# A STUDY OF ADAPTIVE LOAD BALANCING ALGORITHMS FOR DISTRIBUTED SYSTEMS

VOL I

IAN DERRICK JOHNSON

Submitted for the degree of Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

January 1988

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the author's written consent.

The University of Aston in Birmingham

# A STUDY OF ADAPTIVE LOAD BALANCING ALGORITHMS FOR DISTRIBUTED SYSTEMS

Ian Derrick Johnson

Submitted for the degree of Doctor of Philosophy

1988

## Summary

With the advent of distributed computer systems with a largely transparent user interface, new questions have arisen regarding the management of such an environment by an operating system. One fertile area of research is that of load balancing, which attempts to improve system performance by redistributing the workload submitted to the system by the users.

Early work in this field concentrated on static placement of computational objects to improve performance, given prior knowledge of process behaviour. More recently this has evolved into studying dynamic load balancing with process migration, thus allowing the system to adapt to varying loads.

In this thesis, we describe a simulated system which facilitates experimentation with various load balancing algorithms. The system runs under UNIX and provides functions for user processes to communicate through software ports; processes reside on simulated homogeneous processors, connected by a user-specified topology, and a mechanism is included to allow migration of a process from one processor to another.

We present the results of a study of adaptive load balancing algorithms, conducted using the aforementioned simulated system, under varying conditions; these results show the relative merits of different approaches to the load balancing problem, and we analyse the trade-offs between them. Following from this study, we present further novel modifications to suggested algorithms, and show their effects on system performance.

KEY WORDS : Load Balancing
Distributed Operating Systems

*To my parents*

## ACKNOWLEDGEMENTS

I would like to express my gratitude to the following people :

To my supervisor, Dr. A.J. Harget, for his untiring patience and advice.

To Georgie and Jenny, who unravelled my handwriting in order to type the text of this thesis.

To Ian Hardy and Alan Hughes for their guidance on production of the diagrams included in the thesis.

To Neil Toye for his company in the lab.

To Nadia Bendjeddou for her friendship and much-needed moral support during this undertaking.

Finally, I acknowledge the financial support of the SERC of Great Britain

# LIST OF CONTENTS

VOLUME 1. page

Chapter 1 : Introduction 16

Chapter 2 : Distributed Systems 20

2.1 Network Operating Systems 21

2.2 Distributed Operating Systems 23

2.2.1 Communications Primitives 24

2.2.2 Naming Conventions 24

2.2.3 File Systems 25

2.2.4 Fault Tolerance 25

2.2.5 Resource Allocation 26

2.2.6 Example Distributed Operating Systems 27

2.2.6.1 LOCUS 27

2.2.6.2 DEMOS/MP 29

2.2.6.3 The V System 30

Chapter 3 : Processor Allocation Strategies 32

3.1 Distributed Scheduling 32

3.1.1 The Medusa Approach 32

3.1.2 The Wave Scheduling Approach 36

3.2 Static Load Balancing 38

3.2.1 The Graph Theoretic Approach 39

3.2.2 The 0-1 Integer Programming Approach 43

3.2.3 The Heuristic Approach 45

**3.3 Adaptive Load Balancing** 48

3.3.1 Processor Load Measurement 50

3.3.2 State Information Exchange Policy 54

3.3.2.1 The Limited Approach 55

3.3.2.2 The Pairing Approach 55

3.3.2.3 Load Vector Approach 55

3.3.2.4 Broadcast Approach 58

3.3.2.5 Global System Load Approach 59

3.3.3 Transfer Policy 62

3.3.4 Co-operation and Location Policy 66

3.3.4.1 Sender-initiated Approaches 66

3.3.4.2 Receiver initiated Approaches 72

**Chapter 4 : Simulated System Design and Implementation** 76

**4.1 Rationale and Intended Goals** 76

**4.2 Development Environment** 77

**4.3 Simulated Physical Network** 79

4.3.1 The Start-up Process and Simulated Processors 79

4.3.2 Simulated Time Maintenance 82

4.3.2.1 Time Definition 83

4.3.2.2 Processor Synchronisation 85

4.3.3 Communications Medium 87

4.3.3.1 Network Topology 87

4.3.3.2 Network Routing 89

4.3.4 Interprocessor Communications 91

4.4 Distributed Operating System Kernel 95

4.4.1 General Structure 95

4.4.2 User Process Support 96

4.4.2.1 Process Naming 97

4.4.2.2 Kernel Call Interface 98

4.4.2.3 The Process Table 99

4.4.3 Interprocess Communications Mechanism 102

4.4.3.1 Message-Passing Primitives 103

4.4.3.2 Software Ports 104

4.4.3.3 User Message Format 107

4.4.3.4 The Port Table 107

4.4.4 Kernel Call Mechanism 111

4.4.4.1 The User Process View 111

4.4.4.2 Kernel Call and Return 112

4.4.5 Kernel Call Implementation 115

4.4.5.1 Process-related Kernel Calls 116

4.4.5.2 Port-related Kernel Calls 118

4.4.5.3 Message-related Kernel Calls 123

4.4.6 Per-processor Scheduling 127

4.4.7 Process Migration Mechanism 127

Chapter 5 : Simulation Results 131

5.1 Main Goals and Methods of Experiments 131

5.2 Experimental Environment 132

5.2.1 Network Topology 132

5.2.2 Synthetic Workload Generation 135

5.2.3 Load Balancing Algorithm Implementation 137

5.2.4 Performance Metrics and Monitoring 138

**5.3 Independent User Processes** 140

5.3.1 Independent User Process Model 140

5.3.2 Load Balancing Algorithms Implemented 141

5.3.2.1 Processor Load Measurement 141

5.3.2.2 Random Algorithm 142

5.3.2.3 Threshold Algorithm 142

5.3.2.4 Global Average Algorithm 144

5.3.3 Discussion of Results 148

5.3.4 General Observations 153

**5.4 Co-operating Process Groups using Message-Passing** 155

5.4.1 Process Group Model 155

5.4.2 Process Group Synthetic Workload Generation 159

5.4.3 Load Balancing Algorithms Implemented 160

5.4.4 Performance of Load Balancing Algorithms 162

5.4.5 Costs of Algorithm Execution 169

**5.5 Modified Load Balancing Algorithms** 170

5.5.1 Receiver-Initiated Global Average Algorithm 171

5.5.1.1 Implementation 171

5.5.1.2 Performance 173

5.5.2 Limited Broadcast Global Average Algorithm 176

5.5.2.1 Implementation 177

5.5.2.2 Performance 178

5.5.3 Sender/Receiver Global Average Algorithm 181

5.5.3.1 Implementation 181

5.5.3.2 Performance 185

**5.6 Summary** 188

Chapter 6 : Conclusions 190

References 196

VOLUME 2.

Appendix A : Detailed Simulation Results
(using the Independent Process Model) 3

Appendix B : Detailed Simulation Results
(using the Cooperating Process Group Model) 19

Appendix C : Program for the Simulated System 42

## LIST OF FIGURES

**VOLUME 1**


| | | Page |
|---|---|---|
| Fig. 3.1 | Graph Theoretic Module Assignment and Minimum Cutset for Two Processors | 41 |
| Fig. 3.2 | Gradient Surface for the Gradient Model Algorithm in a 9-Processor Network | 61 |
| Fig. 4.1 | System Configuration and Start-up | 81 |
| Fig. 4.2 | Interprocessor Synchronisation Mechanism | 86 |
| Fig. 4.3 | Example Network Topology and Physical Link Table | 88 |
| Fig. 4.4 | Example Route Table and Physical Link Table for Processor 0 | 90 |
| Fig. 4.5 | Interprocessor Communications Mechanism | 94 |
| Fig. 4.6 | Process Table Structure | 100 |
| Fig. 4.7 | Example Interprocess Connection using Ports and Links | 106 |
| Fig. 4.8 | Port Table Structure | 109 |
| Fig. 4.9 | Kernel Call/Return Mechanism | 114 |
| Fig. 5.1 | Square Mesh Topology used in our Experiments and Example Route Table for Processor 3 | 134 |
| Fig. 5.2 | Performance Comparison using the Independent Process Model | 149 |
| Fig. 5.3 | Performance Comparison using the Cooperating Process Group Model | 163 |

Fig. 5.4 Performance Comparison 174
Global Average vs the Receiver-initiated Variant
using the Cooperating Process Group Model

Fig. 5.5 Performance Comparison 179
Global Average vs the Limited Broadcast Variant
using the Cooperating Process Group Model

Fig. 5.6 Performance Comparison 186
Global Average vs the Sender/Receiver Variant
using the Cooperating Process Group Model


**VOLUME 2**


Fig. A.1.1 Mean Load - No Load Balancing vs Threshold 4
using the Independent Process Model
(Load Value = 0.2)

Fig. A.1.2 Load Variance - No Load Balancing vs Threshold 5
using the Independent Process Model
(Load Value = 0.2)

Fig. A.1.3 Load Difference - No Load Balancing vs Threshold 6
using the Independent Process Model
(Load Value = 0.2)

Fig. A.2.1 Mean Load - No Load Balancing vs Threshold 7
using the Independent Process Model
(Load Value = 0.5)

Fig. A.2.2 Load Variance - No Load Balancing vs Threshold 8
using the Independent Process Model
(Load Value = 0.5)

Fig. A.2.3 Load Difference - No Load Balancing vs Threshold 9
using the Independent Process Model
(Load Value = 0.5)

Fig. A.3.1 Mean Load - No Load Balancing vs Random 10
using the Independent Process Model
(Load Value = 0.8)

| | | |
|---|---|---|
| Fig. A.3.2 | Mean Load - Random vs Global Average<br>using the Independent Process Model<br>(Load Value = 0.8) | 11 |
| Fig. A.3.3 | Mean Load - Global Average vs Threshold<br>using the Independent Process Model<br>(Load Value = 0.8) | 12 |
| Fig. A.3.4 | Load Variance - No Load Balancing vs Random<br>using the Independent Process Model<br>(Load Value = 0.8) | 13 |
| Fig. A.3.5 | Load Variance - Random vs Global Average<br>using the Independent Process Model<br>(Load Value = 0.8) | 14 |
| Fig. A.3.6 | Load Variance - Global Average vs Threshold<br>using the Independent Process Model<br>(Load Value = 0.8) | 15 |
| Fig. A.3.7 | Load Difference - No Load Balancing vs Random<br>using the Independent Process Model<br>(Load Value = 0.8) | 16 |
| Fig. A.3.8 | Load Difference - Random vs Global Average<br>using the Independent Process Model<br>(Load Value = 0.8) | 17 |
| Fig. A.3.9 | Load Difference - Global Average vs Threshold<br>using the Independent Process Model<br>(Load Value = 0.8) | 18 |
| Fig. B.1.1 | Mean Load - No Load Balancing vs Threshold<br>using the Cooperating Process Group Model<br>(Load Value = 0.2) | 20 |
| Fig. B.1.2 | Load Variance - No Load Balancing vs Threshold<br>using the Cooperating Process Group Model<br>(Load Value = 0.2) | 21 |
| Fig. B.1.3 | Load Difference - No Load Balancing vs Threshold<br>using the Cooperating Process Group Model<br>(Load Value = 0.2) | 22 |

| | | |
|---|---|---|
| Fig. B.1.4 | Load Variance - Threshold vs Global Average<br>using the Cooperating Process Group Model<br>(Load Value = 0.2) | 23 |
| Fig. B.1.5 | Load Difference - Threshold vs Global Average<br>using the Cooperating Process Group Model<br>(Load Value = 0.2) | 24 |
| Fig. B.2.1 | Mean Load - No Load Balancing vs Threshold<br>using the Cooperating Process Group Model<br>(Load Value = 0.5) | 25 |
| Fig. B.2.2 | Load Variance - No Load Balancing vs Threshold<br>using the Cooperating Process Group Model<br>(Load Value = 0.5) | 26 |
| Fig. B.2.3 | Load Difference - No Load Balancing vs Threshold<br>using the Cooperating Process Group Model<br>(Load Value = 0.5) | 27 |
| Fig. B.2.4 | Load Variance - Threshold vs Global Average<br>using the Cooperating Process Group Model<br>(Load Value = 0.5) | 28 |
| Fig. B.2.5 | Load Difference - Threshold vs Global Average<br>using the Cooperating Process Group Model<br>(Load Value = 0.5) | 29 |
| Fig. B.3.1 | Mean Load - No Load Balancing vs Random<br>using the Cooperating Process Group Model<br>(Load Value = 0.7) | 30 |
| Fig. B.3.2 | Mean Load - Random vs Threshold<br>using the Cooperating Process Group Model<br>(Load Value = 0.7) | 31 |
| Fig. B.3.3 | Mean Load - Threshold vs Preemptive Threshold<br>using the Cooperating Process Group Model<br>(Load Value = 0.7) | 32 |

| | | |
|---|---|---|
| Fig. B.3.4 | Mean Load - Threshold vs Global Average<br>using the Cooperating Process Group Model<br>(Load Value = 0.7) | 33 |
| Fig. B.3.5 | Load Variance - No Load Balancing vs Random<br>using the Cooperating Process Group Model<br>(Load Value = 0.7) | 34 |
| Fig. B.3.6 | Load Variance - Random vs Threshold<br>using the Cooperating Process Group Model<br>(Load Value = 0.7) | 35 |
| Fig. B.3.7 | Load Variance - Threshold vs Preemptive Threshold<br>using the Cooperating Process Group Model<br>(Load Value = 0.7) | 36 |
| Fig. B.3.8 | Load Variance - Threshold vs Global Average<br>using the Cooperating Process Group Model<br>(Load Value = 0.7) | 37 |
| Fig. B.3.9 | Load Difference - No Load Balancing vs Random<br>using the Cooperating Process Group Model<br>(Load Value = 0.7) | 38 |
| Fig. B.3.10 | Load Difference - Random vs Threshold<br>using the Cooperating Process Group Model<br>(Load Value = 0.7) | 39 |
| Fig. B.3.11 | Load Difference - Threshold vs Preemptive Threshold<br>using the Cooperating Process Group Model<br>(Load Value = 0.7) | 40 |
| Fig. B.3.12 | Load Difference - Threshold vs Global Average<br>using the Cooperating Process Group Model<br>(Load Value = 0.7) | 41 |

## LIST OF TABLES

VOLUME 1


| | | Page |
|---|---|---|
| Table 5.1 | Overall System Behaviour using the Independent Process Model (Load Value = 0.2) | 151 |
| Table 5.2 | Overall System Behaviour using the Independent Process Model (Load Value = 0.5) | 152 |
| Table 5.3 | Overall System Behaviour using the Independent Process Model (Load Value = 0.8) | 153 |
| Table 5.4 | Overall System Behaviour using the Cooperating Process Group Model (Load Value = 0.2) | 165 |
| Table 5.5 | Overall System Behaviour using the Cooperating Process Group Model (Load Value = 0.5) | 166 |
| Table 5.6 | Overall System Behaviour using the Cooperating Process Group Model (Load Value = 0.7) | 167 |
| Table 5.7 | Comparison of Average Interprocessor Messages Transmitted per Processor per Second using the Cooperating Process Group Model | 169 |
| Table 5.8 | Overall System Behaviour using the Cooperating Process Group Model Comparing Global Average and the Receiver-initiated Variant (Load Value = 0.2, 0.5 and 0.7) | 175 |
| Table 5.9 | Overall System Behaviour using the Cooperating Process Group Model Comparing Global Average and the Limited Broadcast Variant (Load Value = 0.2, 0.5 and 0.7) | 180 |
| Table 5.10 | Overall System Behaviour using the Cooperating Process Group Model Comparing Global Average and the Sender/Receiver Variant (Load Value = 0.2, 0.5 and 0.7) | 187 |

# CHAPTER 1

## INTRODUCTION

With the traditional Von Neumann architecture reaching the physical limits of its capabilities, the trend in computer systems has been towards distributing processing power over a number of processors executing in parallel and connected via a communications medium. This can be viewed as a natural extension to the multiprogramming mechanism used to share a single CPU between a number of user processes, which, through careful management by an operating system appear to have exclusive access to the CPU, but are in fact being given periods of processor time, interleaved with their peers; thus when more than one processor is available much of this pseudo-parallel execution becomes truly parallel.

However, the advances in distributed processor technology need to be followed by similar developments in the design and implementation of operating systems capable of managing this new environment. The art of operating system design for uniprocessor architectures has made significant steps forward since the "monolithic monitor" concept, but new problems arise for distributed systems. We are interested in such systems where processors are autonomous, each having their own local memory and resources, and therefore the questions of mutually exclusive access to data structures referring to the state of the overall system, synchronisation of operation and general consistency have added complexity over the uniprocessor case.

One of the most important questions which must be resolved to gain the maximum benefits of a distributed system, is that of allocating user processes to physical processors, since these are now a multiple rather than a shared resource. Initial work in this field concentrated on placing individual modules which constitute a user

program on different processors in a static manner, taking into consideration the data flow between modules, and attempting to minimise interprocessor communications costs; thus the thread of control of a program would move from one processor to another. This static allocation method, however, does not take account of the load imposed on processors in the system, and does not adapt to changes in the nature of user process arrival rates. Given this observation, researchers have striven to develop systems where the assignment of processes to processors is performed in a manner which balances the overall system load in order to avoid certain processors being overloaded whilst others remain idle.

Load balancing is an intuitively worthwhile goal, but necessitates some means of maintaining a global view of system activity, and a negotiation mechanism for redistributing processes to nodes on a network where they will most benefit in terms of execution time. A simple approach would be to have a central allocation processor which was periodically sent load information from all other processors, and make process placement decisions based on its last-known state of the system. This approach, however, presents a single point of failure and could create a bottleneck. An alternative method is to distribute the responsibility for load measurement amongst all processors and to allow them to cooperate in making process-to-processor allocation. Again this approach is intuitively worthwhile, but adds complexity to the system in dealing with possibly out-of-date state information.

Studies have been made of the various methods which present themselves for solving the aforementioned problems, but they have generally been restricted to an environment where processes are considered as independent entities, with little or no interaction. The aims of the study which we are conducting are threefold : first we develop a simulated network of loosely-coupled processors, with an operating system

kernel, capable of supporting the creation, execution and destruction of groups of user processes, communicating through a message-passing mechanism; secondly we investigate a number of load balancing algorithms, on the simulated network, whose workload is typified by small, independent processes, in order to gain insight into the detailed operation of such algorithms and the effect this has on overall system performance; finally we investigate load balancing algorithms working in an environment of cooperating process groups, and study the relative merits of each algorithm, together with further improvements which can be made by analysis of the trade-offs involved between creating a stable, balanced system and the costs incurred in bringing this about. The structure of the thesis is outlined below.

In Chapter 2, we present a brief overview of issues which are inherent in the design of "supervisory software" for distributed systems, to describe the environment for which our study is targetted.

Chapter 3 examines the approaches taken by other researchers to the problem of processor allocation; the first two parts of this chapter deal with the initial work in this area, based mainly on static allocation policies; the third part reviews approaches taken to adaptively allocate processes to processors, dependent on the current load on the system, and presents methods which have been used to achieve processor cooperation in this task.

Chapter 4 gives details of the design and implementation of both the simulated loosely-coupled system which we have developed, together with the operating system kernel which runs on each processor.

Chapter 5 describes the manner in which the simulated system was used in order to

investigate a number of different methods for achieving adaptive load balancing under varying workloads.

Finally, in Chapter 6, we offer our conclusions, made on the basis of the study which we have performed, as to the relative merits of the load balancing algorithms investigated.

Appendices A and B give detailed results in graphical form of our study of load balancing algorithms, in environments of independent user processes and cooperating process groups, respectively.

The program listing for our simulated system is included in Appendix C, together with a number of technical implementation notes.

# CHAPTER 2

# DISTRIBUTED SYSTEMS

The potential benefits of distributing the processing requirements of a computer system over a number of individual computation units has long been recognised [Kleinrock85]. Such systems can be categorised in a variety of ways [Enslow77], but we have chosen to group them as being either tightly-coupled or loosely-coupled systems.

In a tightly-coupled system, processors share access to a common set of memory modules, with an arbitration mechanism ensuring that the contents of this shared memory remains consistent and, normally, there will be a central clock used for synchronisation purposes. This environment gives very fast interprocess communications facilities for use in fine-grained parallelism, but suffers from contention problems for the shared memory modules when the number of connected processors increases.

A loosely-coupled system, on the other hand, consists of a number of totally autonomous processors, each with its own clock and local memory which is inaccessible to its peers, connected via an external interprocessor communications medium. A processor will also have its own physical resources, such as disk drives, which are under its control, and thus access to these resources by other processors must be negotiated through the use of interprocessor messages. Since the cost of communications between user processes residing on separate processors in this environment is non-negligible, loosely-coupled systems offer only reasonably coarse-grained parallelism in the development of user applications. It is towards this type of system that the work presented in this thesis is directed, where in addition all

processors are identical and will thus allow code written and compiled for one processor to run on any other node in the network. Hence we are interested in loosely-coupled systems of homogeneous processors.

In order to control a loosely-coupled architecture, operating systems need to be designed to handle the additional complexities of truly parallel execution, and to extract its maximum benefits. In the following sections we present a broad overview of operating system considerations, categorised as Network Operating Systems (which have generally been designed as a networking extension to existing system software) and Distributed Operating Systems (which have been designed from their original conception for a distributed environment).

## 2.1 NETWORK OPERATING SYSTEMS

Network Operating Systems (NOS) can be viewed as a first step from the traditional uni-processor design, towards an Operating System whose whole conceptual basis is centred around a fully distributed environment. The early motivation to build such systems was essentially the ability to login to a remote machine and perform file transfers across the network. The first significant example of a NOS was the ARPANET [McQuillan77] which was developed in the 1970s; nodes on this network are very widely dispersed and its main use is for exchange of technical and research data, and the ability to access specialised hardware belonging to other institutions, whose use is not frequent enough to warrant purchasing such equipment. With the rise to prominence of the UNIX operating system [Ritchie74] there have been many networked versions, with varying degrees of sophistication, developed by research groups and commercial concerns [Blair82; Brownbridge82; Karshmer83; Luderer81;

Rowe82].

NOSs are generally constructed by placing an extra layer of software either "on top of" or "within" an existing conventional operating system, to provide facilities for accessing remote resources; the distributed nature of the system is not hidden from the user, and so such operations as file access and program execution are performed with very little transparency (the actual location of objects is explicitly stated in the naming mechanisms used).

To enable processes to access data remotely, a file system suitable for a network of loosely-coupled nodes is required. In a typical NOS one can identify two main types of file system. The least sophisticated of these provides no access to remote files by system calls, but instead uses simple file transfer software to move the data concerned from the processor on whose secondary storage it is held onto the processor which requires access to it. Examples of this are the "uucp" program [Nowitz79] to copy files from one UNIX system to another, and the "Kermit" package, designed to carry out file transfer across a variety of manufacturer's computer systems. A more sophisticated solution is to establish some form of "super root" for the file system which spans the whole network and can be thought of as being placed "above" each processor's own rooted file system assuming a hierarchical structure; for example a pathname given as "/../mc1/ian/project/kernel.c" will refer to a file called "kernel.c" residing in a directory "ian/project" on the "mc1" processor (with "mcl" placed directly "below" the super root). Two useful examples of such file systems are those of the Newcastle Connection [Brownbridge82] and Sun NFS [SUN86], which are both enhancements of standard UNIX systems and hence we class them as NOSs, since they were not designed from scratch for a distributed environment.

A further noteworthy aspect of a NOS is the question of program execution. A very elementary solution is to require users to explicitly login to a remote machine, and then to execute programs in an entirely local fashion (thus needing very little additional operating system support). Alternatively a facility may be provided enabling the user to execute programs remotely whilst remaining logged in to just one machine; in this case the environment of the program (e.g. current working directory, user privileges etc.) are transferred to the processor on which the program is to execute, and when execution terminates, the environment of the user returns to his local machine. Another approach is to have a system call, say, `create_process()`, to which a parameter is passed, giving the identity of the processor on which the process should execute; this remote process could then make a system call similar to the UNIX `exec()` to execute a program.

## 2.2 DISTRIBUTED OPERATING SYSTEMS

We shall now consider the design of operating systems intended, right from the conceptual stage, to be run on a distributed machine architecture. Such systems are typified by their highly transparent user interface, hiding the underlying structure of the network, providing what appears to be a single, large, powerful machine. A number of important considerations can be identified which are raised by the distributed nature of a collection of autonomous loosely-coupled computers:

- communications primitives
- naming conventions
- file systems
- fault tolerance
- resource allocation

### 2.2.1 Communications Primitives

Since each processor has its own local memory which is inaccessible to its peers, interprocess communication must be achieved via a message-passing mechanism, provided by system calls in the operating system kernel. Considerable effort has been made by researchers [Liskov82] to determine the relative merits of making messaging primitives (i.e. SEND and RECEIVE) function in a blocking or a non-blocking manner. The advantage of non-blocking primitives is that they allow flexibility in program writing, since computation can be performed in parallel while waiting for I/O operations to complete; however this does have drawbacks in that it introduces non-determinism into process execution, because ordering of events may vary for different scheduling situations. Blocking primitives have often been used to implement a client-server model of computation, where a "client process" sends messages requesting a specific operation to be carried out by a "server process"; when the server has processed the request, it sends results back to the client. As can be seen this is similar to the procedure call-return mechanism in a sequential program and has been used to implement Remote Procedure Call (RPC) facilities [Saltzer84].

### 2.2.2 Naming Conventions

In centralised systems, information concerning the state of objects is held in readily available central tables, but in a fully distributed system, this is often not possible and is anyway not desirable since this would create a single point of failure in the network. One solution to this problem, is to elect a processor as a name server which is then used to map locally-known names onto unique network identifiers [Needham82]; additional robustness can be achieved by passing this responsibility onto another processor if the main server goes down. Alternatively a broadcast mechanism could

be used, where the name of the object to be accessed is sent to every other processor, and a reply is sent back from the processor holding that object; however this does tend to create added interprocessor communications overheads.

### 2.2.3 File Systems

In contrast to most NOS file systems, in a true Distributed Operating System, the location of a file is not usually included as part of its pathname (analogous to the fact that in a centralised system, a file's physical location on a disk is not mentioned in that file's name). The operating system implicitly handles the local or remote nature of a file, and it is not necessary for the user to explicitly mount remote files into local directories. As a consequence of this access transparency, files can be freely migrated around the network without changing their accessibility to user processes; this migration has advantages in that it allows easy network reconfiguration, but may result in files being moved far away from the processes wishing to use them, and so this must be managed very carefully.

In order to increase resilience in the face of processor crashes, many systems use replicas of files which can be used when the original becomes corrupted or inaccessible; this introduces the problem of keeping copies up-to-date, i.e. consistent [Svobodova84]; depending on the application's environment this consistency can be either weak (files are eventually up-to-date) or strong (files are guaranteed to always be up-to-date).

### 2.2.4 Fault Tolerance

Due to the separate nature of components in distributed systems, they offer great

potential for building fault-tolerance into the operating system: one processor going down shall not have the catastrophic effect it does in a centralised system; "back-up" processes and processors which are activated when crashes occur can significantly increase the "up time" of the network. Multiple versions of files can also be maintained, and considerable research has been conducted into methods of providing atomic actions on these files [Lampson81], and solving the problem of serialising access to ensure that the user processes' view of a file remains consistent.

### 2.2.5 Resource Allocation

A feature of operating system design which has extra difficulty and importance in a distributed system is that of allocating resources to processes; since processes on one processor can now request and be given resources residing remotely, there must be some form of co-operation strategy employed to ensure that this environment is correctly maintained. Indeed since no central resource tables exist, problems of synchronisation, deadlock and efficient resource management become even more complex and error-prone, as the controlling system may have to deal with out-of-date information regarding the state of the system.

One vital resource whose allocation must be carefully considered is the CPU of each node on the network. Since we now have the potential of running user processes truly in parallel, rather than the pseudo-parallelism of time-shared uni-processor systems, the full benefits of this can only be realised by sensible process-to-processor assignment. We can identify two main approaches to solving this task, and we term these network scheduling policies, and load balancing policies. In a network scheduling policy, some co-operation is needed to ensure that the correct processes are being scheduled on each processor at any one time; hence the problem is viewed as in

a traditional uni-processor system, but with the added complication that the "process table" is now distributed across the network, thus making the finding of available slots a much more difficult exercise. Load balancing policies, on the other hand, assume that each processor takes care of its own process scheduling, independently of other processors, and that the most important consideration is where to place processes on the network and, once placed, whether they should be migrated to an alternative processor as the nature of the system load changes. This area has received much interest and remains a fertile field for research; it will be discussed in considerably more depth in the next chapter, since it forms the basis for this thesis.

### 2.2.6 Example Distributed Operating Systems

In the following sections we present a brief description of three Distributed Operating Systems, which we believe to have features, desirable in the environment for which we are studying load balancing algorithms. In particular such features as distributed file systems (LOCUS), process migration mechanisms (DEMOS/MP) and the creation of cooperating process groups (V-System). This is far from an exhaustive list of such systems currently available, but highlights certain essential considerations.

#### 2.2.6.1 LOCUS

The LOCUS Distributed Operating System [Walker83; Popek85] was developed at UCLA to run on an Ethernet network [Metcalfe76] of 17 Vax-11/750's , in a manner which makes it upward compatible with UNIX. Of particular note in LOCUS, are the mechanisms provided for maintaining a network-wide file system and for creating and executing processes remotely.

The LOCUS file system is a single-rooted tree, consisting of a collection of filegroups, where a filegroup corresponds to a standard UNIX file system. The tree is constructed by a series of mount operations with a considerable amount of replication of directory entries to increase tolerance of the system to individual site failure and network partitioning. The user interface to this tree structure is entirely transparent down to the system call level, in that the physical location of a resource which is being referred to, cannot be discerned from its pathname. All file operations are divided into three logical parts defining the Using Site (the site from where the file is being accessed), the Storage Site (where the file is physically held) and the Current Synchronization Site (the site responsible for controlling mutually exclusive and consistent access to the file). This division is transparent to the user process which uses the standard UNIX file operations in the usual manner, leaving the operating system to resolve all location and consistency issues. Resistance to system faults is increased by allowing file replication. In order to maintain file consistency a shadow page mechanism is used to keep both the original and changed versions of a file until an atomic file commit operation is performed; facilities are also included for resolving conflicts due to inconsistent versions of replicated files after the network has been partitioned and subsequently recovered [Parker83].

Processes can be created with equal ease either locally or remotely, with process location being entirely under the control of the user (thus the operating system does not perform any kind of load balancing). The usual method for remote process execution is via a local `fork()` system call (as in UNIX) followed by a remote `exec()`; a token mechanism is used to support system data structures shared between parent and child processes (such as open file descriptors which are passed to the child upon its creation). UNIX provides a number of software signals to announce either child or

parent process failure, and in LOCUS these have been augmented by signals referring to failure of the node on which a parent or child process is executing.

#### 2.2.6.2 DEMOS/MP

The DEMOS/MP operating system [Powell83], developed at the University of California in Berkeley, is a modification of the DEMOS [Baskett77] system allowing it to run in a loosely-coupled distributed environment; the addition of an efficient process migration mechanism is of particular interest and relevance to our work. The designers of the system intended that such a mechanism would be useful for load balancing considerations, moving processes close to the resources which they use, and providing fault tolerance; they also point out that operating systems which efficiently allow process migration are a rare commodity.

All interactions between processes in DEMOS/MP are message-based, including those between a processor's kernel and its user processes and between kernels resident on separate processors. The kernel of a processor is responsible for controlling access to its own local resources, but kernels cooperate via message-passing to provide a user process with a network-wide transparent interface. Whilst the kernel provides all primitive functions of the system such as process execution and the message-passing mechanism, all higher-level operating system tasks are carried out by a number of system server processes.

The fundamental underlying feature of DEMOS/MP which implements the message-passing primitive operations is that of the "link". A link is an address which specifies the destination process to which a message should be sent, and this address consists of a network-wide unique process identifier, together with the identity of the

processor which was the last known location of that process. Whilst the former of these two parts of a link never changes, the latter will be modified when a process migrates from one processor to another. The migration of a process will thus involve updating all network-wide links to that process (since its last known location will now be different); in addition to this, any messages for the migrant process must be guaranteed delivery, even if they were sent to the process's previous location. In order to deal with the problem of message re-routing, DEMOS/MP places a forwarding address on the processor from which a process has just migrated; any messages which had already arrived for the migrant process are transmitted along with its code and data segments when it moves, and any subsequent messages will be sent on using the forwarding address. Since the designers of DEMOS/MP wished to avoid searching the entire system in order to update all links which are out-of-date due to process migration, whenever a message is forwarded, a further special message is sent indicating the new value for the link for which the message was originally intended. In this manner, links to a migrant process are all gradually brought up-to-date.

Since all requests for service from the kernel are made via the link mechanism, this makes process migration considerably easier, and thus preempting a process and moving it to another location can be performed at very little extra cost (since the only information which needs to be sent to a process's new location, in addition to its code and data, are its swappable and non-swappable state which amount to 600 bytes and 250 bytes respectively).

**2.2.6.3 The V-System**

The V-System [Berglund86; Cheriton84] was developed at Stanford University as a distributed operating system for a number of SUN workstations connected via

Ethernet. Its basic philosophy is to create an environment where inexpensive processes can run with an efficient interprocess communications mechanism. The V-Kernel carries out operations which can be categorized as process and memory management, interprocess communication and device management; all other system resources are controlled by server processes which implement a client/server model of interaction, by using synchronous send and receive primitives to pass messages between a requesting user process (the client) and the appropriate server process (the server).

Closely cooperating lightweight processes are created as a "team", executing in a team address space under the control of team server processes; the team address space is used for fine-grain data sharing, and all processes in a single team must execute on the same workstation. Processes communicate by using synchronous send and receive primitives to exchange messages directly, or to pass permissions to read from or write to their address space. A number of logically related processes can be collected together to form a process group with a unique group identifier, and the members of such a group may execute on different processors; within a group, the system supports either one-to-one communication or one-to-many communication (i.e. one process broadcasting a message to its entire group [Cheriton84]).

The process group mechanism is used to enable a user to request remote execution of one or several of his processes. All of the team servers are considered to be one large group, and so a request for remote execution is simply broadcast using the one-to-many communications facility, and team servers on available processors reply by announcing their willingness to accept work; when a suitable destination processor has been found, the team server on that processor is sent a message requesting it to begin execution of the required task.

# CHAPTER 3

# PROCESSOR ALLOCATION STRATEGIES

Since it has been shown [Zhou86b] that the CPU is the resource which is most contended for in a computer system, an efficient mechanism must be found for performing processor allocation in a distributed system. Below we present previously studied approaches to this problem, in three main categories :

- distributed scheduling
- static load balancing
- adaptive load balancing

## 3.1 DISTRIBUTED SCHEDULING

The question of how to schedule processes in a distributed manner is normally posed for tightly-coupled multiprocessor architectures [Gonzalez77; Tuomenoksa82; Kain79; Hwang85] where there exists some easily maintainable source of global time control. In loosely-coupled systems it is usual for each processor to independently apply a local scheduling policy to the processes which have been assigned to it in a globally-agreed fashion. The following two approaches illustrate important points in the choice of a processor allocation strategy.

### 3.1.1 The Medusa Approach

Work conducted by Ousterhout [82] for the Medusa Operating System [Ousterhout80] running on the Cm* multi-microprocessor [Swan77] is of particular interest, since it

tackles problems of obtaining maximum parallelism and minimising process waiting due to interprocess communications costs, which are non-negligible in loosely-coupled systems. Ousterhout states that, in a uni-processor system, assuming a client-server model of process interaction, a process which requests a service from one of its peers must relinquish the CPU so that the servicing process may be allowed to run, thus incurring the overhead of two context switches for this operation. If, however, we have a multiple processor system and the two processes are executing on different processors, these context switches may not be necessary, since they may run truly in parallel. In order to gain the full benefits of this fact, processes which interact in this manner, form a process working set and should be scheduled at the same time, similar to the way in which page working sets should be co-resident in virtual memory management systems [Denning80]. Identifying process working sets dynamically is too difficult and time-consuming so in Medusa the programmer must specify this statically by grouping them into a task force. A task force is said to be coscheduled if all its runnable processes are executing simultaneously on different processors, otherwise it is said to be fragmented.

Ousterhout strives to maximise the degree of coscheduling using three different allocation and scheduling strategies; his model assumes a system consisting of P processors each having Q slots available in their process table, providing a system-wide process space of size $P \times Q$; the assumption is also made that no task force has greater than P processes.

In Ousterhout's first algorithm, known as the matrix method, the available process space is organized conceptually as a Q-row, P-column 2-dimensional matrix, where column p represents the process slots on processor p, and row q contains one process

slot per processor. When a task force arrives at the system, a scan is made of the rows of the matrix, starting at 0, until one is found with enough free slots to accommodate all the processes making up the task force. The scheduling policy also follows the matrix arrangement: in a particular time slice t ($0 \leq t < Q$), row t is given the highest priority for scheduling on each processor; since task forces are assigned to single rows, this method ensures that all processes in the task force are coscheduled. This procedure continues until all rows have been dealt with, so that after one such sweep, every task force has been coscheduled once, then another sweep begins at row 0 and so on. When a row has been chosen for scheduling, a processor p may find that the corresponding slot [q,p] is either empty or that the allocated process is blocked waiting for some external event (such as input from an interactive terminal); in this case p scans its column of the matrix until a runnable process is found and executes it instead; these processes are known as alternates, and they will generally only constitute a fragment of another task force to the one being coscheduled. Selection of alternates is also performed by the other two algorithms described below.

Since the matrix method assigns task forces only to a single row of the 2-dimensional process space, it results in the creation of a number of unused slots, to which processes will not be allocated, since they cannot accommodate a whole task force; this phenomenon is akin to internal fragmentation in paged memory management schemes. In an attempt to alleviate the rigidity of the above allocation strategy, the second algorithm proposed by Ousterhout, named the continuous algorithm, considers process space as being a contiguous sequence of slots, where P consecutive slots belong to different processors. When a task force is to be assigned to the available machines, a window of width P slots is placed at the left-hand end of the sequence and moved along it (stopping when an empty slot is at the left most end of the window, with a full slot directly to its left) until the number of empty slots (not necessarily

contiguous) is sufficient to allocate the entire task force. The P-slot window is also used for scheduling purposes. For each time slice, the window is moved to the leftmost process of a task force, which has yet to be coscheduled. Moving the window in this manner, rather than singly on to the next task force regardless of its previous scheduling activity was found to give the most equal treatment to both small and large task force sizes. If slots within the scheduling window are empty or contain unrunnable processes then alternate selection is used as in the matrix method. The continuous algorithm packs task forces more tightly than the matrix algorithm and hence reduces internal fragmentation; however as unallocated holes, dispersed over a wide distance, are used for newly arriving task forces, this results in external fragmentation, and Ousterhout shows by simulation that this phenomenon seriously degrades performance.

In order to remove external fragmentation, the undivided algorithm, the third proposed by Ousterhout, uses the same method for processor allocation, except with the proviso that a task force's processes must occupy contiguous slots in the available space. Although this algorithm does not pack processes quite so densely as the continuous method, it does reduce external fragmentation.

In the simulation experiments conducted over a wide variety of system parameters, it was found that the undivided algorithm performed consistently the best of the three, with the continuous algorithm performing worst, probably due to its external fragmentation problems; the matrix method's performance was not substantially worse than that of the undivided approach, and, due to its ease of implementation, it was chosen for the Medusa operating system.

### 3.1.2 The Wave Scheduling Approach

Van Tilborg and Wittie [Van Tilborg81] have developed a method of assigning processes to processors for the MICROS operating system [Wittie80] running on the reconfigurable, multi-microprocessor system MICRONET at the State University of New York [Wittie78]. For simplicity, they assume that nodes in the network are monoprogrammed, and that all processes to be executed are of unit-size. Process arrival is in the form of task forces, which consist of sets of related processes, co-operating to perform a specific task. Given the above assumptions, the problem of processor allocation becomes one of locating sufficient idle nodes in the network to accommodate an arriving task force; the approach used, utilises the hierarchical control structure of MICROS (which is a logical hierarchy and hence does not assume any particular physical interconnection topology) to carry out this allocation. Task forces may arrive at any "manager" node in the network, whose job it will be to find enough idle "worker" nodes on which to execute them.

When a task force requiring S worker nodes arrives at some node n (S being statically specified by the programmer), this node becomes its Task Force Manager (TFM) and strives to reserve for it, the resources it needs. Node n will have a reasonably up-to-date view of available workers in the subtree which it manages, since status information regarding the number of idle nodes is regularly passed up the "hierarchy of command". The TFM computes a value $R \geq S$, which is the number of workers it needs to reserve in order to be fairly sure that it will receive sufficient workers to execute the task force. Choice of R must be very carefully calculated, since if R is too big then a large number of unrequired nodes will be reserved, only to be relinquished later; if R is too small then the necessary number of nodes S may not be reserved, resulting in an unsuccessful scheduling attempt. The request for R nodes is then

divided into a number of subrequests, and these are passed down the hierarchy to the TFMs subordinate managers; this wave of requests permeates down until it reaches the lowest-level managers which have accurate, fully up-to-date information regarding worker availability. These managers then reserve the required number of workers and report this fact to their superior manager. In order to avoid deadlock, TFMs wait for a time-out period, and then assume that their request could not be satisfied. If at the end of the time-out period, the number of worker nodes reserved is greater than S, then the TFM sends a message back down the hierarchy, telling all managers to begin execution of the task force, and to release any unnecessarily reserved workers; otherwise it sends a message releasing any reserved workers and puts the task force back in the queue to try again later. If the number of unsuccessful scheduling attempts exceeds some threshold, then the task force is passed one level up the hierarchy to increase its chances of finding sufficient resources.

Using this method, the average cost ($\overline{T}_S$) of successfully reserving a task force of size S, can be expressed as:

$$\overline{T}_s = C_h * ( \overline{n}_f * \overline{F} + \overline{n}_x )$$

where

$C_h$ = cost of reserving a single node at level h of the hierarchy

$\overline{F}$ = mean number of failed scheduling attempts

$\overline{n}_f$ = mean number of nodes reserved on a failed scheduling attempt

$\overline{n}_x$ = mean number of excess nodes reserved on a successful scheduling attempt.

These mean values obviously depend on the probability of finding any particular node

idle at a particular time t; since each high-level manager regularly receives summary information regarding node utilisation, it is possible to estimate the above probability with reasonable accuracy and then to calculate an optimal value of R by the equation :

$$R_{opt} = \underline{a} * S$$

where $\underline{a}$ is some simple function of node utilisation. Van Tilborg and Wittie found that even constant values for $\underline{a}$ (calculated from previous experience) were sufficiently good for node utilisation up to about 70%. The performance of the algorithm was shown to perform satisfactorily relative to a Markov queueing model of a central scheduler having total knowledge of system-wide node availability.

## 3.2 STATIC LOAD BALANCING

The approaches to allocating processes to processors presented above, do not explicitly take into consideration the execution costs of such processes, neither do they make allowance for the overheads incurred by communicating with their peers or accessing files across machine boundaries. Typically in a computer network, interprocessor communications costs are significant relative to intraprocessor costs and will have a substantial effect on system performance. In order to reduce these overheads, an allocation strategy must be designed to calculate an assignment of processes to processors which minimizes execution and communications costs.

Early solutions to this problem assume a program to consist of a number of modules, which may be run separately on any processor in the network, and which exchange data by some communications mechanism. The cost of executing a particular module

on a particular processor is assumed to be known a priori, as is the volume of data which will flow between modules. The problem then becomes one of assigning modules to processors in an optimal manner within the given cost constraints. Since such approaches do not consider the current state of the system when making their placement decisions, they are referred to as static load balancing algorithms and can be grouped into three major categories: graph theoretic, 0-1 integer programming and heuristic. Although these approaches have many aspects in common, this categorisation illustrates the different conceptual views taken of the problem.

### 3.2.1 The Graph Theoretic Approach

In the graph theoretic approach [Stone77; Stone78; Rao79; Bokhari81] to static load balancing, a program's modules are represented by nodes in a directed graph, with edges in the graph used to show intermodule linkages and weights on these edges giving the cost of sending the appropriate volume of data from one module to another, if they reside on separate processors (intraprocessor communications costs are normally assumed to be zero).

Stone [77] recognised the similarity between this model of program structure and work carried out for commodity flow networks. Such networks consist of a number of source nodes which are capable of producing an infinite quantity of some commodity, which is directed to a number of sink nodes, capable of absorbing this infinite quantity; edges from source to sink, via intermediate nodes, represent a commodity flow through the network and weights on these edges give the maximum capacity of an edge. The sum of the net flows out of the source nodes, and hence into the sink nodes, is termed the value of the commodity flow. A feasible flow through the network has the following properties:

i) the sum of flows into a node is equal to the sum of flows out of that node

ii) flows into sinks and out of sources are non-negative

iii) flows do not exceed capacities

If a flow is found which is maximum among all feasible flows, then it is known as the maximum flow. A cutset determines the set of edges which, when removed from the network, totally disconnect source and sink nodes, and the weight of a cutset is defined as the sum of the capacities of edges in the cutset.

Fig. 3.1 shows an example of such a network, where circles labelled with letters represent modules of a program, and edges between these modules are labelled with numeric weights to show the amount of data passing from one module to another; processors are represented by the two nodes S1 and S2. In this example two-processor system, edges are added from each "module" node to each "processor" node, such that the weight of an edge from a "module" node to (say) $S_1$ gives the execution cost of that module on processor $P_2$ and vice versa (if a module cannot be executed on a processor then the weight on the appropriate edge is set to infinity). Execution costs of a module will vary from processor to processor dependent on facilities available, for example specialised floating point hardware. The minimum cutset for the network in fig. 3.1 is shown as a broad line. Stone shows that a cutset in this graph represents a particular module assignment ($S_1$ and $S_2$ in separate partititons), and that the weight of a cutset is therefore equal to the cost of the corresponding module assignment. Hence an optimal assignment can be found by calculating the minimum weight cutset in the graph, using an algorithm developed by Ford and Fulkerson [62]. If one of these two processors has limited memory, then the

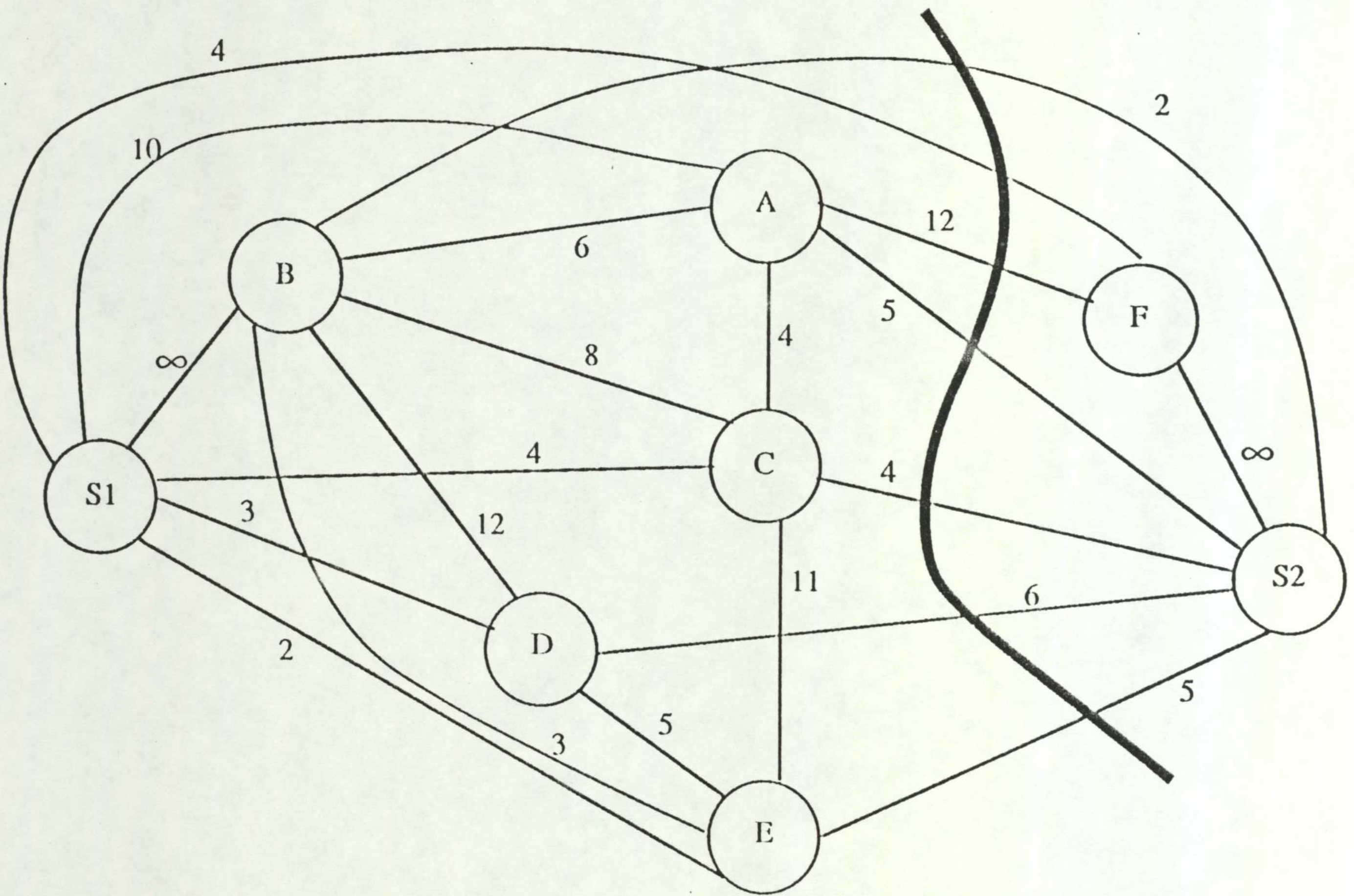



Fig. 3.1 Graph Theoretic Module Assignment and Minimum Cutset for Two Processors

problem becomes much more complex, but it has been shown that techniques exist for complete solution of certain problems under memory constraints and a reduction in complexity for others [Rao79].

This network partitioning method can find an optimal assignment reasonably efficiently for two processors, and can theoretically be extended to an n-processor network, where n cutsets need to be found; this can be done by exhaustive enumeration, but its computational complexity makes it thoroughly undesirable. Stone suggests that the n-processor problem could be considered as a number of two-processor assignments, but significant difficulties exist in this solution, since nodes may be placed outside a minimum cost partition by successive two-processor solutions.

Although the max-flow, min-cut algorithm presented above is not easily extendable to n-processor systems for general programs, it has been shown [Bokhari81] that an efficient implementation is possible if the program has a tree-structured call graph (known as an invocation tree) for its modules. Again assuming that all execution and communications costs are known, an assignment graph can be constructed from the invocation tree, where each node represents assignment of a module to a processor shown by a pair of numbers (ie (i,j) means that module i is resident on processor j); an edge between nodes (i,p) and (j,q) has a weight equal to the cost of assigning module j to processor q, given that module i has been assigned to processor p. An algorithm which then finds the least costly path through the assignment graph, constructs an assignment tree which gives the optimal module-to-processor allocation; such an algorithm has been shown by Bokhari [81] to execute in time $0(mn^2)$ where m modules are assigned to n processors.

Chou and Abraham [82] also use a graph model of module assignment, with nodes in the graph representing modules, but edges are used to indicate precedence relations. They introduce probabilistic branch points where the flow of the program may follow any one of two or more branches under a probability constraint; at fork points, execution will continue along each of the possible branches. A semi-Markov process with rewards is used to model dynamic execution of the program and this is augmented by "policies" which indicate module-to-processor assignments. By iteratively examining the possible state transitions under each policy it is possible to find an optimal assignment for n-processor systems. Concurrent module execution is also built into the model. The disadvantage of this method is that it relies heavily on the accuracy of available data regarding program behaviour and the authors recognise that load balancing which adapts its placement decisions dynamically with the current system state is desirable, but would be too costly using their policy iteration algorithm.

### 3.2.2 The 0-1 Integer Programming Approach

Due to the limitations of the graph-theoretic approach, other researchers [Chou82] have adopted an integer programming method for processor allocation. Again in this model, it is necessary to identify the execution and interprocess communications of modules, hence the following quantities are used:

$C_{ij}$ : coupling factor = the number of data units transferred from module i to module j

$d_{kl}$ : interprocessor distance = the cost of transferring one data unit from processor k to processor l.

$q_{ik}$ : execution cost = the cost of processing module i on processor k.

If i and j are resident on processors k and l respectively, then their total communications cost can be expressed as $C_{ij} * d_{kl}$. In addition to these quantities the assignment variable is defined as:

$$X_{ik} = \begin{cases} 1, & \text{module i is assigned to processor k} \\ 0, & \text{otherwise} \end{cases}$$

Using the above notation, the total cost of processing a number of user modules is given as:

$$\sum_{i} \sum_{k} \left( q_{ik} X_{ik} + \sum_{l} \sum_{j} \left( c_{ij} * d_{kl} \right) X_{ik} X_{jl} \right)$$

The major advantage of a programming solution to static load balancing is that constraints can be easily incorporated into the model, which is difficult, if not impossible, using graph theoretic techniques. For example, to ensure that processor k has sufficient memory available to process modules assigned to it the following constraint can be applied:

$$\sum_{i} M_i X_{ik} \leq S_k$$

where $M_i$ = memory requirements of module i

$S_k$ = memory capacity of processor k

Constraints such as real-time requirements and processor speeds can also be expressed

in a similar manner. Module allocation can then be performed by minimising the above cost equation subject to the constraints imposed, by non-linear programming techniques, or further constraints can be added to linearise the problem. It has been shown [Chu80], however, that on a CDC 6000 series mainframe, a problem involving 15 processors and 25 modules will take a few minutes to solve, and should hence be performed off-line in realistic environments.

An alternative approach is to use a branch and branch method [Ma82] to construct and search in a depth-wise manner, a tree of possible assignments; hence for allocating m modules, a tree of m levels is constructed where a branch at each level represents assignment of that module to a particular processor. As this tree is being expanded, the constraints imposed on a solution are applied via branching nodes, thus eliminating the necessity to further expand certain branches, since they do not satisfy the constraints. A path from the root of such a tree to a lowest level node represents a complete assignment for all modules, and the optimal assignment is the lowest cost path. It is known that an optimal solution to searching a tree is an NP-complete problem, but Ma et al [82] show that the elimination of certain branches using constraints reduces the complexity considerably.

The major disadvantage of integer programming techniques using constraints, is that they are heavily parameterised and require substantial effort on the part of a system designer in specifying which constraints should apply in order to achieve a realisable load balancing solution.

### 3.2.3 The Heuristic Approach

Since finding optimal solutions to the module assignment problem is so

computationally expensive, a number of heuristic approaches have been proposed to find a suboptimal solution [Gylys76; Efe82; Lo84]. The essence of these algorithms is to identify clusters of modules which pass a large volume of data between them and to place them on the same processor.

One such algorithm [Gylys76] finds the module pair with most intermodule communication and examines whether the constaints imposed on their execution allows them to be coresident on one processor. This continues until all possible pairings are found, but has the problem that it does not guarantee that the resultant number of clusters found will not be greater than the number of available processors.

A variation on this approach again due to Gylys [76] is to define a "distance function" between modules, which is a measure of communications between two modules i and j, relative to communications between i and all other modules and j and all other modules. Using this function a "centroid" of a possible candidate cluster can be found, and an iterative algorithm is then used to join modules having the lowest valued distance function from the centroid into the centroid's cluster. The centroid is then appropriately adjusted to take account of this addition. The iterations are stopped when an upper limit is reached or when no module clusters change.

This concept has been extended by Efe [82]. In his algorithm, clusters are formed as above, whilst recognising that certain "attached" modules must be executed on a single processor or a subset of the available processors, and hence these modules can form the "centre" of a cluster. When clustering is completed, a queue length constraint can be imposed on each processor, and modules are then moved from processors whose load lies above the expected average plus some tolerance threshold onto a similarly underloaded processor, whilst still maintaining the restrictions of interprocessor

communications cost. Efe shows that an optimal assignment with respect to execution cost and communications cost may not necessarily result in the most efficient assignment when queue lengths are considered.

This additional constraint is also investigated by Lo [84], who identifies an extra cost to be considered, resulting from the contention for shared resources, such as CPU cycles in a multiprogramming environment, which she terms the interference cost defined as :

$$I_q(i,j) = I_q^p(i,j) + I_q^c(i,j)$$

where

$I_q(i,j)$ = total interference cost of executing modules i and j on processor q

$I_q^p(i,j)$ = processor interference cost of i and j on processor q

$I_q^c(i,j)$ = communications interference cost due to contention for the communications mechanism by i and j running on q

Lo also states that an increase in parallelism (i.e. running i and j on separate processors) reduces interference costs and hence assignments should have a weighting which consists of execution, communications and interference costs. This is achieved by limiting the number of clusters which can be assigned to a single processor.

It has recently been suggested [Chu87] in a study of static load balancing algorithms for real-time systems, that since queueing delays at a processor are non-linear with increasing load, then the major limiting factor on system performance is a single bottleneck processor which becomes overloaded; hence improvement can only be achieved by minimising the chances of a bottleneck being created. A heuristic method of ensuring this, is to create module clusters in a manner similar to Lo [84], and then to solve the equation:

$$\min_{x} \left\{ \max_{1 \leq r \leq s} \left[ \text{EXEC } (r;x) + \text{IPC } (r;x) \right] \right\}$$

where EXEC (r ; x) is the total execution costs of modules assigned to processor r, and IPC (r ; x), the total communications costs of this assignment.

Simulation runs conducted using this method performed significantly better than previous results obtained from simply minimising execution and communications costs.

### 3.3 ADAPTIVE LOAD BALANCING

The limitation of static load balancing is that this method assigns processes to processors in a once-and-for-all manner and solutions require a priori knowledge of program behaviour; most approaches ignore the effects of interference in a system comprising multiprogrammed nodes. Livny and Melman [82] have shown that in a distributed system the probability that at least one process is waiting for service at one node, whilst at least one processor is idle can be calculated as:

$$P_{wi} = \sum_{i=1}^{N} \binom{N}{i} Q_i H_{n-i} = (1-P_o^N)\ (1-P_o^N - (1-P_o)^N)$$

where

| | |
|---|---|
| $Q_i = P_o^i$ | is the probability that i processors are idle |
| $H_i$ | is the probability that i processors are not idle and one process waits for service |
| $P_o$ | is the probability that a processor is idle |
| N | is the number of processors |

They propose that if $P_{wi}$ can be reduced by transfering processes from one processor to another, then the expected turnaround time for processes in the system will also be reduced; it is also to be noted that for systems with greater than 10 nodes, and with loads ranging from moderately light to moderately heavy, $P_{wi}$ is high, unless process transfer is performed.

This important result, and the conclusions which can be drawn from it, suggest that if the current state of the system can be observed, then by maintaining a balanced load, performance improvement can be achieved. We choose to refer to algorithms which dynamically react to system state in this manner, as adaptive load balancing algorithms. Many researchers have suggested that such algorithms are the most effective way of managing processor allocation [Tantawi85; Carey85; Leland86].

Although the adaptive approach is intuitively worthwhile, a number of new questions are raised. A load balancing algorithm must ensure that it has a reasonably up-to-date

view of the system state; this could be achieved by using a centrally-located allocation processor [Zhou86a], but this gives a single point of failure, and so a fully-distributed solution is favoured ; however care must be taken that co-operation between different processors does not overload the communications mechanism used, as load information is exchanged. In addition, since an adaptive load balancing algorithm transfers processes from lightly-loaded to heavily-loaded processors, it must guard against instability [Kratzer80], caused by many processors all sending processes to the same lightly-loaded node, making it heavily-loaded. In this case processes will spend most of their time migrating around the network, fruitlessly looking for a suitable execution location; this phenomenon has been termed processor thrashing, and is analogous to thrashing in virtual memory management schemes.

We identify the following components of an adaptive load balancing algorithm, and review some approaches taken to providing efficient implementations of these components, and the relevant issues involved:

- processor load measurement
- information exchange
- transfer policy
- co-operation and location policy

### 3.3.1 Processor Load Measurement

In order to begin making sensible placement decisions for processes in a multi-computer environment, it is necessary to have available from the operating system a measure of the current load on each processor; due to the loosely-coupled nature of the systems we are considering in this study, this measure will be calculated independently by each processor, and then communicated through the network to its

peers. Since the value representing a processor's load will be frequently calculated during normal operation, it must be efficiently evaluated and be a reasonable indicator of what service a process will receive running on that processor. Also the value should adapt swiftly to changes in load state, but not so much that out-of-date information will be held at other locations in the network [Alonso86]. If possible, the method of load measurement used in a policy should be generalizable so that it can be used in a variety of operating system environments.

One simple solution to this question is to use a specialised load estimation program [Ni85], which constantly runs, determining the time intervals between periods where it successfully acquires use of the CPU; if the interval is great, then processor load is high, and conversely, if it is small, then this indicates low processor load. Although this approach is very easily implementable, it suffers from the problem that it relies heavily on the local process scheduling discipline used, and may therefore not provide a sufficiently accurate estimate of load; additionally it introduces a further process onto each processor, which goes against the principle of trying to improve system performance.

A measure used by the Maitre d' [Bershad85] load balancing program is the UNIX five-minute average which gives the length of the run queue exponentially smoothed over a five-minute period. This value gives a gross indication of processor activity but does not respond quickly to load changes; a quantity which does so, is the number of processes ready to run on the CPU at a given instant (instantaneous processor load), but this will fluctuate very rapidly, because many processes may be waiting for I/O operations to complete, thus giving the false impression of a lightly-loaded processor; this problem can further be exacerbated when process migration is introduced to offload processes from a heavily-loaded processor, since these will not yet be included

in the recipient processor's ready queue. The problem is thus one of maintaining stability. We adopt the definition of a stable system due to Kratzer and Hammerstrom [80], as being one where the costs of adaptive load balancing do not outweigh its benefits over a system using no load balancing. Such a situation would be caused by a large number of fruitless migrations, resulting from use of out-of-date or inaccurate state information. In order to use a measure with a reduced fluctuation, it has been suggested [Krueger84] that the instantaneous load value should be averaged over a period at least as long as the time necessary to migrate an average process. Extra stability is then introduced by using a virtual load value, being the sum of the actual load on a particular processor augmented by the number of processes currently in transit to that processor.

The local load measurement used by Barak and Shiloh [85a] is a further enhancement of the instantaneous load value. A time period t is divided into a number, $\mu$, of atomic time units or quanta of length q; if $W_i$ is taken to be the number of ready processes on a processor in the time interval $(q_{i-1}, q_i)$, $i = 1, 2, \ldots, \mu$, and if $\omega$ of the $\mu$ quanta were unavailable due to operating system overhead, then the load over time t (denoted by $V_t$ ) can be given as :

$$V_t = \frac{\sum_{i=1}^{\mu} W_i}{\mu - \omega}$$

Bryant and Finkel [81] have proposed that if the remaining service time of processes can be estimated (i.e. the time which they still require to complete execution), this value can be used to calculate the expected response time for a process arriving at a

processor, and that this is an indication of processor load. They investigated the use of probability distributions in the evaluation of remaining service time, of a process at time t (denoted by $R_E$ (t)) but found that a simple and quickly calculated value is to assume that $R_E(t) = t$ (in other words a process is expected to require the same service time as it has already received). If J(P) is used to denote the set of jobs resident on processor P, and we take a job $K \notin J(P)$ then the expected response time of K on processor P (denoted by $RSP_E(K,J(P))$) is calculated using the following algorithm:

```
R : = RE (tk);
for all j in J(P) do
begin
        if RE(tj) < RE(tk)
        then R := R + RE(tj)
        else R := R + RE(tk)
end;
RSPE(K,J(P)) := R;
```

Hence this method of calculation can be used to provide an estimate of a processor's load by evaluating $RSP_E(K',J(P))$, where K' is a job whose remaining service time is equal to the average overall service time of processes in the network.

The queue of ready processes is not the only queue which gives an indication of the activity on a processor. Ferrari [85] has studied the possibility of using a linear combination of all main resource queues as a measure of load, where coefficients in

this equation are given by the total time a process spends in a particular queue. Employing a queueing model of each processor, and using mean-value analysis, he seeks to define the response time of a given UNIX command as a single-valued function of the current load, in terms of resource queue lengths. In other words, given the expected resource usage of a command "C" on an otherwise empty processor it is possible to calculate how "C" will perform when a known mix of commands is concurrently executing with it. Unfortunately, Ferrari's "load index" (the name he gives to this measure of processor load) assumes a steady-state system, and the queueing theoretic analysis used only holds for certain queueing disciplines for resources. This may well not apply in a practical system [Zhou86b], since process arrival and departure are dynamic in nature. In addition, since the calculation of the load exerted by a command is dependent on its known resource usage, changes in command code will necessitate changing the coefficient values in the load index equation. In fact Cabrera [86] has suggested that load balancing algorithms based on command names will be detrimental to user process performance.

### 3.3.2 State Information Exchange Policy

In order for an adaptive load balancing policy to make placement decisions for processes arriving for service at a particular node, there must be a mechanism by which information regarding processor load (whose measurement was discussed in the previous section) is passed throughout the network. Since the network architecture is loosely-coupled, this information will vary in its degree of accuracy of the true system state since it will be out-of-date, but accuracy must be sufficient to avoid instability (as defined previously); however, frequent load exchange will result in added overhead and will, in the extreme, lead to performance degrading to a level worse than that achievable without load balancing.

#### 3.3.2.1 The Limited Approach

It has been suggested [Eager86] that load information exchange can be very limited whilst still achieving the goal of maintaining a global view of overall system load. In their study of the merits of very simple load balancing policies, Eager et al [86] propose that load information from other processors in the network should only be requested when an individual processor believes itself to be overloaded based purely on local data; in their threshold and shortest algorithms a number of random processors are probed, in an attempt to find a processor to which processes can be offloaded. They showed via simulation that performance improvements are possible even with this limited exchange policy.

#### 3.3.2.2 The Pairing Approach

A "pairing" approach has been put forward by Bryant and Finkel [81]. In their algorithm, each processor cyclically sends load information to each of its neighbours in an attempt to "pair" with a processor whose load differs greatly from its own; the load information sent consists of a list of all local jobs, together with jobs which are currently migrating to the sender of the load message. Under this scheme, the number of such message exchanges and pairings is reduced by introducing a relaxation period when the loads of all neighbours have been queried, in order to avoid excess overhead in this policy.

#### 3.3.2.3 Load Vector Approach

Some researchers have chosen to maintain a load vector in each processor, which

gives the most-recently received load value for a limited subset of the other processors in the network [Hac86]. Load balancing decisions can then be made on the basis of the relative difference between a processor's own load, and the loads of those held in the load vector.

In the adaptive load balancing algorithm [Barak85a] developed for the MOS distributed operating system [Barak85b] at the Hebrew University of Jerusalem, such an information exchange policy has been studied in detail. A load vector L is used, of size $v$, where the first component contains a processor's own local load value, and the other $v$ - 1 components contain load values for a subset of the other processors. Updating the load vector is performed periodically; every unit of time t (which is a parameter of the algorithm) each processor executes the following operations:

| | |
|---|---|
| 1 | Update own local load value |
| 2 | Choose a random processor i |
| 3 | Send the first half of the load vector to processor i. |

When a processor receives a portion of another processor's load vector, it merges this with its own load vector using the mappings:

$$L[i] \rightarrow L[2i] \,,\; 1 < i < v/2 - 1$$

$$L_R[i] \rightarrow L[2i+1] \,,\; 0 < i < v/2 - 1$$

where $L_R$ is the received portion.

It can be seen from this description that the subset of processors whose load values will be known, changes as the above merging is carried out and a load vector may

contain duplicate entries for a particular processor; it is essential to choose an appropriate value for $v$, large enough to ensure that sufficient information is available to each processor, but small enough to avoid unnecessary overheads. In order to investigate choice of both $v$ (the load vector size) and t (the update interval), it was shown that using the strategy described above, at least $\log_2 v$ load vectors need to be received in order to guarantee that a processor's vector is totally updated in time interval T; in a system with a large number of processors the probability that processor X will be selected at the next update period by k processors is approximately equal to 1/(ek!). Using these results, Barak and Shiloh tabulated a number of possible values of $v$, together with the probabilities that a processor's load vector will be updated in a particular time interval. This allows the designer to tune the load exchange mechanism to the characteristics of the system.

An alternative load vector approach is taken in the distributed drafting algorithm [Ni85]. The load of each processor is considered to be in one of three states: light, normal or heavy; a processor holds its most recent view of the load state of its neighbours in a load vector, which is updated when a state transition occurs; possible transitions are light-to-normal, normal-to-light, normal-to-high or high-to-normal (L-N, N-L, N-H, H-N). It would be possible for a processor to broadcast its new state every time a transition occurred, but this would greatly increase network traffic and the algorithm is intended to be network topology-independent, hence it should not rely on an efficient broadcast mechanism being available. A strategy has been developed to minimise the traffic caused by state transitions, whilst still maintaining a reasonably up-to-date account of the state of all neighbouring processors. In order to avoid many messages being sent when a processor frequently changes between H-load and N-load or between L-load and N-load, an "L-load" message is only broadcast to

its neighbours as the N-L transition is made, if the previous processor state was H-load. If a similar approach were taken to N-H transitions, Ni et al [85] show that this could have a detrimental effect on the performance of their algorithm; they choose to broadcast N-H transitions and only notify neighbours of an H-N transition when process migration between two processors is being negotiated, thus further reducing message traffic. They also note it would be easy to relax these restrictions if the underlying communications network allowed efficient broadcasting.

#### 3.3.2.4 Broadcast Approach

Certain adaptive load balancing policies adopt broadcasting for their load exchange component. In the Maitre d' system [Bershad85], daemon processes (i.e. processes whose sole purpose is to listen for and react to events in the system) play a prominent role. One such process executes on each processor and periodically examines the UNIX five-minute load average and decides whether this will permit user processes to be imported from other processors; a processor will then broadcast this availability, and the appropriate daemon process maintains a list of processors currently willing to accept work. Since load balancing is fairly "coarse-grained" and only applies to certain long-running processes which have been modified to run under Maitre d', this method is adequate and does not exert a lot of extra load on the communications subsystem.

Livny and Melman [82] have made a detailed study of algorithms using a broadcast approach. In the simplest of these algorithms, when the load state of a processor changes (in other words on the birth or death of a process), its load value is broadcast throughout the network, thus each processor has an accurate view of all other processors' loads delayed only by the speed of the communications network. They

found that this accuracy of information improves performance for small numbers of processors, but as the size of the network increases, then the additional communications overhead results in performance degradation. To overcome this overhead, a modified version of the information policy was introduced which purely broadcasts a message when the processor becomes idle, thus announcing willingness to accept migrating processes. It should be noted that this form of policy is only applicable to networks with a broadcast communications medium.

#### 3.3.2.5 Global System Load Approach

The information exchange policies presented thus far, have dealt with the exchange of the load values of particular processors. It has been suggested [Krueger84] that rather than exchanging specific local load values, processors should strive to calculate the load on the whole system and to adjust their own load relative to this global value. This approach has the desirable feature of being more able to detect overall heavy and light loads on the system, in which case attempting to move processes from one processor to another may be of little benefit.

The "above average" algorithm [Krueger84] developed at the University of Wisconsin uses a policy which exchanges each processor's view of the global average system load. Whenever a processor's local load is significantly different from this average, and it is unable to find another processor whose load is in the complementary state it modifies its value for the global average, and broadcasts this fact to all other processors; for example if a processor is overloaded and is unable to find an underloaded processor, then the global average value should be increased. The amount by which processor loads are permitted to differ from the average before load balancing is attempted, needs to be set at a level which is not so small that processors

spend most of their time migrating processes in order to maintain their load close to the average, and not so large that many possible fruitful migrations are neglected.

The Gradient Model algorithm [Lin87] views global load in terms of a collection of distances of each processor from a lightly-loaded processor. If the distance between two processors i and j is denoted by $d_{ij}$ then in an N-processor network, the diameter of the network is defined as:

$$D(N) = \max \{d_{i,j} ,\ i \text{ and } j \in N\}$$

A processor i has a gate value $g_i$ which is set to zero if the processor is lightly-loaded or $W_{max}$ otherwise where $W_{max} = D(N) + 1$. The proximity ($W_i$) of a processor is calculated as its minimum distance from a lightly-loaded processor hence:

$$W_i = \min \{d_{i,k} \text{ over } k, \text{ where } g_k = 0\}$$

$$\text{if } \exists\ k \mid g_k = 0$$

$$W_i = W_{max}, \text{ if for all } k,\ g_k = W_{max}.$$

Since a processor's distance from itself is zero, the proximity of a lightly-loaded processor is zero. Global load can then be represented by a <u>gradient surface</u> which is the set of all proximities $GS = [W_1, W_2, W_3, \ldots W_N]$. Such a measure of global load is useful, since it not only gives a network-wide indication of light-loading, but also gives a route to a lightly-loaded processor with minimum cost, from anywhere in the network.

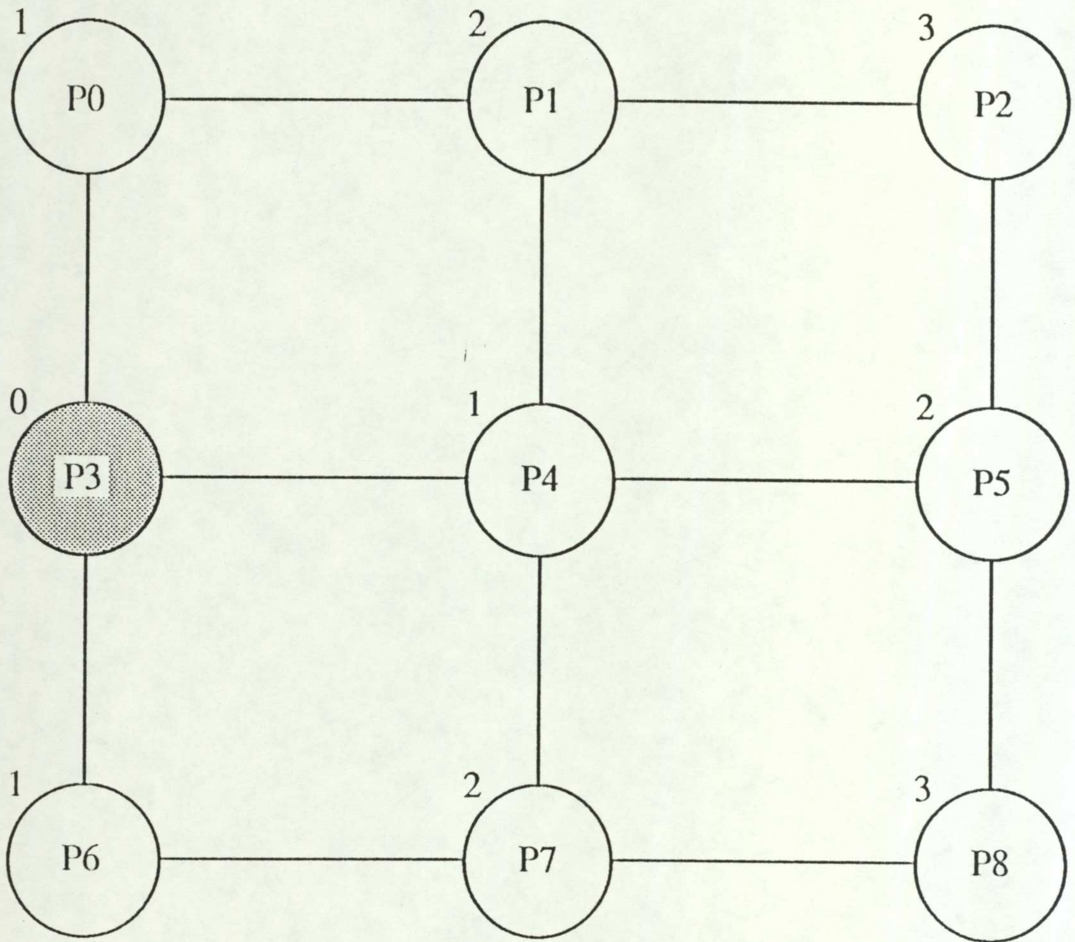



Fig. 3.2 Gradient Surface for the Gradient Model Algorithm in a 9-Processor Network

Since proximities cannot be calculated with absolute certainty due to communications delay in the network, it is necessary to define an approximation to a proximity, which is termed the propagated pressure. This approximation is based on the fact that the information received from a processor's direct neighbour is likely to be more accurate than information obtained from further away in the network. Hence the propagated pressure of processor i ($P_i$) is calculated as:

$$P_i = \min \{g_i, 1 + \min \{P_j \text{ over all } j, \text{ where } d_{i,j} = 1\}\}.$$

So the propagated pressure of a lightly-loaded processor will be zero (as expected), and that of a moderate or heavily-loaded processor will be one greater than the smallest propagated pressure of its direct neighbours. The collection of all propagated pressures is termed the propagated surface and approximates the gradient surface. This method quickly reacts to the absence of lightly-loaded processors in the network, and thus prevents fruitless migrations when the entire network is under moderate or heavy load. An example gradient surface for a 9-processor network, is shown in fig.3.2, where a lightly-loaded processor (P3) is shaded, and the values associated with each processor give its proximity ($W_i$).

### 3.3.3 Transfer Policy

The transfer policy component of an adaptive load balancing algorithm deals primarily with the questions of deciding under what conditions is migration of a process from one processor to another to be considered, and if it is, then which processes are eligible for migration.

A very simple, but effective, method for determining when process movement may occur is to use a static threshold value [Eager86], which, when exceeded indicates that a processor's load is too heavy and work may need to be offloaded to another node in the network. This threshold should be chosen by experimentation to find a load value where performance degrades sharply without load balancing; if the threshold is too high then processors will remain heavily loaded for too long, but if it is too low many pointless migration attempts will occur. Lin and Keller [87] use two thresholds in order to categorise a processor's load as light, moderate or heavy, and consider migration only when the heavy threshold is exceeded.

The above transfer policy strives to identify overloading to trigger process migration, but some researchers [Ni82] have taken the opposite view that an underloaded processor should seek to accept processes from its peers in order to balance the load over the network. One extreme version of this policy is to only consider migration to a processor when it becomes idle. In the distributed drafting algorithm [Ni85] possible process migration is triggered when a processor makes the state transition from normal to light-load, whereupon it indicates its willingness to offload processes from processors whose current state is shown as heavy in its most-recently received entry in the load vector.

Some transfer policies approach this question by using the difference of a processor's load from that of its peers as the major criterion for process migration. In Stankovic's algorithms [Stankovic84] this difference is calculated explicitly and if it exceeds some bias then migration is a viable proposition. The above average algorithm [Krueger84] also bases its transfer policy on difference in load, but achieves this implicitly by maintaining a global average load value, and considering migration when local load differs from the average by a tunable acceptance threshold. Another method used

which implicitly considers load difference is to periodically examine the response time which local processes would receive if migrated to another processor, based on the current estimate of the remote processor's load; if the response time would be significantly better, taking into account the overhead of migration, then movement of that process is desirable [Barak85a].

Once it has been established that a process, or a number of processes, need to be executed remotely, part of the transfer policy is to decide which processes should be moved. A simple and easily implementable method is to only consider newly-arriving processes for migration; this is only really applicable where a threshold is being used in the transfer policy, so that an arriving process causing the threshold to be exceeded is the one chosen for migration. Migrated processes can be treated exactly the same as newly-created ones, but Eager et al [86] have shown that this introduces instability into the system, and under heavy loads, processes may be constantly passed around, trying to find a suitable destination processor; this problem can be alleviated by limiting the number of times that a process is permitted to migrate (Ni et al [85] chose to limit migration to once only).

If the main criterion used in the transfer policy is migrating a process to a processor where its response time will be improved, then the process to be sent is taken as the one which will benefit most from remote execution, assuming that a method is available for estimating a process's remaining service time. In order to be of maximum use, the transfer policy should only migrate processes which have been executing for some minimum amount of CPU time on a particular processor [Barak85a]; this will help maintain stability and prevent a process from migrating too often.

Krueger and Finkel [84] have established a number of essential considerations when choosing a process to migrate:

1. Migration of a blocked process may not prove useful, since this may not effect local processor load.
2. Extra overhead will be incurred by migrating the currently scheduled process.
3. The process with the best current response ratio can better afford the cost of migration.
4. Smaller processes put less load on the communications network.
5. The process with the highest remaining service time will benefit most in the long-term from migration.
6. Processes which communicate frequently with the intended destination processor will reduce communications load if they migrate.
7. Migrating the most locally demanding process will be of greatest benefit to local load reduction.

All of these factors are of varying importance on the effectiveness of a transfer policy, and a load balancing algorithm should incorporate those features which best fit the system environment. Using a preemptive transfer policy (in other words one which migrates executing processes) has considerable advantages in that it adapts more quickly to changes in processor load; however some load balancing algorithms [Ni82; Bershad85] do not use preemption, either because the operating systems for which they have been designed do not support such a facility, or because the costs of such migration are believed to be too high. These costs are significantly reduced if the distributed operating system has been developed with process migration in mind [Powell83].

### 3.3.4 Co-operation and Location Policy

Once mechanisms for measuring local processor load, exchanging load values and deciding when process migration should occur have been established, a load balancing algorithm must define a method by which processors co-operate to find a suitable location for a migrating process. Many categorisations of co-operation and location policies are possible, but we choose to group them into sender-initiated (where an overloaded processor attempts to find an underloaded processor) and receiver-initiated (where the reverse applies) since we feel that this captures the most fundamental differences in approaching the load balancing problem [Eager85].

#### 3.3.4.1 Sender-initiated Approaches

Initiating load-balancing from an overloaded processor is by far the most studied method of co-operation policy. Eager et al [86] examined the question of what level of complexity was appropriate for a load balancing algorithm, by evaluating the performance of three very simple policies which make their migration decisions based purely on local load information. Their goal was not to identify a suitable algorithm in absolute terms, but to analyse the relative merits of varying degrees of complexity; the transfer policy used in all three algorithms is a simple static threshold policy. The simplest of their algorithms chooses a destination processor at random for a process migrating from a heavily-loaded processor, and the number of times that a process is permitted to migrate is limited to once only. Since this policy has no regard for whether the destination processor itself is equally loaded or more heavily-loaded than the source processor, an enhancement was proposed for a second algorithm known as threshold. Under this policy a random destination processor is chosen as before, and this processor is then sent a probe message to determine whether migration of a

process would cause that processor's load to exceed the static threshold; if so, then another processor is chosen and probed, until either an appropriate destination is found, or the number of probe messages sent is greater than a statically set limit; if this limit is exceeded then the process is executed locally. The probe limit was introduced in order to prevent unbounded probing when the global system load is high. In the third algorithm investigated known as shortest, an attempt was made not only to determine whether a potential destination processor would have a load above the threshold, but also to establish the "best" destination; this was achieved by polling a fixed number of processors, requesting their current queue lengths and selecting the processor with the shortest queue.

In order to evaluate the performance of these algorithms under simulation a queueing model was used, and to establish boundary conditions a k-processor network was modelled as k independent M/M/1 queues and an M/M/k queue, to represent the no load balancing and optimal load balancing cases respectively. Results indicated that all three algorithms provided substantial improvement over no load balancing, and further that threshold and shortest provided extra improvement beyond a system load of above 0.5 (where system load of 1.0 is used to denote saturation); also the difference in performance between threshold and shortest was found to be negligible, with shortest performing slightly better. This led the authors to conclude that simple policies are adequate for adaptive load balancing, and that gains to be obtained from additional complexity are questionable.

The Maitre d' load balancing system [Bershad85] uses daemon processes running on each processor to implement its co-operation and location policy. It works on a client/server basis where a local daemon known as maitrd runs on the client processor and negotiates remote process execution with a garçon daemon running on the server

ASTON UNIVERSITY LIBRARY

processor; in a typical configuration all nodes in the network can be both clients and servers depending on their relative loads. Communication between maitrd, garçon and application processes modified to run under Maitre d' is achieved using the socket mechanism of UNIX 4.2 BSD. When the user requests execution of an application, the local maitrd process receives a message at a known socket address; if the UNIX five-minute load average for the local processor is lower than a static threshold, then the application will run locally; if the load average exceeds the threshold, a message is sent to a remote garçon process, which has announced its processor's availability for importing work, again using sockets. The policy used to choose a remote processor is simply the one to which a remote request was least recently sent. When the garçon process accepts the request, it forks another copy of itself to act as a controller for the remote application process and a socket connection is set up back to the originating processor. In this manner both original maitrd and garçon processes can continue listening on their control sockets for further requests. Although this method showed significant performance improvements on the University of California VAX machines, it suffers from a need for application processes to be explicitly modified to use load balancing, and will only work for processes which use their standard I/O channels in a "well-behaved" manner; there is also considerable difficulty in dealing with faulty processes which are part of a pipeline.

Stankovic [84] has proposed three algorithms which are based on the relative difference between processor loads. The information exchange policy used in all three, is to periodically broadcast local load values. In the first of these algorithms, the least-loaded processor is chosen as a potential destination for migrating processes if the difference between that processor's load and the local load exceeds a tunable bias value.

A similar method is used in Stankovic's second algorithm, but in this scheme a processor compares its load with each other processor in turn; for all differences greater than a value bias1, one process will be migrated there; if the difference exceeds bias2 then two processes are moved. In order to prevent instability, a static limit is imposed on the total number of processes that can migrate in a single pass through the load vector.

The third algorithm developed uses exactly the same policy as the first, except that when migration occurs to a particular processor this fact is recorded and no subsequent migrations will be performed to that processor for a time window of ∂t even if it is found to be underloaded. Results were obtained through simulation, and considerable analysis was carried out on the effect of changing the tunable biasses and time window length. The major conclusion of this analysis was that parameter choice is a difficult and crucial question for the algorithms studied, and the second algorithm gave greater performance improvement than the others, only if appropriately tuned. This leads to the suggestion that an algorithm should, as much as possible, adapt to its environment by using variable parameters which do not need to be statically assigned.

The distributed scheduling algorithm designed at the University of Wisconsin [Bryant81], also used the load difference between processors to achieve load balancing. When a processor finds that it has more than two processes currently resident it enters what is termed the "pairing state", where it attempts to locate one of its neighbours to which it can offload processes; in the pairing state, queries are sent cyclically to all neighbours, and a neighbour will respond by accepting pairing if it is sufficiently underloaded (ie it has less than two processes executing on it). Once a pair has been established both processors reject any further queries from their neighbours,

and process migration is performed using expected service time improvement as explained in the previous section on transfer policies. The pair is then broken by common consent. An enhancement to this approach was investigated, whereby queries made during pairing would not be rejected if the querier's load differed significantly from that of the queried processor, but would be postponed until after the pairing state was left, in order to avoid missing possible fruitful migrations. A simulation of 25 processors connected in a square mesh topology showed that this method results in an evenly loaded network, and that even when a lightly-loaded processor is surrounded by heavily-loaded neighbours it does not get "swamped" by migrating processes.

A later algorithm [Krueger84] from the same group of researchers at the University of Wisconsin uses a globally-agreed average load value (as previously described) to negotiate process migration between processors. When a processor becomes overloaded, it broadcasts this fact and waits for an underloaded processor to respond with a message accepting migration of a process to it. The underloaded processor increases its local load value by the number of migrant processes which it believes it is going to receive in order to prevent subsequent overloading, but reduces its load when a timeout period expires indicating that another processor was chosen for migration. If the overloaded processor is unable to find a suitable location for offloading work, then it assumes that the global average value is too low and broadcasts an increased value. The advantage of this approach is that it adapts better to load fluctuations than the co-operation and location policies which migrate processes based on a static threshold for under- and overloading. It has the desirable feature of diminishing load balancing effort when the system is in a generally stable state, and increasing this effort when anomalies in load distribution occur. Simulation of a 40-processor network showed that this method drastically improved both mean processor load and average process

response ratio.

The Gradient Model load balancing algorithm [Lin87] uses the concept of a pressure surface approximation of the current network load distribution to make its process migration decisions. When a processor has calculated its own local load an action is taken depending on whether this is light, moderate or heavy. If local load is light then the processor's propagated pressure is set to zero and propagated pressures from neighbours are ignored; if load is moderate, the propagated pressure is set to one greater than the smallest propagated pressure of all direct neighbours, but no migration is attempted. Migration occurs only at heavy load: if the propagated pressure of all neighbours indicates that no lightly-loaded processors exist, then the network is saturated and migration will serve no purpose; if however this is not the case, then a process is migrated to the neighbour whose propagated pressure is minimal. It should be noted that using this method, processes are not necessarily migrated directly to a lightly-loaded processor, but by definition are guaranteed to migrate towards them following the route implied by each processor's propagated pressure. The algorithm hence strives to achieve global load balancing by a series of local migration decisions; if the intended destination of a migrant process becomes overloaded whilst it is in transit, the algorithm will react to this and divert the process elsewhere. The use of the bounding value $W_{max}$ (as described in the previous section on transfer policies) prevents unnecessary migrations and also serves as an indicator that a processor has failed.

In the MOS distributed load balancing algorithm [Barak85b], a quite complex combination of elements are taken into consideration when choosing a destination processor for local processes, which are caused to contemplate migration in a

round-robin manner by a special system process. Firstly an estimation is made of the response time that a process can expect if it executes on each of the processors whose load is currently held in the periodically-exchanged load vector. This estimate includes communications costs with other processors, where the process's resources may reside and also a weighting to take into account the overheads of transferring the process based on its size. The processor chosen is the one which apparently offers the best expected response time. Tests using this algorithm were conducted using a network of four PDP-11 computers connected by a 10Mbit/second communications ring; a number of I/O-bound and CPU-bound processes were used and observed speed-up in processing was tabulated for various initial process placements; these results showed significant improvement over the no load-balancing case, but it is not clear what effect a larger network, with a greater variety of process behaviour would have.

#### 3.3.4.2 Receiver-initiated Approaches

Receiver-initated location policies work with underloaded or idle processors requesting processes from more heavily-loaded parts of the network. They have been less well-studied than sender-initated policies, but Eager et al [85] showed that they can perform well, especially at high overall system load. Similarly to their work on sender-initiated policies they investigated the level of complexity necessary to achieve efficient load balancing; as in their other study they based their policies on a simple threshold transfer policy. In an algorithm which they term "receiver", when the load on a processor falls below the static threshold (T), it polls random procesors to find one where transfer of a process from that processor would not cause its load to be below T; again unsuccessful probes are constrained by a static probe limit. In an attempt to remove the overhead of migrating an executing process, a modification to

the above approach, known as the reservation policy was investigated. When a processor's load falls below the threshold T it polls its peers exactly as in the "receiver" policy, but instead of accepting a process currently running on a particular processor, a reservation is made to migrate the next newly-arriving process, provided that no other reservations are already pending; a static probe limit is used as above. Simulation results using a queueing model indicated that despite the reservation policies avoidance of costly preemptive migration it did not perform as well as the receiver policy; it was also noted that the receiver policy performed better than an equivalent sender-initiated algorithm at loads greater than 0.7.

The distributed drafting algorithm [Ni85] is also an example of a location policy where lightly-loaded processors seek work from their heavily-loaded neighbours. Since the algorithm is intended to be network topology-independent, the processors which are candidates for migration are defined with regard to communications costs. When a processor enters the light-load state, a "draft request" message is sent to all heavily-loaded candidate processors which then respond with a "draft age" message; the draft age is calculated by considering the characteristics ("ages") of all processes which are suitable for migration and will have a value of zero if the processor's load is no longer heavy. When the original drafting processor has received all such draft ages, it selects the processor which sent the highest draft age value and sends it a "draft standard" message based on the draft ages received. Finally, the receiver of the draft standard will then migrate any of its processes to the drafting processor whose ages exceed the standard; if there are no such processes it replies with a "too late" message. The major drawback of this approach is that it contains a large collection of parameters which must be carefully tuned to suit the network topology (e.g. draft ages, draft standard, time-out periods). A five-processor simulation showed that this method can perform well, with correct parameter choice.

In a study of load balancing algorithms in broadcast networks, Livny and Melman [82] proposed two receiver-initiated policies. Under the first of these a node broadcasts a status message when it becomes idle and receivers of this message carry out the following actions:

Assuming $n_i$ denotes the number of processes executing on a processor i :

1 if $n_i > 1$ continue to step 2, else terminate algorithm.

2 Wait $D/n_i$ time units, where D is a parameter depending on the speed of the communications subsystem; by making this value dependent on processor load, more heavily-loaded processors will respond more quickly.

3 Broadcast a reservation message if no other processor has already done so (if this is the case terminate algorithm).

4 Wait for reply.

5 If reply is positive and $n_i > 1$, migrate a process to the idle processor.

It was thought that this broadcast method may overload the communications medium, so a second algorithm was proposed which replaced broadcasting by polling when idle. In this algorithm the following steps are taken when a processor's queue length reaches zero:

1 Select a random set of R processors ($a_i, \ldots, a_R$) and set a counter j = 1.

2 Send a message to processor $a_j$ and wait for a reply.

3 The reply from $a_j$ will either be a migrating process or an indication that it has no processes.

4. If the processor is still idle and $j < R$, increment j and go to step 2 else stop polling.

A large number of queueing model simulations with varying numbers of processors were performed, and it was found that both algorithms resulted in similar improvements in process turnaround time and similar overall communications costs.

# CHAPTER 4

# SIMULATED SYSTEM DESIGN AND IMPLEMENTATION

## 4.1 RATIONALE AND INTENDED GOALS

From the previous chapter, which reviewed a number of different methods for tackling the question of load balancing, it can be seen that the approaches taken for each component of a load balancing algorithm are legion, and that an analysis of how these components interact and the trade-offs to be considered, are complex. It is necessary to investigate the underlying nature of the problem and to examine how the design of a policy to achieve performance improvement through redistribution of the load across a number of processors is influenced by system characteristics and overheads.

Many studies of load balancing have made simplifying assumptions in order to use queueing theoretic models in their analysis [Zhou86a]; the overheads of executing a load balancing algorithm are often ignored [Cabrera86], and the costs of interprocessor communication are not included in the model, or if they are then they are only considered in the context of messages used for load balancing [Krueger84], thus excluding messages generated by user processes executing on a processor which does not hold the resources accessed by those processes. When algorithms are used in real distributed systems, their performance is often only studied for very small numbers of processors, with a limited workload environment [Barak85a].

Due to the apparent limitations of queueing models [Ni85], simulation has suggested itself as a viable method for analysing the complex nature of a loosely-coupled distributed system [Reed83], and we have adopted it for our work in that area. Our goal is thus first to provide a flexible simulation vehicle for studying the behaviour of

such a system, and then to use this vehicle to investigate the adaptive load balancing problem. The simulated system is structured in a manner which allows the specification of a number of parameters defining the physical characteristics of the network, and provides support for the execution of a variety of user processes in this environment. Our simulation is thus divided into three major components:

- the simulated processors connected via a communications mechanism, with a specifiable speed and topology
- a distributed operating system kernel, providing fundamental facilities to user processes running on the network
- a means of developing and analysing the performance of load balancing algorithms under varying system loads.

By using a completely simulated system of this kind, we are not constrained by available hardware aspects, or the limitations of attempting to add load balancing features to an operating system which was not originally designed with this in mind.

## 4.2 DEVELOPMENT ENVIRONMENT

The simulated system was developed and implemented on an NCR Tower XP running UNIX System V, with 2Mbytes of user-addressable RAM and two 35Mbyte hard disks. Due to the large number of processes created during system execution the UNIX kernel was relinked with an increased process table and open file descriptor table size, and the optional shared memory system calls were included. All software for the simulation was written using the C programming language and totals some 10,000 lines of source code. C was chosen as an implementation language since it

provides a well-defined interface with UNIX, and has features appropriate for the development of operating system software. A listing of the program is given in Appendix C.

The system is structured in a way which will optimise its flexibility and allow us to study a variety of approaches to adaptive load balancing, by creating an abstract machine environment at the lowest level and writing the higher-level software (such as provision of kernel calls for interprocess message-passing) in terms of a number of primitive routines which simulate physical processor operation. Various characteristics of processor speed, communications speed, network size, local processor scheduling and performance monitoring are parameters of the system held in a collection of "#include" files.

Each of the three major components of the system listed in Section 4.1 was designed, tested and debugged incrementally to ensure its correct operation, before being combined for our experiments on load balancing algorithms. Thus we began by creating the simulated processors connected via a network topology and tested them by sending a number of interprocessor messages through the network, gathering considerable trace data to ensure that they reached their correct destination via the correct route. This trace data also involved checking that the passage of real time functioned as required.

Having established the underlying network in this manner, the operating system kernel was developed, and each function provided for user processes in the system was incrementally added and tested, by writing a number of typical user process groups with a large variety of interactions. To ensure that all system data structures were correctly maintained, these were regularly dumped to trace files and examined before

and after a particular operating system function was performed.

Finally we tested the modules of our kernel which support the implementation of load balancing algorithms, including a mechanism for performing process migration. Again a number of user processes were written and executed on our system, and we migrated these processes across the simulated network, monitoring their progress, to verify that they behaved as expected. Whilst recognising that in a highly concurrent, complex system such as ours, it is not possible to test every eventuality, we are sure that the above method ensures the validity of our results.

## 4.3 SIMULATED PHYSICAL NETWORK

The "simulated network" module of the system deals with the creation of simulated processors, the maintenance of a global time source, and elementary interprocessor communications following a specified topology, and hence includes most of the code which would need to be modified to port the software onto a "real" network (in fact the code to simulate the passage of real time would no longer be necessary at all).

### 4.3.1 The Start-up Process and Simulated Processors

It was decided that processors in the simulated network should each be represented by a separate UNIX process, with every processor running the same program (hence they are homogeneous), providing the underlying architecture on which to run the kernel of an operating system designed for a distributed environment. This method was chosen since it neatly encapsulates the concept of autonomous processors with no shared memory, where data exchange can only occur by a message-passing mechanism using

a communications medium. These processes are created from the main start-up routine, which is called when the system is first activated. This routine takes information regarding the number of processors in the network, the number of user processes which will be created during this simulation run, and the rate at which these arrive, from a main configuration file (these values can also be input interactively from a terminal if required); thus these parameters can easily be changed to create different environments. Once these values have been established the start-up routine creates a file of user process arrival times for each processor and a number of named pipes which simulate interprocessor communications. Both of these aspects are described in more detail in later sections.

Having set up this environment the number of processors specified in the configuration file are created using the `fork()` system call; in order to distinguish between processors each one is given a unique "machine identifier" (an integer value) which will be used during interprocessor communications and as an extension to all filenames relating to a particular simulated processor. This value is stored in the global variable `this_mc`. The UNIX process identifiers (pids) of all forked processes are held in the main start-up routine to allow limited debugging of the system if a future modification causes a processor "crash" (e.g. addition of extra operating system facilities); the pids are also passed as an array indexed on machine identifier to every processor, since they will be used in the kernel call and process migration mechanisms of the distributed operating system.

The first task of each processor is to open its own trace file which is written to periodically to give performance information (trace files have the name "trace" followed by the machine identifier number), and will also contain appropriate error

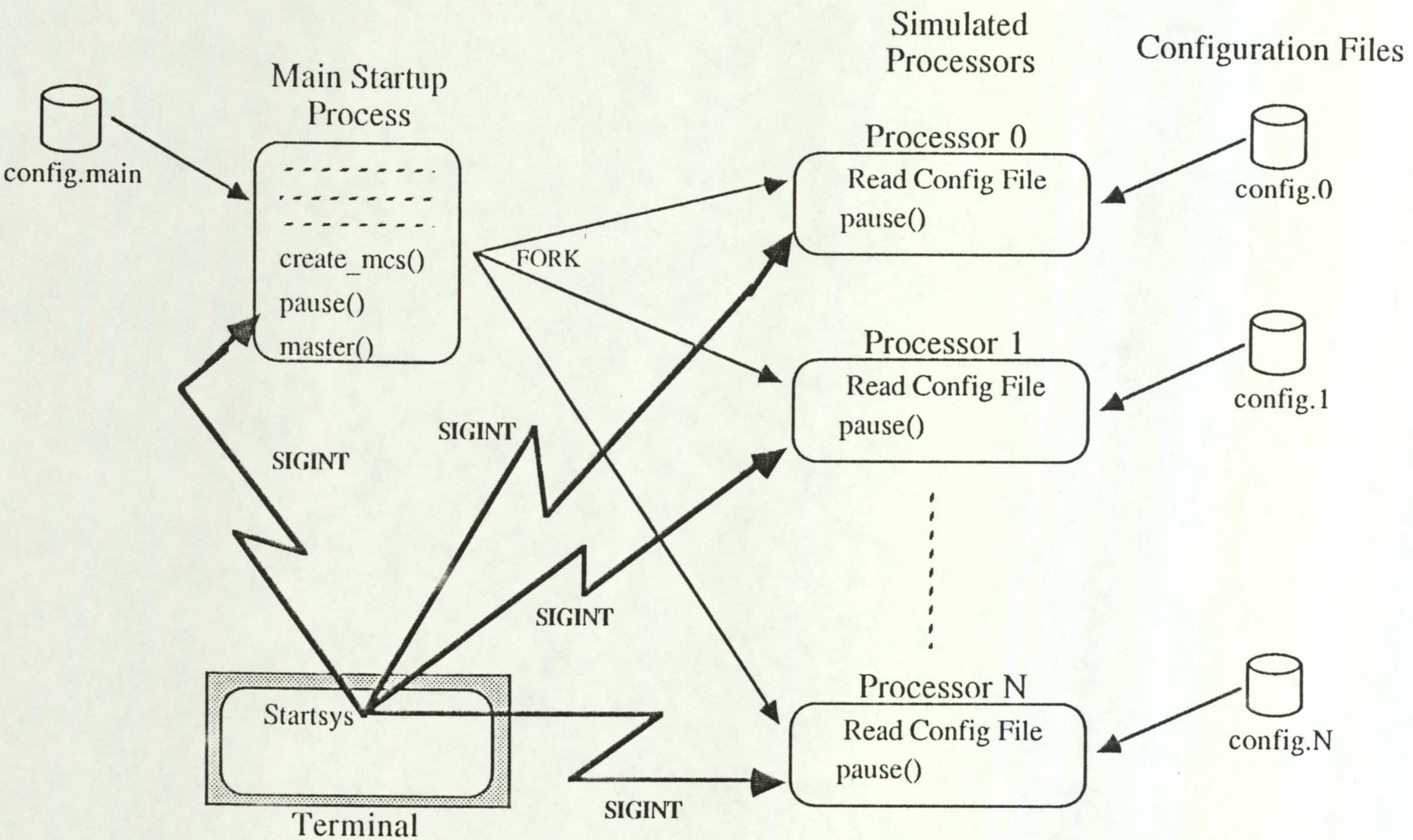



Fig. 4.1 System Configuration and Start-up

messages if a system fault occurs. The simulated network topology is then created, by reading relevant data from per-processor configuration files (named "config" with an extension formed by the machine identifier) and opening the necessary named pipes (as described later).

When the network has thus been configured all processors wait to be booted by making a `pause()` system call; booting is achieved by sending a SIGINT software signal to all processors, either from the keyboard (usually the RUBOUT key) or from a provided shell script `startsys` which scans the UNIX process table and uses `kill()` to create the desired effect. This ensures that all processors begin executing at the same time (a fact which is important for the maintenance of simulated real time, as described below). An interrupt handler in all created UNIX processes which catches the SIGINT signal, either begins running the distributed operating system kernel (if the process is a simulated processor) or a master routine (if it is the start-up process); the purpose of the master routine is to maintain global simulated time consistency. An outline of the operations performed at system start-up is shown in fig. 4.1.

### 4.3.2 Simulated Time Maintenance

A fundamental aspect of network activity which must be accurately simulated is the passage of real time, used for establishing the interval between user process arrivals, overheads incurred by interprocessor communications, process execution time (both user time and OS time), overheads of carrying out load balancing and process migration, and for providing performance evaluation information. This is achieved in our simulation by maintaining a local view of time held independently by each

processor and, since these are running as asynchronous processes on a multiprogrammed system and will update their value for time at varying rates, the start-up process is used to ensure that a sense of consistent global time is correctly maintained.

#### 4.3.2.1 Time Definition

Each processor has a variable `sys_real_time` (of type `double`) which it uses to record the passage of simulated real time; the unit of time used is microseconds, so incrementing this variable by one represents the elapsing of one microsecond of real time. A parameter of the system (the manifest constant AVE_INST) is introduced to specify the length of time in microseconds required to execute one machine instruction, and this is then used as the basic unit by which to update time as necessary. Each operation of a simulated processor which would take some period of real time can then be defined in terms of a number of average instructions (for example sending a message onto the communications medium, or processing a user's kernel call).

The basic routine in the simulated system which maintains local time (ie the processor's own view of the quantity of real time which has passed since it was booted) is `time_update()`. It is called at every point in system operation where simulated time needs to be advanced, and is passed the number of microseconds which have elapsed, together with an indication as to whether the elapsed time is due to user process or operating system activity, and, if the former is the case, a pointer to the user process in the simulated kernel's process table. This routine is responsible for maintaining several timing aspects of the system, the most fundamental of these being to increment `sys-real-time` by the amount of time that has elapsed; it is also

necessary to update various per-process items of timing information. If the elapsed time is due to user process activity, then `time_update()` appropriately increments a count for that process (using the process table pointer) regarding both the amount of time for which it has been executing since it was created, and also the time that execution has been continuing on this processor. This quantity may be different to total execution time if a preemptive load balancing policy is being used, and will be useful for deciding which process to migrate to ensure that processes spend at least a minimum period of time on a particular processor before becoming eligible to be moved. The above values relate to the currently executing process, but time information is also recorded for all processes which are resident on a processor when real time is incremented. This information concerns the total time that a process has existed in the system (regardless of which processor it has previously been running on) and the total time that it has been resident at its current location; hence `time_update()` scans all process table entries adding the elapsed time to these two entities, whether elapsed time is user time or operating system time.

For certain load balancing strategies, time needs to not only be considered in terms of a single instant, but must be divided into a number of quanta, so that local processor load during a particular quantum can be noted in order to provide a load value averaged over a number of such quanta [Barak85b]. To this end, an array of time quanta, where each element holds details for a single quantum is manipulated by `time_update()`; the length of a time quantum in microseconds and the number of past quanta which are retained, are tunable parameters of the system; hence when `sys_real_time` is incremented, the appropriate quantum entries are updated, wrapping around to the beginning of the array when it is full, to guarantee that only the most recent NQUANTA entries are kept (where NQUANTA is the manifest constant

specifying how many quanta should be retained). When an entire quantum has elapsed, the number of non-blocked user processes currently resident on the processor is recorded for load balancing policy purposes. In addition, a note is made of the amount of OS time used in a quantum, to provide details of quanta when the processor was unavailable to user processes due to OS overheads.

#### 4.3.2.2 Processor Synchronisation

If the simulated processors were permitted to update their view of real time in an unrestrained manner, since they will receive variable periods of service from the UNIX scheduler, there would be no consistent value for simulated real time over the whole network; consistency is hence achieved in our system by using the start-up process to synchronise processor operation. Under this scheme every processor is left to independently update its local time value for a limited period of time; this period is specified by a parameter (SYNC_INTERVAL), which should be kept small to ensure that processors' time values do not differ by more than a few microseconds. The start-up process and all processors are attached to a shared memory segment which contains a time at which processors must synchronise with each other, and an array of flags indicating which processors have reached the synchronisation time. As soon as a processor's `sys_real_time` exceeds the next synchronisation time, it sets its flag in the shared memory segment and waits (using `pause()`) until it is informed by the start-up process to continue. The start-up process constantly scans the shared memory segment array of flags until all are set (in other words all processors have reached the synchronisation time), and then establishes the next synchronisation time (incrementing it by SYNC_INTERVAL) and sends each waiting processor a software signal to instruct it to continue operation. In this manner all processors' simulated real

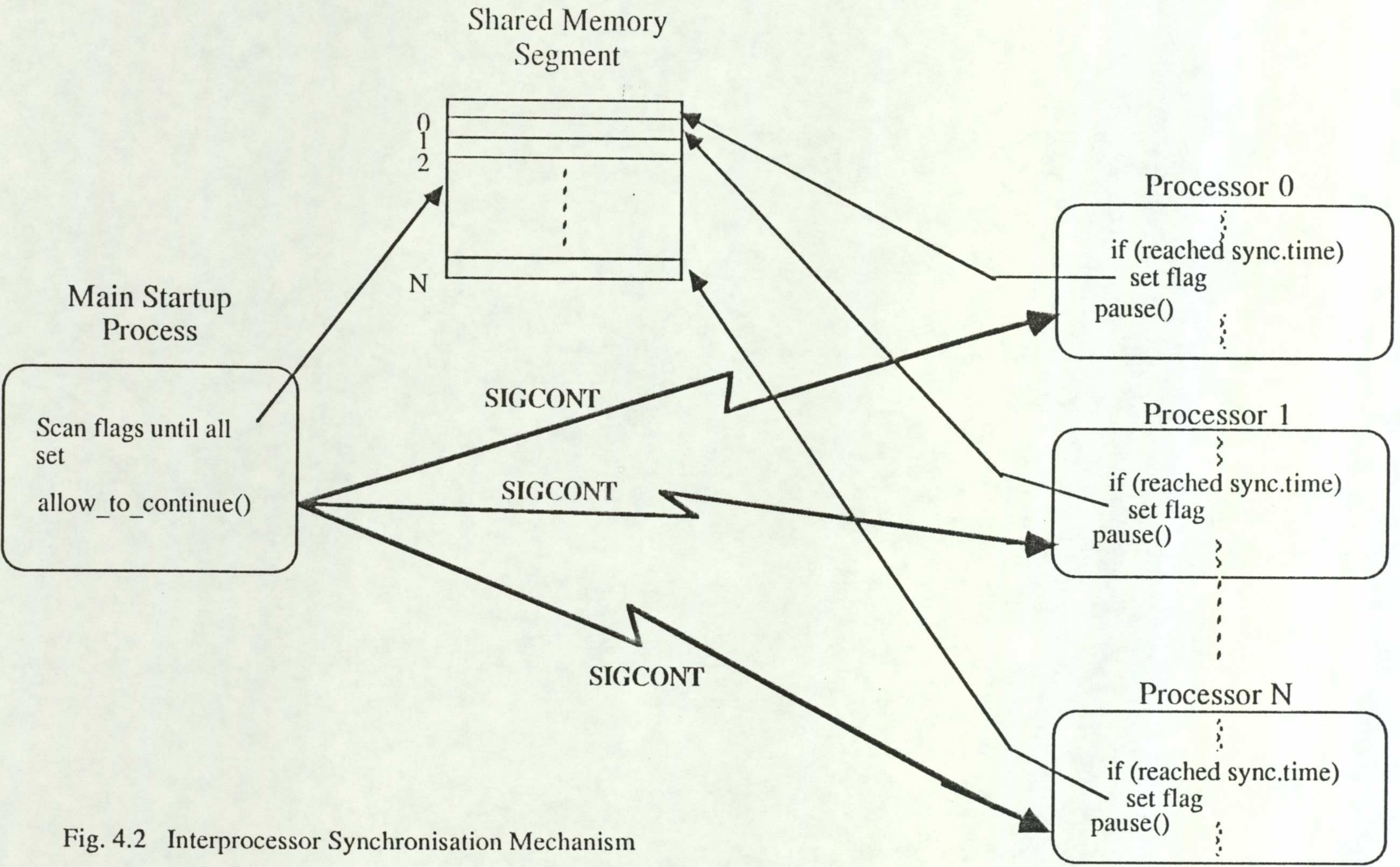



Fig. 4.2 Interprocessor Synchronisation Mechanism

time values are always within less than SYNC_INTERVAL microseconds of each other.

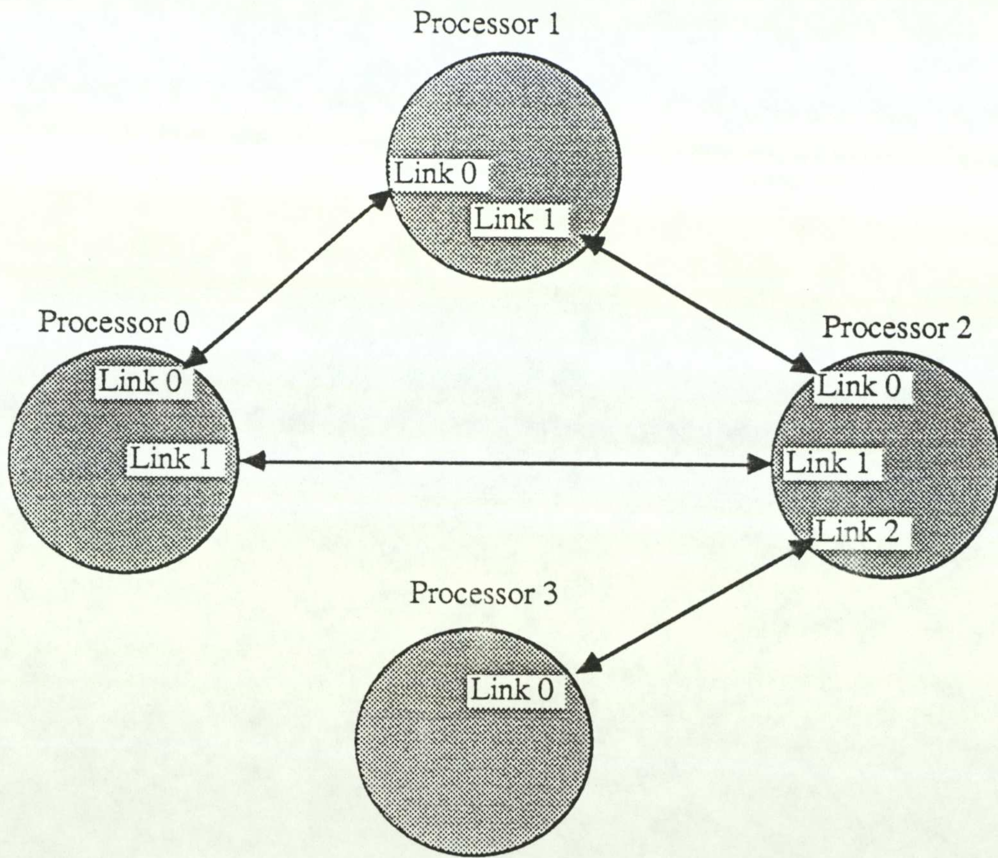
A diagrammatic outline of the operation of this synchronisation mechanism is shown in fig. 4.2.

#### 4.3.3 Communications Medium

As previously stated, since we are studying loosely-coupled distributed systems, all communications between processors must be achieved without the use of shared memory, and hence using a message-passing mechanism on an external communications medium. It was decided that using named pipes to simulate interprocessor links is an appropriate way of creating a network environment with these characteristics; thus when a processor reads from a named pipe it is simulating receiving a message on a communications link, and writing to a pipe simulates message transmission; since pipes in UNIX are uni-directional, a duplex connection between two processors is represented by two pipes, with the processors being readers or writers of the pipes as appropriate. The necessary pipes for communications simulation are all created by the start-up process, before the processors are forked; pipe names are formed by taking the string "own_p" and appending the machine identifier of the processor which will read the pipe (e.g. the pipe read by processor 0 is "own_p0"); as soon as processors are booted they open their "own" pipes for reading, to accept incoming messages.

##### 4.3.3.1 Network Topology

The network topology is specified separately for each processor in terms of its direct




Physical Link Table for Processor 0

| Link no | Neighbouring processor id | Pipe fd |
| --- | --- | --- |
| 0 | 1 | → File descriptor for writing to processor 1's pipe |
| 1 | 2 | → File descriptor for writing to processor 2's pipe |
| ⋮ | ⋮ | ⋮ |
| MAXLINK | | |

Fig. 4.3 Example Network Topology and Physical Link Table

neighbours and routes to all other processors; this information is either entered interactively from the terminal or stored in per-processor configuration files, each of which has the name "config" with the machine identifier as an extension (e.g. "config.0" for processor 0); see appendix C for a description of the configuration file format.

When a processor is first booted, it establishes connections to its immediate neighbours; the number of such links is the first value read from the configuration file, followed by the machine identifier of the neighbouring processor on each successive link. In order to be able to send messages to a neighbour, its communications pipe is opened for writing. Details concerning links are held in a table (named `phys_link[]` in the program listing), indexed on link number, where each entry contains the machine identifier of the neighbour on the corresponding link, together with the UNIX file descriptor returned when the neighbour's pipe was opened for writing; the maximum number of possible links a processor can have is a parameter of the system (MAXLINK). This procedure ensures that processors can only send messages directly to their immediate neighbours. Fig. 4.3 shows an example interconnection topology, with the corresponding physical link table for one of the processors given.

#### 4.3.3.2 Network Routing

In order for processors to communicate with their peers, a mechanism to route messages through the network has been introduced to the simulated system. Since our interest is not primarily in adaptive network routing algorithms, we have adopted a simple fixed routing approach. Each processor maintains a routing table (named

Route Table

| Processor id | Distance | Via link |
|---|---|---|
| 0 | ∞ | ∞ |
| 1 | 1 | 0 |
| 2 | 1 | 1 |
| 3 | 2 | 1 |
| ⋮ | ⋮ | ⋮ |
| MAXROUTES | | |

Link Table

| Link no | Neighbouring processor id | Pipe fd |
|---|---|---|
| 0 | 1 | → File descriptor for writing to processor 1's pipe |
| 1 | 2 | → File descriptor for writing to processor 2's pipe |
| ⋮ | ⋮ | ⋮ |
| MAXLINK | | |

Fig. 4.4 Example Route Table and Physical Link Table for Processor 0

`route_table[]` in the program listing), indexed on machine identifier, where each entry gives an index into the physical link table, thus specifying the link on which a message should be sent for all other processors in the network; the entry also contains an integer value giving the distance in "hops" of each processor from the sender; this information is held in the configuration files as link/distance pairs, ordered by machine identifier. Using this method a route to a particular processor is not fully-specified at the sending processor; the sender only knows the identity of the immediate neighbour to which it must route a message, and does not know the identity of any further intermediate processors which will be used in the message's full path to its destination. This approach was adopted since it will allow routes to be easily changed without considerable updating of tables, if an adaptive routing algorithm were later added to the system. An illustration of how the route table and physical link table interact is shown in fig. 4.4, giving the appropriate table entries for Processor 0 assuming the topology of fig. 4.3 .

### 4.3.4 Interprocessor Communications

Once the basic network structure has been established as described above a means is needed by which to send and receive messages over the simulated communications medium. The lowest level I/O in the system is provided by two general utility functions `pwrite()` and `pread()`; `pwrite()` writes a specified number of bytes from a given address onto a pipe whose UNIX file descriptor is given as a parameter; similarly `pread()` reads a specified number of bytes, from a given pipe into memory at a given address. Both of these functions consider messages as uninterpreted byte streams (interpretation being provided by higher level functions), and perform low-level error checking (eg whether the relevant pipe is open, or whether read/write

errors occur on that pipe). All functions which require I/O operations on the pipes representing interprocessor connections, use these two functions to perform them.

Messages which are exchanged between processors are defined as having a header and a body. The message header (em_hdr) contains an indication of the message type (an integer constant specifying whether the message is for a particular operating system function, or is a user interprocess message), the identity of the sending processor, the intended destination processor, and (for certain messages only) the name of the user process on whose behalf the message is being sent. The message body is of variable size and contains fields specific to its particular type (see appendix C for further details).

When a message needs to be transmitted through the network, both the header and the body are passed to the high-level I/O function TX(); this function uses the identity of the intended destination processor held in the message header as an index into the routing table, in order to find the correct interprocessor link, on which the message should be sent. The appropriate pipe file descriptor taken from the physical link table is passed to pwrite(), together with the message itself (which is transmitted as an uninterpreted byte stream) and the message size. In order to simulate the overheads involved in message transmission, a parameter of the system (TX_BYTE_TIME) is used to indicate the amount of time needed to send one byte of information onto a communications link; this therefore represents the speed of the communications medium, and is passed to time_update() multiplied by the size of the message to be sent. In addition, a fixed software overhead for using a communications protocol (PROTOCOL_TIME) is included, measured in terms of the number of machine instructions needed to send or receive a single message; this value is also a parameter

of the system. Since certain messages need to be sent to all other processors on the network, a broadcast() function has been provided; however this does not mean that an underlying multicast mechanism is available [Frank85], and is in fact purely a series of transmissions to each processor; because this research is not concerned with network communications algorithms, no attempt has been made to include a more "intelligent" broadcasting method, but the system is written in a way which would make its addition easy.

Receipt of a message at a processor is controlled by the I/O function RX(). This function is responsible for receiving the message header and then, depending on the type of message which is being received, for calling pread() to read the message body into a suitably large area of memory. Similarly to TX(), each time a message is received the fixed software protocol time, and the parameter indicating the time required to receive one byte (RX_BYTE_TIME) are taken into account by calling time_update() as appropriate. Since messages may pass through several intermediate processors before reaching their destination, the function check_forward() is called every time a message arrives at a processor; this function examines the destination processor identifier in the message header, and retransmits the whole message if it has not arrived at its destination.

The interactions of TX(), RX(), pwrite() and pread() are illustrated in fig. 4.5.

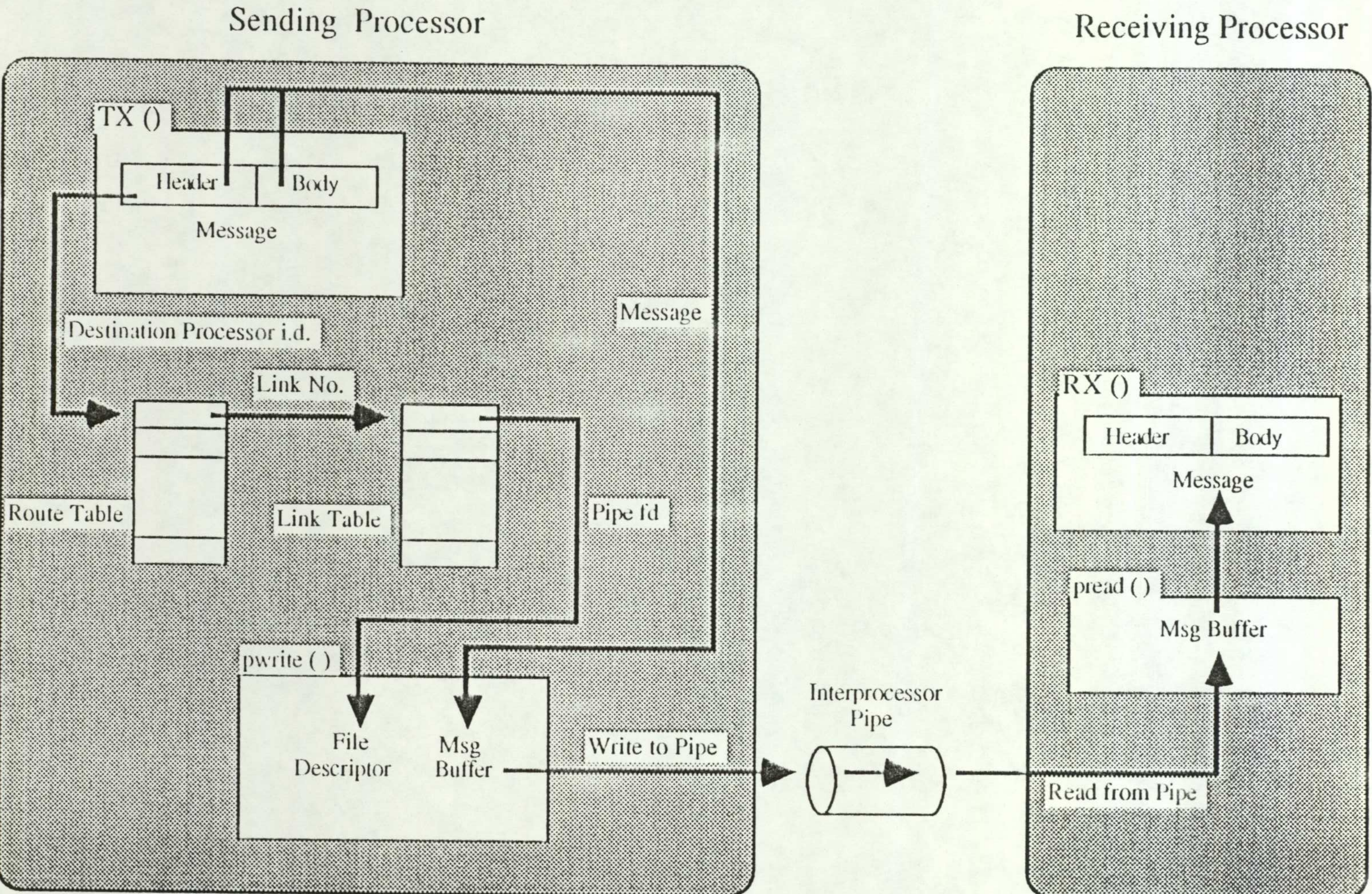



Fig. 4.5 Interprocessor Communications Mechanism

## 4.4 DISTRIBUTED OPERATING SYSTEM KERNEL

### 4.4.1 General Structure

The simulated environment described above provides a mechanism for creating a number of processors connected via a communications medium in a chosen network topology. In order to control this environment we need an operating system which will use the interprocessor communications and time management facilities of each processor, and allow the execution of user processes. To this end we have developed an operating system kernel, with many of the features which we considered desirable from the systems reviewed in Chapter 2, which runs independently on each processor, thus making control of the network totally distributed; global information regarding the network state can thus only be maintained by co-operation between all of the autonomous kernels using the underlying message-passing facilities. The kernel in each processor is identical and begins running as soon as the simulated network is booted using software signals as previously described; when messages are exchanged, the identity of the sending and receiving kernels is established using their relevant machine identifier numbers.

The principal motivation behind the design of the kernel, and in fact that of the whole simulated system, is to provide an environment in which to experiment with load balancing strategies, and hence the features which we have included are not intended to be a full set of kernel services, but are tailored to our needs; however, considerable effort has been made to allow realistic user processes to be written and run on the system, thus distinguishing our approach from a purely mathematical model simulation.

The main functions of our kernel are thus:

a) creation, execution and destruction of user processes

b) a message-passing mechanism for interprocess communication

c) a means for kernels to exchange global state information

d) local process scheduling

e) transparent process migration

f) performance monitoring

The requests for service which the kernel receives can be grouped into two categories:

a) dealing with a user process kernel call

b) dealing with a message from a remote kernel

Each request is identified by an integer constant which is used as an index into two arrays of pointers to functions (`kcvec[]` for kernel calls, and `emvec[]` for external messages); in this manner, further kernel services can easily be added by writing a function to deal with that service and then entering its address in the relevant array. The current system deals with 11 different types of kernel call, and different external messages; a more detailed description of the kernel's operation is given in later sections.

### 4.4.2 User Process Support

User processes, which represent the workload submitted to the simulated system, also run as separate UNIX processes under the control of our distributed operating system kernel. Since the typical user process environment which we envisage for our system,

would consist of sets of processes co-operating to achieve a common goal, we introduced the concept of process groups. Execution of a process group begins with the synthetic workload generation part of the system creating a parent process, thus simulating a user requesting program execution from an interactive terminal, typically through a process similar to the UNIX "shell"; the parent process then creates a number of child processes to act as servers providing a service to perform the desired task, thus forming a "process group" [Cheriton85].

#### 4.4.2.1 Process Naming

A process naming convention was chosen which, whilst providing network-wide unique process names, would not directly associate a name with the processor on which a process is executing; this allows processes to migrate freely around the network and to still be readily identified.

As machine identifier numbers are guaranteed to be unique, and given that a process group must at least begin executing on a single machine, we adopted a process naming scheme which consisted of the machine identifier of the originating processor, together with a "process group number" (where a new group number is allocated each time the synthetic workload generation function is called) in order to uniquely identify a process group, and then a unique character string to name a process within that group. This combination of process group and name within group, guarantees network-wide uniqueness of process names. Hence we use the following C data types to specify a process name (`PROCN`):

```
typedef struct {
                int gmc    ; /*original mc on which group was created*/
                int gnum ; /*group identifying no.*/
                } PGRP;

typedef struct { PGRP pgroup  ; /*process group identifier*/
                char pname [MAXPNAME] ; /*process identifying string within
                                                                  group*/
               } PROCN ;
```

#### 4.4.2.2 Kernel Call Interface

User processes execute in our system by making requests for service via "kernel calls". Similarly to the method used to simulate interprocessor communication, we employ named pipes to implement the kernel call mechanism; when a user process makes a kernel call, it writes its request on its processor's pipe and, when the kernel has processed the call, the user process receives the result on a further return pipe. In order to make this interface transparent to user processes, a number of library routines are provided which need to be linked with the user process code before execution. These library routines present a function call interface for making kernel calls, and deal with sending and receiving on the correct pipes and returning the result; hence the user process view of our kernel call mechanism is similar to that of UNIX system calls. For a more detailed description of the exact operation of this mechanism, see section 4.4.4 on "Kernel Call Mechanism".

We believe that using UNIX processes to simulate user processes on each processor, with a controlled interface to our kernel creates a realistic environment in which to carry out our experiments, and that the use of pipes for the kernel call/return

mechanism accurately mirrors a separate kernel and user address space.

#### 4.4.2.3 The Process Table

Since the processors in our system are totally autonomous and loosely-coupled, and since we wish to restrict the overheads caused by excessive exchange of information between machines, we have adopted the approach that each processor maintains information concerning only those processes which are resident on it. This implies that any global information regarding for example the current number of processes on a particular processor, or any process-specific information, can only be maintained through the information exchange policy of the load balancing algorithm which is being used. All local process information is held in the kernel's process table (named `process_table[]` in the program listing), and the maximum number of possible entries in this table is a parameter of the system (MAXPROCS), where each entry refers to a single user process. The process table is a fundamental data structure for our implementation and a diagram showing its most important fields is given in fig. 4.6.

The process' name (field `proc_name`) is essential since it identifies the process in a network-wide unique manner. As processes may migrate from one processor to another under the control of a load balancing algorithm, a record is kept of the machine identifier number of the processor where a process was first created (field `orig_mc`), for implementation-specific purposes, since user processes are created by a UNIX `fork()` system call, and failure to execute a `wait()` call by the original creating processor would result in many "zombie" UNIX processes being left in the system. The process' size is given in bytes, to be used to calculate the overheads caused by

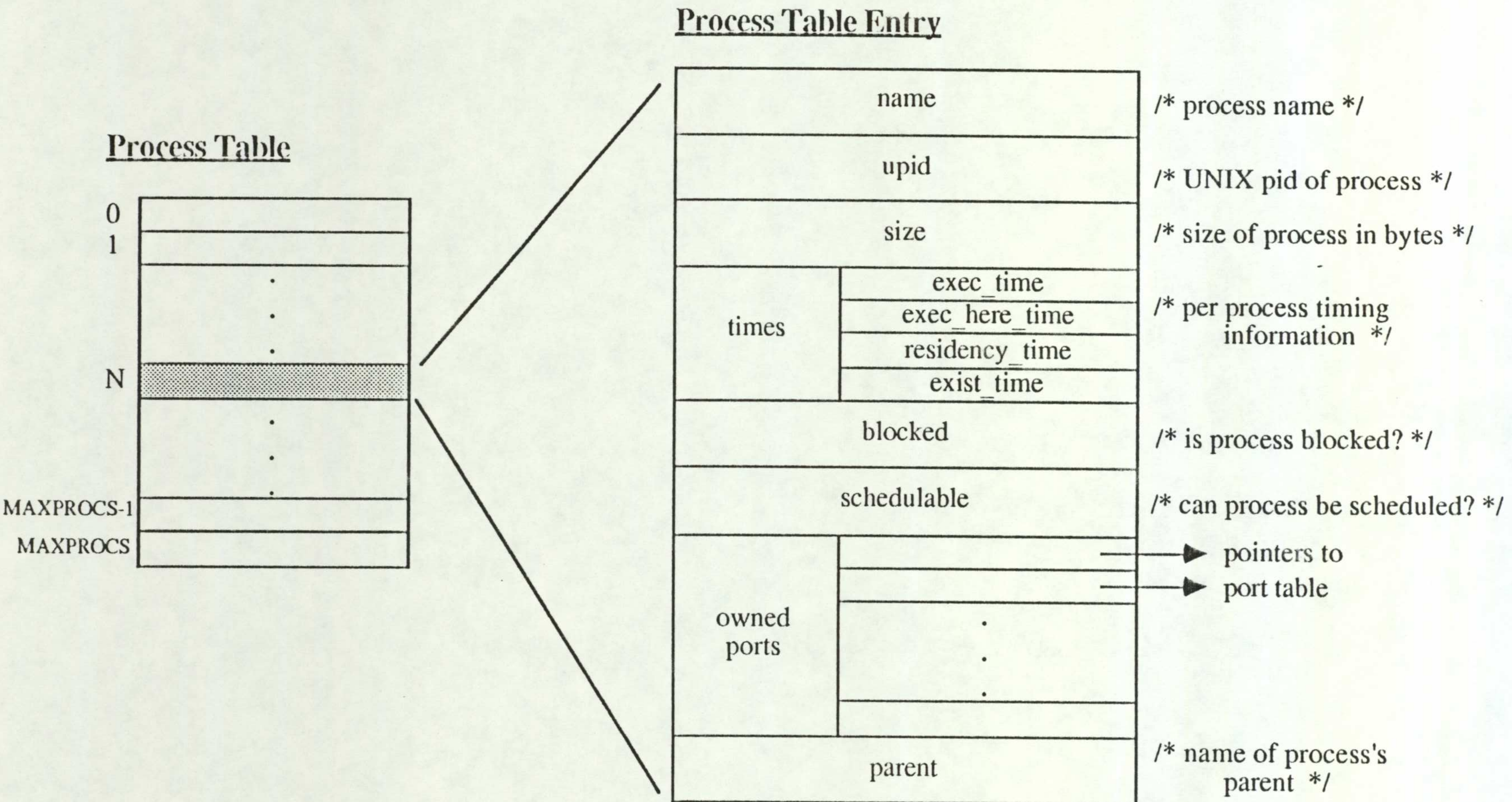



Fig. 4.6 Process Table Structure

process migration.

In order for our kernel to schedule its processes correctly two flags are included for each process entry, one to indicate if the process can be scheduled (it may have been chosen for migration and be waiting to find a destination processor, in which case it should not be locally scheduled) and another to indicate if the process is blocked waiting for an external event (for instance it may be waiting to receive confirmation of a kernel call which resulted in a request being sent to a remote processor for service). The UNIX identifier of the user process is also included for scheduling and kernel call/return purposes as described later.

Timing information is held for each process (appropriately updated as simulated real time elapses), which is used for performance evaluation, and may be used for the more sophisticated load balancing algorithms. This information concerns the total real time that has elapsed since the process was created (`exist_time`), the total time that the process has spent executing, i.e. the number of CPU cycles it has used (`exec_time`). Also included are totals of the amount of time that a process has been resident on its current processor, because it may have migrated (`residency_time`), and the time it has spent executing on that processor (`exec_here_time`). These are the per-process values which are maintained by the simulated processor function `time_update()`.

Since our system is intended to support an interprocess communications mechanism, a process table entry also has an array of pointers into the software port table, whose structure is discussed later, holding a record of the ports which a process currently owns, and through which it will communicate with its peers. The number of such

ports which are owned is also held in each process table entry (in the field ownp_length).

### 4.4.3 Interprocess Communications Mechanism

In order to allow user processes to co-operate in the parallel solution of a common goal, a means must be provided for them to efficiently exchange data. In a shared memory multiprocessor environment this could be achieved through the use of shared variables, but the loosely-coupled nature of our network makes this approach infeasible; in fact, we note that shared variables introduce problems of mutual exclusion and synchronisation when concurrently accessed, and thus result in unnecessary complexity [Manning80]. We have thus adopted a message-passing mechanism for performing interprocess communication.

Message-passing has a number of desirable attributes. As each user process executes in its own disjoint address space, it can only operate on its own local data; with messages being the only way of performing data transfer, the interface between processes, and the side-effects which they may have, can be clearly defined and controlled by the programmer. Messages can either be used purely as a means for distributing data, or to enable one process to request service from another. The latter of these two possibilities can be used by processes to implement a remote procedure call mechanism, where the requesting process sends a message containing an indication of the service required, together with any input parameters, and the serving process returns its results via a further message back to the requester. It is thus possible to "disguise" the message-based interaction in terms of procedure calls, if this is more familiar to the programmer. Access to critical variables is guaranteed to be

mutually exclusive, since they are passed in messages between processes, and hence can only be operated on by one process at a time; process synchronisation can also be achieved using message exchange. In such a system the verification of a set of concurrent processes can be achieved by verifying each process independently, considering its external effects purely in terms of its message sending and receiving activity.

#### 4.4.3.1 Message-Passing Primitives

The kernel of an operating system which supports message-passing between user processes must provide primitives for sending and receiving interprocess messages [Liskov79]; these primitives can operate in either a blocking or a non-blocking fashion, and, since it has been suggested that both modes are appropriate to different application environments [Manning80], our kernel allows user processes to perform blocking or non-blocking send and receive operations (which we henceforth denote as `b_send`, `nb_send`, `b_rcv`, and `nb_rcv` respectively). A `b_send` operation is used if the calling process requires an acknowledgement of message receipt (since the sender will block until the message has arrived at its destination or an error is reported), but an `nb_send` allows the calling process to continue regardless of what happens to the sent message. A `b_rcv` operation blocks the calling process until the expected message arrives, whereas an `nb_rcv` is just an announcement of willingness to accept a message (which may not necessarily arrive); the willing process continues its execution and is signalled upon message receipt. Thus user applications can be constructed using any combination of the available modes to work in a fully-synchronous (`b_send/b_rcv`), half-synchronous (`b_send/nb_rcv`, `nb_send/b_rcv`) or fully-asynchronous (`nb_send/nb_rcv`) manner, to suit their

needs.

User processes send messages in a totally location-independent manner; it is the kernel's responsibility to deal with whether the intended destination for a message is local or remote; if the former is the case then message sending is performed by appropriate manipulation of internal pointers by the kernel, and in the latter case the message is transmitted onto the network and routed to its correct destination.

#### 4.4.3.2 Software Ports

When a message is sent, a means must be provided for specifying that message's destination. A flexible mechanism for establishing endpoints for communication which has been proposed is that of the "port" [Silberschatz81]; we have used this approach in our kernel. In avoiding using process names in a message's intended destination (as is used by Hoare [78] in CSP), the mechanism is more transparent, in that a sending process is not concerned with the identity of the process which will receive the message and will service it; messages are sent and received using the port concept. Each port has a single owner, which is the only process permitted to receive messages on it, but messages may be sent to multiple ports belonging to other processes. A typical use of a software port in this environment would be to associate a separate port for each service which a process provides to a potential client; a message arriving on a particular port is thus a request for its corresponding service. In order to ensure that messages sent to a port are of the kind which the receiving process expects, each port is assigned a "type" when it is created by its owner process; the type of a port restricts messages which arrive at that port to those of the appropriate type, and is a user-defined integer constant. This restriction only applies to messages sent to a port; a message of any type can be sent from a particular port, regardless of that port's

type.

Having defined ports as endpoints for communication, the kernel must provide a means by which user processes can connect their ports, thus allowing message passing to be performed. We have chosen to use the concept of a uni-directional link to implement this connection mechanism. If a process, P1, wishes to send messages from one of its ports, port1, to another port, port2, which may be owned by any other process (or indeed owned by the sending process), it must create a link from port1 to port2. This link will permit messages to flow from port1 to port2, but not vice versa; if a bi-directional connection is required then the owner process of port2 must create a link from it to port1. This link mechanism allows user applications to be developed with well-defined interfaces and data flow connections to suit their requirements. An example process group with ports connected in various combinations is shown in fig. 4.7.

User process sending and receiving of messages can thus be expressed in terms of operations on ports and links. A "receive" kernel call (`b_rcv` or `nb_rcv`) applied to a port owned by the calling process, results in any pending messages which have arrived at that port being delivered to the caller (ie copied into its address space); a "send" kernel call (`b_send` or `nb_send`) is made through a port via a link, and results in the user-supplied message being sent to the destination port which has been connected on that link. Hence in the example in fig. 4.7, if Process B performs a receive operation on its port B1, then it will be delivered a message which may have come from either Process C's port C2 via its link L1, or from Process A's port A1, via its link L1. Also in this figure, if Process C performs a send operation through its port C1's link L1, the message will be sent to Process A's port A1.

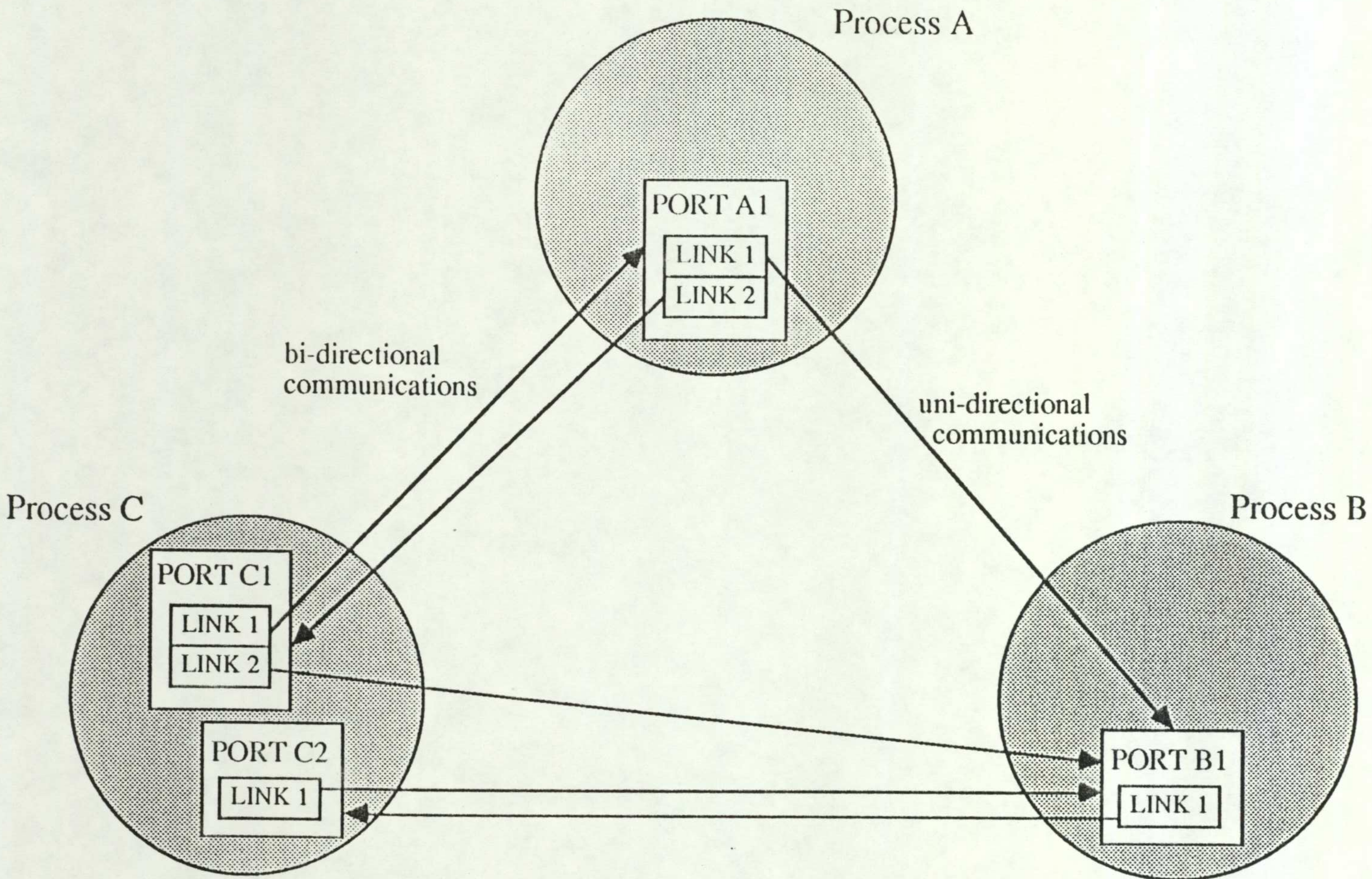



Fig. 4.7 Example Interprocess Connection using Ports and Links

#### 4.4.3.3 User Message Format

Messages passed between user processes in our system consist of two parts: a header and a body. The message header contains all information necessary to guarantee correct delivery and possibly acknowledgement; this information is given in five fields: the first two of these fields specify the names of both the destination and source port of the message (the source being included in case the receiving process needs to know where the message came from, or if the message was sent in blocking mode and therefore receipt needs to be acknowledged). The three remaining fields give an indication of the length of the message in bytes (which is used to calculate the time necessary to transmit and receive the message), a flag which records whether the message was sent in blocking or non-blocking mode, and finally a user-defined integer giving the type of the message, which will be used to check that it is valid to be received on a particular port.

The message body is an uninterpreted stream of bytes; for the purposes of our study we have used a small fixed-size array of characters for this part of a message, but the system can easily be modified to handle large dynamically allocated message bodies.

#### 4.4.3.4 The Port Table

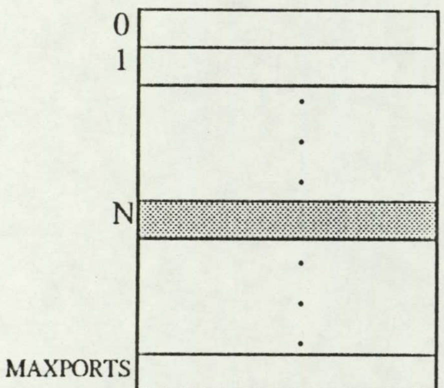
Associated with each software port across the network is a corresponding entry in the "port table" (named `port_table[]` in the program listing), a copy of which is maintained by the kernel of each processor. The pointers in a process table entry which refer to the ports owned by a user process (see previous section on the process table), are thus set up to point at entries in this port table. The maximum number of

ports which can be extant at any one time is a parameter of the system (MAXPORT).

Since ports will migrate from one processor to another, if their owner processes migrate, the kernel must know the port's name, the identity of its owner process and the identity of the processor on which the port is currently resident, for all created ports (both local and remote). This will enable the kernel to correctly route messages sent between ports, and to keep this routing transparent to user processes. In order to make ports uniquely identifiable on a network-wide basis, each port is given a name (specified by the user process when it creates the port) and whenever this port is referred to by the kernel it is taken together with the group identifier of its owner process (which is guaranteed to be unique). By enforcing the rule that no two ports used by a process group can have the same user process-specified name, port names are thus unique across the whole network.

Further information (as described below) needs to be held for all ports which are local to a particular kernel, to allow them to be successfully linked to other ports (both local and remote) and to deal with message-sending and -receiving operations. By maintaining full information on local ports and only essential details regarding remote ports, we have attempted to reduce the overheads caused by increased interprocessor communications which are necessary to keep each kernel's view of the port table consistent. A diagrammatic representation of the structure of the port table is shown in fig.4.8.

In order to record the interconnections between ports, each port table entry holds two arrays which specify all links which the corresponding port has to or from other ports (these are fields `links_to` and `links_from` respectively); the arrays contain



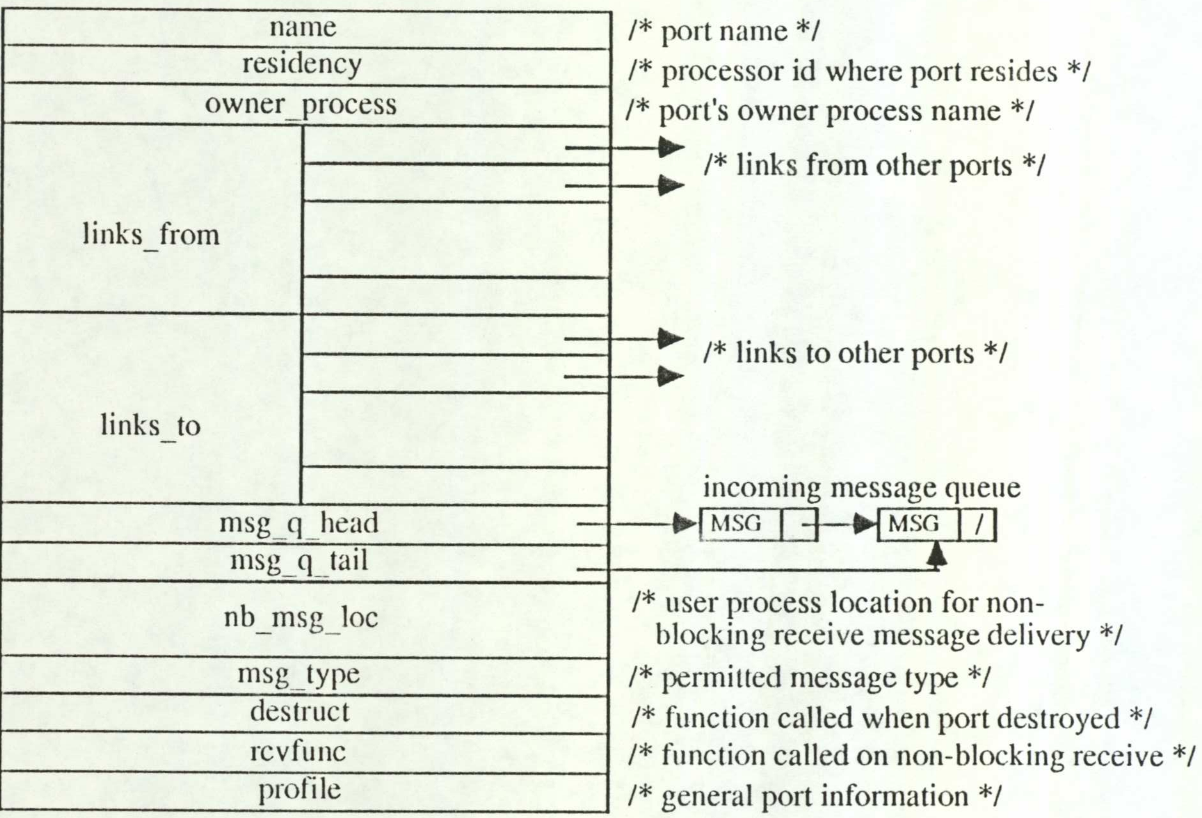




Fig. 4.8 Port Table Structure

pointers to other port table entries (whose corresponding ports may be both local or remote), and their sizes are parameters of the kernel (MAXLTO and MAXLFROM respectively).

Since messages may arrive at a port before the owning process has performed a corresponding receive operation (either blocking or non-blocking), a queue of such messages is maintained in a linked list for each port, with pointers to both the head and the tail of the queue. In our implementation, when a process requests receipt of a message it is given the message at the head of the queue, and further arriving messages are appended to the tail of the queue; hence message receipt is on a FIFO basis, but this could easily be modified if, for example, a priority scheme was deemed more desirable. Also held for each port is an integer value giving the type of message which can be received on that port (`msg_type`); this value is specified by the owning process and is used to check that the type of messages arriving at the port is correct.

Port entries also contain a number of pointers into the owning user process address space. When a message arrives at a port, and the port's owner has requested a non-blocking receive operation on that port, the location in the owner's address space where the message should be deposited is held in the pointer field `nb_msg_loc`; in addition, the owner process may have specified that a particular function should be called upon receipt of a message (in non-blocking mode), and the address of this function is held in the field `rcvfunc`. Finally, when the port is destroyed at the request of its owner, a pointer in the corresponding port table entry (the field `destruct`), gives the address of the user process function to be called upon port destruction.

In order for the kernel to maintain the current state of a port, various counts are held in a port's `profile`. This gives values for the number of ports to which a port is linked, the number of links connected from other ports, and the current length of the incoming-message queue; also held in the profile are two flags, indicating whether non-blocking or blocking receive operations are pending for this port, in fields `nb_pending` and `b_pending` respectively.

### 4.4.4 Kernel Call Mechanism

#### 4.4.4.1 The User Process View

As previously mentioned, user processes interact with the kernel of the processor on which they are resident through UNIX named pipes; each processor has two pipes for this purpose, one for receiving kernel call requests, and one for returning corresponding results. Since many user processes will be sharing these pipes, the kernel enforces mutually exclusive access to them using a "lock" file.

User processes view requests for service from the kernel as normal C function calls (as in the UNIX system call mechanism); the value returned from such "kernel calls" will vary depending on the service being requested, but we have adopted the general convention that a return value of -1 (cast to the appropriate type if the expected value is not an integer) denotes failure of the kernel call. Similarly to UNIX, we use a global integer variable (called `err` in our system) to indicate to the user process the reason why a particular kernel call failed; also as in UNIX, this variable holds the reason for failure of the last failed kernel call, and is not reset when subsequent successful calls are made.

In order to implement this "function call" interface, two pre-compiled files of routines ("kcalls.o" and "interface.o") must be combined with a user process when it is linked. The first of these files deals with packaging the parameters for each kernel call into a form convenient for transmission to the kernel via a named pipe, and with correctly returning results to the user process (setting `err` appropriately if an error has occurred). The second file implements the low-level details of interaction with the kernel through names pipes (both sending requests for service and receiving results) and with various software signals which are used for sychronisation.

When a user process is created its first action must be to call a provided function (`set_up()`) which sets a number of global user process variables, which are used for implementing the kernel call mechanism, the most important of these being the UNIX file descriptors of the call/return pipes and the UNIX process identifier of the processor on which the user process is running; these values are passed to the user process through the `main()` function argument list `argv[]`.

#### 4.4.4.2 Kernel Call and Return

Given the above environment, a user process makes a kernel call in the following manner:

return-value = Kcall (parameters)

where Kcall is any of the available kernel calls, namely:

cproc ( ) - create a user process

exit-proc ( ) - end process execution

| | |
|---|---|
| cport ( ) | - create a software port |
| dport ( ) | - destroy a previously created port |
| lport ( ) | - create a uni-directional link between two ports |
| uport ( ) | - remove a previously created port-to-port link |
| b_rmsg ( ) | - receive a message (blocking) |
| nb_rmsg ( ) | - receive a message (non-blocking) |
| b_smsg ( ) | - send a message (blocking) |
| nb_smsg ( ) | - send a message (non-blocking) |

When the user process executes a statement of the above type, this results in a call to the appropriate kernel call interface routine (linked in "kcalls.o"). It is this routine's responsibility to place the parameter list for the kernel call into a "parameter block" which is a suitable C `struct` data structure. This parameter block (`p_blk`), together with an integer representing the required kernel call (`kc_type`) are then written to the kernel through its named pipe; the user process then pauses waiting for the kernel to service its request. This sequence of events can be thought of as placing an integer function code in a machine register (representing the operating system service required), and pushing parameters onto the stack.

The kernel uses `kc_type` to index its vector of available services (`kc_vec[]`). The appropriate service routine reads its `p_blk` from the pipe (which is akin to popping it from the stack), and then carries out its required action, provided this does not violate any constraints (in which case the kernel call has failed). In order to satisfy a particular request, the kernel may need to co-operate with kernels on other processors

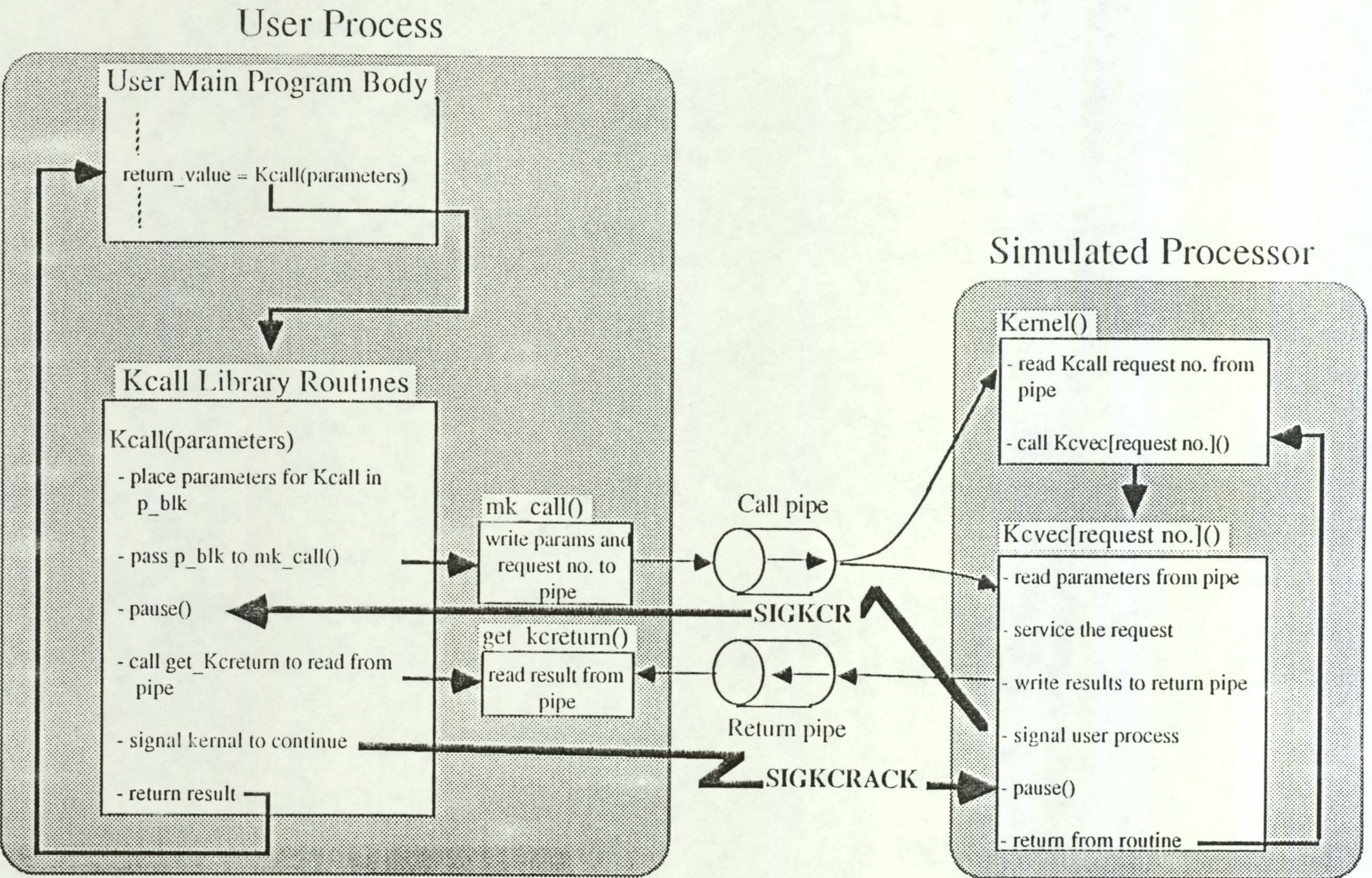



Fig. 4.9 Kernel Call/Return Mechanism

(for example linking two software ports together which reside at different locations in the network) but this remains totally transparent to the calling user process. Results of the kernel call are written back on a separate pipe and a software signal is sent to the user process to notify it of this fact; the kernel pauses until the user process acknowledges receipt of such results.

When the user process receives the software signal announcing completion of the kernel call, an interface routine reads the result from the return pipe and, after signalling the kernel to acknowledge receipt, the kcall routine returns the result to the calling user process. These actions can be thought of as popping results from the stack.

Finally when the kernel is sure that the results have successfully been received, it updates the passage of real time by an amount appropriate to the execution of the particular kernel call which was invoked; this will include any overheads incurred by sending messages to a remote kernel if this proved necessary. The full kernel call mechanism is summarised diagrammatically in fig. 4.9.

### 4.4.5 Kernel Call Implementation

In this section we will discuss in more detail, the actions of the kernel on receiving a request for service from a user process, and present pseudocode descriptions of the routines in the kernel which provide the services listed above, and also those which deal with interprocessor messages used to perform remote operations. The reader is referred to appendix C, for further details in the full program listing.

Each kernel call routine can be summarised as carrying out the following four steps:

i) read kernel call parameters from the appropriate named pipe

ii) perform the necessary operations on behalf of the calling process

iii) return results to the calling process

iv) update the passage of real time appropriately for this kernel call

We have chosen to group kernel calls into three categories, namely:

- process-related
- port-related
- message-related

#### 4.4.5.1 Process-related Kernel Calls

The kernel calls which deal with user process creation and destruction are `cproc()` and `exit_proc()`. `Cproc()` is used to create a new user process and expects as parameters character strings giving the process' name and the name of the executable UNIX file in which the program to be run resides; on successful process creation, `cproc()` returns the network-wide unique name for that process. `Exit_proc()` is called by a user process when it wishes to terminate execution. The pseudo code description of these kernel calls is:

```
CPROC ( )        /* routine for creating a new user process */

{
    if (process is of a new "group" created by the "shell")
        Assign it a new group number;
    Increment process counts;
    Enter information into a new process table entry, ie
        - name
```

```
        - originating processor
        - parent process name

    FORK a new process:

    Child /* ie user process */              Parent /* ie Kernel */

    Set up argument list for process         Enter UNIX pid of new process
    which will be used for making            in process table;
    Kernel calls;

    Return SUCCESS to caller;
    Exec new program
                                             Make newly created process schedulable;

                                             /* if load balancing is to be performed on
                                                process creation then code is inserted
                                                here.
                                             */
}

EXIT_PROC ( ) /* routine to deal with user process exit */
{
    if (calling process still has open ports)
        return FAIL to calling process;
    else
    {
        return SUCCESS to calling process;
        if (calling process was first created on this processor)
                wait for process to die;
        else /* process originated elsewhere, ie it migrated */
                send notification of process exit to originating processor;
        Re-initialise process table entry;
        Decrement process counts;
    }
}

EXIT_MSG ( )        /* routine to deal with notification of existing process
                       which originated on this processor */

{
    Wait for process to die;
    /* This is necessary for implementation reasons, since under UNIX a child
       process will remain in a "zombie" state upon death, unless it is waited for
       by its immediate parent */
}
```

#### 4.4.5.2 Port-related kernel calls

Kernel calls which concern manipulation of a user process's software ports are `cport()`, `dport()`, `lport()` and `uport()`. `Cport()` is called to create a port (which will be owned by the calling process), and expects as parameters the port's name, an indication of the type of message which can be received on that port, the address of a user process routine to be executed when the port is later destroyed, and the address of the user process routine to be invoked when a message arrives at the port following a non-blocking receive operation. If successful, `cport()` returns the port identifier, which should be used in subsequent kernel calls which relate to that port (e.g. receiving messages on it), which is similar to use of a UNIX open file descriptor. When a port is created, the creating kernel broadcasts its name and message type to all other processors.

`Dport()` deals with a user process request to destroy one of its previously created ports, and expects the relevant port identifer as a parameter, returning a flag indicating the success or failure of the port destruction request. If `dport()` is successful, then the identity of the destroyed port is broadcast to all other processors, allowing them to remove the port from their local port tables. `Cport()` and `Dport()`, together with the kernel routines to deal with the broadcast messages which they generate are shown in pseudocode below:

```
CPORT ( )          /* routine for creating a port */

{
        if (process owns too many created ports)
            return FAIL to calling process;
        Set pointer to new port table entry in calling process's
            owned ports list;
        Enter port information in new port table entry, ie
            - name
```

```
            - owner process name
            - message type for this port
            - pointer to user process's port destruction function
            - pointer to user process's function called for non-blocking
                        receive/message arrival;
        Broadcast port's name and message type to all other processors;
        Increase migration size of calling process;
        Return Port Identifier to calling process;
}


CPORT_MSG ( )  /* routine to deal with receipt of a message from a remote processor
                  announcing port creation */

{
        if (too many ports exist in the network)
            SYSTEM ERROR;
        else
        {
            Enter information for remote port in new port table entry, ie
                        - name
                        - owner process name
                        - message type
                        - processor on which port is resident
        }
}


DPORT ( )          /* routine for destroying a port */

{
        if ("port to be destroyed" does not exist)
            return FAIL to calling process;
        else
        if (messages pending in "port to be destroyed")
            return FAIL to calling process;
        else
        {
            Remove port from calling process's owned ports list;
            Decrement calling process's migration size;
            Decrement number of ports in table;
            Re-initialise port table entry;
            Return address of destruction function to calling process;
            Broadcast notification of port destruction to all other processors;
        }
}

DPORT_MSG ( )  /* routine to deal with receipt of a message from a remote processor
                  announcing port destruction */

{
        Find specified port in local port table;
```

```
        Re-initialise port table entry;
        Decrement number of ports in local port table;
}
```

The two kernel calls which deal with linking and unlinking ports are `lport()` and `uport()` respectively. `Lport()` expects as parameters the port identifier of the port owned by the calling process from which the link is to be created (which we term the "link from" port), and the name of the port to which the "link from" port is to be linked (which we term the "link to" port); if the link operation is successful `lport()` returns a unique identifier for the newly-created link, to the calling process. If the two ports to be linked are resident on different processors, the kernel deals with the necessary exchange of interprocessor messages to achieve the link, transparently to the calling process.

`Uport()` is called to remove a previously created link, and can only be invoked by the owning process of the "link from" port; it expects the port identifier of the "link from" port, and the identifier of the link to be removed as parameters. Again, any interprocessor message exchange due to the linked ports being on separate processors is handled by kernel routines. The pseudo code to deal with port linking and unlinking is thus:

```
LPORT ( )  /* routine for creating a link between two ports */
{
      if ("link from" port does not exist)
          return FAIL to calling process;
      else
      if ("link to" port does not exist)
          return FAIL to calling process;
      else
      if ("link from" port has too many links to other ports)
          return FAIL to calling process;
      else
      {
          if ("link to" port is local)
```

```
		{
			if ("link to" port has too many links from other ports)
				return FAIL to calling process;
			else
			{
				Update link information for "link to" port;
				Increment "link to" port's owner's migration size;
				Update link information for "link from" port;
				Increment calling process's migration size;
				Return SUCCESS to calling process;
			}
		}
		else /* ie "link to" port is remote */
		{
			Mark calling process as blocked;
			Send a message requesting a port link to the processor on which
			"link to" port is resident;
		}
	}
}


LP_REQ ( ) /* routine to deal with receipt of a message from a remote processor
		requesting a port link */

{
	Find the "link to" port in the port table;
	Find the "link from" port in the port table;

	if ("link to" port has too many links from other ports)
	    Send a FAIL message back to the processor which sent this request;
	else
	{
	    Update link information for "link to" port;
	    Increment "link to" port's owner's migration size;
	    Send a SUCCESS message back to the processor which sent this request;
	}
}


LP_ACK ( ) /* routine to deal with receipt of a SUCCESS message from a
		remote processor to which a port link request had been sent */

{
	Find "link to" port in the port table;
	Find "link from" port in the port table;
	Update link information for "link from" port;
	Increment original calling process's migration size;
	Mark calling process as unblocked;
	Return link identifier to caller;
}
```

```
LP_NACK ( )        /* routine to deal with receipt of a FAIL message from a remote
                   processor to which a port link request had been sent */

{
      Mark original calling process as unblocked;
      Return FAIL to calling process;
}

UPORT ( )  /* routine for unlinking two previously linked ports */

{
      if ("link from" port does not exist)
          return FAIL to calling process;
      else
      if ("link to" port does not exist)
          return FAIL to calling process;
      else
      {
          Update "link from" port's link information;
          Decrement calling process's migration size;
          if ("link to" port is local)
          {
                Update "link to" port's link information;
                Decrement "link to" port's owner's migration size;
                Return SUCCESS to calling process;
          }
          else /* ie "link to" port is remote */
          {
                Mark calling process as blocked;
                Send message requesting a port unlink to processor on which "link
                to" port is resident;
          }
      }
}

UP_REQ ( ) /* routine to deal with receipt of a message from a remote processor
                   requesting a port unlink */

{
      if ("link to" port no longer exists)
          Send a FAIL message back to the processor which sent this request;
      else
      {
          Find "link from" port in the port table;
          Update "link to" port's link information;
          Decrement "link to" port's owner's migration size;
          Send a SUCCESS message back to the processor which sent this request;
      }
}
```

```
UP_ACK ( )/* routine to deal with receipt of a SUCCESS message from a
                remote processor to which a port unlink request had been sent */

{
    Mark original calling process as unblocked;
    Return SUCCESS to calling process;
}

UP_NACK ( )     /* routine to deal with receipt of a FAIL message from a remote
                processor to which a port unlink request had been sent */

{
    Mark original calling process as unblocked;
    Return FAIL to calling process;
}
```

#### 4.4.5.3 Message-related Kernel Calls

Message-related kernel calls deal with sending and receiving of messages by user processes through their software ports.

When a user process wishes to receive a message on one of its owned ports, either in a non-blocking or blocking mode, it uses the kernel calls `nb_rmsg()` and `b_rmsg()` respectively. `Nb_rmsg()` takes as parameters the identifier of the port on which the message is to be received, and a pointer into the calling process's address space, where the message is to be delivered; it returns a flag indicating success or failure of this operation to the calling process. `B_rmsg()` expects just the appropriate port identifier as a parameter, and returns a pointer to the subsequently delivered message in the caller's address space. For both of these routines, if the received message was sent in blocking mode, then the sending process is notified by the kernel of message receipt. The routines are described in pseudo code below:

```
NB_RMSG ( )      /* routine for performing a non-blocking receive operation */
{
      if ("receive" port does not exist)
          return FAIL to calling process;
      else
      if (a receive is already pending on this port)
          return FAIL to calling process;
      else
      {
          return SUCCESS to calling process;
      if ("receive" port has a message in its queue)
      {
          if (message in queue was sent in blocking mode)
          {
                    if (sending process is local)
                            Inform sending process of message receipt;
                    else /* ie sender is remote */
                            Send "notification of receipt" message to processor on
                            which sending process is resident;
          }
          Return message and address of function to deal with non-blocking receive, to
          calling process;
          Remove message from head of port's queue;
          }
          else /* no messages have arrived on "receive" port */
                    mark port as pending a non-blocking receive;
      }
}

B_RMSG ( )       /* routine for performing a blocking receive operation */
{
      if ("receive" port does not exist)
          return FAIL to calling process;
      else
      if ("receive" port has a message in its queue)
      {
          Return message to calling process;
          if (message in queue was sent in blocking mode)
          {
                    if (sending process is local)
                            Inform sending process of message receipt;
                    else /* ie sender is remote */
                            Send "notification of receipt" message to processor on
                            which sending process is resident;
          }
          Remove message from head of port's queue;
      }
      else /* no messages have arrived on "receive" port */
      {
          Mark calling process as blocked;
          Mark port as pending a blocking receive;
      }
}
```

Sending a message between two linked ports in either blocking or non-blocking mode is achieved by a call to `b_smsg()` or `nb_smsg()` respectively. Both of these routines expect as parameters the identifier of the sending port, the identifer of the link from that port on which the message is to be sent, the length and type of the message, and finally the message text itself. `Nb_smsg()` returns an indication of success or failure immediately to the calling process (since in non-blocking mode the user process does not wait for acknowledgement of message receipt); `b_smsg()` returns immediately if the send operation cannot be performed, but blocks the calling process until the message is received, if the send operation is valid. A further routine is provided in the kernel, which deals with the delivery of a message which was transmitted through the network, as the two ports involved in the message exchange were resident on different processors (i.e. the routine `usr_msg()`). We present the pseudo code for the `nb_smsg()` and `usr_msg()` routines below; (`b_smsg()` has been omitted here for conciseness since it is essentially identical to `nb_smsg()` without the immediate return to the calling user process):

```
NB_SMSG ( )        /* routine for performing a non-blocking send operation */

{
      if ("source" port does not exist)
            Return FAIL to calling process;
      else
      if ("destination" port does not exist)
            Return FAIL to calling process;
      else
      if (message type is incorrect for "destination" port)
            Return FAIL to calling process;
      else
      {
            Return SUCCESS to calling process;
            if ("Destination" port is local)
            {
                  if (a non-blocking receive is pending on "destination" port)
                  {
```

```
                Cancel non-blocking receive pending;
                Return message and address of function  to deal with
                non-blocking receive to owner of "destination" port;
            }
            else
            if (a blocking receive is pending on "destination" port)
            {
                Cancel blocking receive pending;
                Return message to owner of "destination" port;
            }
            else /* ie message is not awaited */
            {
                Append message to tail of "destination" port's queue;
            }
        }
        else /* ie "destination" port is remote */
            Send the message to processor on which "destination" port is
            resident;
    }
}


USR_MSG ( )     /* routine for dealing with arrival of a user message from
                   another processor */

{
    if (a non-blocking receive is pending on "destination" port)
    {
        if (message was sent in blocking mode)
            Inform sending process;
        Cancel non-blocking receive pending;
        Return message and address of function to deal with non-blocking receive
        to owner of "destination" port;
    }
    else
    if (a blocking receive is pending on "destination" port)
    {
        if (message was sent in blocking mode)
            Inform sending process;
        Cancel blocking receive pending;
        Return message to owner of "destination" port;
    }
    else /* ie message is not awaited */
    {
        Append message to tail of "destination" port's queue;
    }
}
```

### 4.4.6 Per-processor Scheduling

Since the processors on our simulated network are intended to be fully autonomous, scheduling of user processes resident on a particular processor is totally under the control of the local kernel - this means that we will be considering load balancing from a network-wide perspective, with no account of the effects of local scheduling. This approach has been taken since co-operation at the process scheduling level will create added overhead, and needs very strict synchronisation between processors.

The kernel enforces a scheduling policy using UNIX software signals; thus user processes pause when waiting to be scheduled, and are woken up by a signal from the scheduling routine in the kernel. Any appropriate policy can be chosen for selecting the next process to be scheduled, but for the purposes of our study we have chosen a simple round-robin mechanism.

### 4.4.7 Process Migration Mechanism

In order to experiment with different load balancing algorithms, the kernel must provide a means by which a process can be moved from the processor on which it is currently resident to a different processor chosen by an appropriate location policy. Since load balancing algorithms may require migration to be either preemptive (ie a process can be moved after it has begun execution) or non-preemptive (ie a process can only be moved as soon as it is created), our mechanism can be used in both of these modes. By far the most complex of these two alternatives is preemptive migration, since it is necessary for the kernel to re-route a migrating process's logical communications paths (which have been established through creating and linking ports) in a manner which is transparent to that process, and all of its peers (it would be

unreasonable, and highly undesirable, for user processes to be written such that they explicitly allow for possible migration to a different processor from the one on which they were originally created). The only restrictions which we have imposed on this mechanism is that a process cannot be migrated if it is currently scheduled, or is blocked waiting for completion of a kernel call (which may be due to the necessity to wait for an acknowledgement from another processor); we believe that it would be inadvisable to attempt process migration under these two circumstances. Since our kernel call/return mechanism works using named pipes and software signals, then from the user process point of view, migration merely involves changing the pipes which it uses, from those of the processor on which it is resident, to those of the processor to which it is being migrated; the kernels of both processors are left to deal with updating relevant tables and ensuring that the residency of all processes and their associated ports remains consistent.

When the load balancing algorithm being used on a particular processor wishes to invoke process migration, it calls a kernel utility function `migrate()`, and passes to it as parameters a pointer to the process table entry of the user process to be migrated, and the identity of the processor to which migration should be performed. `Migrate()` sends a software signal to the user process to inform it that it is being migrated, and passes it details of its new pipes to be used for making kernel calls and for receiving results of these calls (i.e. those of the processor to which it is migrating) and the UNIX process identifier of its new processor (which it needs for signalling purposes). A signal handling routine provided in the user process library file "interface.o", which has been linked into the user process deals with receiving this new information, and hence this remains transparent to user-written code. The kernel of the processor from which the process is migrating then constructs a message

(mig_info), into which it places all information regarding the migrating process together with all details concerning its owned ports (ie their names, links, and any messages which have arrived but have yet to be delivered to the process). This message is then transmitted to the processor on which the migrating process is to reside, the corresponding process table entry is re-initialised and all of the process's owned ports are marked as being resident at their new location; finally the new location of such ports is broadcast to all processors to allow them to correctly route subsequent user messages. At this point the migrating process is considered to have left its original processor, and to be in transit to its new processor.

The receiving kernel of a "migrating process" message (mig_info), installs the relevant information in its local process table, and updates its port table to include that process's owned ports with any user messages which were in the ports' queues; in order to maintain consistency of the interconnection of user process ports, their link information (i.e. pointers into the port table are also appropriately updated). The migrating process is now treated exactly as a normal local process, and becomes available for scheduling; thus all future kernel calls made by the process will now be serviced by its new processor's kernel.

This entire mechanism is summarised in the pseudo code shown below:

```
MIGRATE (process, processor)        /* routine to migrate a process to another
                                       processor */
{

    Put process table entry into a message (mig_info)

    for (each of process's owned ports)
    {
        Put port name in mig_info;
        Mark corresponding port table entry's residency as new processor i.d.
        Put port's link information in mig_info
        Put port's message queue in mig_info
    }
```

```
        Send migrating process a SIGMIG signal, together with details of its
        new processor;

        Send mig_info message to process's new processor;

        Decrement local process counts;

        Re-initialise process table entry;

        Broadcast new location of process's owned ports;
}


PINFO_MSG ( )           /* routine to deal with the arrival of a migrant process */

{
        Get process table information from mig_info message;

        for (each of process's owned ports)
        {
            Get port's details from mig_info and enter into port table;
            Resolve internal pointers for port links;
            Get port's message queue from mig_info and install it;
        }

        Increment local process counts;

        Send process a SIGCONT signal to allow it to continue on this processor;
}
```

# CHAPTER 5

# SIMULATION RESULTS

## 5.1 MAIN GOALS AND METHODS OF EXPERIMENTS

Previous studies of load balancing algorithms have mainly concentrated on their performance relative to a system using no load balancing, rather than comparing different algorithms and establishing the qualities which make one algorithm superior to another; in addition, these studies often do not incorporate all of the overheads caused by executing such algorithms, both in terms of information exchange and process migration.

Our main goal in this study is to provide an analysis of the relative merits of a number of adaptive load balancing algorithms under differing workloads, and with both a simple and more complex model of user processes which are executing on a loosely-coupled distributed system. Other studies have considered processes as independent quantities, and have not analysed the effect of load balancing algorithms when processes are submitted to the system in groups, cooperating to achieve a common goal, and hence requiring a means of exchanging information. We have chosen to provide message-passing primitives in our distributed operating system kernel to allow such information exchange.

To this end we have structured the study in three main phases. The first phase was designed to validate our model, by reproducing results obtained by Eager et al [86], and also to examine more of the underlying behaviour of load balancing algorithms, dealing with very simple user processes. The second phase introduces a more

complex type of user process to the system, with groups of such processes arriving at any node in the network and cooperating through message-passing. Finally, the third phase takes the load balancing algorithm with the best performance and analyses variations on its operation to investigate how it can be tailored to a particular environment (for example, measures which should be taken if the communications medium used cannot cope with the heavy load imposed on it by some algorithms).

Since our simulation is largely parameterised and hence allows easy experimentation with different environments, the number of possible combinations of parameters used needs to be limited, to enable us to make meaningful comparisons; for this reason, certain parameters (especially regarding the network topology and communications medium speeds) were held fixed across all experiments, whilst others (such as system load) were varied and their influence on the load balancing algorithms under investigation was noted.

## 5.2 EXPERIMENTAL ENVIRONMENT

Below we describe the choice of values for the major parameters of our simulated network, and the generation and execution of user processes to give a synthetic workload to the system.

### 5.2.1 Network Topology

We have chosen to use a point-to-point network in our study, since shared communications media tend to limit the number of processors which can be interconnected efficiently. The particular point-to-point topology which we have used

is a square mesh, which we consider to be easily extended and to offer a reasonable level of connectivity [Wittie81]; each node in the network has at most four links to its immediate neighbours. Since the number of processes which can coexist on the NCR Tower at any one time has an absolute upper limit, and there is also an upper bound on the number of file desciptors which can be opened simultaneously (even when operating with "super user" privileges), we had to restrict the number of simulated processors in the network to nine - hence we use a 3 x 3 square mesh topology. We consider, however, that this is of sufficient size for our purposes of experimenting with load balancing algorithms in a point-to-point environment.

Our simulation allows the use of any routing algorithm for transmitting messages through the network; since the goals of our experiments are concentrated around load balancing algorithms, we used a fixed routing algorithm, which uses routes established manually before the system is booted. Thus when the system is started, each processor is passed an indication of which of its physical links it should use to transmit messages destined for other processors on the network. In fig. 5.1, we show the 3 x 3 mesh topology used, together with details of fixed routing; processors are shown as circles, with arcs representing physical interconnections. Each processor is given an identifying integer in the range 0-8, and routes are shown as arrows along a physical link between processors. For example as shown in fig. 5.1, a message sent from processor 0 to processor 8 follows a route via processors 1, 2, and 5 before arriving at its destination. The manner in which routing is represented for each processor is illustrated by showing the appropriate data structure for processor 3 in the aforementioned figure.

Processor and communications medium speeds are also tunable parameters of our simulation. For our study we chose the processor speed to be 1 MIPS, which we

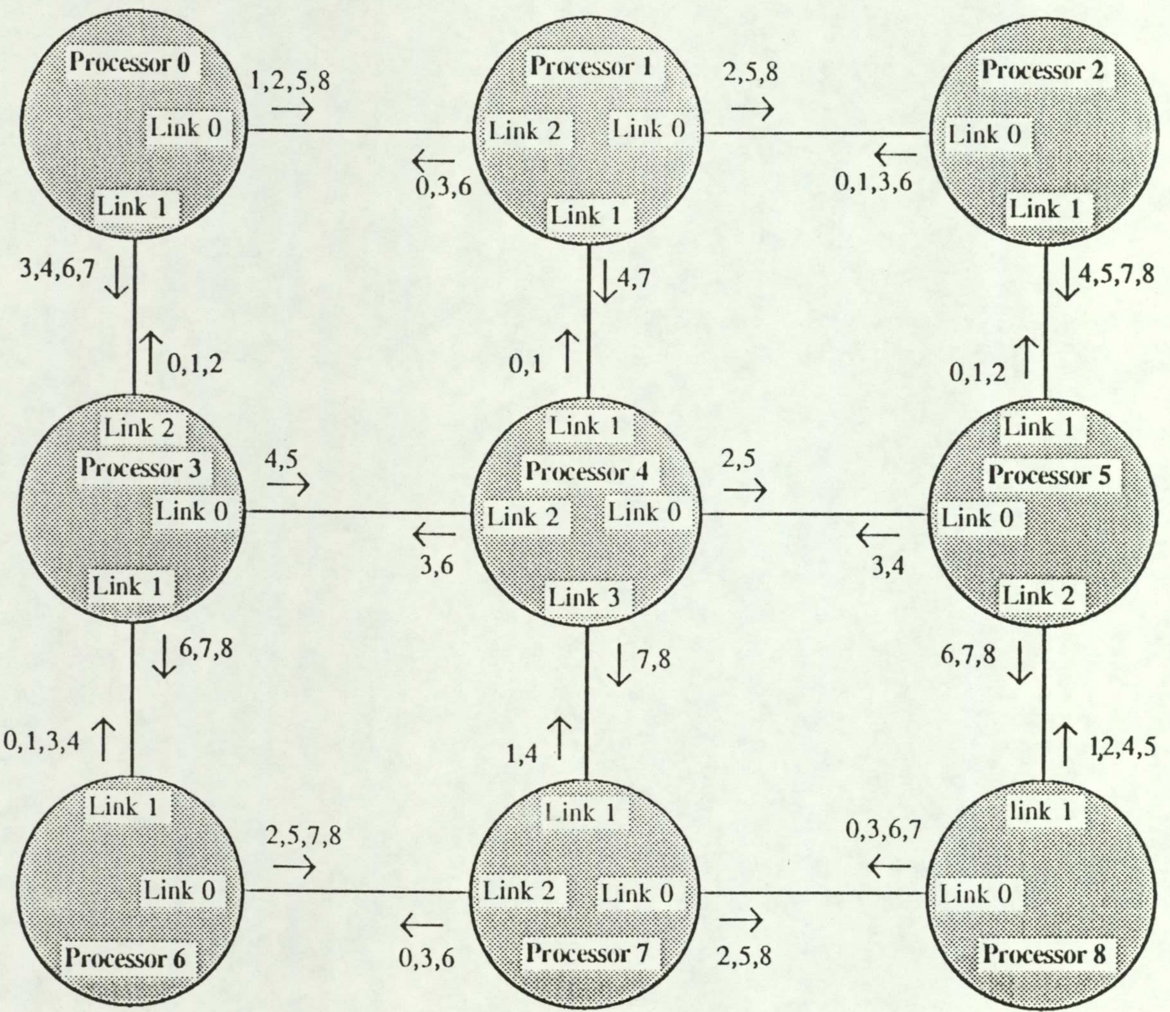



Fig. 5.1 Square Mesh Topology used in our Experiments and Example Route Table for Processor 3

**Route Table for Processor 3**

| Processor | Link |
|---|---|
| 0 | 2 |
| 1 | 2 |
| 2 | 2 |
| 3 | --- |
| 4 | 0 |
| 5 | 0 |
| 6 | 1 |
| 7 | 1 |
| 8 | 1 |

consider a reasonable value for existing machines. The overhead due to use of the communications medium can be divided into two parts - a fixed overhead due to the time necessary for executing the system software dealing with communications protocols, and a further overhead depending on the time taken to transmit variable length messages (which can be expressed as message-length x time necessary to send/receive one byte). We set the fixed software overhead per message receipt/transmission at 1000 machine instructions, and the physical speed of the communications medium to be 1Mbyte/sec; again these were based on values which we considered reasonable.

### 5.2.2 Synthetic Workload Generation

As previously mentioned, a synthetic workload arriving for service at our simulated network is generated by providing each simulated processor with a file containing the arrival times of user processes. Each occasion that a processor's value for the passage of real time is updated, the appropriate per-processor file is inspected to establish whether a new user process is due to arrive; if this is the case a call is made to the kernel routine `cproc()`, passing the name of the file holding the process's executable image, together with a unique process group identifier; this can be thought of as a UNIX-like shell process which is servicing user-submitted commands from a terminal connected to a particular processor.

In order to place varying loads on the network we used the well known formula for system load ($\rho$) [Lavenberg83], with processes arriving at any one of m available processors :

$$\rho = \lambda \, E[S] \, / \, m \, \gamma,$$

where $\lambda = 1/E[T]$,

using the following notation :

$\rho$ = system load    ($0 \leq \rho \leq 1$ for a stable system)

$E[S]$ = expected service time for a process

$\gamma$ = service rate for each processor

$\lambda$ = process arrival rate

$E[T]$ = mean interarrival time

Given the above equations, we can vary system load by shortening or lengthening the mean interarrival time for user processes; the main configuration file for the network ("config.main") specifies to our simulation the system load value required for one simulation run, together with the total number of processes which will pass through the system.

We consider the arrival of user processes to our system as a Poisson arrival process, with interarrival times being random, exponentially distributed and with mean $1/\lambda$. A Poisson arrival process with arrival rate $\lambda$ can be decomposed into m independent streams [Kobayashi78], each being a Poisson process, with rate $r_k \lambda$, $k = 1,2, ....m$, where $r_k$ is the probability that stream k will be chosen for an arriving process; we use these properties to generate process interarrival times for each processor, which are then stored in their per-processor job arrival files. Before the simulated network is

booted, the initial start-up process establishes arrival times of processes to the network from an exponential distribution. This is achieved by generating a uniformly distributed random number U, (using the UNIX 48-bit linear congruential random number generator `erand48()` ) and using the inverse transformation $-\lambda^{-1}\log_e U$ to form an exponentially distributed random number with mean $1/\lambda$. Since processes can arrive at any processor in the network (with equal probability), the time of each arriving process is placed in a randomly chosen processor's job arrival file (jobs0, jobs1, ... jobs8), by using a further uniformly distributed random number. The above procedure ensures that each processor will receive processes with mean arrival rate $\lambda/9$, and exponentially distributed interarrival times.

In order to guarantee that our results would not be dependent on the seed chosen for the random number generator we conducted our initial experiments with a number of different seeds, to assure ourselves that, given sufficient passage of simulated time, the same results would be achieved. This indeed was found to be the case. We then selected one of these seeds for use in all subsequent experiments, with the main criteria being that this seed gave as near to a truly exponential distribution as possible over approximately 3-4,000 seconds of simulated time.

### 5.2.3 Load Balancing Algorithm Implementation

When implementing a load balancing algorithm, the decision must be made whether to have a separate privileged process running outside the operating system kernel dealing with all load balancing considerations, and using kernel calls to migrate processes and exchange load information, or to embed load balancing code into the routines of the

kernel itself. For purely practical considerations we chose the latter approach, since it provides easy access to all kernel data structures, and hence eases implementation. Thus for each load balancing algorithm which we investigated, code for the algorithm was inserted, either in the main `kernel()` routine or in the process creation routine `cproc()` as was appropriate to the particular algorithm (i.e. depending on whether the algorithm is invoked only on process creation or periodically). In order to facilitate changing from one algorithm to another, we added a number of utility routines to the kernel for measuring processor load, migrating processes, and for exchanging processor load information across the network; in this manner, modification of the kernel when changing the load balancing algorithm required only the insertion of calls to the appropriate utility routines.

### 5.2.4 Performance Metrics and Monitoring

To establish the differences in performance between load balancing algorithms and to analyse the reasons for these differences, we need to identify useful performance metrics and provide a means for recording these during simulation runs. It has been shown [Baumgartner85] that the average time needed for processes to execute to completion is a good measure of the efficiency of processor allocation, hence we have taken this as the principal metric for our performance evaluation of load balancing algorithms. Graphs showing the relative performance of load balancing algorithms are included in the main text of this chapter.

In order to study performance differences in more detail we also took measures of the mean load imposed on a single processor (since reducing this value should result in performance improvement) and also the variance of individual processors' load from

this mean (since this gives an indication of how well an algorithm redistributes load across the network); as we are dealing with a small sample (load values for 9 processors), variance was calculated as $\Sigma(x-\bar{x})^2/(n-1)$. In addition to these metrics, we chose to measure the absolute difference in load between the most heavily-loaded and the least heavily-loaded processors - this provides a good indication of whether an algorithm results in one overloaded processor whilst another processor remains underloaded [Livny82]; it may be suggested that this difference is of more fundamental importance than the measure of processor load variance - a reasonable algorithm should significantly reduce the difference. For our later experiments, which include user processes exchanging messages, we added a measure of message traffic in the network, and also the rate of process migration from one processor to another.

In all of the simulation runs which were conducted, the system was allowed to continue until the average per-process execution time converged sufficiently to provide a meaningful value for comparison to be made between algorithms (i.e. the average process execution time for each processor differed by less than 2%, which for most load balancing algorithms was after approximately 2-4,000 seconds of simulated time). Over this period, details of the above metrics were written to a number of trace files, with each processor having its own file (trace0, trace1, ... trace8). This means that not only were we able to look at overall network behaviour and performance, but also that of each individual processor; these values were sampled every 10 seconds of simulated time in our experiments.

We present the results of our experiments in the following sections, both in graphical and tabular form where appropriate; tables summarising system behaviour are presented in the main text. For reasons of clarity, graphs showing overall network

behaviour are given for the first 1,000 seconds of simulated time, however these are suitably representative of the differences between performance of the load balancing algorithms which we implemented, over the entire period of the simulation run. These graphs are presented in Appendices A and B, and references to them in the text give the appropriate figure number beginning with the letters A and B respectively. It should be noted that since each processor has a "shell" process constantly resident on it (and not eligible for migration), then the minimum number of processes on a processor at any time is one; hence when the mean load per processor is shown in our graphs as having a value of one, then this represents the state where no user processes are currently extant in the system.

## 5.3 INDEPENDENT USER PROCESSES

Our initial experiments were conducted in an environment where user processes execute with no message-passing, thus we term these independent processes.

### 5.3.1 Independent User Process Model

Following the choice of parameters in the work on simple load balancing algorithms by Eager et al [86], we set the expected service time required by a user process (E[S]) at one second, to experiment with different algorithms in a simple environment. All processes were considered to be CPU-intensive and did not use any I/0 kernel calls (i.e. message-passing calls). In order to simulate this activity we added a further kernel call `do_processing()` to our system, allowing a process to execute for a specified number of microseconds, where this value is passed as a parameter. Each

call to `do_processing()` updates the execution time of the calling process, and also the passage of real time together with the existence and residency times for all processes residing on the servicing processor. We set the time slice for our round-robin scheduling algorithm at 50 msecs (chosen through experimentation), and so a one-second process was simulated by making 20 calls to `do_processing()` with 50,000 microseconds as a parameter. The time slice of 50 msecs was maintained in all further simulation runs.

To simulate the overhead of transferring the executable image of a migrant process from one processor to another, since we are dealing with a model of small, simple processes, the size of a typical user process was chosen to be 10K bytes.

### 5.3.2 Load Balancing Algorithms Implemented

For the purposes of the analysis of load balancing algorithms in a simple process environment, we implemented two simple non-preemptive algorithms which we term Random and Threshold (similar to those used by Eager et al [86]) and one more complex, preemptive algorithm which we term Global Average (based on the Above Average algorithm of Krueger and Finkel [84], but with slight modifications). Below we describe the algorithms and the results which were obtained.

#### 5.3.2.1 Processor Load Measurement

For the two simple algorithms (Random and Threshold), we defined the load on a particular processor to be the number of processes resident on the processor at the time of measurement. Since the algorithms are non-preemptive, load measurement was only

performed on the arrival of a new process. For the preemptive algorithm, Global Average, a more complex measure of processor load was used; this was defined as the number of non-blocked processes (the "actual" load) plus the number of processes known to be currently in transit to a processor (the "virtual" load), averaged over a period of time divided into discrete quanta (we chose a period of ten 20 millisecond quanta, since Krueger and Finkel [84] state that this should be at least as long as the time taken for a process to migrate).

#### 5.3.2.2 Random Algorithm

The Random load balancing algorithm uses no information exchange at all between processors; when the load on a particular processor is found to exceed some threshold, a newly arriving process is sent to a randomly chosen alternative processor, regardless of that processor's own load value. We set the threshold to 2, which we found to be the most effective under all workloads, and we seeded each processor's uniform random number generator with that processor's identifying integer, to ensure that the same sequence of random numbers was not generated in all processors (this would have led to processes being transferred to the same processor).

#### 5.3.2.3 Threshold Algorithm

The Threshold algorithm uses very small amounts of information exchange to negotiate the movement of processes from heavily-loaded processors to their more lightly-loaded peers. When a processor's load exceeds a fixed threshold, it chooses another processor at random and sends it a probe message requesting whether migration of the process which caused the overloading on the sender, would result in the probed processor's load being above the threshold. If the reply to the probe

indicates that the probed processor's load is below the threshold, then the newly-arrived process is migrated there, otherwise another random processor is selected, and this procedure continues until either a suitable destination is found or a static probe limit is exceeded (in which case the process must be executed locally).

We chose the fixed threshold to be a load value of 2, as in the Random algorithm, and the probe limit to be 3 (as in Eager et al [86]). In order for a processor to handle probing on behalf of multiple processes at one time, our implementation of the Threshold algorithm uses the probe limit on a per-process level - in other words when a process arrives which causes a particular processor's load to exceed the threshold, probing on behalf of that process begins independently of any probing which was already being carried out on behalf of processes which arrived earlier.

The following pseudo-code shown below summarises the operation of this algorithm. On process arrival each processor executes the following :

```
if (processor_load > THRESHOLD)
{
        send probe to randomly chosen processor
}
```

On receipt of a probe a processor takes the following action :

```
if (processor_load+1 > THRESHOLD)
{
        send a refusal for a migrating process
}
else
{
        send an acceptance for a migrating process
}
```

Finally when a processor receives a reply to a previous probe, the following steps are executed :

```
if (processor_load > THRESHOLD)
{
        if (reply = = acceptance)
        {
                migrate the process which caused the probing
        }
        else
        {
                if (number_of_probes < PROBE-LIMIT)
                {
                        send probe to another processor
                }
                else
                {
                        execute process locally
                }
        }
}
else
{
        execute process locally
}
```

#### 5.3.2.4 Global Average Algorithm

In the Global Average algorithm, as previously stated, the measure used for processor load is the sum of a processor's actual load together with its virtual load (the number of processes in transit), all averaged over a time period. Each processor maintains a value for the average load across the whole network and strives to keep its own load to

within a pre-defined acceptable range around this average value; whenever a processor believes that this average is not accurate it broadcasts a new value, which is then used by all other processors as the current global average. A processor considers that it is overloaded if its load is above the average by more than the acceptable range, and underloaded if its load falls below the average by more than this range.

The global average is maintained using a number of "timeouts". If an overloaded processor cannot find an underloaded partner to which it can migrate one of its processes within a specified timeout period, then the global average value is too low and must be increased. A similar mechanism is used for underloaded processors, in order to decrease the global average when it is found to be too high. Timeouts are also used in conjunction with a processor's virtual load; since many processors may attempt to accept processes from an overloaded processor, each acceptance increases the virtual load of the accepting processor to avoid that processor receiving a large number of migrant processes. When a processor announces its willingness to receive extra work, it sets a timeout which, if it expires without the expected process arriving, will indicate that the particular process was migrated to another processor, and will hence never arrive at the accepting processor. This ensures that a migrant process is not waited for indefinitely, to no avail.

In order to implement this algorithm we added a timeout mechanism to our kernel, where timeouts are of three types : `too_low`, `too_high` and `awaiting_process`. We decided that for optimisation reasons a processor could only have one `too_low` or `too_high` timeout pending at any one time. However for `awaiting_process` timeouts we need a different mechanism; since it is desirable for an underloaded processor to be able to announce its availability to more

than one overloaded processor (provided that this does not result in that processor being inundated with migrating processes), we allowed `awaiting_process` timeouts to be queued. When an `awaiting_process` timeout needs to be set, it is added to the tail of the queue, and if a timeout expires, or the awaited process arrives, one timeout is removed from the head of the queue.

The operation of the algorithm is thus driven by any of the following events : a processor becomes underloaded or overloaded, a timeout expires, a migrating process arrives, or the global average value needs to be updated. Below we present the possible events shown in italics, together with the actions which must be performed when these events occur :

*a process detects that it is overloaded :*

```
if (too_high timeout not pending)
- broadcast a too_high message
- set a too_high timeout
```

*a processor detects that it is underloaded :*

```
if (too_low timeout not pending)
- set a too_low timeout
```

*a processor receives too_high message :*

```
if (processor is underloaded)
{
    - send accept_process message
    - queue an awaiting_process timeout
    - increment processor's virtual load
}
- cancel any pending too_low timeouts
```

*a processor's too_low timeout expires :*

if (processor still underloaded)

- broadcast a new lower global average value

*a processor's too_high timeout expires :*

if (processor is still overloaded)

- broadcast a new higher global average value

*a processor receives a new global average value :*

- update locally hold value for global average
- cancel any too_low or too_high timeouts which are pending

*a processors awaiting_process timeout expires :*

- decrement processor's virtual load

*a processor receives a migrating process :*

- dequeue one awaiting_process timeout
- decrement processor's virtual load
- install the migrating process as a local process

A number of parameters need to be established for this algorithm to work efficiently; one of these is the timeout period, which must be set at a value which represents an amount of time sufficient for a message to pass from an overloaded processor to an underloaded processor and vice versa; in other words when an acceptance message does not arrive at an overloaded processor within the timeout period, since the processor broadcast its plight, then it can be assumed with reasonable certainty that such a message will never arrive and this implies that no suitable destination for a migrating process can be found without raising the global average load value. In our system we set this timeout period to 200 milliseconds. We used an acceptance range of 1.0 around the global average value (since Krueger and Finkel [84] suggest the range should be at least the size of the load imposed by a single process), and chose

the amount by which to increment or decrement this value, when this proves necessary, by 0.5 (where one non-blocked process imposes a load of 1.0 on its processor).

### 5.3.3 Discussion of Results

In fig. 5.2, we present the average time taken for a process to complete its execution under varying system loads using the algorithms discussed in the preceding sections. In addition to this graph, we also present more in-depth analysis of process behaviour in the form of the mean number of processes per processor, the variance around this mean, and the absolute difference between the load of the most-heavily loaded processor and that of the least heavily loaded processor. For reasons of conciseness we shall henceforth refer to these values as mean load, load variance, and load difference respectively. The values were sampled at low load (0.2), moderate load (0.5), and high load (0.8), and are plotted against time on our graphs. The motivation behind considering load variance and difference was taken from the Unbalance Factor due to Livny and Melman [82]. In their work, the Unbalance Factor (UBF) is defined as:

$$UBF = \max \{(n_i - n_j)/n_j, \text{ for all } i,j \leq N\}$$

where $n_i$ denotes the number of tasks on processor i, and N denotes the total number of processors in the network. Henceforth we shall refer to a system exhibiting low load variance or load difference as one showing a good balance factor.

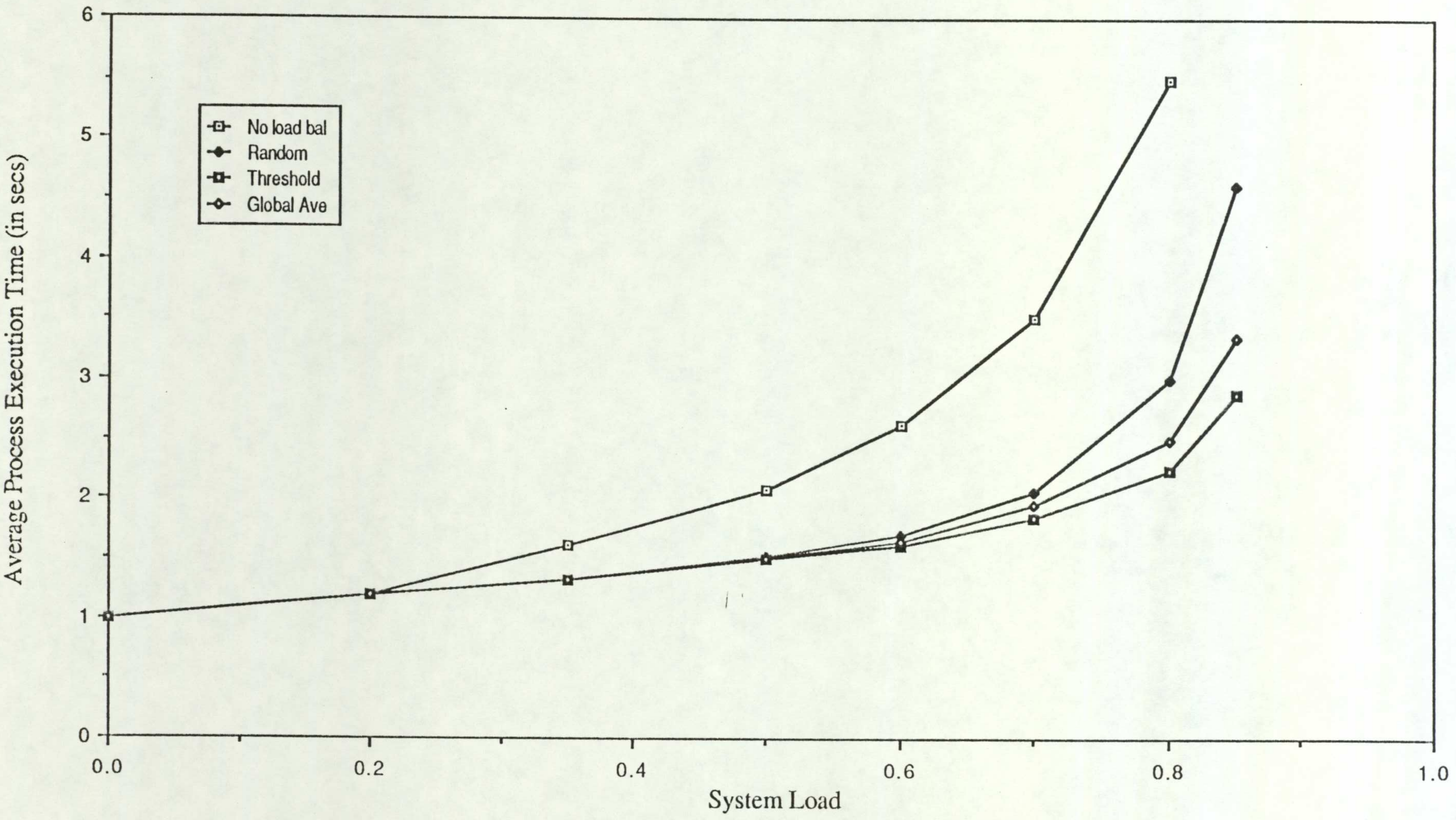



Fig. 5.2 Performance Comparison using the Independent Process Model

It can be seen from fig. 5.2, that at a system load of 0.2, there is no difference in the average execution time of processes running on the system whether load balancing is used or not (which is what we would intuitively expect, since at such a low load value, processor overloading is a fairly rare phenomenon). It is interesting, however to observe that even at this low system load, the behaviour of the system in terms of mean load, load variance and load difference, is improved when any of the three load balancing algorithms being investigated is used. Since there is no discernible difference in system behaviour between these algorithms, we choose to plot the no load balancing case against the threshold algorithm. Fig. A.1.1 shows that the mean load is marginally higher when using no load balancing, as against using the threshold load balancing algorithm; Fig. A.1.2, shows the improvement in the system's balance factor by using a load balancing algorithm, since the load variance across the network remains constant and low, thus removing the occasional "peaks" of variance experienced with a system using no load balancing; moreover, fig. A.1.3, shows that the threshold algorithm results in the load difference only rarely exceeding a value of one (the load exerted by one process), which indicates that, even at light system load, use of a load balancing algorithm shows a marked improvement over the no load balancing case. We deduce that the reason why these improvements in the system's balance factor do not translate into improvements in average process execution time at a light system load (0.2), is that the overheads incurred by executing the load balancing algorithm itself and by transferring a process from one processor to another are sufficient to cancel out the benefits of more evenly distributed load. The behaviour of the system at low load is summarised in Table 5.1.

| Statistics / Algorithm | Mean Load | Load Variance | Load Difference |
|---|---|---|---|
| No Load Balancing | 1.309 | 0.441 | 1.16 |
| Random | 1.204 | 0.219 | 1.01 |
| Threshold | 1.204 | 0.214 | 1.01 |
| Global Average | 1.204 | 0.227 | 1.09 |

Table 5.1 Overall System Behaviour using the Independent Process Model (Load Value = 0.2)

Returning to fig. 5.2, we note that for a system load value of 0.5 and above, the average process execution time when no load balancing is employed, continues to rise sharply, whilst all three load balancing algorithms show a less drastic degradation in their performance. Similarly to our analysis of system behaviour at a load of 0.2, we sampled mean load, load variance and load difference at a system load of 0.5 for the no load balancing case and using the random, threshold and global average algorithm. Again there is no discernible difference in the behaviour of the three algorithms investigated and so threshold was chosen to be displayed against no load balancing. Here we note a marked improvement in mean load, as shown in fig. A.2.1; naturally, reducing the overall load imposed on the system at any one time is a significant factor in the observed difference in average process execution time between using no load balancing and using the threshold algorithm; the improved balance factor of the system is even more marked, as exemplified by the load variance, as shown in fig. A.2.2 and by the fact that the threshold algorithm maintained the load difference of the system at a level rarely exceeding the equivalent of two user processes (fig. A.2.3). The above behaviour is summarised in Table 5.2.

| Statistics / Algorithm | Mean Load | Load Variance | Load Difference |
|---|---|---|---|
| No Load Balancing | 2.091 | 2.107 | 3.16 |
| Random | 1.769 | 0.693 | 1.97 |
| Threshold | 1.742 | 0.609 | 1.93 |
| Global Average | 1.750 | 0.694 | 1.95 |

Table 5.2 Overall System Behaviour using the Independent Process Model (Load Value = 0.5)

As we would expect, it is for load values above 0.7 that the benefits of load balancing can truly be observed. It is also at this point that a distinction can be drawn between the three algorithms which were implemented for this first set of experiments. We chose to sample mean load, load variance and load difference at a system load of 0.8, which, from fig. 5.2 can be seen to be where a significant performance difference emerges between the three algorithms. Fig. A.3.1 shows that even the very simple random algorithm considerably reduces overall mean load on the network, whilst as illustrated in fig. A.3.2, the global average algorithm gives even further improvement; the performance of global average and threshold, with respect to mean load is almost identical as shown in fig. A.3.3. Similar behaviour can be observed regarding the balance factor of the system at high load; fig. A.3.4 illustrates the high and erratic load variance when no load balancing mechanism is used, and also indicates the improvement obtained by the random algorithm. By inspection of figs. A.3.5 and A.3.6, it can be seen that global average and threshold both give further improvement to the balance factor than the random algorithm, with global average being the better of the two. A similar performance ranking, with respect to the load difference of the system is exhibited in figs. A.3.7, A.3.8, and A.3.9. The behaviour of the system at

high load is summarised in Table 5.3.

| Algorithm \ Statistics | Mean Load | Load Variance | Load Difference |
|---|---|---|---|
| No Load Balancing | 4.983 | 21.059 | 10.31 |
| Random | 3.138 | 4.188 | 4.32 |
| Threshold | 2.817 | 1.449 | 3.26 |
| Global Average | 2.893 | 1.208 | 2.98 |

Table 5.3 Overall System Behaviour using the Independent Process Model (Load Value = 0.8)

### 5.3.4 General Observations

From these results we can see that for system loads above a moderate load of 0.5 a system which uses no load balancing mechanism suffers severe performance degradation, and the benefits of load balancing are evident. This has been shown to be due to periods of very heavy overloading of certain processors whilst others remain underloaded (in the no load balancing case) illustrated by the load difference graphs which we have presented; also the general distribution of load is unbalanced, as seen from the load variance graphs. This difference in performance is not experienced at low load, partly because the load balancing algorithms are not often invoked at light system loading, and partly because the added overheads of executing a load balancing algorithm is not justifiable enough for the balanced load distribution which they bring.

It is to be noted that, although the random algorithm does not attempt to make "sensible" choices for candidate processors to which to offload processes from an overloaded processors, the problems of high load variance is reduced using this

algorithm as opposed to the no load balancing case. However, as we would expect, at high system load, this algorithm still causes moments of high load variance and load difference. We believe that the reason why random performs well at moderate load (but not at high load) is that the probability of finding a randomly-chosen processor which is not itself overloaded is quite high, when moving processes from an overloaded processor.

The algorithm which performed best over all system load values was threshold. It can be observed that the poor balance factor which was still evident in the system when using random load balancing, is considerably reduced when using threshold, since it guards against migrating processes to an already overloaded processor by its probing mechanism. Since threshold maintains the simplicity of the random algorithm, and hence incurs very little extra overload from its limited information exchange, we can deduce that in this simple environment, threshold is quite adequate for bringing significant performance improvement.

An interesting point should be noted regarding the performance of global average in these experiments. Although the algorithm uses more co-operation between processors in the network, and hence uses a more accurate picture of overall load, the performance of global average is only superior to that of random for high system loads, and does not equal the improvement shown by threshold. Despite the fact that the network load is marginally more evenly distributed at high loads using global average rather than threshold, this improved balance factor is counteracted by the overheads of the message exchange necessary to maintain the globally agreed average load value in such a volatile environment. It should be pointed out that, since processes do not remain in the system long, this more complex algorithm is not able to show its true potential. We expect the results to be different for our experiments to be

presented later in this chapter, where processes arrive in groups, co-operating via message-passing to achieve a common task.

Thus our results from a simple environment, where short-lived processes execute independently of their peers, support the view of Eager et al [86], that added complexity in a load balancing algorithm does not give corresponding performance improvement (in fact complexity is seen to degrade performance in this environment).

## 5.4 CO-OPERATING PROCESS GROUPS USING MESSAGE-PASSING

The second stage of our experiments with load balancing algorithms was designed to examine their relative performance in a system where processes form logical groups which exchange interprocess messages. In the following sections we present the changes made to our experimental environment in order to carry out this study and the results which were obtained. The characteristics of the network (communications speed, network topology, processor speed, number of simulated processors) all remained unchanged for this series of simulation runs. We measured the network mean load, load variance and load difference at sampled intervals of 10 seconds of simulated time as in the previous sections of this chapter, but also recorded the number of process migrations and the number of messages transmitted across the network, to add further analysis to our results.

### 5.4.1 Process Group Model

Our distributed operating system kernel was designed to provide facilities for the creation and execution of logical groups of processes where each group has an

identifier which is guaranteed to be unique across the network. For the purposes of our experiments on load balancing algorithms we chose an environment where the workload consists of groups of either two, three or four processes.

A group is formed by creating a group parent process which then uses the kernel call `cproc()` to create the appropriate number of child processes in the group. Such a process group is thus a representation of a user application which has been structured as one controlling parent process, which requests service from its children in order to divide its task into a number of concurrent transactions. Each such transaction consists of a message being sent from the parent process to one of its children requesting a service provided by that child; when a child receives a request it services that request and returns a result to the parent via a reply message. This model is not purely a Remote Procedure Call interface, since the children also perform processing of their own when they are not dealing with parental requests for service. Hence this can be viewed as a number of processes which rendezvous at particular points when they need to exchange data.

The interprocess communication necessary to implement the above process group model was performed using the software port mechanism of our kernel. The parent process of each group creates its own port, to which it gives the name "parent_port", and uses this to send all messages to its children and also for receiving their replies. Each child process in a group also creates its own port, giving it the name "child_port", together with an integer specifying to which child in the group this port belongs (for example in a three-process group, consisting of a parent and two children, child1 uses child_port1 and child2 uses child_port2). Two-way communication between parent and child processes is thus established by the parent

linking its port to each of its childrens' ports and vice versa.

Since in a real application written in the style of this process group model, the message traffic between the parent process and its children will not necessarily be symmetrical (i.e. the parent will communicate more frequently with one child than with others), we need a "weighting" scheme to determine the number of request/reply transactions which the parent undertakes with each child process. In order to impose an expected service time on the process group the parent process was written to perform a constant number of transactions with its children (this constant, MAX_TRANS, being set at 10 for our experiments). So as to achieve asymmetrical interprocess communication, the "weighting" scheme was implemented in the following manner : in a two-process group all parental transactions are with its single child; in a three-process group MAX_TRANS/3 transactions are directed to one child process and (2xMAX_TRANS)/3 to the other; in a four-process group the division of transactions between child processes is MAX_TRANS/6, (2xMAX_TRANS)/6, and (3xMAX_TRANS)/6. All transactions between a parent and its children were initiated by first generating a uniform random number and then transforming this appropriately to give the above "weighting" for each child. When the parent finishes its MAX_TRANS transactions it sends a termination message to each of its children, which then exit cleanly. The entire operation of every process in a group was written using the message-passing facilities provided by our kernel.

To limit the effect of many random parameters in the system we fixed the size of message sent between parent and child processes at 1K bytes; these messages are sent in non-blocking mode and received in blocking mode by all parent and child processes. We present below a pseudo-code description of the parent and child process in a group :

**Parent Process**

```
Create ("parent-port")
Create (appropriate number of children)
Link ("parent-port" to childrens' ports)
for (i = 0; i <MAX_TRANS;i++)
{
    do_processing (250 milliseconds) /*CPU-intensive processing*/
    send_msg (to randomly-chosen child) /*non-blocking*/
    do_processing (250 milliseconds)
    receive_msg (reply from child) /*blocking*/
}
for (j=0; j < num_children; j++)
    send_msg (termination message to child j)
Unlink ("parent_port" from childrens' ports)
Destroy ("parent_port")
Exit
```

**Child Process**

```
Create ("child_port")
Link ("child_port" to "parent_port")

forever
{
    do_processing (250 milliseconds) /* CPU-intensive processing */
    receive_msg (request from parent) /* blocking */
    do_processing (250 milliseconds) /* processing parent request */
    send_msg (reply to parent)
    if (message is "termination message")
    {
        Unlink ("child_port" from "parent_port")
        Destroy ("child_port")
        Exit
    }
}
```

### 5.4.2 Process Group Synthetic Workload Generation

In our study of independent processes, the interarrival times of processes were generated from an exponential distribution to form a Poisson arrival process. Since the workload on the system arrives in groups under our process group model described above, we chose to allow groups to arrive with exponentially distributed interarrival times, lengthened by a constant multiplier to account for the extra load imposed on the system; thus for our study on load balancing algorithms in a process group environment, the arrival of a group is a Poisson arrival process with rate $\lambda$, but the creation of processes within a group is at the instant of that group's arrival; it has been shown [Lavenberg83] that under this workload, expected response time is poorer than when the arrival of individual processes is Poisson.

As previously stated, we concentrated on studying two, three and four-process groups, and so our synthetic workload generation phase performed at system startup, was augmented to generate arrival times, together with a random integer (either 2, 3 or 4) drawn from a uniform distribution to indicate the size of the arriving group; hence over a sufficiently long period of simulated time, the expected number of groups of each size will be equal. The constant multiplier for interarrival times was set at the average time necessary for a process group to execute to completion, assuming that no interference is experienced due to other processes in the system; through experimentation this value was found to be 9 seconds.

Thus the synthetic workload generation phase can be summarised as follows :

- generate random size process group (2, 3 or 4)

- generate exponential time since last group arrival
- multiply interarrival time by expected process group execution time

These steps are repeated until arrival times have been generated for the number of process groups required by a particular simulation run.

### 5.4.3 Load Balancing Algorithms Implemented

In order to continue our study of load balancing algorithms by examining their behaviour in a cooperating process group environment, we implemented the same algorithms presented in previous sections, (i.e. random, threshold and global average) together with a modification of the threshold algorithm, which was designed to see whether performance of this algorithm would alter if it were operated in a preemptive manner (we term this algorithm "preemptive threshold"). We present here, details of any necessary changes made to the original algorithms (mainly in terms of parameter choice) in order to allow them to function as efficiently as possible in this new workload environment, and also a description of the "preemptive threshold" algorithm.

During initial trial runs of the two simple, non-preemptive algorithms (random and threshold), we found that the original threshold value of 2, defining the load value above which a processor considered itself overloaded, thus needing to transfer newly processes to another processor was inadequate to deal with processors arriving in groups. In fact at moderate and heavy system load, using a threshold value of 2, these algorithms gave little improvement in system performance over the no load balancing case, and in certain pathologically high peaks of general system load, they actually resulted in a performance degradation due to instability (as defined in Chapter 3). We attribute this fact to the threshold value being set too low, thus causing many attempted

migrations from an overloaded processor with no consideration as to the distribution of migrating processes; recall that the random algorithm moves a process to a randomly-chosen alternative processor, and that threshold will try to negotiate many fruitless migrations through its probing mechanism, if all processors are loaded near to or above the threshold value. By experimentation, we concluded that in the process group environment a threshold value of 4 was appropriate (i.e. that a processor considers itself to be overloaded if there are more than 4 processes currently executing on it). Thus the random and threshold algorithms were modified to use this new threshold value for our subsequent simulation runs; otherwise they remained unchanged, from their description in sections 5.3.2.2 and 5.3.2.3.

It should be noted that this indicates a fundamental flaw of these two simpler algorithms : it is indeed true that in the simple environment of independent, short-lived processes, we saw evidence that a simpler algorithm was sufficient to give performance improvements; however, in moving to a more realistic environment which more accurately mirrors the way in which user applications are written to benefit from the parallelism provided by a distributed computer system, the algorithm parameters need modification, and thus they do not adapt easily to changing conditions.

In contrast to this, we found (again during initial trial runs) that the global average algorithm required no modification to operate under the process group workload, and as will be seen in the following sections provided substantial performance improvements; so henceforth when we refer to this algorithm, it is as described in section 5.3.2.4 (including the choice of values for its parameters).

The preemptive threshold algorithm functions in a similar manner to the threshold

algorithm, transferring processes to another processor, when a particular processor's load exceeds a fixed threshold value; however checking for overloading is not performed upon the arrival of a new process, but is invoked periodically, thus allowing a process which is already executing on a processor to be interrupted and migrated to a suitable alternative location. We used the same threshold value of 4, and set the period between invocations of the algorithm to be 2 seconds (which was chosen through experimentation). Any period shorter than this was found to result in too many fruitless migration attempts, particularly at heavy system load, and a longer period did not provide sufficient chance for a processor to detect its overloaded state, or to offload multiple processes within a reasonable time span if it became very heavily-loaded. Processes were only considered for migration if they were not blocked and also not currently executing (in other words, processes in the ready queue).

### 5.4.4 Performance of Load Balancing Algorithms

We studied the performance of the algorithms described above (random, threshold, preemptive threshold and global average) under varying system loads and, similarly to the results presented in section 5.3.3, we took a measure of the average time which was necessary for a process to execute to completion. We also included simulation runs using no load balancing to provide a "worst case" analysis.

Each simulation run was allowed to continue until the aforementioned average converged to within a range allowing comparison between the performance of different algorithms (i.e. the average for each processor in the network did not differ by more than approximately 2% from that of its peers). These results are presented in fig. 5.3.

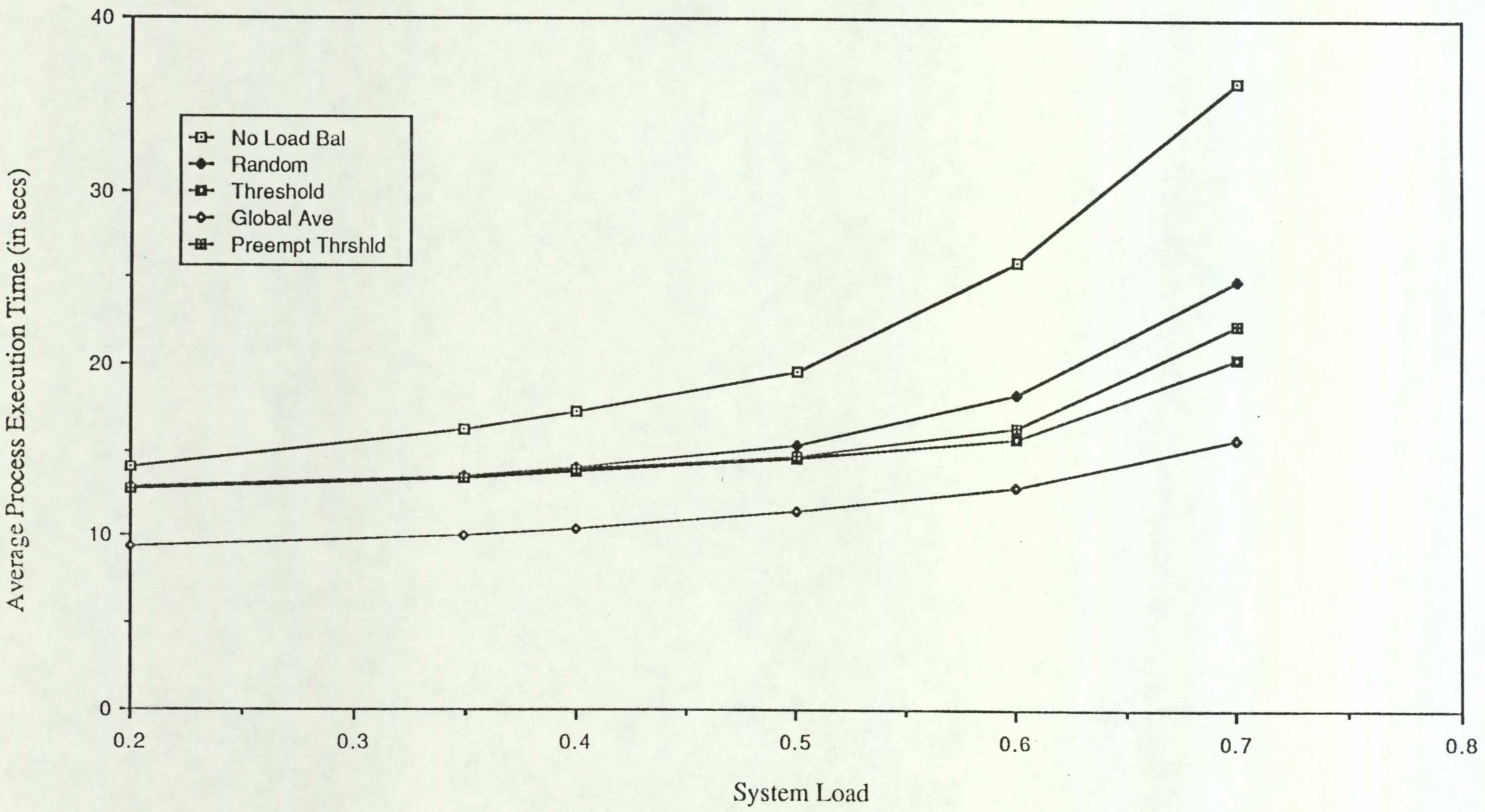



Fig. 5.3 Performance Comparison using the Cooperating Process Group Model

The maximum system load which we were able to simulate was 0.7, due to the bursty nature of process arrival, which creates severe overloading in pathological cases (not only of our simulated system, but also the NCR Tower on which the simulations were conducted).

From fig. 5.3, we can see that when no load balancing is used, the performance of the system degrades markedly for system loads greater than 0.5. The more simple algorithms (random, threshold and preemptive threshold) all improve significantly on the no load balancing case, but (in contrast to the results we obtained using a system whose workload comprised short-lived, independent processes) the global average algorithm provides constantly superior results over all values of system load.

In order to examine in more detail the reasons for these performance differences we again sampled mean processor load (the average number of processes on one processor in the network at any one time), processor load variance (the variance between the number of processes on each processor) and the load difference (the load on the most heavily-loaded processor minus that of the least heavily-loaded). In addition we sampled the number of migrations which occurred, when using a particular load balancing algorithm and the number of interprocessor messages sent during the simulation run; these give greater insight into the behaviour of each algorithm and its associated costs. All of these values were taken at 10 second intervals over a period of 1,000 seconds of simulated time, and for low, moderate and heavy system loads (0.2, 0.5 and 0.7 respectively). We present these results in a graphical form in Appendix B (figs. B.1.1-B.1.5, B.2.1-B.2.5 and B.3.1-B.3.12), and an overall summary in the text of this section (Tables 5.4-5.7).

| Statistics / Algorithm | Mean Load | Load Variance | Load Difference | Average Migrations per Second |
|---|---|---|---|---|
| No Load Balancing | 1.625 | 1.910 | 3.16 | 0.0 |
| Random | 1.582 | 1.443 | 2.67 | 0.026 |
| Preemptive Threshold | 1.577 | 1.437 | 2.74 | 0.031 |
| Threshold | 1.576 | 1.437 | 2.67 | 0.026 |
| Global Average | 1.437 | 0.449 | 1.48 | 0.261 |

Table 5.4 Overall System Behaviour using the Cooperating Process Group Model (Load Value = 0.2)

It should be observed from Table 5.4 that even at low system load (0.2), there is a benefit in balance factor to be gained from using any of the load balancing algorithms, shown by the improvement in load variance and load difference; however it is only the global average algorithm which results in a significant reduction in mean load. A more detailed view of these aspects of system performance can be seen in figs. B.1.2 and B.1.3. Since at such a low system load there is no discernible difference in behaviour between threshold, preemptive threshold and random, we chose to plot only threshold against the no load balancing case as being representative of all three simple algorithms. Figs. B.1.4 and B.1.5 illustrate that global average gives further improved balance factor over the simple algorithms, even at low system load.

Table 5.5 shows a summary of system performance at moderate load (0.5). This illustrates that the trends observed at low load are similar as load increases; at a load value of 0.5, we can see that without any form of load balancing, the system is increasingly poorly balanced, and that now the mean load on the system as a whole, is

significantly reduced by using any one of the algorithms which were implemented.

| Algorithm \ Statistics | Mean Load | Load Variance | Load Difference | Average Migrations per Second |
|---|---|---|---|---|
| No Load Balancing | 3.405 | 9.862 | 7.93 | 0.0 |
| Random | 2.880 | 2.683 | 4.25 | 0.360 |
| Preemptive Threshold | 2.766 | 2.463 | 4.00 | 0.366 |
| Threshold | 2.728 | 2.507 | 3.92 | 0.284 |
| Global Average | 2.433 | 0.880 | 2.55 | 1.269 |

Table 5.5 Overall System Behaviour using the Cooperating Process Group Model (Load Value = 0.5)

Again global average shows a far more significant improvement both in terms of balance factor and reduction in mean load. A more detailed picture of these phenomena can be seen in figs. B.2.1-B.2.5.

At heavy system load (0.7), the difference in behaviour between the chosen load balancing algorithms is even more apparent, as summarised in Table 5.6. In order to better illustrate these differences we plotted the performance of pairs of algorithms to allow direct comparison between them; the pairs we selected are : no load balancing vs random, random vs threshold, threshold vs preemptive threshold and threshold vs global average (figs. B.3.1-B.3.12). At this load value, we note that the more complex global average algorithm again offers a good balance factor and hence predictability of process response time; we also note that the addition of preemption to the threshold algorithm results in a slight loss of stability, shown by the degraded balance factor together with an increased number of process migrations.

| Statistics / Algorithm | Mean Load | Load Variance | Load Difference | Average Migrations per Second |
|---|---|---|---|---|
| No Load Balancing | 5.874 | 27.885 | 14.15 | 0.0 |
| Random | 4.705 | 4.228 | 5.70 | 0.875 |
| Preemptive Threshold | 4.039 | 3.630 | 5.25 | 0.646 |
| Threshold | 3.942 | 3.271 | 4.84 | 0.463 |
| Global Average | 3.412 | 1.315 | 3.19 | 2.492 |

Table 5.6 Overall System Behaviour using the Cooperating Process Group Model (Load Value = 0.7)

A number of important points emerge from these results, indicating reasons for the differences between the average process execution time observed for each algorithm. Global average outperforms all of the simpler algorithms, even at very low load; we can attribute this improvement at low load, to the fact that for the threshold and random algorithms, the threshold value had to be set at 4 to avoid fruitless migration attempts at high system load, but this value removes the possibility of finding better load distribution when system load is low.

To illustrate this point, consider a totally idle network; if a process group of size 4 arrives at one processor, then random and threshold will not attempt to move any of the 4 constituent processes since the fixed threshold load value has not been exceeded; even if no other processes arrive elsewhere in the network, under these two algorithms, none of the 4 processes will be moved to another processor. Using the global average algorithm on the same example, the idleness of the network will be detected, and an arriving group of 4 processes will rapidly be redistributed amongst the available processors. We believe that this is a significant factor in explaining the superior performance of global average for low system loads. We can also see from

Table 5.6 why global average performs better than threshold at high system load (0.7); note that the mean load on the network for the three simple algorithms remains around a load value of 4 processes, and that this is the value of the fixed threshold which we used. Thus most probes to investigate the possibility of process migration will fail, since the probed processors' loads are also above the threshold value; in contrast global average continues to redistribute load, even as general system load increases, and also provides a more reliable mechanism for finding a relatively lightly-loaded processor in an otherwise heavily-loaded system.

If we examine the average number of migrations performed per second throughout the network under each of the four load balancing algorithms (as shown in Tables 5.4, 5.5 and 5.6), we can see that although random causes more migrations than threshold, its overall performance both in terms of balance factor and average process execution time is worse. We can conclude from this that these extra migrations are made fruitlessly (which is what we would expect, since processes are not assigned an alternative processor in an "intelligent" manner, but purely at random). Moving to the global average algorithm we see that the average number of migrations per second (at all system loads) is considerably higher than for the other algorithms, and that the balance factor shown by the load variance and load difference values is better. We can thus conclude from this that the additional migrations are made at appropriate times when overall system load begins to become unbalanced (in fact closer inspection of the simulated system during its operation, revealed that this was indeed the case, with dramatic changes in workload resulting in highly increased migration activity).

A further noteworthy point is that the preemptive threshold algorithm actually degrades performance relative to its non-preemptive version; we can see that it results in a less stable system at heavy load. We can conclude from this that purely introducing

preemption into an algorithm is not suffient in order to improve its performance; this must be linked with the ability to adapt to varying circumstances (as exemplified by the global average algorithm).

| Algorithm \ Load Value | Load 0.2 | Load 0.5 | Load 0.7 |
|---|---|---|---|
| No Load Balancing | 1.512 | 3.773 | 5.192 |
| Random | 1.617 | 4.980 | 8.057 |
| Preemptive Threshold | 1.703 | 5.865 | 9.333 |
| Threshold | 1.628 | 4.779 | 7.181 |
| Global Average | 4.531 | 17.044 | 28.608 |

Table 5.7 Comparison of Average Interprocessor Messages Transmitted per Processor per Second using the Cooperating Process Group Model

### 5.4.5 Costs of Algorithm Execution

In order to establish the extra costs incurred by the load balancing algorithms studied, we recorded the number of interprocessor messages generated by each processor on the network during each second of the 1,000 second simulation period. These values are shown in Table 5.7. We can see that the three simple algorithms do not differ greatly in the additional message traffic which they produce; global average, however, requires on average approximately 3 times as many interprocessor communications. It is of note that despite the added overheads incurred by this message traffic, the algorithm still outperforms its simpler counterparts at all load levels (as shown in previous sections). These are not only messages used by the algorithm itself but also interprocess messages sent between cooperating user processes which have been

migrated to different processors, since in our implementation we are unable to measure separately kernel and user-generated messages. Despite this drawback, the figures shown do serve as broad indicators of the total communications cost imposed on the system. These costs are of importance when considering use of a particular algorithm in a particular network with a given communications medium speed. The costs should be viewed in conjunction with the performance results presented in previous sections, and we only claim that they are valid for our given experimental environment. Experimentation with other network speeds and topologies are a source of further research.

## 5.5 MODIFIED LOAD BALANCING ALGORITHMS

Having gained considerable experience of the operation of load balancing algorithms under varying conditions, and based on the analysis of the detailed behaviour of the simulated network when load balancing is applied, the final phase of this study was designed to investigate modifications which could be made to the algorithms which we implemented in order to further enhance performance improvement. Since the global average algorithm was shown to create a system with a good balance factor, we elected to take its basic philosphy of maintaining a network-wide view of processor load, and to modify the way in which pairs of processors are chosen to consider entering into process migration negotiations. Due to apparent drawbacks of the fixed threshold parameters of the simpler algorithms, we decided that they did not warrant any further investigation (in fact, the only reasonable modification to these algorithms would be to make the threshold variable, which is effectively the essence of the global average algorithm). For all of the simulation runs used to carry out this study, we retained the cooperating process group model of synthetic workload generation.

### 5.5.1 Receiver-Initiated Global Average Algorithm

In our implementation of Global Average, process migration was initiated by a processor when it detects the fact that it is overloaded relative to the globally-agreed overall system load value. This means that in reply to its broadcast message attempting to find an underloaded partner, the overloaded processor may have to deal with a number of such replies, thus imposing even further load upon it in processing these replies. It has been suggested that at overall high system load, it is more reasonable to expect a lightly loaded processor to seek an overloaded partner from which to accept a number of processes; Eager et al [85] investigated this approach for very simple load balancing algorithms.

#### 5.5.1.1 Implementation

Our first modification to the Global Average algorithm was thus to examine whether this phenomenon would apply to a more complex environment than has been previously studied, and we term this modification the Receiver-Initiated Global Average algorithm. Using this approach, when a processor detects that it is underloaded, it sets a `too_low` timeout and broadcasts its availability for accepting processes from an overloaded processor. An overloaded processor, on receiving such a "too_low" message, sends back a "too_high" message indicating that it wishes to offload a process to the sender of the "too_low" message; if the original sender is still underloaded then process migration will take place. We felt that this slightly lengthy procedure of acknowledgements was more desirable than simply immediately migrating a process from an overloaded processor on receipt of a "too_low" message,

since the latter could lead to the originally underloaded processor being swamped with incoming migrant processes.

As before, the algorithm functions on events concerning under- or overloading and the receipt of interprocessor messages, and we present below a list of these events, shown in italics, together with their associated actions :

*a processor detects it is overloaded* :

```
set too_high timeout
```

*a processor detects it is underloaded* :

```
broadcast availability
set too_low timeout
```

*a processor receives a too_low message* :

```
if (overloaded)
{
    send too_high message to sender of too_low message
}
cancel too_high timeout if pending
```

*a processor receives a too_high message* :

```
if (underloaded)
{
    send accept message to sender of too_high message
    add awaiting_process timeout to queue
    increment virtual load
}
    cancel any too_low timeout pending
```

*a processor's too_low timeout expires* :

broadcast new lower average

*a processor's too-high timeout expires* :

broadcast new higher average

*a processor receives a new average value* :

update local copy of average

cancel any too_low or too_high timeouts pending

*a processor's awaiting_process timeout expires* :

decrement virtual load

*a processor receives a migrant process* :

remove one awaiting_process timeout from the queue

decrement virtual load

install process

#### 5.5.1.2 Performance

In fig. 5.4, we present the average process execution time for the network under a range of system loads, comparing the original global average algorithm with our receiver-initiated variant. From this we can indeed see that this modification leads to performance improvement at heavier system load, but we also note that the results

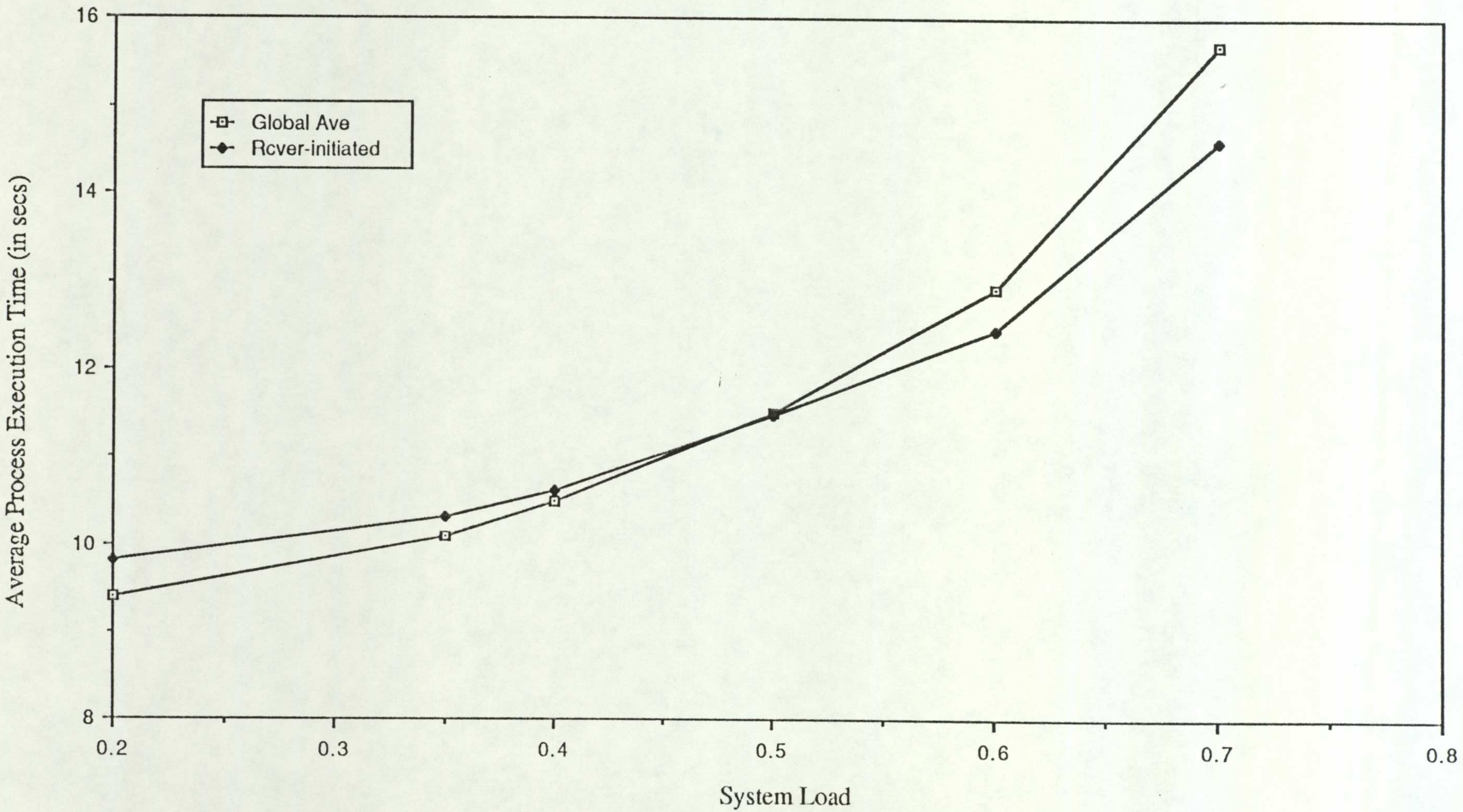



Fig. 5.4 Performance Comparison - Global Average vs the Receiver-initiated Variant using the Cooperating Process Group Model

System Load 0.2

| Statistics / Algorithm | Mean Load | Load Variance | Load Difference | Average Migrations per second | Average interprocessor messages per processor /sec |
|---|---|---|---|---|---|
| Global Average | 1.437 | 0.449 | 1.48 | 0.261 | 4.531 |
| Receiver-initiated | 1.449 | 0.529 | 1.61 | 0.235 | 5.558 |

System Load 0.5

| Statistics / Algorithm | Mean Load | Load Variance | Load Difference | Average Migrations per second | Average interprocessor messages per processor /sec |
|---|---|---|---|---|---|
| Global Average | 2.433 | 0.880 | 2.55 | 1.269 | 17.044 |
| Receiver-initiated | 2.415 | 0.909 | 2.54 | 1.072 | 19.941 |

System Load 0.7

| Statistics / Algorithm | Mean Load | Load Variance | Load Difference | Average Migrations per second | Average interprocessor messages per processor /sec |
|---|---|---|---|---|---|
| Global Average | 3.612 | 1.315 | 3.19 | 2.492 | 28.608 |
| Receiver-initiated | 3.563 | 1.368 | 3.19 | 2.127 | 22.752 |

Table 5.8 Overall System Behaviour using the Cooperating Process Group Model Comparing **Global Average** and the **Receiver-initiated Variant** (Load Value = 0.2, 0.5 and 0.7)

from the receiver-initiated global average algorithm are poorer for low loads. The differences in behaviour of these two algorithms are compared in Table 5.8, showing mean load, load variance, load difference and process migration rate together with the intensity of interprocessor message traffic. From these values we can see that the receiver-initiated version of global average gives an improved balance factor at high load, and a reduction in the mean load value; however at low load the system's balance factor degrades using this modification. It should also be noted from Table 5.8, that the receiver-initiated global average algorithm reduces the message traffic in the system at high load (0.7); this can be attributed to the fact that a high overall system load will result in less broadcast messages, since this modification of Global Average relies on underloaded processors to broadcast their availability. Conversely the performance degradation at low level is due to the presence of many lightly-loaded processors in the system broadcasting "too-low" messages.

### 5.5.2 Limited Broadcast Global Average Algorithm

Since the value for global average load is maintained in our system by broadcasting its updated level when a processor detects that it is no longer accurate, and also overloaded processors broadcast a message to indicate their plight, the number of messages necessary to allow the algorithm to function correctly is high. We observed from closer inspection of the migration behaviour of processes from an overloaded processor that, although a "too_high" message is broadcast to all other processors, there is a very high probability that if a number of underloaded processors exist in the network, then the nearest one to the overloaded sender of the broadcast message will reply first, thus accepting a migrant process; this means that other underloaded processors will wait in vain to receive the migrant process until the awaiting-process timeout period has elapsed.

#### 5.5.2.1 Implementation

In order to reduce the number of such broadcast messages, and to limit the distance which a migrant process will be moved, we introduced a further variant to the global average algorithm, whereby an overloaded processor sends a "too_high" message only to its immediate neighbours; these neighbours will only forward the message, in turn to their neighbours if they are themselves either overloaded or at least their load is above the global average load value. To prevent this forwarding from continuing indefinitely we added an "ageing" feature to "too_high" messages which only permitted them to be re-transmitted once; in other words an overloaded processor sends the message to its immediate neighbours and these can then forward the message to their neighbours if necessary, but the message will not travel any further through the network. For larger networks, the number of possible re-transmissions could be increased since it is merely a parameter of the algorithm. We term this variant, the Limited Broadcast Global Average algorithm.

Implementation of this modification consisted of changing the actions carried out upon detecting overloading and receiving a "too_high" message; all other actions remained unchanged. Below we present a description of the modified actions associated with these two events :

*a processor detects it is overloaded* :

```
set forward_it field of "too_high" message to maximum number of "hops"
send "too_high" message to immediate neighbours
set "too_high" timeout
```

*a processor receives a "too_high" message* :

```
decrement forward_it field of "too_high" message
if ( !underloaded )
```

```
{
        if ( forward_it field != 0  )
          forward "too_high" message to immediate neighbours
}
else
{
        send accept message to original sender of "too_high" message
        add awaiting_process timeout to queue
        increment virtual load
}
```

### 5.5.2.2 Performance

The graph in fig. 5.5 showing average process execution time for the original global average algorithm, and for our Limited Broadcast variant reveals that indeed the overall performance of the system is improved under this scheme. If we look in more detail at the behaviour of the network as shown in Table 5.9, we can see that under all these sampled load values, the Limited Broadcast variant does considerably reduce the amount of message traffic in the system, without significantly degrading the general balance factor (as shown by the load variance and load difference values). In fact we note that at moderate load (0.5), the system's balance factor is slightly improved using this variant; we attribute this to there being little need at this load to seek migration further than the immediate proximity of an overloaded processor, and that doing so merely results in less fruitful migrations being made. It should also be noted that at high load (0.7), the performance of the two algorithms (shown in fig. 5.5) begins to converge; we believe this is due to the fact that at high overall system load, an overloaded processor will need to communicate with more distant processors in order to find an underloaded partner, hence the slight degradation in balance factor of the Limited Broadcast variant, although this is amply compensated for by the reduction in

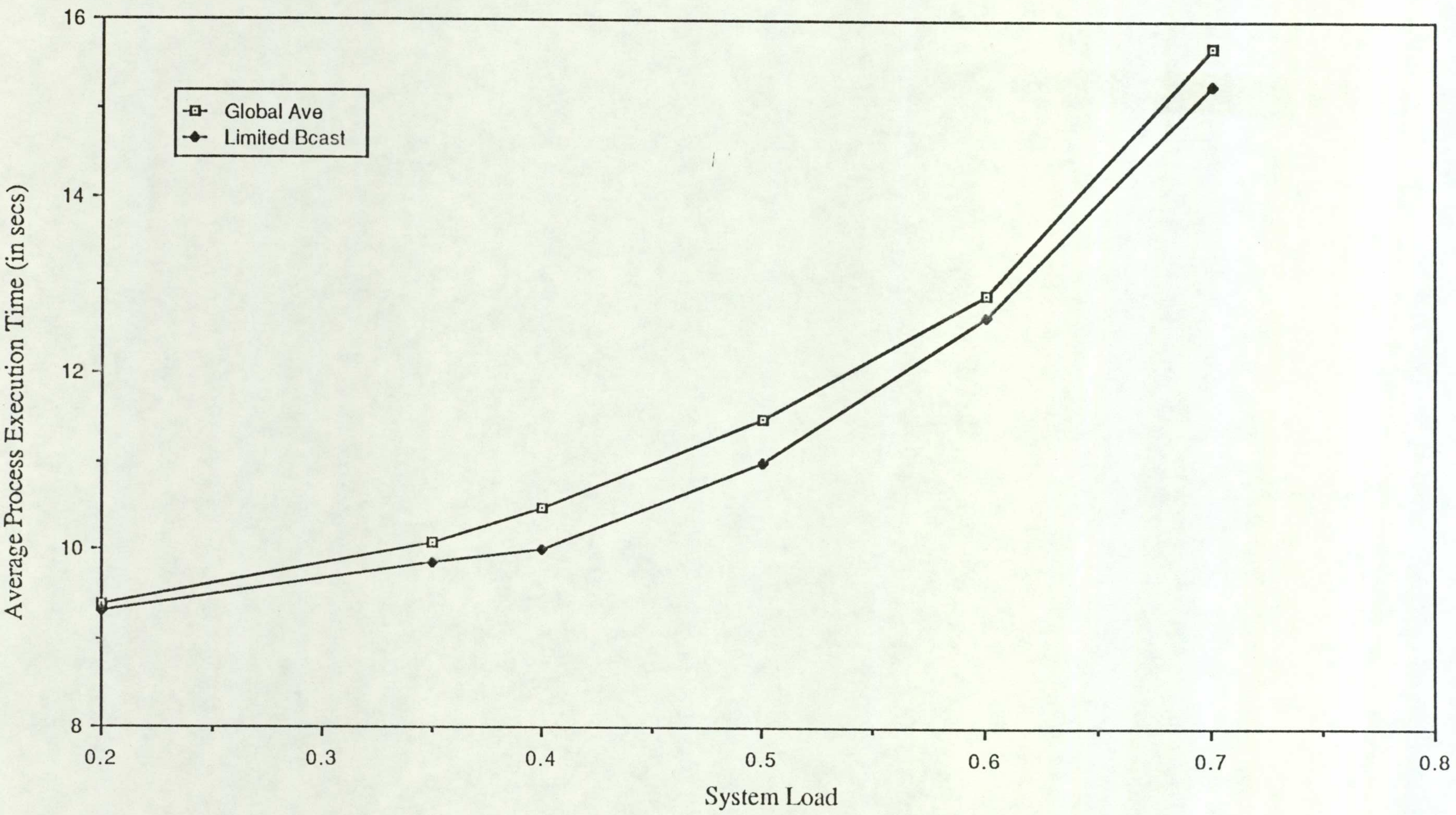



Fig. 5.5 Performance Comparison - Global Average vs the Limited Broadcast Variant using the Cooperating Process Group Model

System Load 0.2

| Algorithm \ Statistics | Mean Load | Load Variance | Load Difference | Average Migrations per second | Average interprocessor messages per processor /sec |
|---|---|---|---|---|---|
| Global Average | 1.437 | 0.449 | 1.48 | 0.261 | 4.531 |
| Limited Broadcast | 1.443 | 0.469 | 1.55 | 0.308 | 4.151 |

System Load 0.5

| Algorithm \ Statistics | Mean Load | Load Variance | Load Difference | Average Migrations per second | Average interprocessor messages per processor /sec |
|---|---|---|---|---|---|
| Global Average | 2.433 | 0.880 | 2.55 | 1.269 | 17.044 |
| Limited Broadcast | 2.354 | 0.826 | 2.45 | 1.287 | 14.120 |

System Load 0.7

| Algorithm \ Statistics | Mean Load | Load Variance | Load Difference | Average Migrations per second | Average interprocessor messages per processor /sec |
|---|---|---|---|---|---|
| Global Average | 3.612 | 1.315 | 3.19 | 2.492 | 28.608 |
| Limited Broadcast | 3.485 | 1.376 | 3.26 | 2.631 | 23.752 |

Table 5.9 Overall System Behaviour using the Cooperating Process Group Model Comparing **Global Average** and the **Limited Broadcast Variant** (Load Value = 0.2, 0.5 and 0.7)

message traffic and the reduced mean load value.

We suggest that the difference in performance would be more marked if the communications medium were slower than the one we have used in our simulation, and that therefore the limited Broadcast Global Average algorithm would be suitable for such environments.

### 5.5.3 Sender/Receiver Global Average Algorithm

Examining the average process execution time graph of the global average algorithm and its receiver-initiated variant (fig. 5.4), we notice that the two curves cross at a system load value of 0.5, thus we can draw the general conclusion that for loads less than this value, the sender - initiated (i.e. the original) version of global average is the one which we should use and that for loads greater than 0.5 the receiver - initiated version is preferable. It should also be noted that an overall system load which is heavy (e.g. 0.7) does not mean that the workload arrival is at a constant rate; there are moments of peaks of heavy load and troughs of low load, which combine to give an average system load of 0.7. A similar observation applies for a low system load value of say 0.2.

#### 5.5.3.1 Implementation

Observing the above facts about the behaviour of the load on our simulated network, we designed a further variant of the global average algorithm (termed sender / receiver global average) which exploits the benefits of sender- and receiver-initiated global average, by switching to the version appropriate for the current overall system load; when overall system load is high, then the algorithm operates in a receiver - initiated

fashion, and changes to a sender - initiated algorithm when system load falls. Since the actual value of system load being used in a particular simulation run is a parameter of our simulation and hence not accessible to a load balancing algorithm, it was necessary to define heavy or low load in terms of the current global average value maintained in each processor at any one instant of time; by experimentation we determined that a global average value of 3.5 was a good indicator of the point which separates heavy from light load. This value was used in our algorithm to decide whether it should operate in receiver - initiated or sender - initiated mode. However this value alone is not sufficient to gain improvement in performance, since we wish to avoid the state where the algorithm frequently oscillates between its two modes, when the global average fluctuates between values just above 3.5 and just below it; in fact, it is apparent that such behaviour would be unacceptably unstable. In order to avoid this potential instability we defined a tolerance range around the load breakpoint to ensure a more gradual transition from one mode of operation to the other; so for example if the current load on the system is detected to be low (and hence the sender - initiated mode is being used), the changeover to receiver - initiated mode will only be made when the global average value exceeds the load breakpoint plus the tolerance range; similarly, when load is considered to be high (and hence the receiver - initiated mode is in use), the changeover to sender - initiated mode will only be made when the global average value falls below the load breakpoint minus the tolerance range. Again by experimentation, we elected to use a tolerance range of size 0.5, around the load breakpoint, which gave a sufficiently certain indication that either the low load or high load state had been detected and that the system would remain in that state sufficiently long to benefit from the particular qualities of either a sender - initiated or receiver - initiated mode of operation.

The algorithm is driven by events, and uses a variable `policy` to indicate whether sender or receiver - initiated mode is in force as described below :

```
a processor detects it is overloaded :
        if (policy == RECEIVER_INITIATED)
        {
                set too_high timeout
        }
        else /* policy == SENDER_INITIATED */
        {
                broadcast "too_high" message
                set too_high timeout
        }

a processor detects it is underloaded :
        if (policy == SENDER_INITIATED)
        {
                set too_low timeout
        }
        else /* policy == RECEIVER_INITIATED */
        {
                broadcast "too_low" message
                set too_low timeout
        }

a processor receives a too_low message :
        if (overloaded)
        {
                send "too_high" message to sender of "too_low" message
        }
        cancel any pending too_high timeout

a processor receives a too_high message :
        if (underloaded)
```

```
        {
                send accept message to sender of "too_high" message
                add awaiting_process timeout to queue
                increment virtual load
        }
        cancel any pending too_low timeout
```

*a processor's too_low timeout expires* :

```
        broadcast new lower average value
        if ( policy == RECEIVER_INITIATED
              && new average value < LOAD_BREAKPOINT - TOLERANCE )
                policy = SENDER_INITIATED
```

*a processor's too_high timeout expires* :

```
        broadcast new higher average value
        if ( policy == SENDER_INITIATED
              && new average value < LOAD_BREAKPOINT + TOLERANCE )
                policy = RECEIVER_INITIATED
```

*a processor receives a new average value* :

```
        update local copy of average
        cancel any pending too_low or too_high timeouts
        if ( new average value > LOAD_BREAKPOINT + TOLERANCE
              && policy == SENDER_INITIATED )
                policy = RECEIVER_INITIATED
        else if ( new average value < LOAD_BREAKPOINT - TOLERANCE
              && policy == RECEIVER_INITIATED )
                policy = SENDER_INITIATED
```

*a processor's awaiting_process timeout expires* :

```
        decrement virtual load
```

*a process receives a migrant process* :

```
        remove one awaiting_process timeout from the queue
        decrement virtual load
        install process
```

#### 5.5.3.2 Performance

In fig. 5.6, which shows a comparison of the average process execution time for the network under a range of system loads, between the original global average algorithm and our sender / receiver variant, we can see that the variant offers improvement over all but the lightest of system loads. By examining the more detailed behaviour of these two algorithms at low (0.2), moderate (0.5) and high (0.7) loads as presented in Table 5.10, we observe that the sender / receiver variant results in a more balanced system, with a reduced mean load value, and also reduces the message - traffic passing through the communications medium.

We would expect performance of this variant to be similar to the original global average algorithm at low load, since the algorithm would be operating almost constantly in sender-initiated mode, since the global average value will very rarely reach the load breakpoint where it will switch into receiver-initiated mode. We note also that at high system load (0.7), the performance of sender / receiver global average is similar to that of the receiver-initiated variant, since at such a load level the mode of operation will almost constantly be receiver-initiated and only rarely will overall system load fall below the load breakpoint. It is interesting to note that at a load value of around 0.5, the average process execution time of the original global average algorithm and our sender / receiver variant are close to being the same value; we attribute this to the fact that a load value of 0.5 represents the point at which the globally - maintained average load value is close to our chosen load breakpoint, and hence switching between sender and receiver-initiated modes will be reasonably frequent and perhaps not really beneficial.

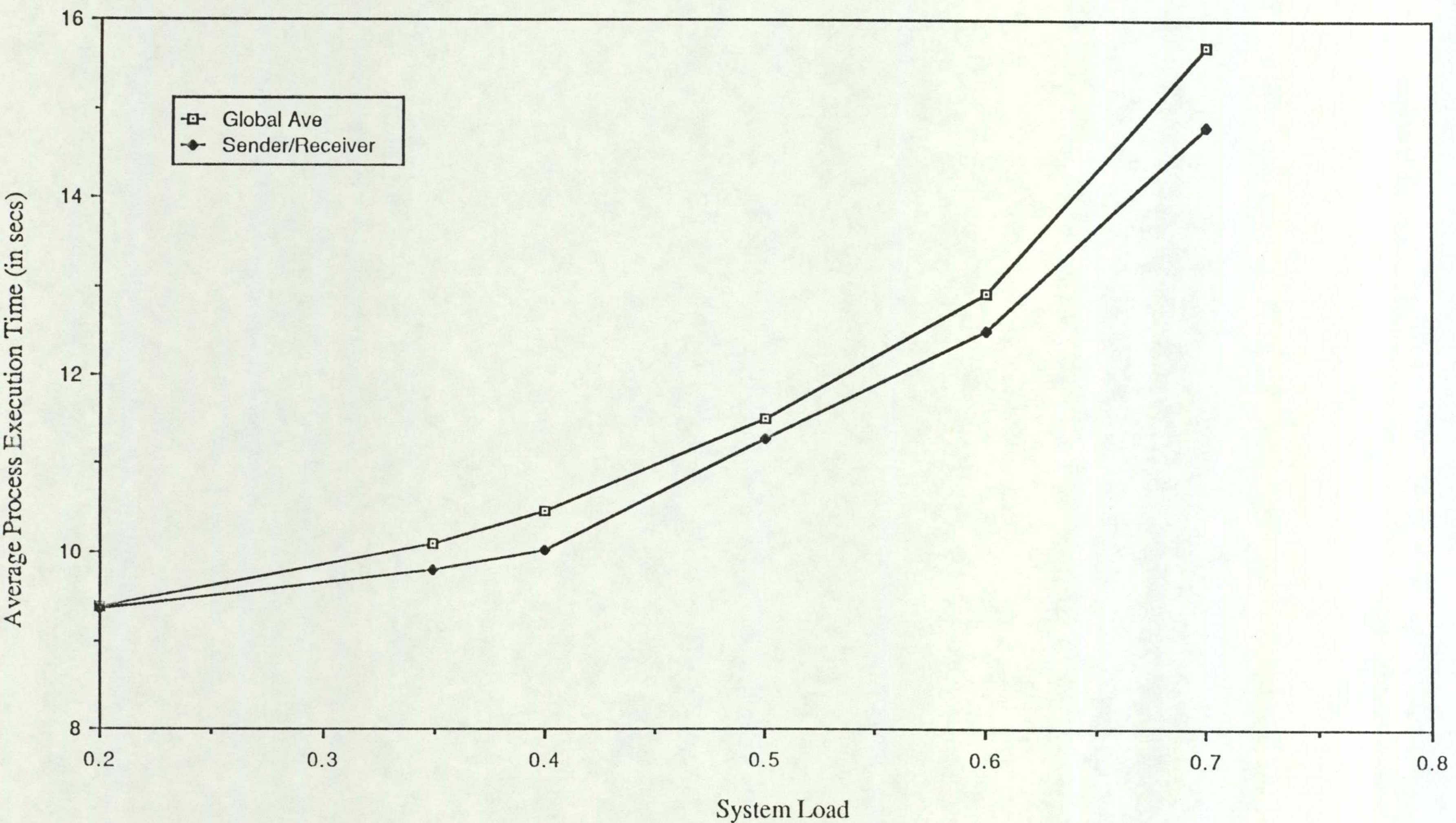



Fig. 5.6 Performance Comparison - Global Average vs the Sender/Receiver Variant using the Cooperating Process Group Model

System Load 0.2

| Statistics / Algorithm | Mean Load | Load Variance | Load Difference | Average Migrations per second | Average interprocessor messages per processor /sec |
|---|---|---|---|---|---|
| Global Average | 1.437 | 0.449 | 1.48 | 0.261 | 4.531 |
| Sender/Receiver | 1.433 | 0.455 | 1.52 | 0.274 | 5.012 |

System Load 0.5

| Statistics / Algorithm | Mean Load | Load Variance | Load Difference | Average Migrations per second | Average interprocessor messages per processor /sec |
|---|---|---|---|---|---|
| Global Average | 2.433 | 0.880 | 2.55 | 1.269 | 17.044 |
| Sender/Receiver | 2.399 | 0.793 | 2.46 | 1.238 | 16.605 |

System Load 0.7

| Statistics / Algorithm | Mean Load | Load Variance | Load Difference | Average Migrations per second | Average interprocessor messages per processor /sec |
|---|---|---|---|---|---|
| Global Average | 3.612 | 1.315 | 3.19 | 2.492 | 28.608 |
| Sender/Receiver | 3.558 | 1.192 | 3.05 | 2.297 | 27.861 |

Table 5.10 Overall System Behaviour using the Cooperating Process Group Model Comparing **Global Average** and the **Sender/Receiver Variant** (Load Value = 0.2, 0.5 and 0.7)

### 5.6 SUMMARY

It is illustrative to examine our results in the light of the following four criteria based on the work of Alonso [86]. These criteria are suggested as being a measure of the efficacy of a load balancing algorithm. They are :

- balance factor
- cost
- autonomy of processors
- transparency to user processes.

We note that all of the load balancing algorithms which we have studied allow user processes to run unchanged, and hence they are all transparent. We believe that the improved balance factor brought by the global average algorithm is significant in explaining its superiority over the more simple algorithms which were implemented. It was further observed that the greater the balance in the system, the greater the predictability of user process response time, which is a desirable attribute for most environments. By including the results obtained when using a "preemptive threshold" algorithm, we demonstrated that the use of preemption does not by itself necessarily guarantee extra stability, and indeed we observed that instability and processor thrashing may occur as a result. This was shown by the fact that the "preemptive threshold" algorithm caused more migrations than the simple threshold policy, but consequently degraded overall system performance.

From the tables of results shown in this chapter, we see that the cost of executing the more complex global average algorithm is high in terms of process migration and message transmission, and conclude that it is suitable for a system where

interprocessor communications do not form a substantial bottleneck (recall that in our simulation runs the speed of the communications medium was set at 1Mbyte/second).

We conclude also, that the autonomy of a load balancing algorithm is of vital importance. The only algorithm which we studied where each processor does not control process migration in an autonomous manner is the random algorithm, since a processor cannot refuse to accept a process which has been sent to it by one of its overloaded peers, regardless of its own load state. As we can see from our results, this can lead to periods of overloading and also gives rise to large differences between the loads of the most heavily-loaded and the least heavily-loaded processors.

Our results indicate that it is viable to not only use an algorithm which adapts its fundamental parameters to suit a particular load environment, but that changes to the way in which the algorithm itself operates are possible and that this added adaptability can be achieved at low cost and further improve overall system performance.

# CHAPTER 6

# CONCLUSIONS

Having noted the apparent drawbacks of static load balancing policies, we chose, in this thesis, to study algorithms which adapt to the changing nature of the overall load on a loosely-coupled distributed system, thus exploiting the observations of Livny and Melman [82], that there is a high probability of there being idle processors in the network, whilst others hold a number of processes queueing for service.

In order to carry out this study we have presented details of a simulated distributed system, where we have chosen a 3x3 square mesh topology for its interprocessor connections. To give greater flexibility and to provide a neat and realistic environment, each processor in the system is simulated by a separate UNIX process. We have also described an operating system kernel for this environment, which supports the creation of typical user processes (in groups if desired), with interprocess communication being performed through the use of message-passing between software ports. The system periodically dumps performance information to a number of trace files, to allow us to compare the behaviour of different load balancing algorithms operating under varying system loads. The system implemented, allowed a much greater range of parameter choice, and point-to-point topology, than those chosen here.

Our experiments were conducted in both a simple environment where small user processes execute independently, and also in a more complex environment where groups of two, three and four processes exchange messages in order to divide a given task into a number of concurrent activities. The algorithms which were chosen for investigation show a range of different approaches to the load balancing issue : the

most simple of these (the random algorithm) uses a very simple processor load measure (the number of processes resident on a processor) and performed no information exchange when making process migration decisions; the threshold algorithm uses the same simple processor load measure, but includes a limited information exchange policy (via its probing mechanism) to establish a suitable location for a migrant process; the global average algorithm was included to investigate the possible benefits of a more complex load measure (the number of non-blocked processes on a processor averaged over a time period), and utilised a more complex information exchange policy to maintain a global view of the average load on the system.

Using the independent user process model for our initial study, we saw that, especially at high system loads, all load balancing algorithms gave substantial improvements in average process execution time (which we used as an overall measure of performance) and that we could observe reductions in system mean load, load variance and load difference. We noted also that the improved balance factor at little extra cost, seen when using the threshold algorithm, caused it to perform better than a random placement policy. In addition, we were able to establish that in such a simple environment, the more complex global average algorithm was unable to produce any further improvement over the threshold policy. We attributed this to the extra costs which this algorithm incurred when attempting to maintain process queues of equal length on each processor.

Resulting from this initial series of simulation runs, we can conclude that for a simple process environment, our results support the findings of Eager et al [86], that there is little to be gained by using a complex algorithm for load balancing purposes. In their paper Kurose and Chipalkatti [87] have extended the aforementioned work on the

complexity of load balancing algorithms using a real-time environment, with real-time constraints employed as the indicator for when to migrate processes from one processor to another. The goal of our second series of experiments was to investigate the load balancing issues in a distributed system where user applications are structured as process groups, with members of each group engaging in information exchange through message-passing. We believe that this is a more realistic environment than that commonly studied by other workers, in that one process in the group could be thought of as a file manager, receiving requests for blocks of a file which is resident on its processor, and sending the contents of these blocks as reply messages; the question of I/O overheads had been ignored by Eager et al [86], and other studies have only included the overheads caused by the exchange of messages, explicitly involved in executing a particular load balancing algorithm. For our study of algorithms in this environment, we again used average process execution time as an overall performance metric, with system mean load, load difference and load variance as further indicators, but in addition we measured the rate at which processes were migrated across the network, and the number of interprocessor messages which were transmitted, in order to gain more insight into the extra costs of the chosen algorithms.

From the results which we presented in Chapter 5, we can make a number of important observations. The peformance improvements when using any of the selected load balancing algorithms are notable across a wide range of system loads, even when load is relatively light. We can attribute this to the group arrival of processes to the system, which can lead to heavy overloading; in fact we noted that the system remained overloaded for lengthy periods of time when no load balancing was used, until the backlog of competing processes could be cleared. When moving to the process group environment, we discovered that the fixed threshold used by both the random and threshold algorithms needed modification to cope with the nature of the

workload. This leads us to the conclusion that a load balancing algorithm should be flexible enough for it to move to a different environment and still maintain its essential features.

The modifications to the global average algorithm which we presented in Chapter5, and which were novel, were based on experience of examining the behaviour of the system in the preceding series of simulation runs. The receiver-initiated variant was inspired by a study of receiver-initiated versus sender-initiated versions of very simple load balancing algorithms, due to Eager et al [85], and we found that we obtained similar results in our study; the receiver-initiated variant brought improvement at high system load, but degraded performance at low system load. We can thus conclude that these are valid for more complex algorithms as well as for simple algorithms.

Since the cost of executing a global average algorithm is high in terms of interprocessor communications, the purpose of our limited broadcast variant was to reduce the number of messages generated by the algorithm. Our results in Chapter 5 show that this was indeed achieved and we also note that this reduction in messages, did not result in a significant loss of stability, and that the essential "inertia" of the global average approach was still maintained. We conclude that the approach can thus be easily tailored to further increase performance. It is not clear, however, what the effect of increasing the size of the network would be on this variant, since the global average value may be found to vary in different parts of the network, due to the reduced information exchange. We suggest that a solution to this problem would be to define clusters of processors, each having its own load average, with one processor in the cluster responsible for intercluster communications in order to maintain a network-wide average.

The sender/receiver variant of the original global average algorithm was designed to capitalise on the advantages of sender-initiated and receiver-initiated approaches at low and high loads respectively. We believe that our results show that it is possible for a load balancing algorithm to have a single overall structure, but to modify its operation dependent on the current load state - in fact, in essence this variant switches between two load balancing algorithms, so that Zhou's statement [Zhou86a] that such an operation is too costly is not totally justified.

The results which we have presented were obtained from a simulated system, since this freed us from the constraints of using an existing operating system without facilities for supporting load measurement, message-passing and process migration. Due to the wider availability, now, of software supporting distributed program execution, we intend, as an extension to our study, to investigate the problem of load balancing on real physical systems (such as our network of Apollo DN3000's and a collection of INMOS Transputers). This work will draw on our results, but will be conducted subject to the particular nature of such systems; for example preemptive process migration on a Transputer is a difficult, if not inadvisable task.

During our study, we noted that the global average algorithm possessed a natural inertia, which prevented the processes constituting a group being spread widely across the network; this feature reduces the overheads caused by message-passing between such processes. Of further interest would be the development and analysis of algorithms which explicitly take account of the message-passing behaviour of processes when making migration decisions. This would involve an "on the fly" evaluation of costs similar to that found in the graph theoretic approach to static load balancing. The mechanisms to measure interprocess message traffic have already been included in our kernel and could be used in load balancing algorithms to achieve this

explicit consideration of message-passing costs.

In summary, we conclude that in a complex user process environment, the added complexity of an algorithm which brings more stability and an improved balance factor to the system is evidently worthwhile, and that a further criterion for load balancing algorithms (exemplified by our sender/receiver variant of a global average algorithm) which can be added to the list presented previously in this chapter, is one of adaptability.

# REFERENCES


[Alonso86] Alonso, R., "The Design of Load Balancing Strategies for Distributed Systems", *Proceedings of a Workshop on Future Directions in Computer Architecture and Software*, Charleston, SC, USA, May 1986. pp. 202-207.

[Barak85a] Barak, A. and Shiloh A., "A Distributed Load-balancing Policy for a Multicomputer", *Software - Practice and Experience*, **15**(9), Sept. 1985, pp. 901-913.

[Barak85b] Barak, A. and Litman A., "MOS : A Multicomputer Distributed Operating System", *Software - Practice and Experience*, **15**(8), Aug. 1985, pp. 725-737.

[Baskett77] Baskett F., Howard, J.H., Montague, J.T., "Task Communication in DEMOS", *Proceedings of the Sixth Symposium on Operating Systems Principles*, Purdue, USA, Nov. 1975. pp. 23-32.

[Baumgartner85] Baumgartner, K.M. and Wah, B.W., "The Effects of Load Balancing on Response Time for Local Computer Systems with a Multiaccess Network", *Proceedings of the IEEE International Conference on Communications*, Chicago, USA, June 1985. pp. 262-266.

[Berglund86] Berglund, E., "An Introduction to the V-System", *IEEE Micro*, **6**(4), Aug. 1986, pp. 35-52.

[Bershad85] Bershad, B., "Load Balancing With Maitre d'", Report #UCB/CSD86/276, Computer Science Division (EECS), University of California, Berkeley, California, USA, Dec. 1985.

[Birrell84] Birrell, A.D. and Nelson, B.J., "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, **2**(1), Feb. 1984, pp. 39-59.

[Blair82] Blair, G.S., Hutchison, D. and Sheperd, W.D., "MIMAS - A Network Operating System for Strathnet", *Proceedings of the 3rd International Conference on Distributed Computing Systems*, Miami/Ft.Lauderdale, Florida, USA, Oct. 1982. pp. 212-217.

[Bokhari81] Bokhari, S.H., "A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System", *IEEE Transactions on Software Engineering*, **SE-7**(6), Nov. 1981, pp. 583-589.

[Brownbridge82] Brownbridge, D.R., Marshall, L.F., and Randell, B., "The Newcastle Connection - or UNIXes of the World Unite!", *Software - Practice and Experience*, **12**(12), Dec. 1982, pp. 1147-1162.

[Bryant81] Bryant, R.M. and Finkel, R.A., "A Stable Distributed Scheduling Algorithm", *Proceedings of the 2nd International Conference on*

*Distributed Computing Systems*, Paris, April 1981. pp. 314-323.

[Cabrera86] Cabrera, L-F., "The Influence of Workload on Load Balancing Strategies", *Proceedings of the 1986 Summer USENIX Conference*, Atlanta, GA, USA, June 1986. pp. 446-458.

[Carey85] Carey, M.J., Livny, M. and Lu, H., "Dynamic Task Allocation in a Distributed Database System", *Proceedings of the 5th International Conference on Distributed Computing Systems*, Denver, USA, May 1985. pp. 282-291.

[Cheriton84a] Cheriton, D.R., "The V Kernel : A Software Base for Distributed Systems", *IEEE Software*, **1**(2), April 1984.

[Cheriton84b] Cheriton, D.R. and Zwaenepol, W., "One-to-Many Interprocess Communication in the V-System", *Proceedings of the ACM SIGCOMM 84 Symposium*, 1984.

[Cheriton85] Cheriton, D.R. and Zwaenepol, W., "Distributed Process Groups in the V Kernel", *ACM Trancations on Computer Systems*, **3**(3), May 1985.

[Chou82] Chou, T.C.K. and Abraham, J.A., "Load Balancing in Distributed Systems", *IEEE Transactions on Software Engineering*. **SE-8**(4), July 1982, pp. 401-412.

[Chu80] Chu, W.W., Holloway, L.J., Lan, L.,M-T. and Efe, K., "Task Allocation in Distributed Data Processing", *IEEE Computer*, **13**(11), Nov. 1980, pp. 57-69.

[Chu87] Chu, W.W. and Lan, L.M-T., "Task Allocation and Precedence Relations for Distributed Real-Time Systems", *IEEE Transactions on Computers*, **C-36**(6), June 1987, pp. 667-679.

[Denning80] Denning, P.J., "Working Sets Past and Present", *IEEE Transactions on Software Engineering*, **SE-6**(1), Jan. 1980, pp. 64-84.

[Eager85] Eager, D.L., Lazowska, E.D. and Zahorjan, J., "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing", *Proceedings of the 1985 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, Austin, Texas, USA, Aug. 1985. pp. 1-3.

[Eager86] Eager, D.L., Lazowska, E.D., Zahorjan, J., "Adaptive Load Sharing in Homogeneous Distributed Systems", *IEEE Transactions on Software Engineering*, **SE-12**(5), May 1986, pp. 662-675.

[Efe82] Efe, K., "Heuristic Models of Task Assignment Scheduling in Distributed Systems", *IEEE Computer*, **15**(6), June 1982, pp. 50-56.

[Enslow77] Enslow, P.H., "Multiprocessor Organization - A Survey", *ACM Computing Surveys*, **9**(1), March 1977, pp. 103-129.

[Ferrari85] Ferrari, D., "A Study of Load Indices for Load Balancing Schemes", Report #UCB/CSD86/262, Computer Science Division (EECS), University of California, Berkeley, California, USA, October 1985.

[Ford62] Ford, L.R. and Fulkerson, D.R., *Flows in Networks*, Princeton, NJ: Princeton University Press, 1962.

[Frank85] Frank, A.J., Wittie, L.D. and Bernstein, A.J., "Multicast Communication on Network Computers", *IEEE Software*, **2**(3), May 1985, pp. 49-61.

[Friedrich85] Friedrich, M. and Older, W., "Helix : The Architecture of the XMS Distributed File System", *IEEE Software*, **2**(3), May 1985, pp. 21-29.

[Gonzalez77] Gonzalez, M.J., "Deterministic Processor Scheduling", *ACM Computing Surveys*, **9**(3), Sept. 1977, pp. 173-204.

[Gylys76] Gylys, V.B. and Edwards, J.A., "Optimal Partitioning of Workload for Distributed Systems", *Digest of Papers, COMPCON Fall 76*, Sept. 1976. pp. 353-357.

[Hac86] Hac, A and Johnson, T.J., "A Study of Dynamic Load Balancing in a Distributed System", *Proceedings of the ACM SIGCOMM Symposium on Communications, Architecturs and Protocols*, Stowe, Vermont, USA, Aug. 1986. pp. 348-356.

[Hoare78] Hoare, C.A.R., "Communicating Sequential Processes", *Communications of the ACM*, **21**(8), Aug. 1978, pp. 666-677.

[Hwang85] Hwang, K. and Briggs, F.A., *Computer Architecture and Parallel Processing*, New York, McGraw-Hill, 1985.

[Kain79] Kain, R.Y., Raie, A.A., "Multiple Processor Scheduling Policies", *Proceedings of the 1st International Conference on Distributed Computing Systems*, April 1979, pp. 660-668.

[Karshmer83] Karshmer, A.I., Depree, D.J. and Phelan, J., "The New Mexico State University Ring-Star System : A Distributed UNIX Environment", *Software - Practice and Experience*, **13**(12), Dec. 1983, pp. 1157-1168.

[Kleinrock85] Kleinrock, L., "Distributed Systems", *Communications of the ACM*, **28**(11), Nov. 1985, pp. 1200-1213.

[Kobayashi78] Kobayashi, H., *Modeling and Analysis : An Introduction to System Performance Evaluation Methodology*, Reading, Mass., USA, Addison-Wesley, 1978.

[Kratzer80] Kratzer, A. and Hammerstrom, D., "A Study of Load Levelling", *Proceedings of the IEEE Fall COMPCON*, 1980. pp. 647-654.

[Krueger84] Krueger, P. and Finkel, R., "An Adaptive Load Balancing Algorithm for a Multicomputer", Computer Sciences Technical Report #539, University of Wisconsin, Madison, Wisconsin, USA, April 1984.

[Kurose87] Kurose, J.F. and Chipalkatti, R., "Load Sharing in Soft Real-Time Distributed Computer Systems", *IEEE Transactions on Computers*, **C-36**(8), Aug. 1987, pp. 993-1000.

[Lampson81] Lampson, B.W., "Atomic Transactions", in Lampson, B.W.(ed.), *Distributed Systems - Architecture and Implementation*, Springer-Verlag, Berlin, 1981. pp. 246-265.

[Lavenberg83] Lavenberg, S., *Computer Performance Modeling Handbook*, New York, USA, Academic Press, 1983.

[Leland86] Leland, W.E. and Ott, T.J., "Load-balancing Heuristics and Process Behaviour", *Proceedings of the ACM SIGMETRICS Conference*, May 1986. pp. 54-69.

[Lin87] Lin, F.C.H and Keller R.M., "The Gradient Model Load Balancing Method", *IEEE Transactions on Software Engineering*, **SE-13**(1), Jan. 1987, pp. 32-38.

[Liskov79] Liskov, B., "Primitives for Distributed Computing", *Proceedings of the 7th ACM SIGOPS Symposium on Operating Systems Principles*, Dec. 1979. pp. 33-42.

[Liskov82] Liskov. B., "On Linguistic Support for Distributed Programs", *IEEE Transactions on Software Engineering*, **SE-8**(5), May 1982, pp. 203-210.

[Livny82] Livny, M. and Melman, M., "Load Balancing in Homogeneous Broadcast Distributed Systems", *Proceedings of the ACM Computer Network Performance Symposium*, April 1982. pp. 47-55.

[Lo84] Lo, V.M., "Heuristic Algorithms for Task Assignment in Distributed Systems", *Proceedings of the 4th International Conference on Distributed Computing Systems*, 1984. pp. 30-39.

[Luderer81] Luderer, G.W.R., Che, H., Haggerty, J.P., Kirlis, P.A. and Marshall ,W.T., "A Distributed UNIX System Based on a Virtual Circuit Switch", *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, Pacific Grove, California, USA, Dec. 1981. pp. 160-168.

[Ma82] Ma, P-Y.R., Lee, E.Y.S. and Tsuchiya, M., "A Task Allocation Model for Distributed Computing Systems", *IEEE Transactions on Computers*, **C-31**(1), Jan. 1982, pp. 41-47.

[Manning80] Manning, E., Livesey, N.J. and Tokuda, H., "Interprocess Communication in Distributed Systems : One View", *Proceedings of the IFIP Congress*, Tokyo, Japan, Oct. 1980. pp. 513-520.

[McQuillan77] McQuillan, J.M. and Walden D.C., "The ARPA Network Design Decisions", *Computer Networks*, **1**(5), Aug. 1977. pp. 243-289.

[Metcalfe76] Metcalfe, R.M. and Boggs, D.R., "Ethernet : Distributed Packet Switching for Local Computer Networks", *Communications of the ACM*, **19**(7), July 1976, pp. 395-404.

[Needham82] Needham, R.M. and Herbert, A.J., *The Cambridge Distributed Computing System*, Reading, Mass., Addison-Wesley, 1982.

[Ni82] Ni, L.M., "A Distributed Load Balancing Algorithm for Point-to-Point Local Computer Networks", *Proceedings of COMPCON Computer Networks*, Sept. 1982. pp. 116-123.

[Ni85] Ni, L.M., Xu, C-W., Gendreau, T.B., "A Distributed Drafting Algorithm for Load Balancing", *IEEE Transactions on Software Engineering*, **SE-11**(10), Oct. 1985, pp. 1153-1161.

[Nowitz79] Nowitz, D. A., "Uucp implementation description", Sect. 37 in *UNIX Programmer's Manual, Seventh Edition, Vol. 2*, Jan. 1979.

[Ousterhout80] Ousterhout, J.K., Scelza, D.A. and Sindhu, P.S., "Medusa : An Experiment in Distributed Operating System Structure", *Communications of the ACM*, **23**(2), Feb. 1980, pp. 92-105.

[Ousterhout82] Ousterhout, J.K., "Scheduling Techniques for Concurrent Systems", *Proceedings of the 3rd International Conference on Distributed Computing Systems*, Miami/Ft.Lauderdale, Florida, USA, Oct. 1982. pp. 22-30.

[Parker83] Parker, D., Popek, G.J., Rusidin, E., Chow, J., Edwards, D., Kiser, S. and Kline, C., "Detection of Mutual Inconsistency in Distributed Systems", *IEEE Transactions on Software Engineering*, **SE-9**(5), May, 1983.

[Popek85] Popek, G.J. and Walker, eds., *The LOCUS Distributed Systems Architecture*, Cambridge, Mass., MIT Press, 1985.

[Powell83] Powell, M.L. and Miller, B.P., "Process Migration in DEMOS/MP", *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, Dec. 1983, pp. 110-119.

[Rao79] Rao, G.J. and Stone, H.S., "Assignment of Tasks in a Distributed Processor System with Limited Memory", *IEEE Transactions on Computers*, **C-28**(4), April, 1979, pp. 291-298.

[Reed83] Reed, D.A., "A Simulation Study of Multimicrocomputer Networks", *Proceedings of the 1983 IEEE International Conference on Parallel Processing*, Aug. 1983, pp.161-163.

[Ritchie74] Ritchie, D.M. and Thompson, K., "The UNIX Time-Sharing System", *Communications of the ACM*, **17**(7), July 1974, pp. 1905-1929.

[Rowe82] Rowe, L.A. and Birman, K.B., "A Local Network Based on the UNIX Operating System", *IEEE Transactions on Software Engineering*, **SE-8**(2), Feb. 1982, pp. 137-145.

[Saltzer84] Saltzer, J.H., Reed, D.P. and Clark, D.D., "End-to-End Arguments in System Design", *ACM Transactions on Computer Systems*, **2**(4), Nov. 1984, pp. 277-278.

[Shrivastava82] Shrivastava, S.K. and Panzieri, F., "The Design of a Reliable Remote Procedure Call Mechanism", *IEEE Transactions on Computers*, **C-31**(7), July 1982, pp. 692-697.

[Silberschatz81] Silberschatz, A., "Port directed communication", *Computer Journal*, **24**(1), Jan. 1981, pp. 78-82.

[Stankovic84] Stankovic, J.A., "Simulations of Three Adaptive, Decentralized Controlled, Job Scheduling Algorithms", *Computer Networks*, **8**(3), June 1984, pp. 199-217.

[Stone77] Stone, H.S., "Multiprocessor Scheduling with the Aid of Network Flow Algorithms", *IEEE Transactions on Software Engineering*, **SE-3**(1), Jan. 1977, pp. 85-93.

[Stone78] Stone, H.S. and Bokhari, S.H., "Control of Distributed Processes", *IEEE Computer*, **11**(7), July 1978, pp. 97-106.

[SUN86] SUN Microsystems, *Networking on the SUN Workstation*, Mountain View, California, USA, 1986.

[Svobodova84] Svobodova, L., "File servers for network-based distributed systems", *ACM Computing Surveys*, **16**(4), Dec. 1984, pp. 353-398.

[Swan77] Swan, R.J., Fuller,S.H. and Siewiorek, D.P., "Cm* - A modular multi-microprocessor", *Proceedings of AFIPS 1977 NCC*, **46**, Arlington, Va, USA, AFIPS Press, 1977. pp. 637-644.

[Tantawi85] Tantawi, A.N., Towsley, D., "Optimal Static Load Balancing in Distributed Computer Systems", *Journal of the ACM*, **32**(2), April 1985, pp. 445-465.

[Tuomenoksa82] Tuomenoksa, D.L. and Siegel,H.J., "Analysis of Multiple-Queue Task Scheduling Algorithms for Multiple-SIMD Machines", *Proceedings of the 3rd International Conference on Distributed Computing Systems*, Miami/Ft.Lauderdale, Florida, USA, Oct. 1982. pp. 114-121.

[VanTilborg81] Van Tilborg, A.M. and Wittie, L.D., "Wave Scheduling : Distributed Allocation of Task Forces in Network Computers", *Proceedings of the 2nd International Conference on Distributed Computing Systems*, Paris, April 1981. pp. 337-347.

[Walker83] Walker, B., Popek, G., English, R., Kline, C. and Thiel, G., "The LOCUS Distributed Operating System", *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, Dec. 1983. pp. 49-70.

[Wittie78] Wittie, L.D., "MICRONET : A Reconfigurable Network for Distributed Systems Research", *Simulation*, Nov. 1978, pp. 145-153.

[Wittie80] Wittie, L.D., "MICROS, A Distributed Operating System for MICRONET, A Reconfigurable Network Computer", *IEEE Transactions on Computers*, **C-29**(12), Dec. 1980, pp. 1133-1144.

[Wittie81] Wiitie, L.D., "Communication Structures for Large Networks of Microcomputers", *IEEE Transactions on Computers*, **C-30**(4), April 1981, pp. 264-273.

[Zhou86a] Zhou, S., "A Trace-Driven Simulation Study of Dynamic Load Balancing", Report #UCB/CSD/87/305, Computer Science Division (EECS), University of California, Berkeley, California, USA, September 1986.

[Zhou86b] Zhou, S., "An Experimental Assessment of Resource Queue Lengths as Load Indices", Report #UCB/CSD/86/298, Computer Science Division (EECS), University of California, Berkeley, California, USA, June 1986.