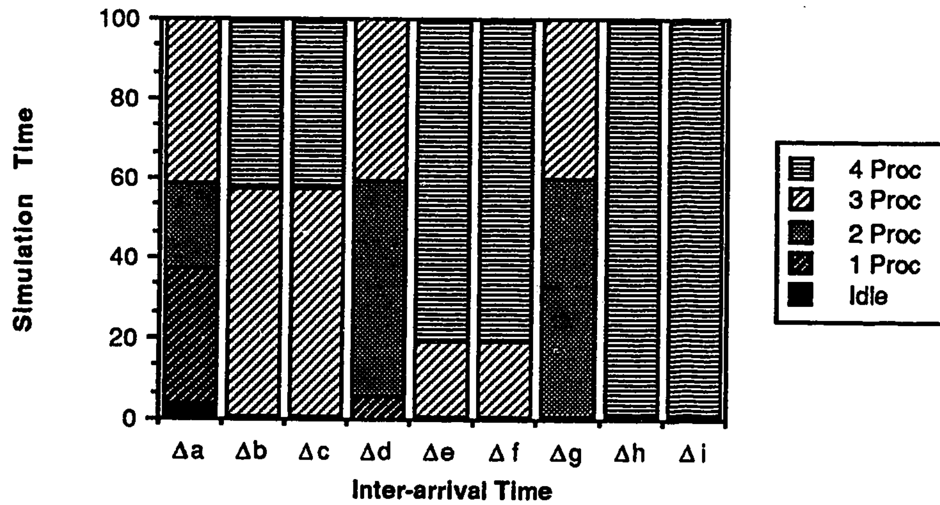


Figure B.7: Degree of Parallelism Tandem4

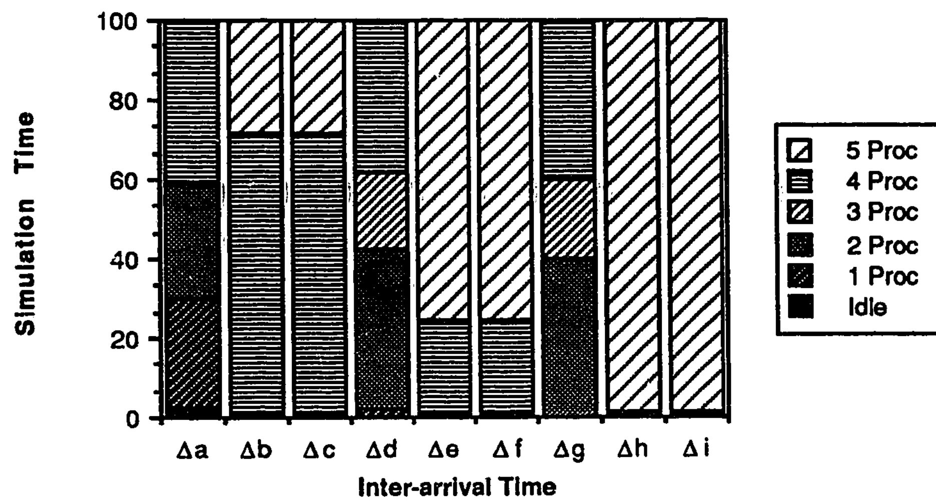


Figure B.8: Degree of Parallelism Tandem5

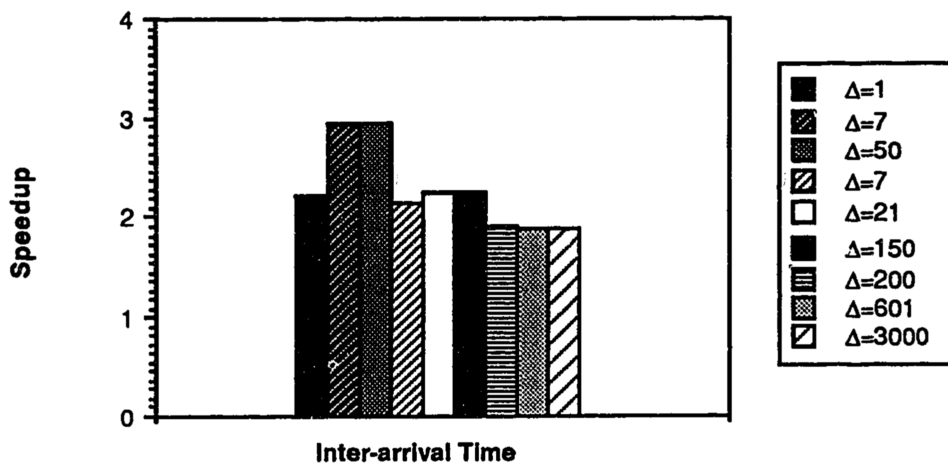


Figure B.9: Speedup Topology Feedforward1

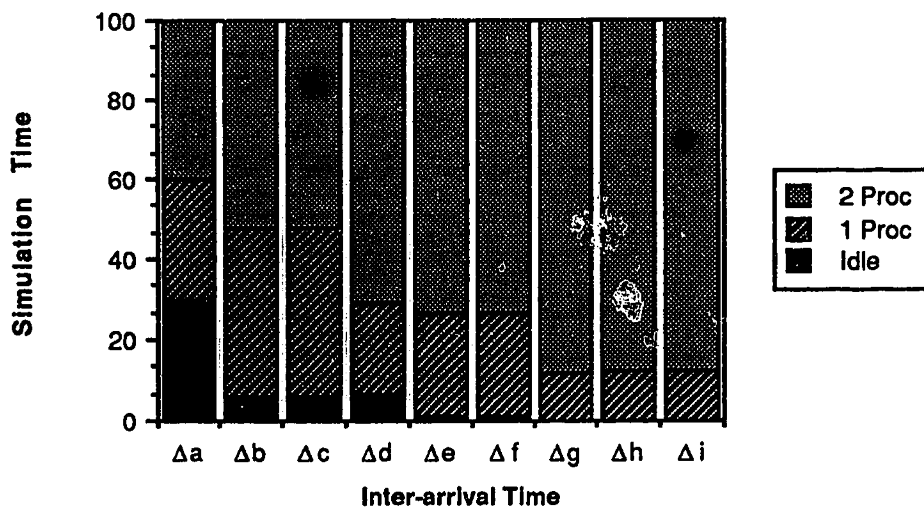


Figure B.10: Degree of Parallelism Topology Feedforward1

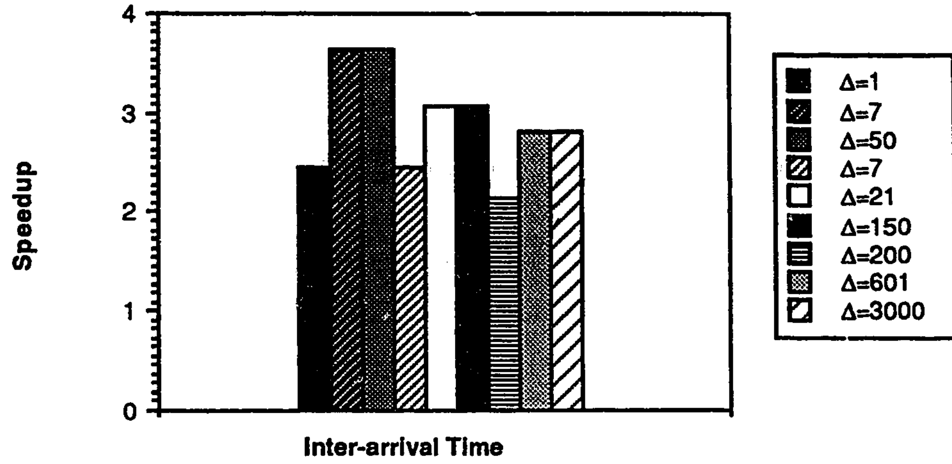


Figure B.11: Speedup Topology Feedforward2

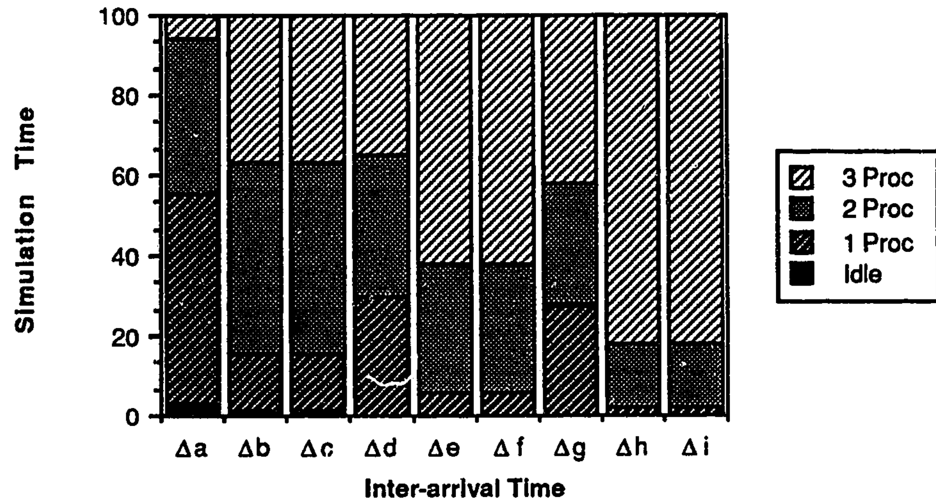


Figure B.12: Degree of Parallelism Topology Feedforward2

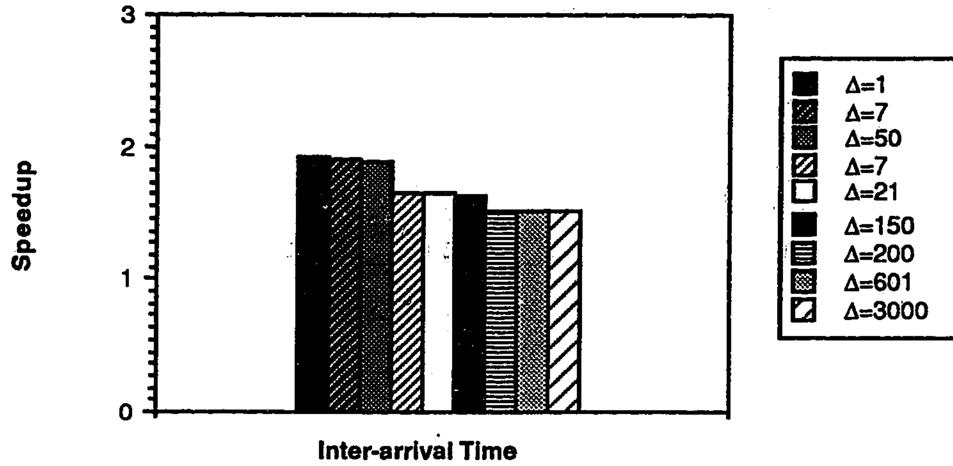


Figure B.13: Speedup Topology Feedforward3

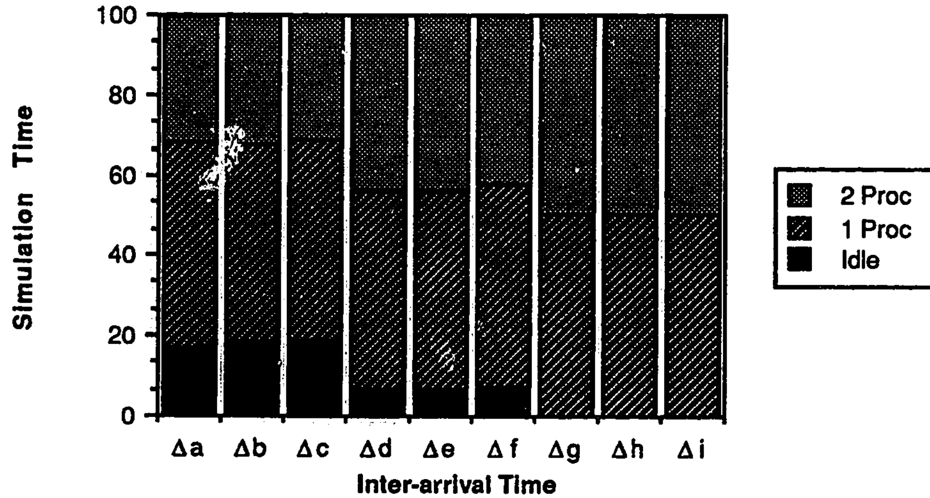


Figure B.14: Degree of Parallelism Topology Feedforward3

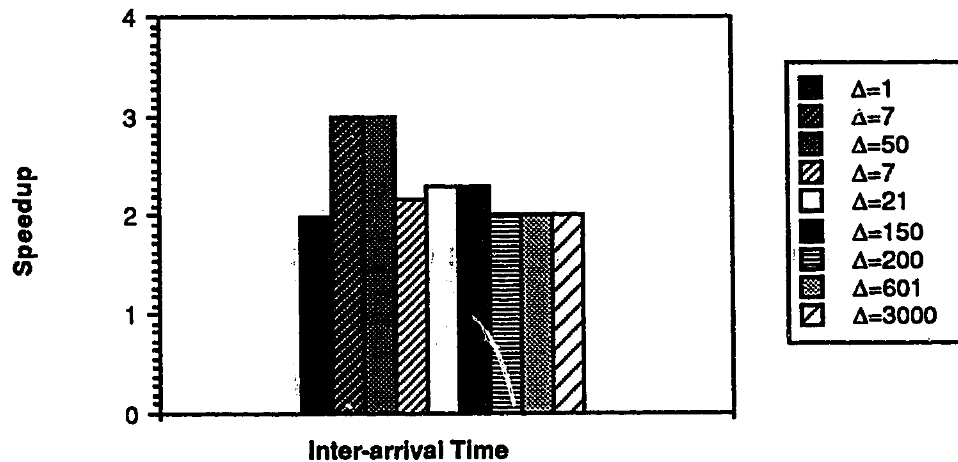


Figure B.15: Speedup Topology Feedforward4

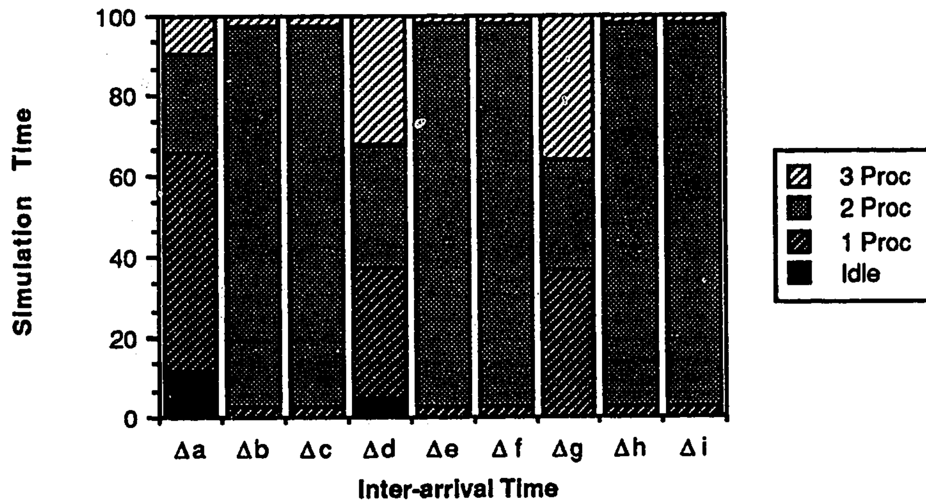


Figure B.16: Degree of Parallelism Topology Feedforward4

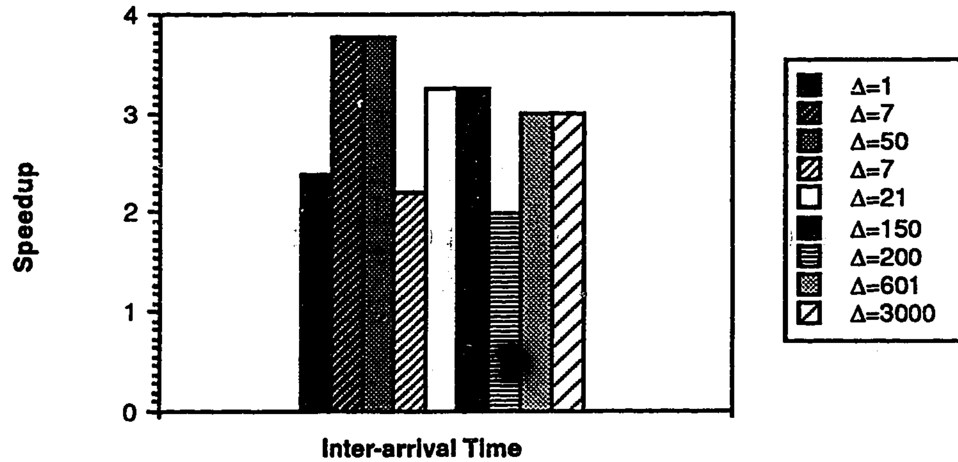


Figure B.17: Speedup Topology Feedforward5

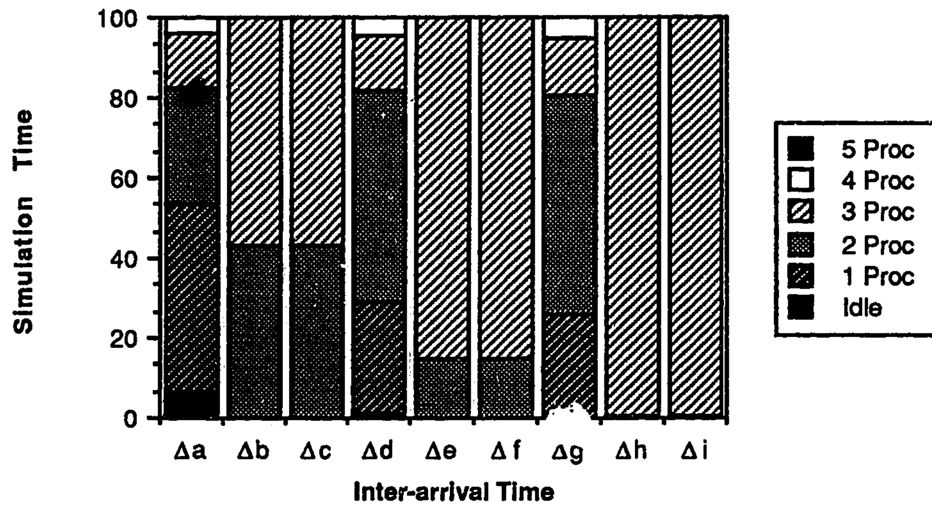


Figure B.18: Degree of Parallelism Topology Feedforward5

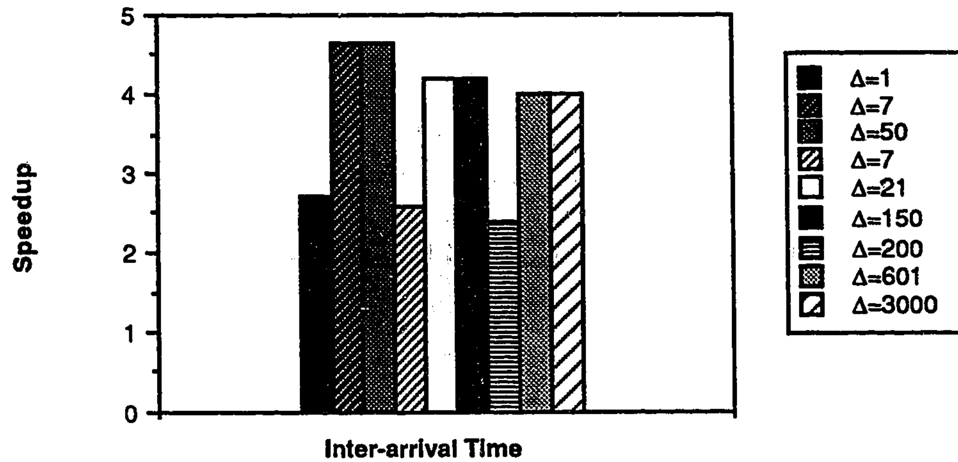


Figure B.19: Speedup Topology Feedforward6

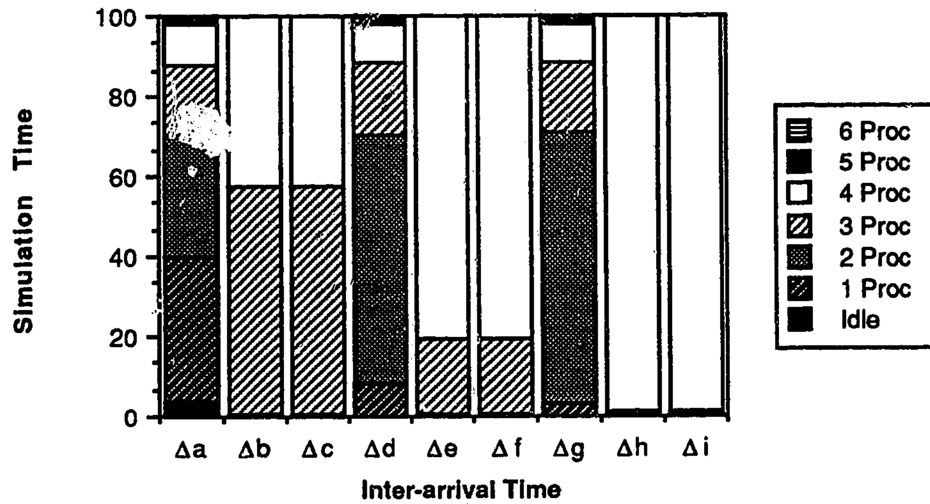


Figure B.20: Degree of Parallelism Topology Feedforward6

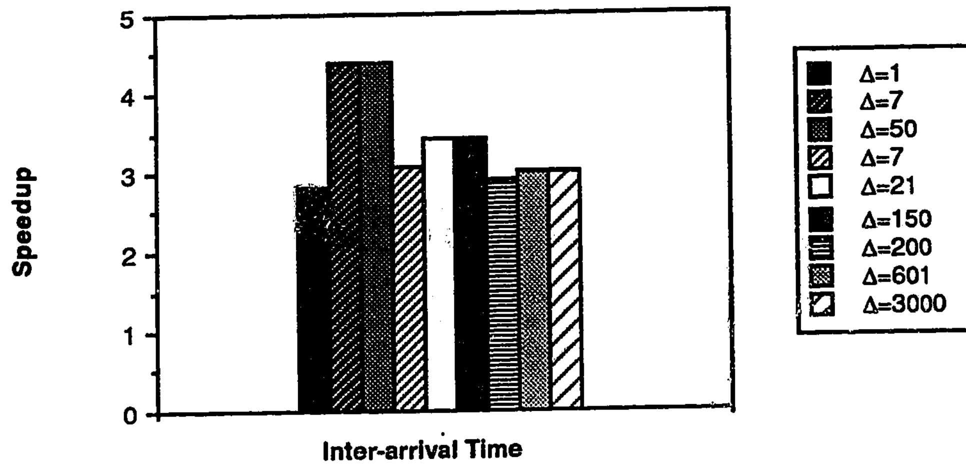


Figure B.21: Speedup Topology Feedforward7

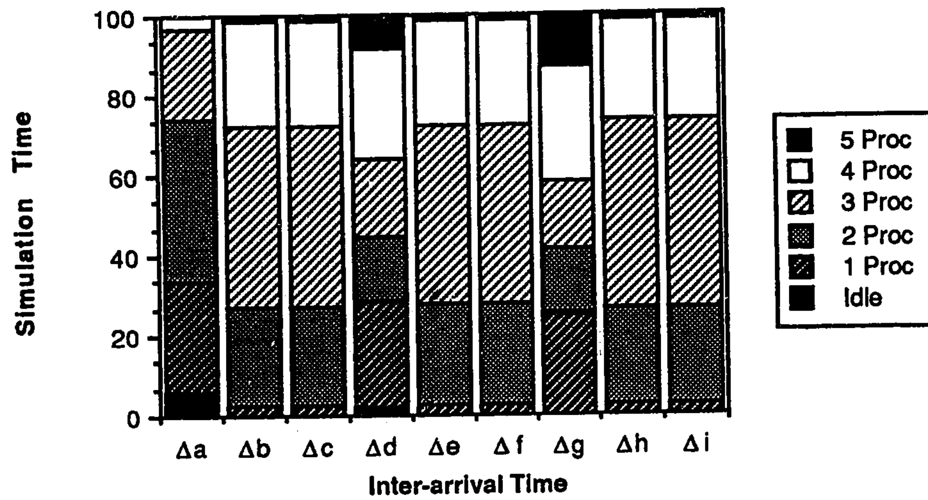


Figure B.22: Degree of Parallelism Topology Feedforward7

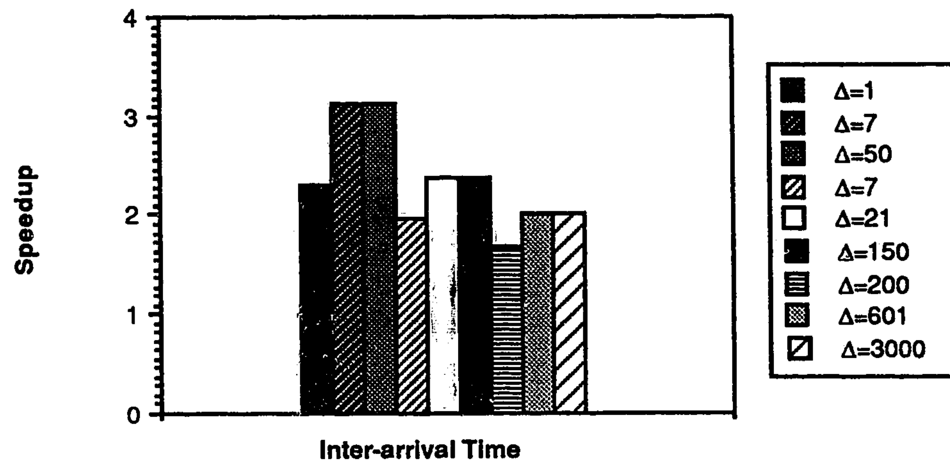


Figure B.23: Speedup Topology Feedforward8

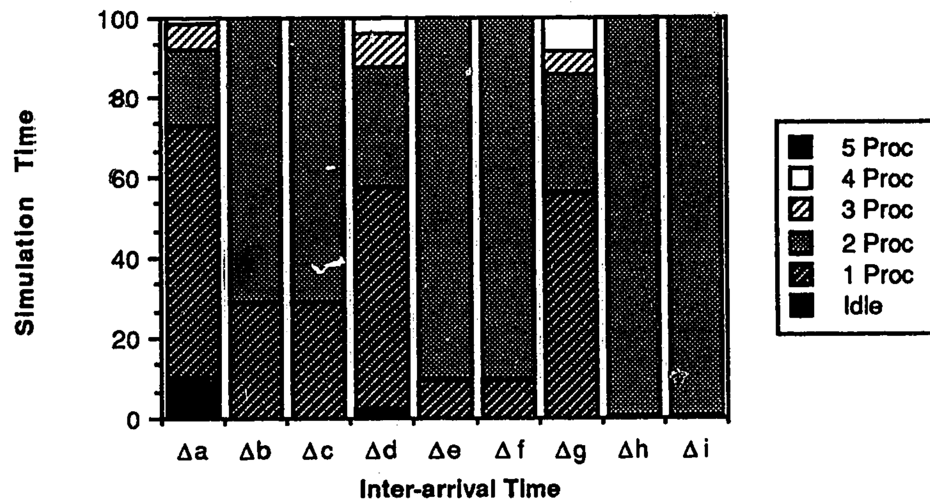


Figure B.24: Degree of Parallelism Topology Feedforward8

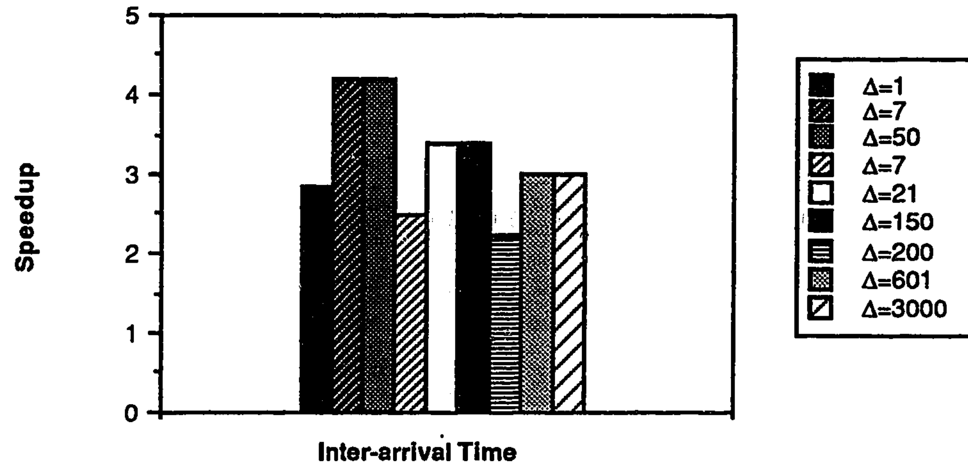


Figure B.25: Speedup Topology Feedforward9

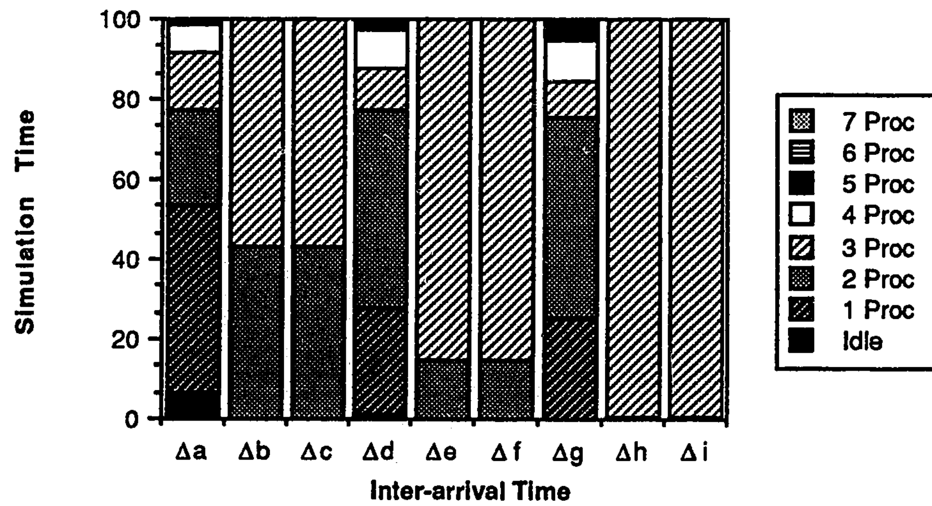


Figure B.26: Degree of Parallelism Topology Feedforward9

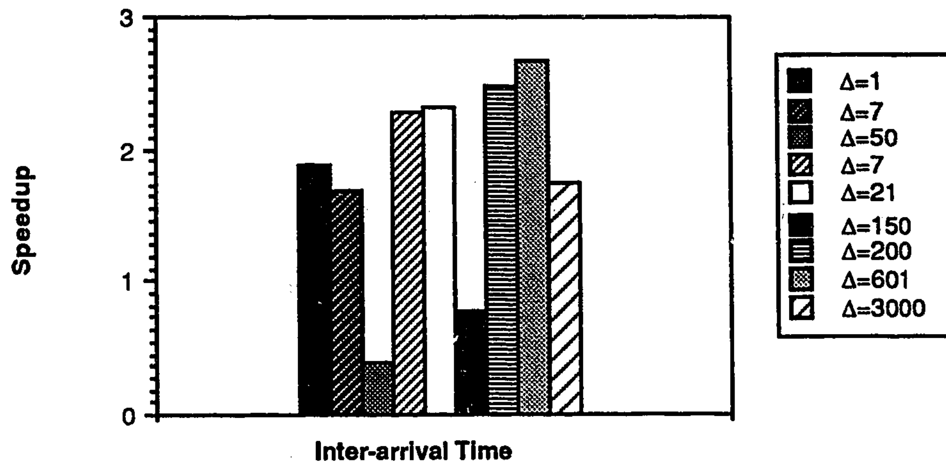


Figure B.27: Speedup Topology Feedback1

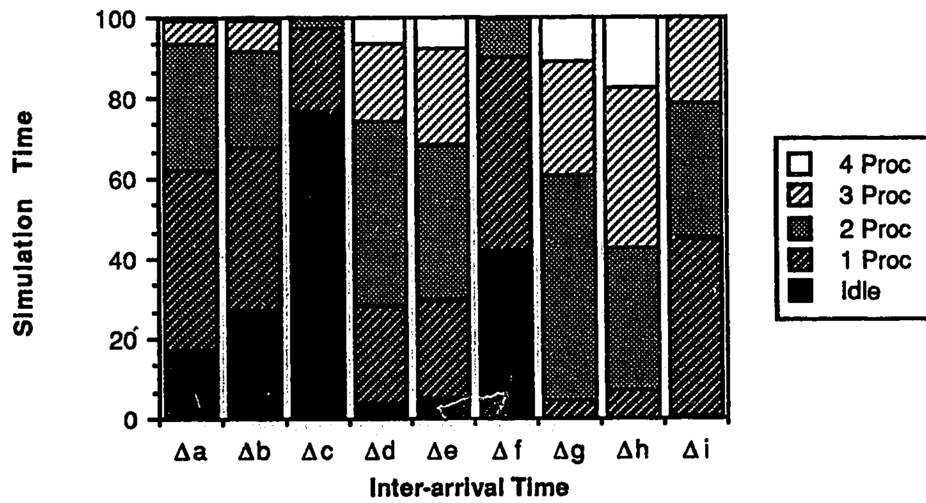


Figure B.28: Degree of Parallelism Topology Feedback1

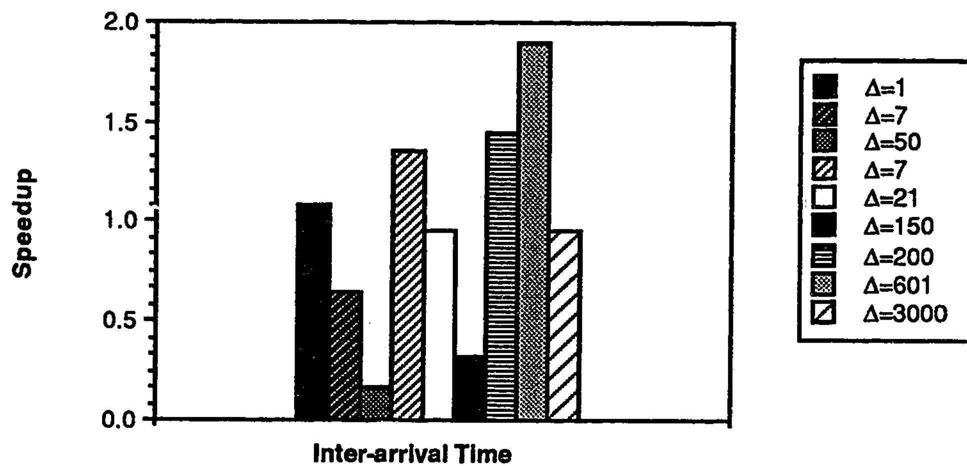


Figure B.29: Speedup Topology Feedback2

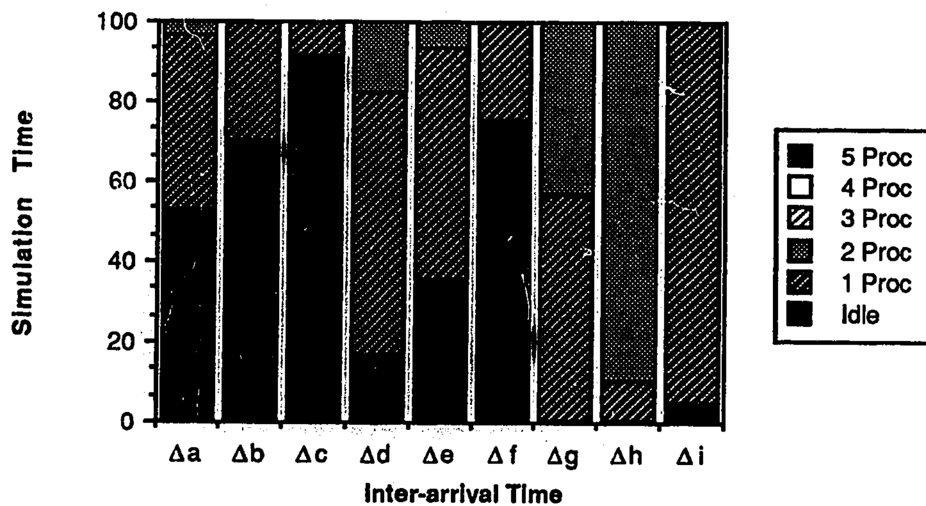


Figure B.30: Degree of Parallelism Topology Feedback2

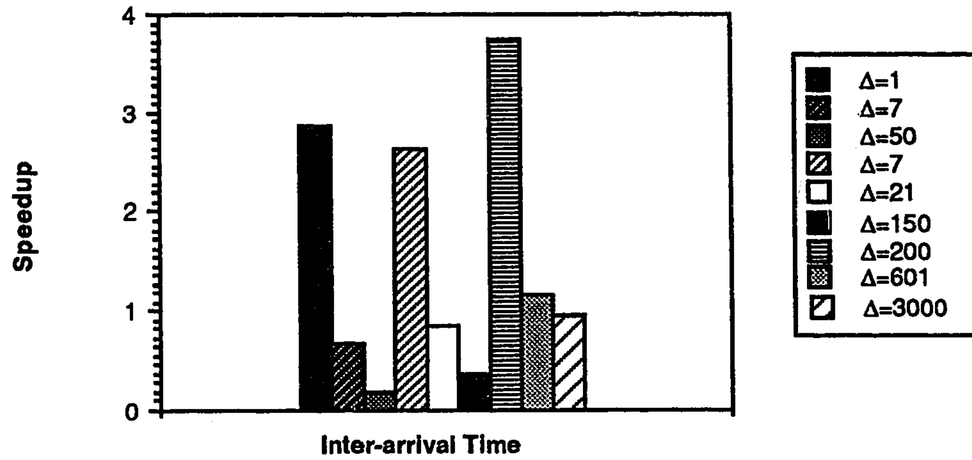


Figure B.31: Speedup Topology Cluster

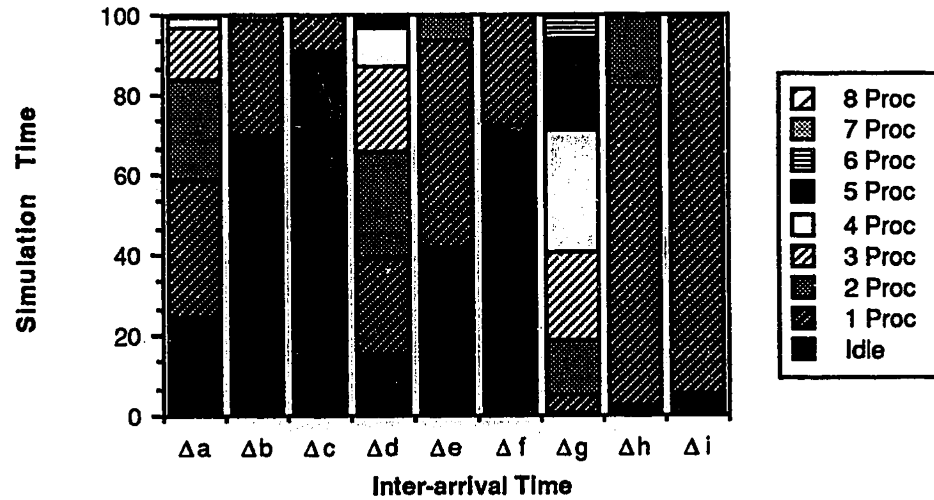


Figure B.32: Degree of Parallelism Topology Cluster

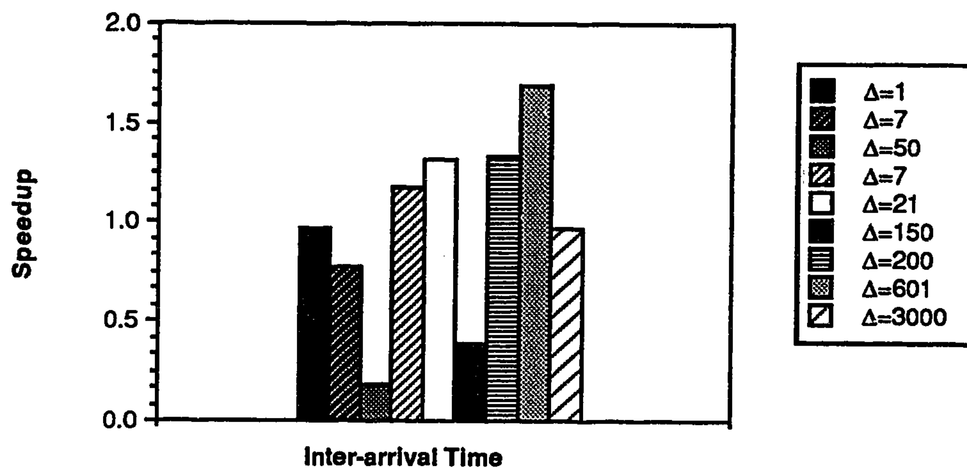


Figure B.33: Speedup Topology Central Server

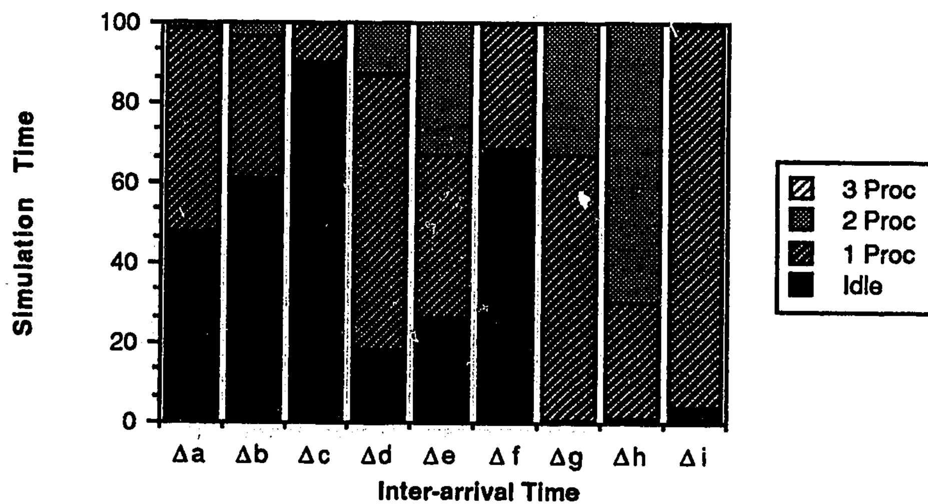


Figure B.34: Degree of Parallelism Topology Central Server

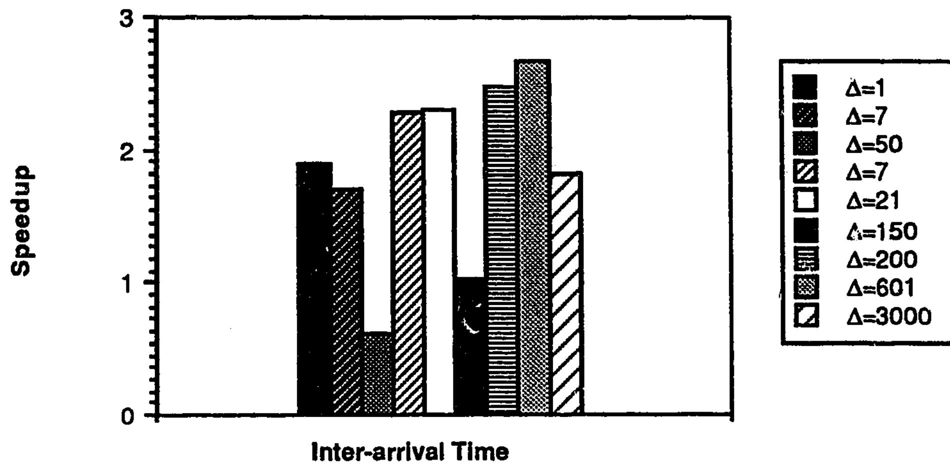


Figure B.35: Speedup Topology Feedback1 (2cNULL)

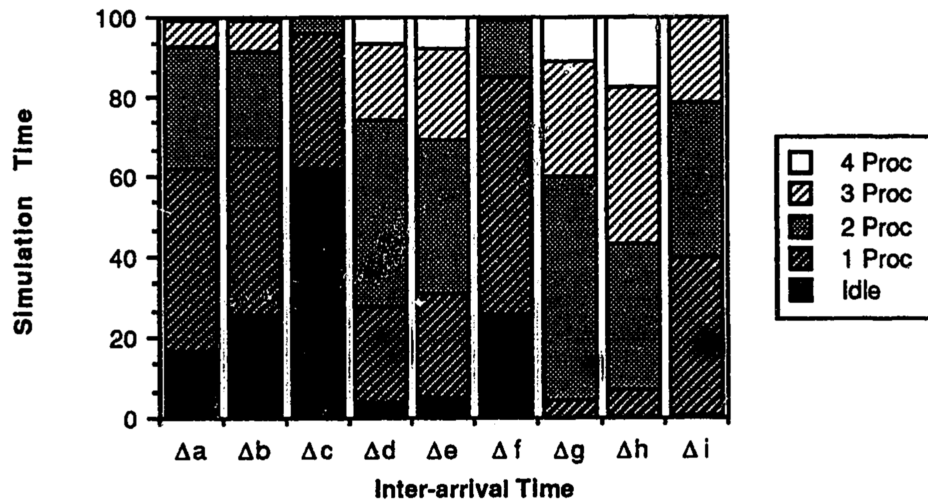


Figure B.36: Degree of Parallelism Topology Feedback1 (2cNULL)

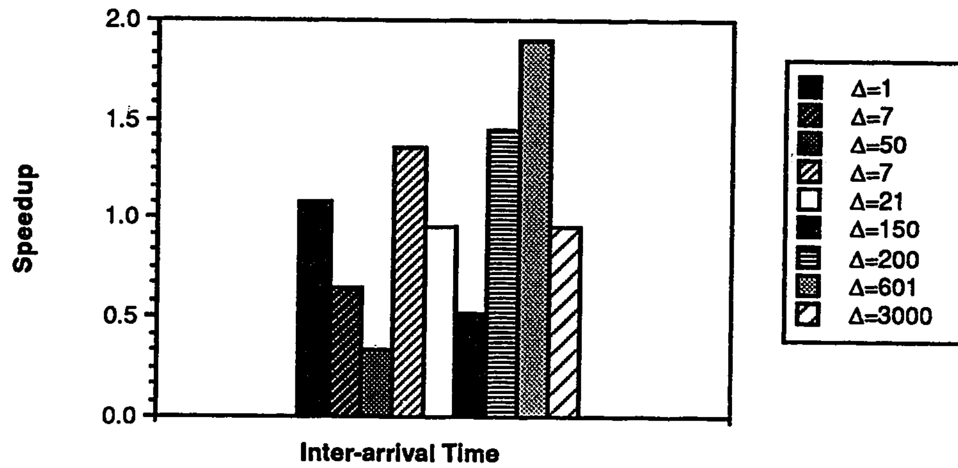


Figure B.37: Speedup Topology Feedback2 (2cNULL)

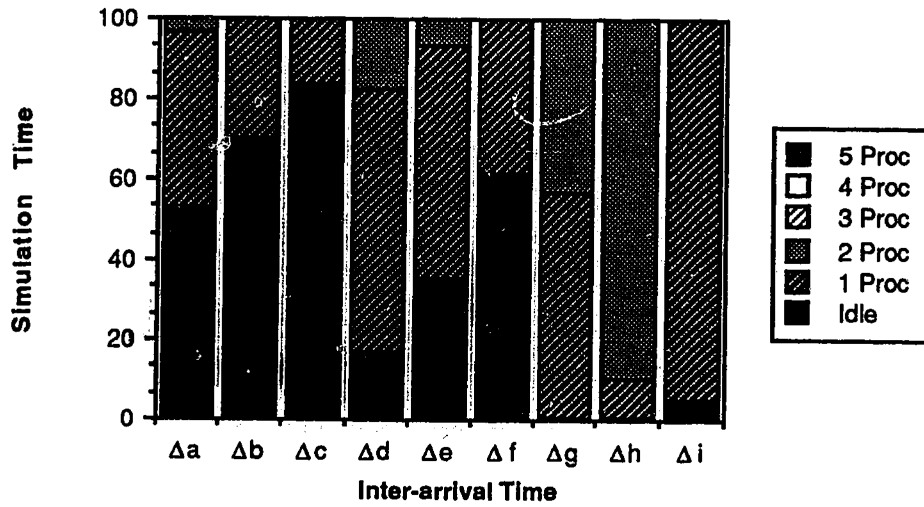


Figure B.38: Degree of Parallelism Topology Feedback2 (2cNULL)

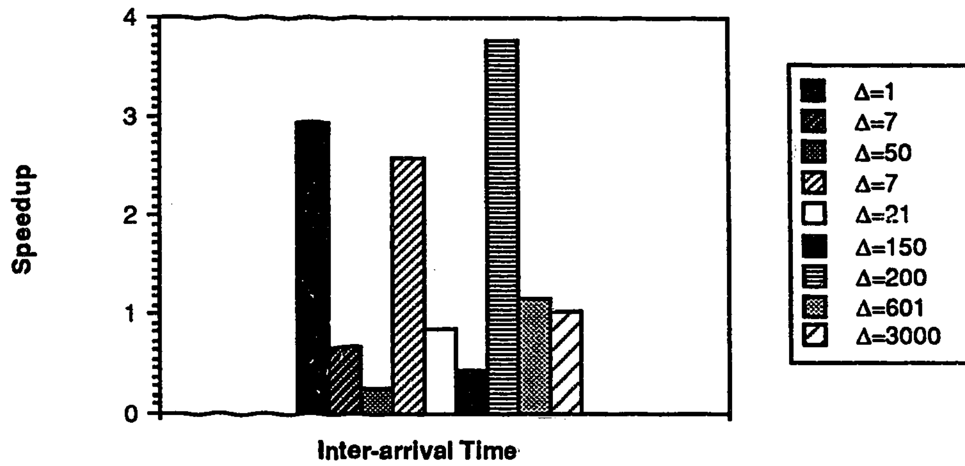


Figure B.39: Speedup Topology Cluster (2cNULL)

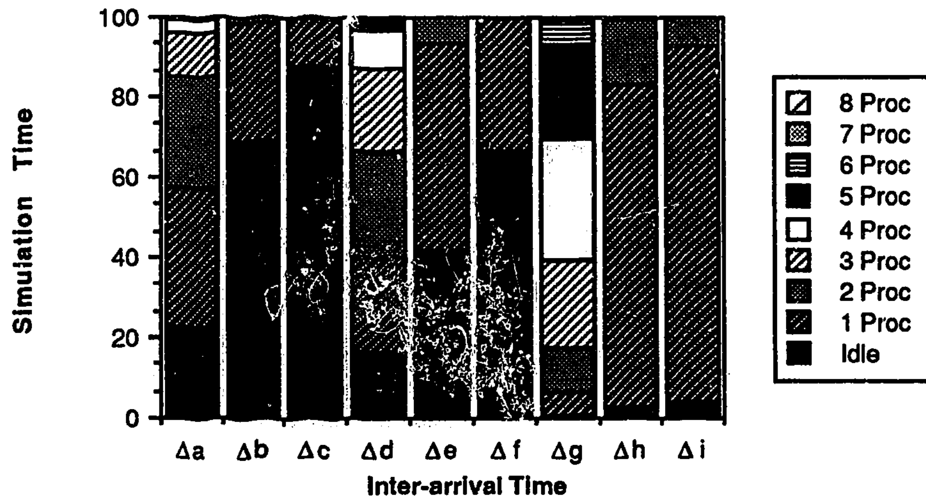


Figure B.40: Degree of Parallelism Topology Cluster (2cNULL)

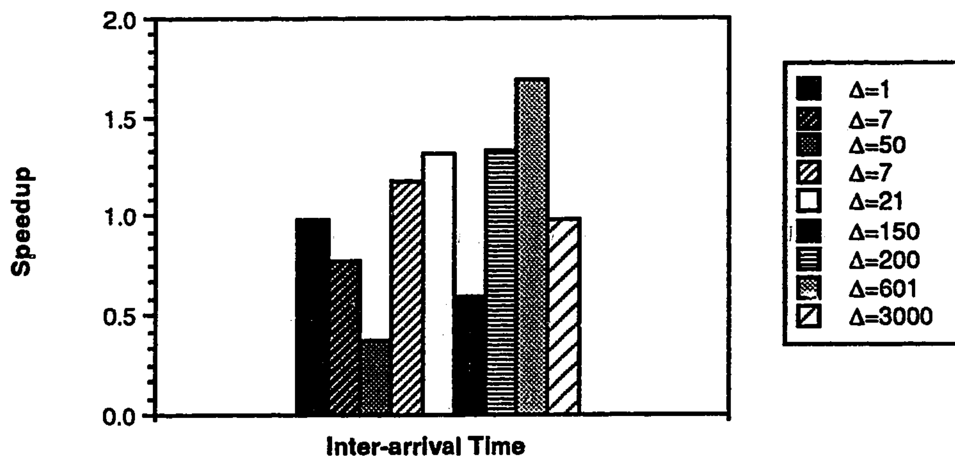


Figure B.41: Speedup Topology Central Server (2cNULL)

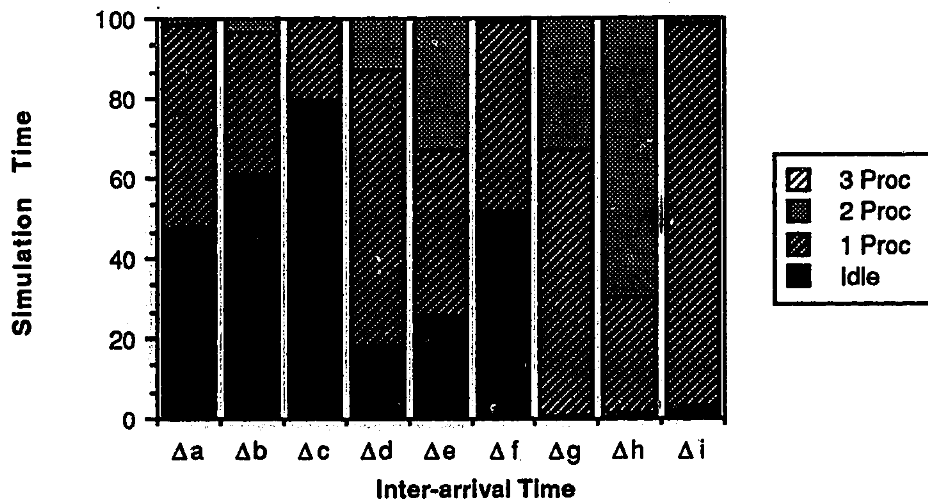


Figure B.42: Degree of Parallelism Topology Central Server (2cNULL)

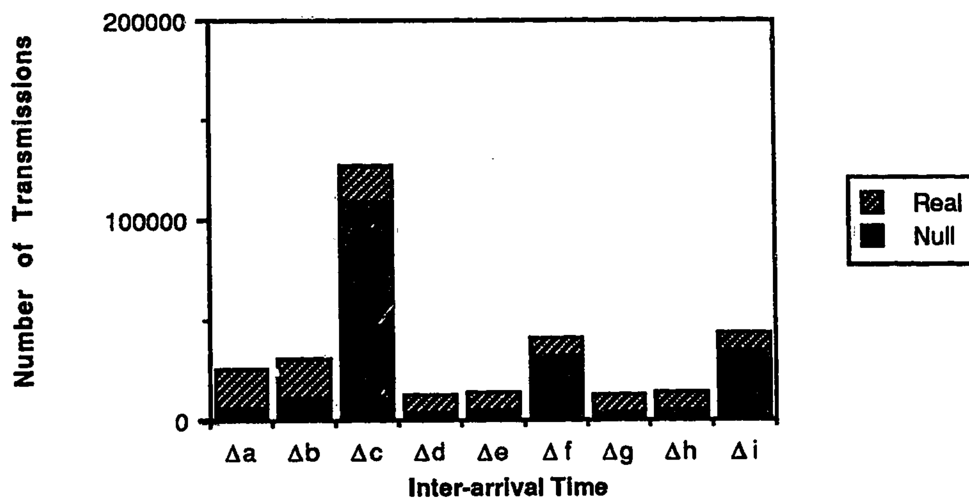


Figure B.43: Null/Real Transmissions Topology Feedback1

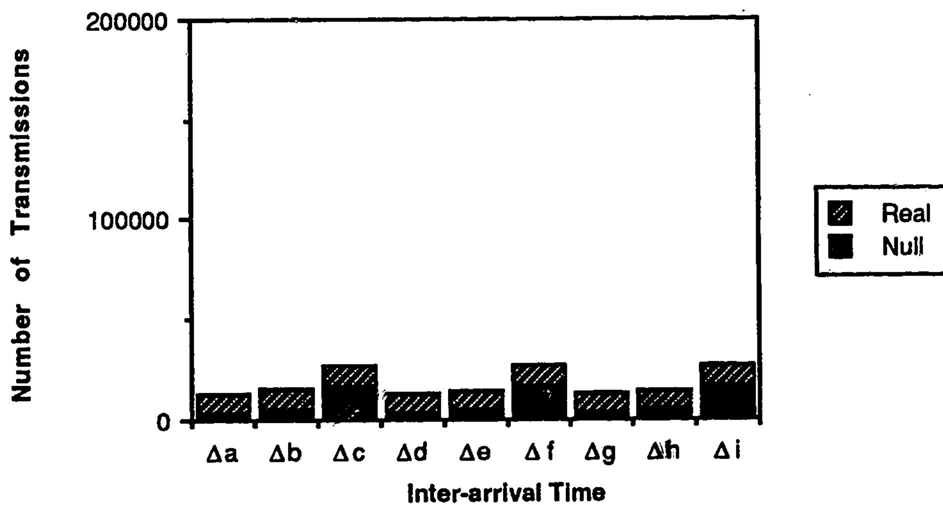


Figure B.44: Null/Real Transmissions Topology Feedback1 (2cNULL)

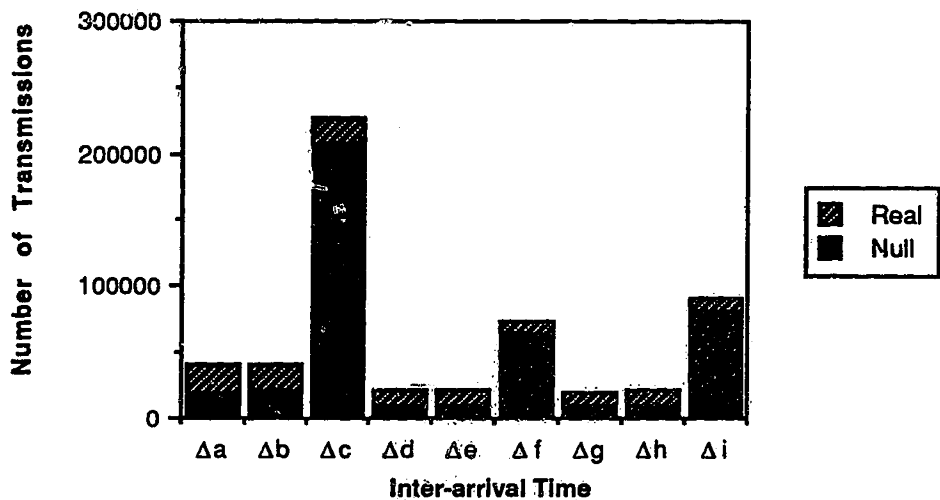


Figure B.45: Null/Real Transmissions Topology Feedback2

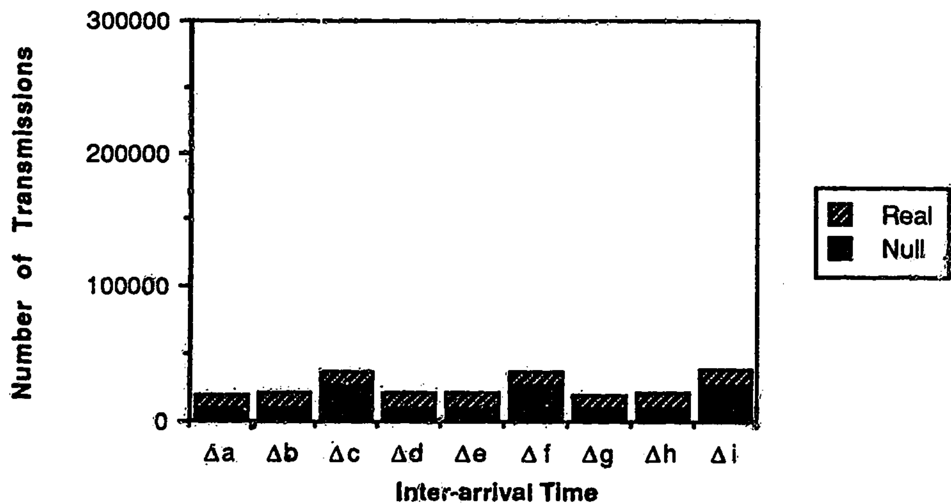


Figure B.46: Null/Real Transmissions Topology Feedback2 (2cNULL)

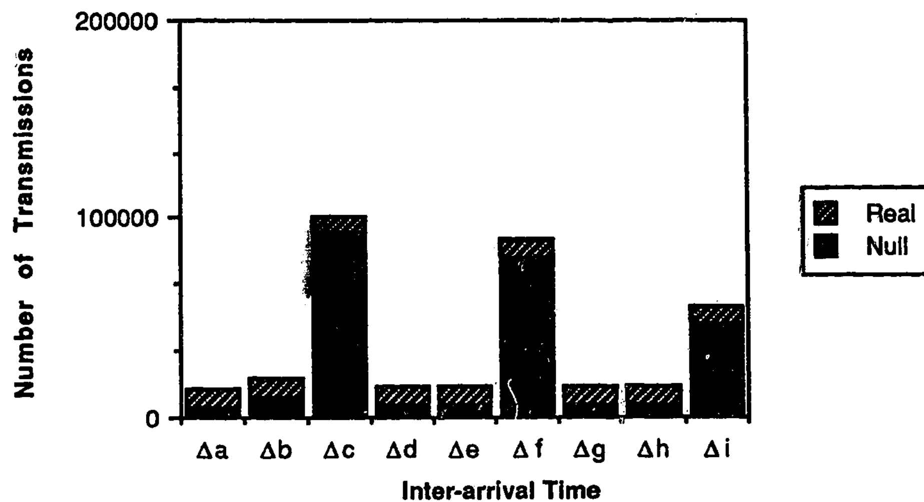


Figure B.47: Null/Real Transmissions Topology Central Server

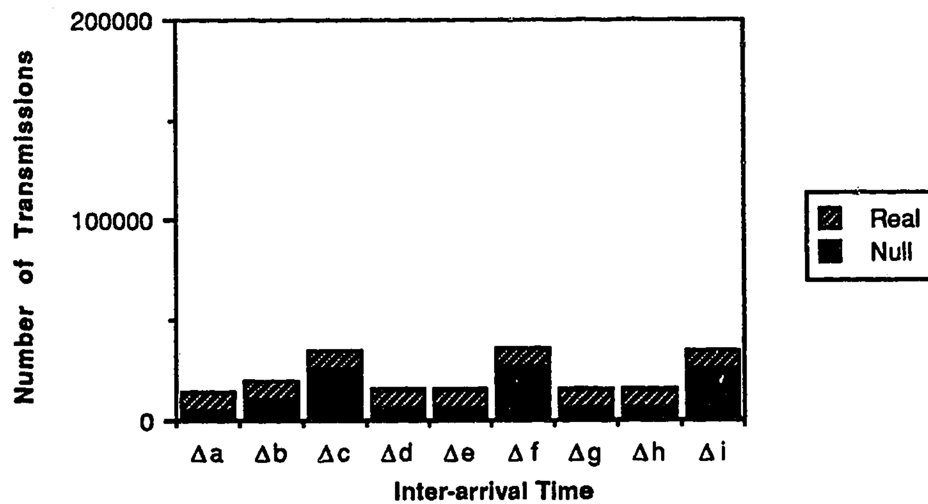


Figure B.48: Null/Real Transmissions Topology Central Server (2cNULL)

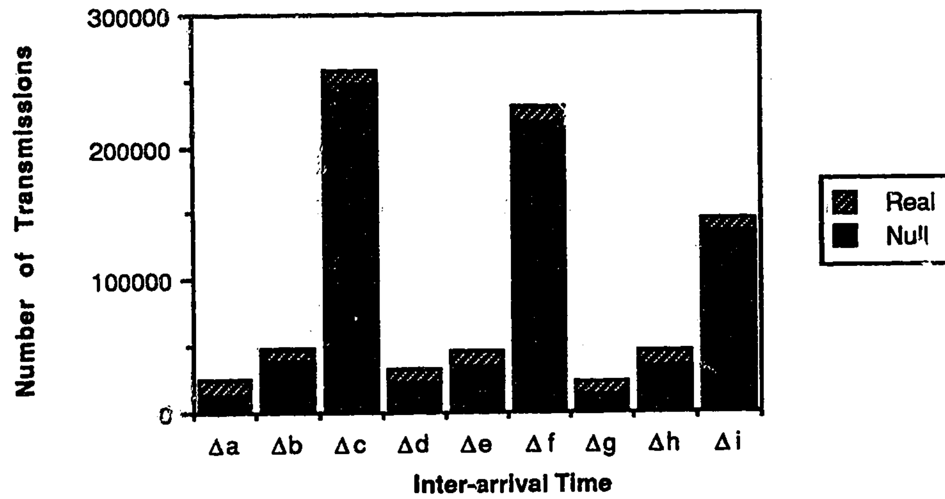


Figure B.49: Null/Real Transmissions Topology Cluster

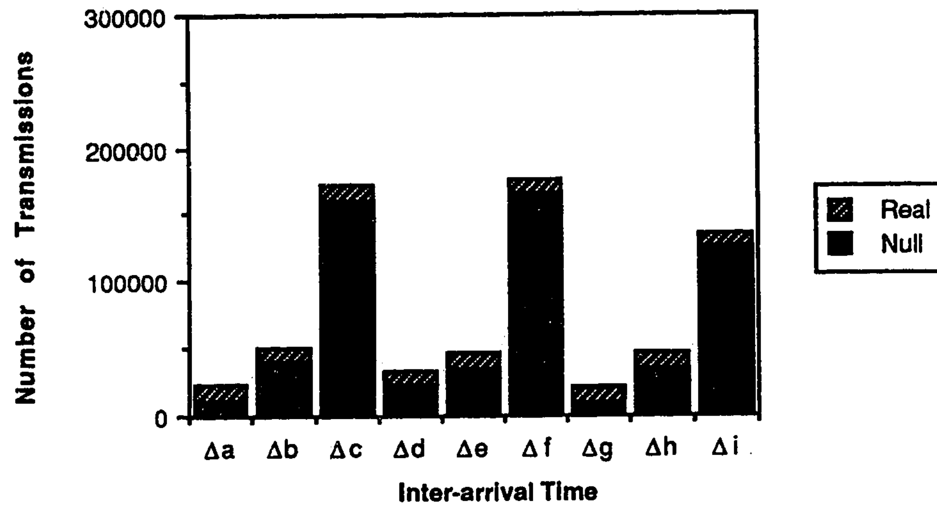


Figure B.50: Null/Real Transmissions Topology Cluster (2cNULL)