THE RELATIVE PERFORMANCE OF SCALABLE LOAD BALANCING ALGORITHMS IN LOOSELY-COUPLED DISTRIBUTED SYSTEMS

VOL I

RUPERT ANTHONY SIMPSON

Submitted for the degree of Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

July 1994

The University of Aston in Birmingham

THE RELATIVE PERFORMANCE OF SCALABLE LOAD BALANCING ALGORITHMS IN LOOSELY-COUPLED DISTRIBUTED SYSTEMS

Rupert Anthony Simpson

Submitted for the degree of Doctor of Philosophy

1994

Summary

The computer systems of today are characterised by data and program control that are distributed functionally and geographically across a network. A major issue of concern in this environment is the operating system activity of resource management for different processors in the network. To ensure equity in load distribution and improved system performance, load balancing is often undertaken.

The research conducted in this field so far, has been primarily concerned with a small set of algorithms operating on tightly-coupled distributed systems. More recent studies have investigated the performance of such algorithms in loosely-coupled architectures but using a small set of processors.

This thesis describes a simulation model developed to study the behaviour and general performance characteristics of a range of dynamic load balancing algorithms. Further, the scalability of these algorithms are discussed and a range of regionalised load balancing algorithms developed. In particular, we examine the impact of network diameter and delay on the performance of such algorithms across a range of system workloads. The results produced seem to suggest that the performance of simple dynamic policies are scalable but lack the load stability of more complex global average algorithms.

KEYWORDS: Distributed Resource Scheduling
Load Balancing Protocols

To Nadine and Perry,

and Mum and Dad

ACKNOWLEDGMENTS

I would like to express my thanks to the following people who helped me at various stages during the development of the system and the preparation of the thesis:

To my supervisor Dr. A. J. Harget for the advice given, and suggestions made at every stage of the project. This was particularly important in sustaining project momentum, given the mode of study undertaken.

To the technical support staff for their untiring willingness to provide system support and finding the additional disk space and processing power required by the model. In particular, I would like to thank Mary Finnegan, and Duncan McLean.

To my children for the moments of relaxation, and family and colleagues for their friendship and moral support. And many thanks to Noreen for the time spent proof-reading the document.

Finally, I would like to acknowledge and thank UCE in Birmingham for the financial and practical support given throughout the project. In the latter case, the support of Malcolm Reid and his team in the Design and Print Unit is much appreciated.

TABLE OF CONTENTS

VOLUME I

LIST OF ILLUSTRATIONS

VOLUME I

VOLUME II

LIST OF TABLES

VOLUME I

VOLUME II

CHAPTER 1

INTRODUCTION

As computer manufacturers move towards the delivery of more open computer architectures and portable software products, the performance of computer systems has become the most important criterion in procurement decisions of many users. It is often the case that as the performance of computer systems improves, users find themselves in a position to tackle problems that were previously considered to be intractable. In the quest for ever-increasing system performance the adoption of very large scale integration(VLSI) technology and material has enabled computational power to increase by a factor of 10 on average, every five years [Hockney88]. Many have argued that this rate of increase, through technology alone, could not be sustained and ultimately was subject to an upper physical limit which, most systems in the 1980s had almost reached. According to Lewis [Lewis92], one is likely to see the maximum switching speed of silicon reached during the 1990s and the rapid progress in achieving greater computing speed level off.

It is generally accepted by most researchers that further increases in performance can only be sustained through reorganising the way in which a computer operates such that a greater degree of parallel processing takes place. Whilst the RISC-based technology of the 1990s has focused on performance enhancement via essential but simple instruction sets, it has also become a cost-effective building block for computers with multiple processors, with or without pipelined instruction and data streams. Furthermore, the expansion of network and communications technology in the 1990s has broadened the issue of performance based on tightly-coupled multiprocessor systems to cover systems consisting of two or more autonomous

processors linked together over a well-defined geographical region, be it local, national, or international. Processors can be added to the network by local user communities to service their processing requirements as well as to share resources with other communities on the network. The incremental growth in the system may well improve the overall performance of the system for all its users and thereby, offer enormous computational power at modest cost in contrast to today's multiprocessor supercomputers. However, the realisation of the performance potential of distributed systems is dependent on the design and development of new and effective system software for such environments. Whilst the performance of a local system is dependent on the nature of its workload and the level of interaction across the network with other user communities, the overall effectiveness of the system relies on the ability of the resource scheduler used to predict, identify, and rectify such things as load imbalances and the under-utilisation of expensive system resources.

It is generally the case that in a distributed system, some user-communities on the network have a propensity to generate a more demanding workload for their local processor than others with more modest requirements. Consequently, a disparity in the quality of service received by all users will develop. Therefore, one of the goals of a well-tuned resource scheduler is to exercise both local and remote scheduling disciplines so that the system workload is shared equally between all available processors. The event-chain initiated by the scheduler should result in process migration from the over-utilised to the under-utilised processor sites. Many researchers have shown, and is re-affirmed in this study, that such schedulers result in an overall improvement in performance for all users even in cases where the simplest of implementations is being used. The static allocation methods are the most limited as it requires a priori knowledge of future sites and their workload pattern. In contrast, adaptive (or dynamic) allocation methods are able to respond to changes in

system workload. However, the performance gained is inevitably a trade-off with the expense of collecting, composing, and broadcasting system state information that is accurate and reliable. This is one of the main factors that has been influential in the architecture and implementation of commercial distributed systems.

Many of the schedulers developed and investigated for distributed systems are analogous to traditional computer architectures, characterised by the centralisation of data (such as file servers), and program and control (resource schedulers). Semi-distributed systems are being developed to attain some of the benefits of decentralised architectures using the hardware and software technology of today. The greater challenge is in distributed systems where each processor is autonomous, having its own memory, resources, and operating system, executing asynchronously, and communicating solely by message-passing protocols. A key problem peculiar to such systems is that of ensuring the timeliness and integrity of messages on which critical resource management decisions will be made. The problem is further exacerbated by unpredictable increases in system workload. The task of balancing workload under such circumstances is not trivial. Factors such as workload characteristics, system state, communication delay, and processor characteristics must be considered. These are the issues that this study is primarily concerned with.

A number of researchers have studied and documented a variety of algorithms for effecting load balancing on decentralised (or loosely-coupled) systems under light, moderate, and heavy workloads. However, many of the systems studied are small, and eight or nine-processor configurations are commonplace. Given the success of the algorithms considered by other researchers, the optimal size of the system and the scalability of such algorithms are important design parameters for any resource manager. Furthermore, it is possible that as the diameter of the system increases,

algorithms previously considered inefficient may become more attractive. Finally, it is also important to know the behaviour of such algorithms under extreme load conditions especially in cases where the system is tending towards its saturation level.

The aims of the research conducted here are summarised as follows. Firstly, to investigate the behaviour and general performance characteristics of load balancing algorithms of varying implementation complexity under a variety of workload conditions. Secondly, to examine the impact of network diameter and delay on the performance of such algorithms. Thirdly, to identify efficient and scalable load balancing algorithms. A simulation model of a loosely-coupled distributed system was developed to aid the investigation. The workload of each node consisted of independent, CPU-intensive processes. Given the larger system configurations under consideration, algorithms were developed where each local host attempted to construct and maintain a communicating set of processors participating in load balancing activity over a period of time. The implementation of such algorithms varied in their complexity and performance, in terms of average response time and load stability, compared to adaptive global average algorithms.

The remainder of this thesis is organised as follows: in Chapter Two an overview is given of distributed system architecture with particular consideration being given to the design philosophy and mechanisms required by local hosts engaged in load balancing activities. In addition, a selection of distributed operating system kernels, currently under development are discussed.

In Chapter Three the structural characteristics of load balancing algorithms are examined in greater detail and a general classification for such algorithms presented.

The relative performance of the algorithms are compared and the work and findings of their authors summarised and evaluated.

Chapter Four provides a detailed discussion of the design and implementation of the model. The first part of this chapter focuses on the design and implementation of load balancing algorithms as message-passing protocols using state transition modelling techniques; and the second part is concerned with building the simulated system architecture using object-oriented techniques.

Chapter Five presents a discussion of the methods used to verify and validate the design of the model and its resulting output. In Chapter Six an analysis, and evaluation of the experimental results is presented in terms of the representational accuracy of the model and the relative performance of the load balancing algorithms considered.

Chapter Seven presents a summary of the findings, and suggests areas for which further investigation may be necessary and appropriate.

The appendices include: graphs and statistical data for the main algorithms; monitoring output and supporting descriptions for load balancing protocols; detailed model design descriptions and equivalent program source code.

CHAPTER 2

DISTRIBUTED SYSTEMS ARCHITECTURE

2.1 DEFINITION AND CLASSIFICATION

It is the considered view of many scientists that one of the grand challenges of the 1990s is to increase the speed of computers to the teraflops level so that very large scientific and engineering problems might be solved [Cocke88, Lewis92]. The advent of RISC technology has made it possible for workstation manufacturers to offer within their general product range, relatively cheap single-processor systems with speeds of up to 100 MIPS. Ultimately, performance enhancement by merely reimplementing traditional architecture is subject to physical, technological, and economic constraints. A number of researchers are of the opinion that further increases in performance can only be sustained through reorganising the way in which a computer operates such that a greater degree of parallel processing takes place[Watson82, Dennis80, Gurd85, Trealeaven82]. Thus, computers with multiple processors, have been developed for use in highly specialised applications. However, developments in network technology have facilitated the construction of multicomputer systems where specialised resources of local communities can be made available to other more remote communities of users. Such systems are loosely referred to as distributed systems, and are one of the principal system design innovations to appear in recent times.

Given the variety of possible distributed system configurations, a precise definition or classification for such systems is desirable. In using the classification proposed by

Flynn [Flynn72], one is restricted to the view that all distributed systems have Multiple Instructions that operate concurrently on Multiple Data streams. The taxonomy of Flynn places two quite different classes of computer systems, namely multiprocessor systems and distributed systems, in the same broad category. Fallmyr et al [Fallmyr91] regard a distributed system as distinct from a multiprocessor system by having the following properties: asynchronous parallel processes communicating over links that are subject to delay; no central point of control; inaccurate global state information or global time; and processes which change their communication patterns according to changes in the system state. However, this definition only covers a specific group of systems, commonly referred to as loosely-coupled systems, where processing and data is fully distributed across the available processors. That is, each processor is autonomous and has its own user processes, local memory, system resources, and operating system. As there is no shared memory, or central point of control, co-operating processors must execute asynchronously and communicate solely by message-passing protocols. Distributed systems with a centralised file server or process scheduler may be regarded as semi-distributed, whereas fully distributed systems have decentralised functionality. Therefore, in the context of its common usage, Tanenbaum [Tanenbaum92] defines a distributed system as: "..one that runs on a collection of machines that do not have shared memory, yet looks to its users like a single computer." Primarily, it is the level of imperfection exhibited by the communication network for losing data, being slow, sluggish and untimely in its actions when overloaded that distinguishes one distributed system from another.

2.2 Performance Issues

According to Cocke [Cocke88], the three principal contributors to system performance are the algorithm, compiler, and the system architecture. In the latter

case, performance improvements tend to grow linearly relative to the number of processors used in the architectural design. Whilst algorithm improvements tend to have the most spectacular effect on performance, with performance changes of logarithmic proportions (such as $N_{log}N$) in some cases, exploitation of the algorithm's characteristics in the underlying system architecture tends to result in the creation of highly specialised systems. In many instances, such systems tend to perform badly when applied to different algorithms. Cocke therefore suggests that:

> "..the simultaneous optimisation of these three factors holds the key to the highest possible performance." [Cocke88].

The partitioning of an algorithm at a global level which facilitates operation on its separate components using the multiple processors of the underlying architecture remains a key area of research and beyond the scope of this thesis. However, unresolved questions remain about the underlying system architecture. Researchers into system architecture have generally attempted to identify the total number of processors, and the hardware and system software configurations that can maximise overall system performance. Distributed systems with multiple processors and control software, like its shared memory single-box multiprocessor counterpart, are subject to the laws of diminishing returns, where to continually increase the number of processors would cause a disproportionate increase in hardware costs, and a corresponding decrease in overall system performance as a result of increased communication overheads. However, unlike shared memory multiprocessors where performance degradation may well occur in cases of hundreds of processors, it is conceivable that distributed systems with thousands of processors can sustain performance many orders of magnitude greater than the systems of today.

The three key attributes that a computer architecture should deliver are security, reliability, and performance. Traditional computer architectures, characterised by the centralisation of data, program, and control, were unable to deliver high performance to users at a local level. Aspects of security which were previously considered to be the strength of such systems, have given way to a range of secure distributed systems architecture. In recent years, the choice of architecture is primarily governed by the level of security required for user data and simplicity of administration. On those criteria distributed systems with powerful centralised file servers are popular [Ciciani92].

Distributed systems with centralised servers and/or schedulers are generally more secure and relatively simple to manage. However, such an architecture gives rise to the creation of a central point of failure, and ultimately, communication and performance bottlenecks, with its consequential effects on "slave" processors and their user communities. In contrast, a system in which processing and data is fully distributed across the available processors, supported by peer-to-peer communication, should be more robust and reliable as each processor is autonomous and has its own memory, resources, and operating system. Furthermore, a processor will also have a small number of point-to-point connections to other processors and will be able to route messages via these connections. As there is no shared memory, co-operating processors must execute asynchronously and communicate solely by message-passing protocols. Given this diversity in computer architecture and performance, Cocke's conclusion is particularly poignant:

> "I do not find it discouraging that there seems to be no clear-cut route to high performance. I feel the flexibility of computers will allow us to solve problems in ways not yet envisioned, and will make the future of computing more interesting than the past" [Cocke88].

2.3 Network Hardware

In a distributed system with multiple host processors the configuration and interconnection between nodes may have a significant effect on the performance, cost and reliability of the system. According to Reed et al [Reed87] "a network ill-suited to prevailing communication patterns results in message congestion and excessive communication delays." In terms of the transmission medium, a distributed system can be constructed with different types of cables or directional links which may be differentiated by their bandwidth, and transmission reliability. Generally, coaxial cable has a greater bandwidth than twisted copper wire pairs, but a lesser bandwidth than fibre optic systems. The amount of information carried (bandwidth) by coaxial cable is typically within the region of 10 Megabits per second. Given the transmission medium, the topology of a network will depend to some degree on the architecture of the interconnection network. A bus architecture has a single backplane to which all machines are connected and over which messages are sent and received. Messages transmitted in the form of packets may be addressed to a specific host or to all hosts on the network. In the latter case, commonly referred to as broadcasting, each message contains a special address which will be recognised by all hosts thereby copying the transmitted message to their local message buffers. Generally, broadcast mechanisms of this nature are very efficient for transmitting messages relevant to all hosts, such as the processor state information. That is, the cost of transmission to all hosts via the broadcast mechanism is only marginally more expensive than transmitting to a single host in the case of Local Area Networks (LANs). In contrast to broadcasting, a multicast mechanism is characterised by transmitting messages to a specific subset of the hosts or processes available in the system [Cheriton88]. A common implementation is for transmitted message packets to contain a group identifier which is recognisable to hosts belonging to that specific group. Such

systems become inefficient in cases where: groups are dynamically configurable; there is a need to send messages to a sub-group; or hosts belong to different groups.

In terms of LANs one of the most typical and common network arrangement would be either an Ethernet or IEEE 802.3 single-bus unidirectional broadcast system with a fixed number of hosts. However, it is only able to handle one signal in the cable at any one time from any host. The hardware operation requires that the controller for each host checks to see whether another host is transmitting, and will continue to do so at random time intervals until the potential for dual transmission, message collision, and data loss are avoided. A minimum length for data packets must be imposed to make collisions detectable while the packets involved are being transmitted. This technology, known as Carrier Sense Multiple Access/Collision Detect (CSMA/CD) has been found to be ideal when the network is lightly loaded and the probability of two hosts transmitting at the same time is low [Nam92]. As most Ethernet technology typically operates far below their maximum capacity, synchronisation overheads at such loads are often absent. However, the main problem with this architecture is its behaviour under extreme loads. In such cases, a large number of hosts trying to use the bus at the same time would result in a greater incidence of message collision, loss, and retransmission. Further, the algorithms used to avoid this problem area requires additional processing time and introduces further complexity in network operation. Such delays may be crucial to the timeliness of state information on which load balancing decisions are made. In addition, network failure (such as a break in the bus cable) is liable to result in the whole system going down as there is no opportunity to re-route messages to their intended destinations. Greater reliability can be achieved by the use of a dual-bus unidirectional broadcast system that would allow each host to transmit on the outbound channel whilst receiving on the inbound channel. An alternative strategy proposed by Nam et al

[Nam92] would be to include a collision-avoidance switch using a multichannel broadcast star network. In this topology, the central host (or hub) would only switch the data packet if the destination link is idle, otherwise it must be retransmitted by the source. Nam et al found that whilst the use of a multichannel medium reduced the probability of collisions in CSMA/CD based technology, at low loads the higher data rate reduced the overall response time.

Another widely used LAN standard is based on a token-passing ring topology. It avoids the problem of collision and its detection by ensuring that only the host holding the data packet containing a special pattern of bits (known as a "token") can transmit. The token circulates around the ring continually and is absorbed by the host wishing to transmit. On completion of transmission, the token will be released. In this topology, all stations physically receive the transmitted packet, but only the addressed host will carry out further processing operations on the received data.

To overcome some of the problems presented by a bus-type architecture switched architectures make use of individual cables between machines. Messages are then routed by explicit switching decisions at intermediate hosts before reaching the destination host. An ideal topology is one in which every host is directly connected to every other host. The main advantage is the greater speed and reliability of message transmission even at extreme loads. Should a connection fail alternative paths for re-routing messages will exist between the sender and the receiver. However, the main problem with this arrangement is the packaging constraints and wiring costs which tend to impose a limited number of connections to and from each host. In the hypercube topology for example, the dimension of the cube determines the number of nodes and the number of links to each node. Thus, a three-dimensional hypercube will consist of eight nodes and three communication links to neighbouring nodes.

One advantage of this topology is that the maximum distance for routing messages from source to destination increases logarithmically with the size of the cube. Thus, in the case of a 9-processor hypercube, a maximum of three hops will be required to send a message anywhere within the network. Further, host addressing and message routing is simple to compute and implement. The main weakness of the topology however, is the exponential growth in connections with every increase in the hypercube dimension. Consequently, a 10-dimensional hypercube supporting 1024 nodes, each with ten communication links represents the current technological limit for this arrangement.

The two-dimensional mesh topology used in this study imposes a limit of up to four connections per host. One of the main problems in this arrangement is to ensure that the route to a destination is both the shortest possible, and nodes are evenly loaded as far as routed message traffic is concerned. Without this constraint, the risk remains of certain intermediate nodes becoming the "shortest path" hubs for "send-and-reply" communication between hosts. Thus, the hub processor is likely to spend more time handling or routing message traffic relative to servicing its own process queues.

2.4 Communications Model

Comer [Comer88] views protocols as providing formulae for passing messages, specifying message format, and handling error conditions. The complexity of these formulas can make a significant impact on the performance of a distributed system as the network performance is determined by the characteristics of the protocol used [Tusch92]. A host with limited processing power, must not only meet its local processing requirements but invoke and execute protocol software to assemble, disassemble, and interpret transmitted data. Further, under extreme loads a high

incidence of transmission errors, and subsequent recovery processes will cause the protocol to be invoked and executed more frequently.

The OSI (Open Systems Interconnection) Reference Model published by the International Organisation for Standardisation (ISO/IEC) identifies seven levels of network functionality ranging from the physical to the application layer. The collection of software that represent the "layers" of functionality is commonly referred to as the protocol stack as the concept embodies a clearly defined interface between each layer such that only service requests can be made by any given layer to the layer immediately below it. That is, each layer participates in a dialogue with the layer immediately above it without assuming control of its data or the manner in which it operates. Thus, in some systems a given host will only send data packets upwards from the physical layer to the application layer if it is addressed by the packet. The advantage of the protocol stack model are twofold. Firstly, by having a well-defined interface, the integrity of each layer is ensured. Secondly, greater flexibility and ease of maintenance is achievable as each layer can be replaced or modified independently of other layers providing the interface remains intact.

The main problem with the stack concept is that strict adherence to its principle may well result in performance overheads as each layer is engaged at different stages with messages being passed between distributed applications. Furthermore, the complexity and size of some vendor implementation of a particular layer of functionality may in practice be so inefficient that the discipline of the protocol stack is abandoned. For example, some large commercial users have opted for the IBM SNA protocol, in preference to TCP/IP, for transferring large blocks of data around a network because the overhead imposed was significantly smaller in the case of the former, whereas the latter consisted of around 80,000 lines of code [PC93]. Thus, the greater the time

dedicated by a processor to the execution of the protocol layers, the greater the message transfer delay compared to the pure network access delay for a CSMA/CD based bus technology, for example. Message queues and buffers fill up in the receiving host as the load increases on the network. For certain protocols overflowing message queues, or messages chronologically out of sequence, results in a greater propensity for packets to be lost and congestion becoming a persistent state as senders enter "retransmit message" loops.

Each layer of the protocol stack offers a range of communication services to potential users, from transmission error control to data traffic flow control services. A transport layer protocol such as TCP guarantees error-free delivery of all messages sent, and in the sequence in which they were transmitted. This is made possible through its connection-oriented, and flow control services, supplemented by the use of time-outs for monitoring acknowledgements from recipients. A connection-oriented service means that a connection must be established between host machines or the applications concerned before communication can take place. The flow control service ensures that a sender does not retransmit at a rate in excess of the receiver's ability to process the transmitted data. Whilst guaranteed message delivery is desirable between the hosts of a distributed system, a connection-oriented exchange can impose significant processing overheads in terms of the time taken to establish a connection, and terminating the connection once data transfer is complete.

In contrast to TCP, the UDP protocol offers a connectionless packet delivery service. Although this protocol is much simpler than TCP and avoids the overheads of managing a connection, there is no guarantee that messages sent using the UDP protocol will arrive at their destination. Therefore, for a distributed system which engages in load balancing one could send status information using protocols such as

UDP that offers a connectionless service. However, whilst it may be possible for the system to tolerate the loss of state information pertaining to the workload of one or more of its hosts, the loss of processes in transit would be unacceptable. Therefore, it is imperative that the migration of a process be carried out using a connection-oriented protocol. In cases where multiple protocols are being used on the network, one possible solution is to have self-identifying frames where "the Ethernet interrupt routine uses the packet type field of arriving packets to determine which protocol was used in the packet" [Comer91].

At a functional level, client-server models are popular and useful abstractions for describing the interaction between two or more processing components. For example, when applied to the protocol stack, the software of the upper layer acts as client processes by making requests to the layer below for specific services. The latter will then play the role of server by performing the required service on behalf of the client making the request. However, where the provision of a service is effected elsewhere on the network, the local "protocol" server must itself become a client and make requests to its remote peer for the required service. The client-server model is also applicable to computation and information services and will be used throughout the remaining chapter.

2.5 Distributed Resource Management

The primary goal of a distributed operating system is to coordinate and manage the collection of resources on the network [Goscinski90, Tanenbaum90]. In so doing its activities should be transparent to users and user processes. That is, a process may be moved or scheduled on a remote host without the user having to make an explicit change in the process's name, behaviour, and resources used. However, whilst transparency remains the key attribute of a distributed operating system, a range of

implementations exist, from those where scheduling, data and file service activities are the responsibility of a single processor to those where both data and processing are distributed. For example, in Figure 2.1, if processor A was designated to be the file server, then all client processors B, C, and D would send their file processing requests to A. Correspondingly, if processor A was also responsible for the process scheduling activity of the system then the workload of all client processors would be determined and influenced by the central server.

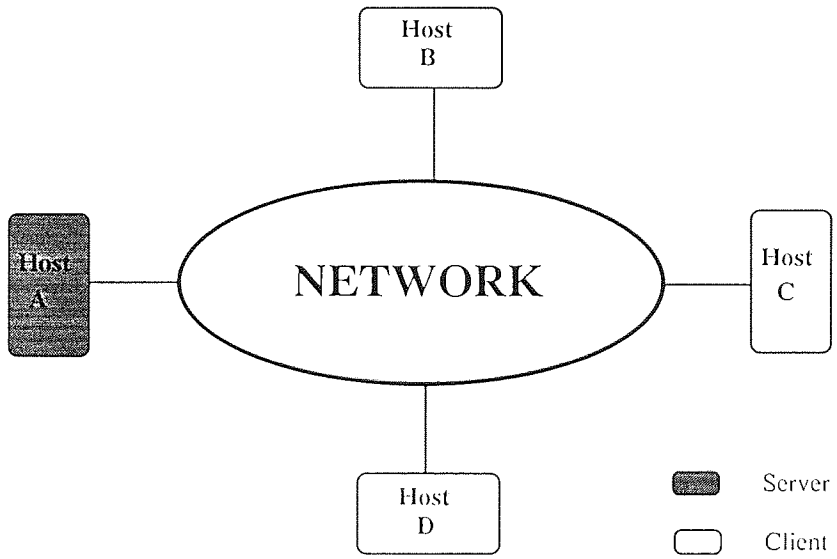


Figure 2.1 Distributed System Model using a Centralised Server

In contrast, a distributed system where both data and process scheduling are distributed would mean that all processors A, B, C, and D would be responsible for the scheduling of their own and the system workload, as well as the integrity of the data files of the system used. For workload redistribution to be effective, Goscinski et al [Goscinski90] argues that such a facility should make decisions on **when** to balance, **which** process or process group to exchange, and **where** in the network the most acceptable destination host resides. However, as illustrated in Figure 2.2, the load balancing facility will need to interact with other system services of the distributed operating system, such as the migration facility, the high-level scheduler,

and the communication sub-system (in the form of the protocol stack). Figure 2.2 shows that the load balancer makes use of both local and remote state information obtained from the local scheduler, and remote hosts respectively. The migration facility manages the transfer of processes to and from remote hosts, and this may in turn be determined by the host (according to its address) selected by the load balancer. The protocol stack will be used by both the load balancer and the migration facility to exchange state or process information.

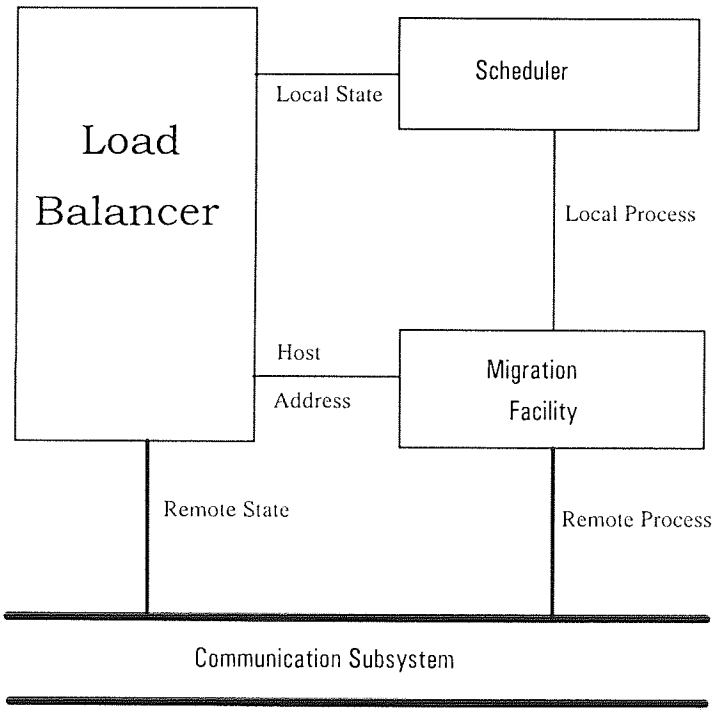


Figure 2.2 The Process Architecture for Load Balancing Mechanisms

On most distributed systems, the software of the communication sub-system resides in the operating system kernel and is shared by all system and user processes in the system. In particular, the system processes of a given host wishing to engage other hosts in the exchange of state information or user processes, will interact with one or more of the relevant active protocol process modules available. Whilst the communication sub-system is an important component of any distributed system,

detailed study is beyond the scope of this thesis. Therefore, the following sub-sections will lay emphasis on the other major components of the workload balancing facility illustrated in Figure 2.2.

2.5.1 Process Scheduler

In a uniprocessor or multiprocessor environment, the goal of a process scheduler is to match executable processes with available processors such that no processor is idle, and no process is kept waiting for significant periods of time. Depending on the application environment and the operating system, pre-emptive, round-robin scheduling may be applicable, whereas in other cases a non-preemptive, shortest job first policy may be appropriate. The execution of the scheduler will impose performance overheads attributable to: the execution time of the scheduling algorithm; and the time it takes to make a context switch between the scheduler and other processes in the system. Singhal et al defines a distributed scheduler as:

> "...a resource management component of a distributed operating system that focuses on judiciously and transparently redistributing the load of the system among the computers such that overall performance of the system is maximized" [Singhal94].

More specifically, Blake [Blake92] sees the scheduling activity as being concerned with assigning processes to processors in multicomputer systems so that the end result is the minimisation of process completion times. Therefore, in a distributed system the role of the scheduler is often synonymous with load balancing. The scheduling overhead in this case, includes the activity of transferring a process from the local host to the process queue of the remote host (analogous to a local context switch) in addition to the execution time of the scheduler itself.

The notion of idle processors in a multicomputer system can be loosely understood to be processors with under-utilised processing capacity, where utilisation is measured against a fixed or variable threshold value. Whilst each host may operate a local scheduling discipline for its own process queue, a tightly-coupled system with a centralised scheduler will schedule work to those hosts with workloads below a given threshold. Some researchers [Ali92, Ahmad90, Lin91] have investigated load balancing in systems using a centralised scheduler. In such systems, the scheduler will use state information collected from all processors and select the processor that will yield the maximum performance advantage to one or more of the migratable processes in the system. In contrast, scheduling in a loosely-coupled system requires each host to execute its own scheduler, and collect its own state information from other hosts [Blake92, Johnson88]. According to Blake,

> "The collection of global information requires a non-trivial amount of overhead, but is essential to develop a moderately effective scheduler." [Blake92]

Therefore, the success of scheduling strategies for distributed systems will be mainly governed by the timeliness and accuracy of the state information collected and used in scheduling decisions. Thus, Blake argues that heuristics must be used in order to constrain the complexity of the scheduler relative to the number of schedulable processes in the system [Blake92].

A further problem for schedulers in distributed systems relates to process granularity. The scheduling of coarse grained processes, consisting of a set of intercommunicating processes, presents additional parameters in the scheduling equation. Firstly, to schedule each member process to any host that becomes "idle" may result in the scattering of set members throughout the network. Whilst, it is possible that a more balanced load is achieved, the average response time may suffer as these processes maintain their communication channels over much greater distances (via implicit

Remote Procedure Calls) and are subject to the prevailing operating environment of the resident host. A refinement on such a strategy is to constrain the scheduling policy such that communicating processes are grouped together via those hosts that have spare processing capacity. Eventually, all processes belonging to the communicating set would reside on the same host, but with a better overall performance than if they had not been migrated in the first place. Nevertheless, even this strategy will incur significant interprocess communication overheads, even though it may only be for a short time. A more favoured strategy used in the study of Johnson [Johnson88], is to make the process group the unit of distribution. Thus, an idle host will only receive work if it has sufficient capacity to accommodate the members of a given process group on the local host. This strategy eliminates the overheads caused by distributing the members of a process group but may result in processor fragmentation. That is, the scheduler is unable to find a host with sufficient capacity to accommodate all the members of the available process groups. The growth in distributed applications using multiple process threads has highlighted the problem of the granularity of scheduling as shared memory is one of the primary means by which communication between threads can be accomplished. Further, as developers and users move towards object-based applications, composed of a collection of objects that communicate using messages, the problem of scheduling in a distributed environment will become more acute.

Nevertheless, whether a centralised or distributed scheduler is being used in a distributed system, an ideal scheduling strategy would be a preemptive one that results in the remote execution on the least loaded processor say, of the local host's longest running jobs or its largest group of intercommunicating processes.

2.5.2 Load Balancing Servers

As indicated in the previous section, load balancing can be viewed as a sophisticated high-level process scheduler for a distributed environment. However, as illustrated in Figure 2.2, the remit of the load balancer is the network of processors. In contrast, the scheduler is primarily concerned with local processors and processes. Thus, as discussed in the previous section, the local scheduler will influence and may decide on the process or group of processes to be selected as candidates for migration. Furthermore, the separation of load balancing and scheduling functions enables such servers to be enhanced, replaced, and maintained with minimal impact on the operation and cost of the other.

It is common to find in the literature no real distinctions being made between load sharing and load balancing. According to Cybenko [Cybenko89], the goal of load balancing is for each processor to perform an equitable share of the total work load. If load sharing is concerned with sharing workload fairly among hosts regardless of origin then it equates to load balancing. In a tightly-coupled multiprocessor environment where communication delays are negligible, load sharing regardless of origin is acceptable. However, in a loosely-coupled environment load balancing strategies are driven by the desire to improve overall system performance. Thus, in the latter case, the origin and state of the remote host must be considered in any load balancing policy. In either case, there is still a need to avoid bottlenecks in the system, and maintain robustness and performance in the presence of partial failures in the system [Fallmyr91]. Many researchers such as Hatch et al [Hatch90] see no difference between load balancing and load sharing, with load sharing being defined as attempting "..to improve system performance by transparently dispersing the individual hosts' workloads throughout the system" [Hatch90]. Therefore, in the following chapters load sharing and load balancing will be used in a similar light.

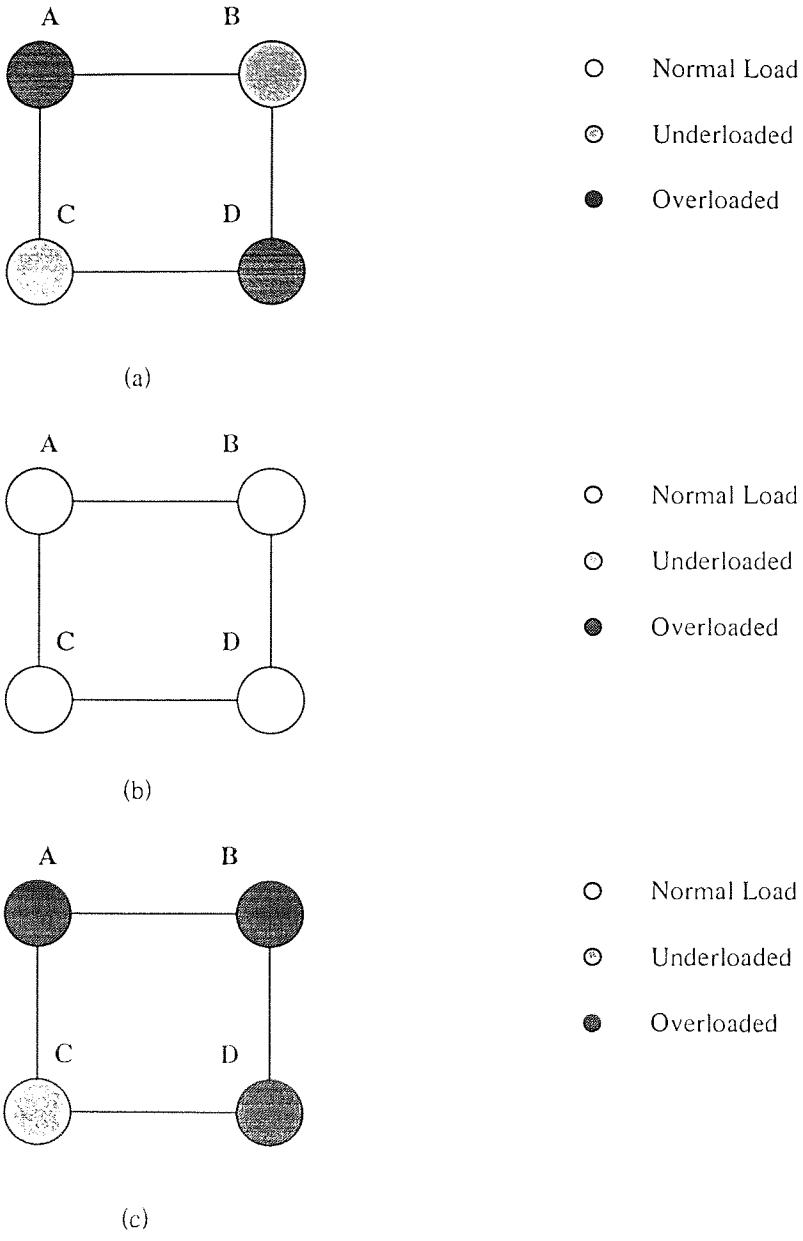


Figure 2.3 Load Balancing Example Using Four Interconnected Processors

The primary aim of load balancing is to ensure that the resources of the system as a whole are utilised in a manner that enables all hosts to deliver a service within a predefined quality range to all its client processes. Figure 2.3a shows a point-to-point distributed system with four homogeneous host processors. The process arrival rate at each host is such that two processors (A and D) are always overloaded and two processors (B and C) underloaded. Thus, whereas the latter are under-utilised and

able to produce response times below a predefined threshold of acceptance for all their processes, the overloaded processors A and D are characterised by excessive use and response times over and above the response threshold. Therefore, the use of load balancing techniques, resulting in the migration of excess processes from host A and host D to the underloaded hosts, may result in all hosts having a normal workload (Figure 2.3b) and delivering a service within a specified quality range. However, it is important that any technique employed does not simply shift the problem of load imbalance from one pair of hosts to the next, or further exacerbate the problem of an already overloaded host, resulting in a further degradation in system performance. In the situation where all hosts are heavily loaded, illustrated in Figure 2.3c, load balancing should still be possible as long as it is able reduce system instability and ensure that no single host is more overloaded than another.

2.5.3 Process Migration Facility

Joosen et al [Joosen90] defines process migration as an operation "causing the execution of the process to continue on another machine with the same functionality as if it was never reallocated". By implication, migration can only be effective if the behaviour of other processes which interact with the migrated process remain unaffected. That is, process migration should be completely transparent to the user, related processes, and the migrated process itself throughout the lifetime of the latter. Joosen et al views transparency as a fundamental component of a process migration facility and defines it in the following manner: "the migration of a process is completely transparent when the process and all processes that cooperate (communicate) with the migrated process execute exactly as if the migration never occurred" [Joosen90] . Considerable processing overheads can result from process migration as the process itself and information on its state must be transferred. The process state information required would include the contents of registers and stacks,

its virtual memory address space, and any file descriptors being used. For example, given a 10Mbps Ethernet communication medium, and a process using 10 megabytes of address space, at least 10 seconds would be required to transfer the memory contents alone. Therefore, process migration can only be justified if the benefit to the process is significantly greater than the cost [Mullender93].

The process migration facility is not load balancing, but the basic tool for the implementation of any load balancing policy. According to Artsy et al [Artsy89], the design and implementation of the facility is problematic due to the inherent complexity of the operations to be performed. That is, "..the mechanism for moving processes must reliably and efficiently detach a migrant process from its source environment, transfer it with its context (the per-process data structures held in the kernel), and attach it to a new environment on the destination machine" [Artsy89]. That is, once the decision has been made to transfer a process from the local host computer to a remote host willing to accept the process, the following generalised process migration steps need to be carried out on the local host:

a) Suspend the selected process, and save its current execution environment.

b) Effect the protocol for the transfer of the process and its state information to the remote host.

c) After remote execution is complete, effect the protocol for receiving the process and its state information.

d) Restore the process's original execution environment.

Likewise, the migration mechanism of the remote host concerned will cooperate in the transfer by effecting its protocol for receiving both the process and its state information. Once the remote host has established an execution environment for the migrated process, the protocol is made complete by the return of an acknowledgement

message to the originating host. The latter can now destroy the execution image of the process it had just migrated. However, the execution profile of the process may dictate the extent to which the local host should preserve its execution environment when the process is resident elsewhere.

The term *residual dependency* was used by Joosen et al [Joosen90] to describe the case where the whole or part of a migrated process's execution environment continued to exist on the originating host computer. In time-limited residual dependency, the migrated process would run on its new host after a finite period of time. In the migration facility outlined residual dependency would come into effect from the time the local host starts the transfer of the process to the time it receives the acknowledgement from the remote host. Independent, CPU-bound processes in a homogenous distributed system tend to exhibit this characteristic. According to Douglis et al [Douglis91b] residual dependency is undesirable due to its general propensity for unreliability, poor performance, and level of complexity. For example, consider the case of I/O bound and cooperating processes. The devices that are being used on the local host may be difficult to migrate, although it may be possible to migrate a virtual representation (in the form of copy files say), along with the process using them, in order to minimise residual dependency. The problem of migration becomes even more complex when the files are shared by two or more local processes. One must either move the file and all the processes sharing it to the remote host, or move the selected process only and retain a high degree of residual dependency. One of the primary costs associated with dependency of this nature is the cost in communication and processing time as more messages are generated in order to redirect and re-route the "mail" belonging to the migrated process. The problem is further exacerbated if the process is migrated more than once, resulting in residual dependency on each new host. That is, system operations such as memory

management and failure recovery are complicated by the distribution of state information among several nodes for any number of processes. To simplify the management of such problems, a number of design constraints have been imposed on the migration facility of distributed operating systems such as the V-system [Cheriton88] and Sprite [Douglis91b]. Joosen et al [Joosen90] found that none of the distributed operating systems investigated were without residual dependency, and concludes that process migration facilities can only be made efficient by introducing a high degree of residual dependency.

Given the potential costs incurred by a process as a result of migration, Hac [Hac91] introduces the concept of the migratable factor, defined as the ratio of the average transfer time of a process to the local average process response (run) time. In their experiments, the migratable factor of CPU-intensive processes was close to zero; mixed CPU-I/O-intensive processes was *0.9*; and I/O-intensive processes a value of *1.3*. One can argue that the higher value of I/O-intensive processes reflect the level of residual dependency between such processes and their originating hosts. Thus, it is conceivable that a process migration facility could decide to transfer only those processes with a migratable factor less than one or alternatively, as a result of the transfer can obtain a value that is less than or equal to its existing value. However, such a mechanism would be complex, and would make demands on the host's processor and memory resources for computing and maintaining the migratable factor for every active process.

2.6 Distributed Operating Systems

Over the years, operating systems have evolved to manage the hardware, software, and data resources of a computer system to ensure efficient and correct operation.

Within a distributed computing environment the operating system's responsibility for the management of input-output, memory, and processor resource must also encompass the scheduling of multiple processors operating in parallel, and the provision of communication services to processes across the network. According to Cheriton, a distributed operating system is characterised by the cooperation of the kernels in different hosts "to provide a single system abstraction of processes in address spaces communicating using a base set of communication primitives" [Cheriton88]. Similarly, Tanenbaum identifies resource transparency as the hallmark of a truly distributed operating system [Tanenbaum90]. Thus, when a process is created, the single "abstract" operating system makes the decision as to the best place to run it. The other characteristic identified by Tanenbaum is the existence of a single, system-wide file system. There is therefore, "no concept of file transfer, uploading or downloading from servers, or mounting remote file systems." [Tanenbaum90]. One of the major problem that such systems must tackle efficiently is the problem of uniquely identifying objects (such as files and migrated processes) in a dynamic, decentralised environment and associating data and addresses with those names [Shaw88]. As techniques for overcoming these problems are current topics for further research, most commercial operating systems provide a collection of modules to facilitate communication across the network. Further, a dedicated file or name server where object names and addresses are statically determined are commonplace. Such systems fall short of the Tanenbaum ideal for a distributed operating system because the user is, and in many cases needs to be, aware that multiple independent computers exist and must deal with them accordingly.

Most distributed operating systems currently under development have attempted to distribute the functionality of the operating system such that only a minimal set of services which need access to privileged functions or devices are implemented in the

kernel (referred to as a "micro-kernel"). For example, the memory manager, process scheduler, device drivers, and communication protocols are usually contained in the kernel. Other services, such as file, directory, and terminal handling are carried out by processes running in user mode and are directly available from elsewhere within the network. The micro-kernel approach has the advantages of giving designers and users greater flexibility in service provision, extendibility and simplicity in accommodating new services and technologies, and ease of maintenance. However, a greater number of context switches are required to effect the range of services offered by the system. Thus, despite the findings of researchers on projects such as the Amoeba project [Douglis91b], supporting research evidence for the improvement in the quality of service provision through micro-kernel architectures remains rather sketchy, sparse and sometimes unclear.

According to Douglis et al [Douglis91a], the two key issues of distributed operating systems research today relates to shared storage and shared processing power. The primary focus in shared storage research is how to implement a distributed file system that can facilitate the sharing of data objects (such as files) amongst all the processors in a network without degrading the overall system performance or forcing users to worry about the distributed nature of the filing system. In the case of shared processing power, the main focus is to identify techniques that can harness the power of the processors and make it available to individual users so that their applications can benefit through improved response or completion times . Whilst it is the case that most distributed operating systems currently adopt similar techniques for managing shared data, Douglis et al [Douglis91a] perceive the differences between such systems to be their model of computation rather than the operating system's software architecture. For example, systems such as Sprite [Douglis91b], and V [Cheriton88]

use a workstation-centred model of computation compared to the shared processor pool model used by the Amoeba system [Tanenbaum90].

In the workstation-centred model users are associated with individual workstations that can provide sufficient power to meet most of their application's processing requirements. Such a model was justified on the basis that any further advancement in processor technology would be to facilitate workstation users with better and faster graphical user interfaces. In contrast, designers adopting the "processing pool" model were driven by the belief that:

> "..it would be easier to place hundreds of processors in racks in a machine room than distribute those processors equally among each user.." [Douglis91a].

Thus, one is able to make available to an individual user the total processing power of the "pool" rather than the power of any one individual workstation, especially in cases where the process to be executed exceeds the processing capacity of the latter. For example, a user program consisting of n independent compilation units can be compiled using a pool of n processors to produce a potential n-fold speedup of the compilation process. The following sections provide a brief overview of some of the distributed systems under development.

2.6.1 Amoeba

The Amoeba distributed operating system model was developed at the Vrije University of Amsterdam with the aim of understanding and developing techniques for the transparent connection of multiple computers [Levelt92, Tanenbaum90]. Based on the "processor pool" model, the Amoeba architecture consists of: intelligent user terminals, specialised servers, a processor pool consisting of 48 single-board

VME-based computers, and 10 VAX CPUs, and gateways which isolate Amoeba from non-standard protocols used over Wide Area Networks.

The operating system is organised around the client-server model of distribution. Conventional operating system services such as file and directory management services have been removed from the kernel and delegated to server processes. The Amoeba kernel, with only communication, scheduling, and memory management responsibilities can make client requests to the delegated servers. These servers may in turn send client requests to the kernel in its role as both client and server. Remote Procedure Call (RPC), implemented directly by the kernel, is used as the primary implementation mechanism for client-server communication.

The performance of the Amoeba RPC implementation was compared to other systems in terms of delay and bandwidth. In this instance, delay is concerned with the time that has elapsed between a client issuing a request and receiving an acknowledgement to that request, and bandwidth is concerned with the data transfer rate between server and client. Experiments were conducted for the local case where both processes reside on the same machine, and for the remote case where each process ran on separate machines and communicated over an Ethernet network. The researchers found that, when compared to the RPC facility of the SUNOS network operating system implementation, the Amoeba's delay was six times better and its bandwidth twice as good. It was also observed that the operation of the Mach's RPC was worse than most distributed operating systems by a factor of three, and ten times slower than the Amoeba.

Amoeba is transparent both in terms of the location of service provision and the underlying processor configuration used. This enables the operating system to service

user requests dynamically by allocating one or more available processors, depending on the degree of concurrency inherent in the request made, from its pool of available processors. A run server is responsible for organising the pool and will attempt to match the service request (in the form of executable binary files) to the available processors according to their current workload and the memory requirements of the application.

In terms of communicating process groups, an Amoeba process consists of multiple threads of execution that are capable of operating in parallel. However, these process threads share the same address space and communicate through their common area. All threads of a given process must therefore reside on the same host and can only be scheduled as a group by the operating system. As a consequence, the scheduling of the individual threads of a process must be handled by the designer of the process itself.

The migration of a process to a processor is mainly governed by information held in the respective process descriptor such as: class of machines, instruction set, and minimum memory requirements. The migration protocol would operate thus:

1) The local host sends process descriptor to the machine where it will be executed.

2) The memory server fetches the code and data segments from wherever they might reside for the remote host.

3) The kernel of the remote host creates the process threads according to the process descriptor data, and invokes them.

4) Remote host creates and returns a capability, that contains process identification and privilege information, to the initiator.

Whilst the basic primitives exist to support process migration for load balancing purposes, the research team on the Amoeba project were undecided about its merit as implied by the following statement:

> "Whether process migration for load balancing is an essential feature or just another frill is still under discussion" [Tanenbaum92].

The sentiment expressed should be viewed in the context of the two critical factors of any process migration facility for load balancing. Firstly, residual dependency becomes more of a problem the greater the number of migrations endured by a process during its lifetime. Secondly, migration for load balancing requires the local host to negotiate with one or more remote hosts. In this scenario, the client-server model where RPCs are the predominant mode of communication may seriously impair system performance as each sequence of packets must be individually acknowledged. It is therefore not surprising that the researchers, having observed that asynchronous RPC allowed considerable parallelism despite its difficulty to program correctly, advised future designers to avoid such mechanisms.

2.6.2 Sprite

The Sprite operating system was developed primarily to run UNIX applications on a network of workstations [Douglis91a]. Sprite is based on the workstation-centred model of computation where owners are pre-eminent on their own machines. However, each station informs a central coordinator process of its availability to receive work whenever it becomes idle or under-utilised. The availability of a station is determined by the length of time between keyboard or mouse input and the number of the runnable processes. At the time of writing, it was the responsibility of the workstation user to decide which tasks to run remotely using the system state information available to the central coordinator [Douglis91b]. However, should the

owner of the selected workstation require further use of its resources then any identifiable "foreign" process is either run at a lower priority or migrated back to their originating host. Douglis et al [Douglis91b] observed an improvement in response time by a factor of five when performing multiple application builds (using UNIX 'make') for a network of 12 workstations.

Process migration is effected automatically whenever "foreign" processes must be returned to their originator or as the indirect consequence of a user-request to transfer a process. In order to contain the problem of residual dependency, the state information on the migrated process is restricted to the process's originating node although a copy can be made available at the new host via the file service. All signals and messages are sent to the process' home machine from where it may be forwarded. Further, to reduce the time that a process remains in the suspended state during migration, the file server is used to receive any modified pages of the process' address space from both the original and the new host, and all other pages are demand-paged in at the new host. It is anticipated that over time relatively few pages will be modified.

Communication between hosts is facilitated by the RPC mechanism. Thus, using an Ethernet network of SparcStation 1 workstations the overheads due solely to the Sprite migration mechanism (migrating a "null" process) was in the region of 76 milliseconds [Douglis91b]. It is notable that the original design of Sprite consisted of independent monolithic kernels running on each host but only sharing file system services, due to the perceived inefficiencies in micro-kernel technologies.

2.6.3 The V-System

The V distributed system started life as a research project at Stanford University. Unlike the Amoeba processor pool model, V was designed to support a cluster of computer workstations connected by a high performance network. The primary tenet of the design philosophy is the creation of a software backplane in which a small distributed kernel and a set of exchangeable value added service modules (such as file servers) can co-exist [Cheriton88]. The researchers focused on optimising the performance of the communication sub-system. Cheriton et al found that 50% of the message traffic fits into short fixed-length messages [Cheriton88]. Therefore, the performance for short RPCs is improved when a user process places the entire message in the CPU registers and is taken out by the kernel for transmission. Along with other optimising features, their implementation of the transport protocol VMTP in the UNIX kernel exhibited improved performance with an 8ms message return trip time and 1.9Mbps data rate using two MicroVaxII machines over an Ethernet link.

In the load sharing policy described by Stumm et al [Stumm88] each host not only broadcast its current state to all other hosts, but also attempts to maintain a list of potential sites for load sharing. In order to reduce the communication overheads state information is only broadcast whenever there is a significant state change, as indicated by the CPU utilisation measure. In addition, a multicast facility is also available to request and distribute load information as part of the distributed scheduling mechanism. Only newly arrived tasks are nominated for transfer as this reduces the overheads of process migration and the potential level of residual dependency. Further, in the V-system kernel communicating processes are considered to be lightweight as each process does not carry the weight of a separate address space.

In the case of the process migration mechanism of the V kernel, elimination of dependency could only be guaranteed if programmers ensured that their processes used globally managed resources. The actual mechanism makes use of a technique known as *precopying* which transfers the bulk of the process state information prior to the suspension of the process. Once suspended, the complete address space is transferred followed by any pages modified whilst the transfer was in progress. However, precopying imposes additional communication overheads and, according to Singhal et al,

> "..provides an advantage to migrating tasks at a performance cost to those tasks left behind at the sending host and those tasks already residing at the receiving host." [Singhal94].

2.6.4 Mach

The Mach technology was designed at Carnegie-Mellon University as a multiprocessor operating system. The key aspects of its design were functional compatibility with the UNIX technology of the day, and the provision of multi-threaded application support [Acetta86]. It was later incorporated as the base technology for OSF/1. The kernel is object-based, which allows the system to exhibit the personality of a particular operating system. Thus, unlike traditional UNIX, the kernel consists of tasks, threads, and memory objects which can be dynamically reconfigured without the need to rebuild the system. However, unlike the Amoeba, network communication is implemented outside the kernel as user-level servers and consequently has an associated performance penalty.

The kernel schedules the individual threads of a process rather than the process itself as each thread will have its own execution and computation state. Although, it is possible to implement Mach as a micro-kernel architecture, no perceived benefits

were envisaged in a multiprocessor environment. Thus, a generalised process migration facility is also absent. The Mach IPC which facilitates communication between task objects consist of ports and messages. A clever optimisation technique is used to pass messages between processes running on the same machine, namely mapping their address space, thus avoiding the copy operation.

The OSF/1 kernel is derived from the Mach operating system, and makes use of the kernel objects to implement the UNIX operating system personality [OSF93]. It attempts to integrate the operating system and the application programming interface such that the services offered can be used directly by application programs. However, application developers generally, do not work directly with tasks and threads. As a future standard for distributed UNIX systems, it does provide a migration path for existing UNIX applications, whilst harnessing the resources of a distributed system.

2.7 Distributed Application Standards

As distributed systems architecture become more commonplace the emphasis has shifted some way towards the development of applications that can best exploit the characteristics of the underlying architecture. In principal, application programmers, system administrators, and end users would have access to an integrated toolset for building systems that are capable of being ported or operated across heterogeneous platforms. Whilst such standards are to be welcomed, they do provide a "blueprint" for building distributed applications with the technology of today. Therefore, it is even more important that distributed operating systems are designed that can shoulder the responsibility of managing processes and process threads in an efficient manner. Load balancing mechanisms and algorithms will be a key feature of such systems. The following sub-sections presents a brief overview of some key standards.

2.7.1 OSF/DCE

The Open Software Foundation (OSF) proposal for a distributed computing environment, (commonly referred to as DCE), presents a collection of services and tools that support the creation, use, and maintenance of distributed applications in a heterogeneous computing environment [OSF92]. The DCE software architecture is based on the Client-Server model of computation using RPC as the basic mechanism for communication between applications. Figure 2.4 illustrates the DCE Architecture on a typical host and consists of general services, distributed file services, and RPC and process thread mechanisms. Services such as data sharing are implemented through the provision of directory and file servers. Given the heterogeneous nature of the network, communication between the DCE's of different host is achieved through the use of a network protocol that is common to all.

The DCE architecture takes on the role of a distributed operating system by ensuring network transparency for distributed applications running on conventional network operating systems. In instances where such an operating system exists, DCE essentially specifies the name, type, format, and structure of distributed programming facilities, including the libraries that implement the Application Programming Interface (API) and program development tools. A distributed application that makes use of facilities such as RPC and process threads of DCE would have such functions mapped directly onto the components of the underlying operating system. However, for systems without the ability to schedule or manage process threads, the application developer must also carry the responsibility for the management, synchronisation, and prioritisation of individual threads. Further, in cases where DCE threads are made available as a user-space library routine, the capability of the underlying operating

system may only be the identification and scheduling of the complete process rather than individual threads.

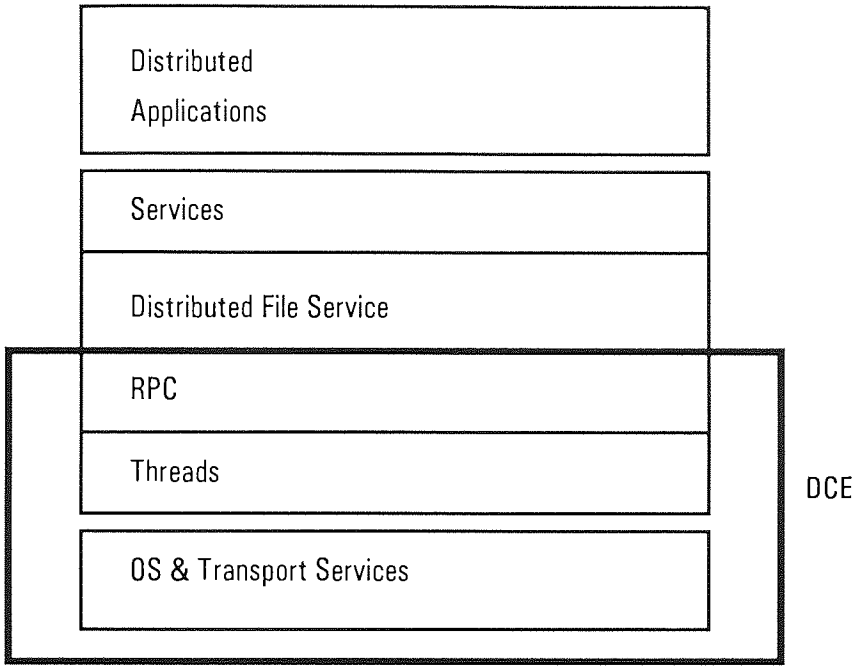


Figure 2.4 The DCE Software Architecture

2.7.2 OMG/CORBA

The Object Management Group (OMG) published its standards for an architecture for object-oriented messaging and referred to it as CORBA (Common Object Request Broker Architecture) [Byron94]. The primary goal was to facilitate distributed object management in a manner that allows objects in a heterogeneous network to communicate transparently. An object may be loosely regarded as the unit of processing defined by the encapsulation of both the computational methods and the data to be used. In addition, an object of any type can request services of other objects via the exchange of messages. The objects managed by CORBA includes "..components of, and output from, operating systems, languages, databases,

networking software, and applications - whether they were built with object technology or not" [Byron94]. In this scenario objects, as units of processing, can vary enormously in their size and complexity. Therefore, one of the primary goals in setting the standards was compatibility with existing applications (where size and behaviour are predictable) with a view to facilitate future object-oriented implementation needs.

The architecture is similar to DCE in terms of its relationship with the native operating system and the toolset provided to application developers. The CORBA structure consist of an Object Request Broker (ORB) kernel, Interface Definition Language, and other APIs. The primary difference is that as an object itself, the ORB and its components can be shared or simultaneously used to build other client objects. CORBA defines standards for name services, the mapping of requesters to providers, the selection and invocation of the appropriate methods for service provision, and its delivery using standard transport protocols such as TCP/IP.

The future of standards such as CORBA is uncertain as there are a number of competing products (or de facto standards) available such as Microsoft's OLE (Object Linking and Embedding), and Architecture Projects Management Ltd's ANSA (Advanced Networked Systems Architecture) [APM93]. However, it may be some time before application developers are able to appreciate and demand the full benefit of object orientations, and vendors can deliver such architectures at a performance premium.

CHAPTER 3

DISTRIBUTED LOAD BALANCING

3.1 POLICIES AND PROCEDURES

In a study conducted by Williams [Williams91] load imbalance is defined to be "the difference between the maximum and minimum numbers of elements per processor compared to the average number of elements per processor". However, in order to correct imbalances in system workload, the local host may need to collect information about the process loads of part or all of the system, decide the most appropriate location to continue execution of a process, and select and transfer a process that would benefit most from execution at that remote location. These tasks entail both communication and processing overheads. Therefore, its policies for collecting and exchanging state information, selecting remote hosts and local processes for the purpose of process migration, should result in low communication overheads and a fast response time.

Researchers such as Lin et al [Lin91], Zhou [Zhou88], and Johnson [Johnson88] have identified three components of any load balancing policy, namely policies for the exchange of state information, the transfer of processes, and the location of recipient processors. Similarly, Goscinski et al [Goscinski90] suggest that a load balancing server should consist of three components to answer the *when*, *which*, and *where* questions of load balancing. Thus, the transfer component would address the question of which process to migrate, the location component answers questions regarding the

destination for migrated processes, and the issue of when to activate load balancing would be governed by the system clock or other specific events.

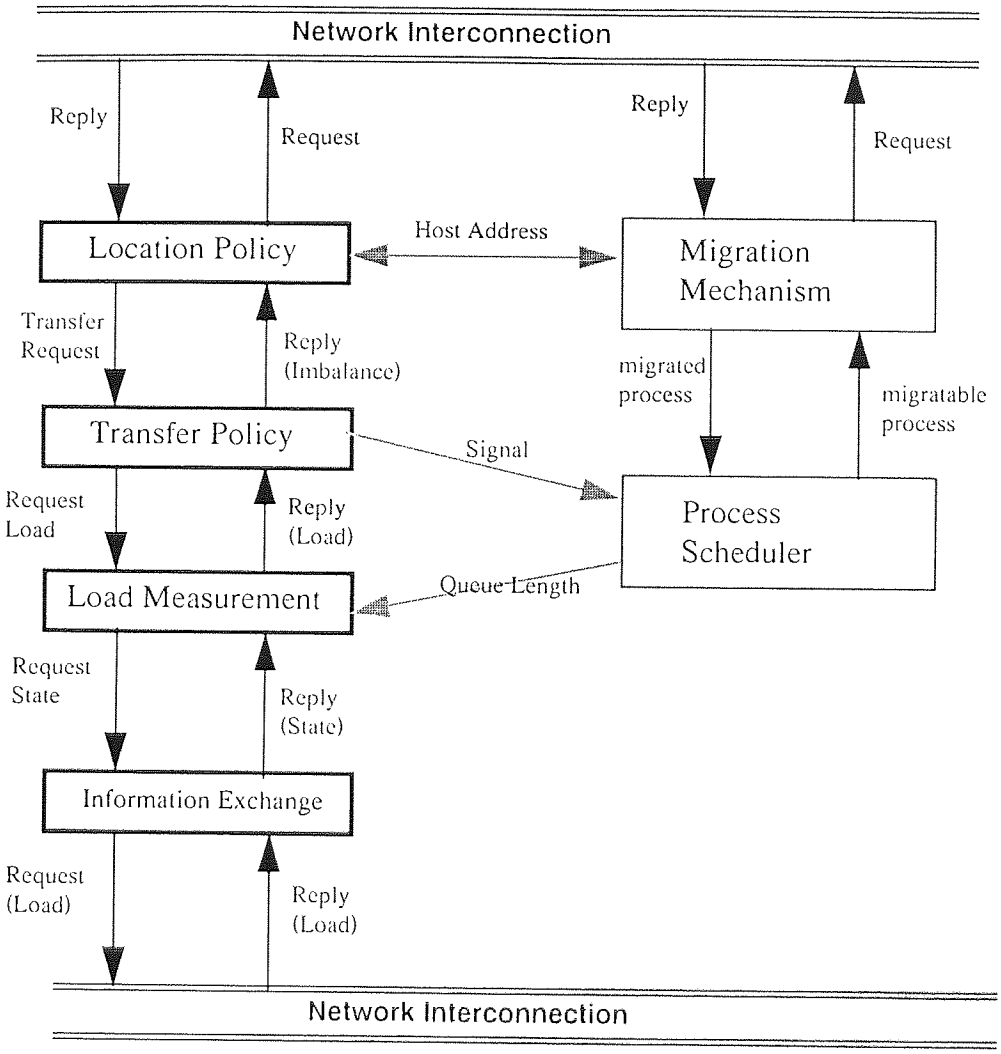


Figure 3.1 Components of a Load Balancing Server

Shown in Figure 3.1 is a more detailed illustration of the load balancing server presented in an earlier Figure 2.2. What is particularly noticeable, and is supported by the definitions presented by Johnson [Johnson88], is that the transfer policy does not make decisions about which process to transfer, since that is considered to be the responsibility of the scheduler, but rather dictates the conditions under which process migration is considered. Thus, the issue of when to activate load balancing is more to

do with the exchange of state rather than process information. For example, in Figure 3.1, the information exchange component requests state information from other nodes in the network. This may in turn be the result of the activities of the load measurement process in its attempt to construct a picture of the global state. The transfer policy component, using local and global state information, will respond by sending a request to the location policy component and signalling the scheduler according to the degree of imbalance detected. The location policy component, having identified a suitable destination for process migration will pass the host address to the migration mechanism to effect the transfer.

In the load balancing policy (referred to as LBC) suggested by Ciciani et al and Lin et al[Lin91], the information exchange and location components are centralised, and the transfer component is partially distributed and partially centralised. One of the processor nodes acts as a central job dispatcher by maintaining a table of process queue lengths for each node. Process migration only occurs on the basis of the state information available and at the request of the central dispatcher to potential senders and receivers [Ciciani88]. Whilst, this strategy requires only partial involvement in the actual transfer of processes between processors, like most centralised load balancing strategies, the system is vulnerable to failure in the processor running the job dispatcher. Therefore, greater reliability can be attained by each processor taking responsibility for effecting and synchronising load balancing activities with its peers. The various policies commonly operated by the components of a load balancing server are discussed in the following sections.

3.1.1 Processor Load Measurement

The monitoring of processor load is an important activity in centralised and distributed systems. The information collected may be used for accounting purposes, load balancing, performance measurement, and configuration management in the short- and long-term. Given the potential impact of these decisions on the service quality of the system, it is essential that the load measurements be accurate and timely. Furthermore, the measure should impose little or no processing or communication overheads. For example, a load measure that is simple to compute, but needs to be sent frequently to all the remote hosts of a large distributed system can impose significant communication overheads. In a loosely-coupled distributed system, each processor will independently compute its local load. Various quantitative and qualitative indicators of processor load are possible along with a variety of methods for computing their values. The simplest load measure is the instantaneous length of resource queues for a given host. The predefined maximum queue lengths would be used to define the capacity of a host processor. The computation of the load is invoked during any process state change such as creation, execution, suspension, and termination. The computation would involve simple increment or decrement operations in accordance with the resource queues concerned. The processing overhead is insignificant as the computation is an integral part of the operating system's resource management operation. However, this measure is inaccurate as it does not take into account processes that are in transit from remote hosts. That is, as migrating processes are not counted on the local host until they arrive, their arrival after the load measurement has been made would make the preceding load reading inaccurate. The time that a process is in transit is likely to be significantly smaller than the load measurement sampling period. Thus, there is no guarantee that the next measurement made would capture such processes in the queues

of the local host. In the study by Barman et al [Barman89], the scheduler of each processor estimated its own load continually, using the formula:

$$Z(t+\Delta t) = Z(t) + a(\Delta t) - s(\Delta t) + xr(\Delta t) - xt(\Delta t) \qquad (3.1)$$

where, $Z(t+\Delta t)$ is the estimated load for the new time frame $(t+\Delta t)$, $a(\Delta t)$ represents newly created local processes, $s(\Delta t)$ is the number of local processes transmitted to the local cpu server, $xr(\Delta t)$ is the number of remote process arrivals, and $xt(\Delta t)$ is the number of processes migrated during the time period Δt. Whilst this method gives a more accurate and stable load measure, by trying to anticipate the creation and movement of processes, it has a number of problems. Firstly, a suitable time frame must be selected which maximises the accuracy of the measurement and minimises the overheads of its computation. Such requirements are often in conflict. Secondly, the method needs to build up knowledge about the system's workload characteristics as light-weight and heavy-weight processes can significantly distort the accuracy of the measure.

In the study by Johnson [Johnson88], the virtual load of a system would be the sum of the local processes, and processes in transit from other nodes. The local host could keep track of its virtual load, by maintaining a count of the total number of processes it has agreed to receive or transmit. The key problem with this method is that under heavy system loadings, it needs a reliable network, resilient communication protocols, and remote hosts that are unlikely to cancel negotiated process agreements. Nevertheless, the virtual load is a more accurate representation of a system's instantaneous load than the length of process queues, and much easier to compute than the measure suggested by Barman et al [Barman89].

The load value should not only be an accurate reflection of current load, but also a means by which one can make reasonable predictions about the future workload [Hatch90]. The research conducted by Ferrari et al [Ferrari87] demonstrated that resource queue lengths, such as CPU, disk, and memory, were the best indicator of current and future system load. Hac et al [Hac90] used the CPU queue length, CPU utilisation, and the number of active jobs to characterise the workload at a given host. State information tables are kept in the form of a load vector for each site. The load vector for host h is computed as follows:

$$w(h) = w(h) + k(h)^2 \tag{3.2}$$

where, w(h) is the length of the load vector, k(h) represents the weighted workload characteristic (CPU queue length , utilisation etc.) for h. Thus, long load vectors would represent hosts that are heavily loaded relative to those with short load vectors.

In the case of light-weight processes, instantaneous load measurements are unreliable and susceptible to rapid change. Therefore, any decisions made on the basis of the load information collected are more likely to be inaccurate and contribute to system instability. An alternative and more accurate load measure is the average system load over a given time period. The difficulty here is in defining a load sampling period that is small enough to capture significant changes in system load, and yet large enough to allow its accurate prediction for succeeding time periods. UNIX systems provide a load average, based on the length of the process run queue, over a five minute period. Given the length of the sampling period, the measure can only provide an indicator to system activity from one five minute period to the next, but is very insensitive to changes in system load within that period. Krueger et al [Krueger84] suggest that system load should be "averaged over a period at least as long as the time

necessary to migrate an average process". Again, the time to migrate a process will very much depend on the system load, the routing possibilities of each intermediate host, and the scheduling discipline employed by a local host for processing messages and processes from their respective queues. The study of Ferrari et al [Ferrari87] identified a four second average subjected to short-term smoothing allowed more accurate decision-making with a resultant improvement in overall system performance.

The length of the message queue for a given host can be a good indicator of its load. However, this is a less reliable measure as it depends on the degree of residual dependency on the local host. That is, cooperating processes, that have been partitioned through migration across the network will continue to generate messages to the local host for processes of that set. Thus, the length of the queue does not reflect the process load of the host. Further, the message queue of a local host in a point-to-point distributed system architecture may contain processes that are waiting to be routed to their destinations.

Hatch et al [Hatch90] examined various load indices for load sharing in a distributed system with differing processor speeds. The researchers concluded that a relative load index based upon the resource queue lengths, and processor speed gave a more accurate characterisation of processor load and resulted in the greatest reduction in average job response times.

3.1.2 Information Exchange Policy

To obtain the aggregate load for a distributed system, the load of each host must be determined and collected. The collection of this information is known as the load balancing information exchange policy. The policy may be implemented in one of two ways. In the first approach, a host could actively seek the load of its remote hosts. That is, each time the local host needs the information, it will broadcast or send individual requests for it, and the receiving hosts will reply in kind with information about its current system load. As load information is only requested when needed, the policy keeps to a minimum the communication overheads resulting from the exchange of state information. Alternatively, each host may volunteer this information by broadcasting to all its remote hosts at periodic time intervals or on the occurrence of specific events such as a change in the systems load. The effectiveness of this policy will primarily be determined by the frequency of events and the chosen time intervals for information exchange. Frequent information exchange provides each recipient host with relatively accurate information regarding system load, but results in a greater volume of messages on the network. The use of flexible time intervals would enable less frequent information exchange at low loadings, and more frequent exchange at high loadings. Hac et al [Hac90] makes use of a flexible exchange policy where the interval for information exchange depends on the load of each host. For example, information can be exchanged less often as long as the load is balanced, and on every occurrence of a load state change. The local host receiving the load packets of its peers, can either use the information to update a locally held load table for all the hosts in the system, or compute a global system load value to be used in any present or future balancing policy. However, the overhead in attempting to identify and predict the future load state, in order to set an appropriate interval for information exchange, may outweigh the advantage of such a policy.

Both strategies for the exchange of information become problematic under extreme system loads and such policies may even accentuate the communication overheads. For example, under heavy system load, information is exchanged more frequently, but with less accuracy, because the load state is changing even before any decision can be made with the data received. Further, there is a greater likelihood of messages being lost or discarded as a result of overflowing message buffers, a saturated communication medium, or "reply" responses being timed out. It may be the case that the load state of systems with long running processes changes less frequently than those with lightweight processes, and the problem of information exchange under such conditions is one of degree. Nevertheless, the problem remains of ensuring that the information exchanged is timely and accurate.

Most bus-based LANs have an efficient implementation of a broadcast mechanism such that message delivery time to all hosts in the network is generally uniform [Blake92]. However, under extreme loads the incidence of message collision and their avoidance can further exacerbate the communication overheads. Given the communication overheads of a fully distributed system various alternative proposals have been suggested for reducing this overhead. Some researchers have suggested a semi-distributed scheme such that certain hosts are nominated for the collection of state information from processors within the network, and are given the responsibility for computing the system load, and re-broadcasting this value to all host processors [Mittal89, Ahmad90]. The main problem with this scheme is its reliance on centralisation of the information exchange policy and its vulnerability to system failure. Alternatively, the design of an efficient multicast system yielding similar performance to the broadcast mechanism retains the fully distributed scheme, but would allow each host to collect partial state information from processors up to a maximum distance from the requesting host. The issue is whether the decisions based

on partial information about the load state is just as effective as those using information that is more complete [Ali92].

3.1.3 Transfer Policy

According to Zhou, "the transfer policy determines the eligibility of a job for load balancing based on the job and the loads of the host" [Zhou88]. That is, the transfer policy determines the process to be migrated and the circumstances in which it takes place. However, Johnson [Johnson88] regards the role of the transfer policy component to be primarily concerned with the conditions under which process migration should be considered. This view is supported by Lin et al [Lin91] who states:

> "The transfer rule is used to determine when to initiate an attempt to transfer a job and whether or not to transfer a job" [Lin91].

Thus, the algorithm's complexity is reduced by delegating the responsibility for selecting a migratable process to one or more schedulers of the local host.

Threshold values are commonly used to flag the conditions under which migration should take place [Eager86]. A local host can be considered to be overloaded, underloaded, or normally loaded depending on its current workload relative to the load threshold value. If the host is not handling a normal load, then an attempt should be made to redress this load imbalance by making better use of the remote hosts. A simple transfer policy consists of a fixed threshold value derived from extensive system traces, modelling, and simulation. Researchers have found that a fixed threshold value of up to two processes yielded a better system performance than much larger values [Johnson88, Zhou88, Eager86]. The advantage of this strategy is its ease of implementation and operation as the policy is primarily concerned with the load of the local host. No knowledge of the load of remote hosts, and the subsequent

average system load, is required to operate the policy. However, there will be periods when all hosts are in an underloaded or overloaded state. In such cases, the fixed threshold policy would result in futile attempts being made to balance the workload. Furthermore, it is even more important than the transfer policy is driven by the load of the local host relative to that of its remote peers. The global average policy suggested by Johnson [Johnson89a] attempts to ensure that each host maintains its load within range of the average system load. If a local host finds its load to be out of range then procedures will be set in motion to remove the imbalance in the system.

The transfer policy can be activated periodically or on the occurrence of a change in load state resulting from activities such as process creation or deletion. In the model developed by Johnson [Johnson88], the global average policy was activated every 200 milliseconds. This period of activation was found to give the best response times for processes with an average execution time of one second. In transferring a process to a remote node, one must ensure that: the process does not continue to migrate aimlessly through the network without ever being executed at any of the nodes to which it is sent. The process should continue to execute at its new location with the same functionality as if the transfer never occurred. Dealing adequately with these problems may place additional costs on systems which employ load balancing.

3.1.4 Location Policy

Once the decision has been taken to initiate a process transfer, the load balancing facility must locate and negotiate with potential remote hosts to be the source of or destination for a migratable process. The complexity of the location policy is dependent on whether the requesting local host maintains, and updates regularly, state information about its remote hosts. Pertinent and current state information, resulting from a timely information exchange policy, would simplify the location policy to the

point of merely selecting the most suitable location address from a local state table. In the algorithm proposed by Hac et al [Hac90], the state information tables provide an easy-to-use reference for the location policy. The site with the longest load vector would indicate the worst location, and the shortest the ideal choice. If the site with the shortest load vector is unwilling to accept the process then the next best sites will be chosen. Failure to find a suitable location may result in the termination of the algorithm.

In load balancing strategies that maintain little or no state information, the location policy must poll potential hosts in order to identify suitable destinations. Having received a number of replies, it can either select the first underloaded host, or wait for all hosts to reply before deciding on the host with the best location, in terms of distance as well as measured workload, for undertaking process migration. Under very heavy or light system load a requesting host which adopts a policy that is based on collecting all replies before selecting potential destinations is more likely to lose those respondents with acceptable conditions for migration. This is primarily because the state of the selected location may well have changed even before the migration mechanism is effected, resulting in the subsequent rejection of a potentially migratable process.

3.1.5 Frequency Of Activation

The execution of the load balancing algorithm represents an additional performance overhead. This is dependent on the frequency of its activation and the complexity of the processing performed. A computation-intensive or message-intensive algorithm will impose a greater overhead than those requiring simple arithmetic computations, little or no inter-processor messaging, and with occasional activation.

Load balancing may be invoked at the point of process scheduling or at regular time intervals which may or may not coincide with the operation of the scheduler. Ideally, the algorithm should be invoked whenever its execution is likely to result in the successful migration of a local process to a remote host. For example, whenever a process is created by the High-Level Scheduler the load balancing algorithm would be invoked by the low-level Scheduler. In this case, known as non-preemptive scheduling, processes that have started their execution on the local host are not eligible for migration. The intention is to begin and complete execution of the newly created process earlier than would be possible on the local host, and also avoid the extra complexity of administering the run-time environment for the migrated process. However, if the algorithm fails to find a remote host, the local host will remain in an overloaded state until one or more local processes complete. However, whilst the execution of the algorithm is kept to a minimum, the creation of subsequent processes, and the continued failure to find a remote host for these new processes may result in the local host being in an overloaded state for an indefinite period.

In cases where load balancing is invoked at regular time intervals, the interval must be such that the overheads do not outweigh the benefits of a successful migration. For example, frequent execution of the algorithm could result in more successful migrations but a reduced throughput on the local host. The primary disadvantage of this form of activation is its expense as it must be supported by pre-emptive process scheduling. It is generally the case that such scheduling carries enormous overheads as it may be necessary to migrate the complete process environment including registers and workspace areas [Singhal94].

3.2 Algorithmic Classification

The algorithms used for implementing a load balancing policy vary in their level of complexity and sophistication. Wah et al [Wah85] used the term "task bidding" to describe the process by which overloaded and underloaded hosts negotiate the transfer of processes. However, today it is more common to refer to this activity, in the case of loosely-coupled systems, as having sender-initiated or receiver-initiated properties. The sender-initiated policy is one in which the overloaded host invokes load balancing in an attempt to find a peer processor that is able to accept its excess workload. It is therefore the responsibility of the overloaded host to identify amongst its peers potential (underloaded) recipients of the excess workload. Zhou [Zhou88] refers to policies of this type as source-initiative algorithms, because the overloaded host must take the initiative to transfer excess workload. In contrast, a receiver-initiated policy is characterised by the underloaded host initiating the load balancing procedure to seek out peer processors from which additional work may be obtained. Such policies are referred to by Zhou [Zhou88] as server-initiative algorithms as it is the underloaded hosts that takes the initiative to be actively engaged in finding amongst its peers, potential (overloaded) senders of any excess workload.

The success or failure of policies in either category is governed by the ratio of the amount of message traffic created by a policy against the number of process migrations that results from that activity. The debate is not so much to do with whether sender-initiated policies are better than receiver-initiated policies, but is more to do with the amount of state information that needs to be collected and maintained, in order to identify overloaded or underloaded processors, and thus increase the likelihood of successful process migrations.

The researcher Philp [Philp90] views the complexity of load balancing algorithms by the amount of communication that results when attempting to identify the least loaded node in the network . Thus, load balancing algorithms can vary from those which collect little or no state information from peer nodes to those that base their location decision on state information gathered from all nodes in the network.

In the following sub-sections, load balancing algorithms are classified according to the state information that must be collected and maintained before policy decisions can be made.

3.2.1 Simple Static Algorithms

A load balancing algorithm is said to be static if it collects no state information. That is, a local host employing such an algorithm will be characterised by the absence of dialogue with other hosts to establish the system state, and also the maintenance of state tables for remote hosts who may or may not broadcast their current state. Therefore, given the absence of state information, the location policy of the local host is primarily governed by probabilistic balancing decisions, without regard for the current state of the system. A very simple processor load measure is used by the algorithm, such as the number of processes resident on the host concerned. Typically, a predefined load threshold level is set and kept constant throughout the operational lifetime of the system. This threshold value may have been derived from simulation studies and system load traces. But, once the load balancing parameters have been established, they become unalterable at run-time.

The balancing algorithm is only effected whenever the load rises above (or drops below) the threshold. Once effected, the algorithm will decide, without interrogating

the intended destination, which process should be migrated to which node. The selection of a process and its destination host may be randomly determined, or based on predefined rules such as migrating the last locally created process to a destination host whose address is the first entry in the locally held remote host address table. The process is then migrated to the destination, which has no choice but to accept it. The policy for selecting a process to migrate must ensure that on migration, any remote host receiving the process does not continue the aimless and indefinite migration of the same process. In this study, the policy adopted was to allow only one migration per process. In Ahmad et al [Ahmad90] a process is allowed to make many migrations until a suitable node is found or a predefined process migration limit is exceeded. Eager et al [Eager86] adopted the method of specifying a maximum number of migrations per process. Alternatively, a process could be migrated a variable number of times during its lifetime providing it was executed at least once on each new host before becoming eligible for any future migrations.

The random load balancing policy is characteristic of this class of algorithms and is commonly cited as the lowest common denominator for all practical load balancing algorithms [Finkel90]. Such algorithms are attractive because the communication overheads are minimal but they are generally incapable of adjusting to the real imbalances of "peaks" and "troughs" of system workload over a period of time. Further, Srimani et al [Srimani92] adapts the policy for real-time processes with execution deadlines. In this environment a migrated process will be lost if the receiving host cannot start its execution immediately. However, before any such jobs are lost, the policy attempts to ensure their execution by issuing pre-emptive priorities.

3.2.2 Simple Dynamic Algorithms

These algorithms are characterised by the collection of small amounts of state information. Similar to the static algorithms, a simple processor load measure is also used, and a predefined load threshold set. Unlike the static algorithms, its invocation will initiate a dialogue between the overloaded (or underloaded) processor and a fixed number of potential recipients. Refusal by all potential recipients as a result of their own workload, will result in the migratable process being executed locally. A willingness to accept a process by any one of the interrogated processes will result in the migration of a selected process to that node. The algorithm proposed by Ahmad et al [Ahmad90] is characterised by nodes collecting state information from their most immediate neighbours, and the migration of processes to the neighbour with the lowest load. Other researchers have used sender-initiated threshold algorithms where state collection is activated on the arrival of a local task [Mirchandaney89, Johnson88]. Remote hosts are polled at random and the first to respond positively is selected. In contrast, an alternative algorithm probes a fixed number of remote hosts, awaits all replies, and select the host with the shortest process queue[Finkel90, Eager86, Philp90].

In the model used by Dikshit et al [Dikshit89] the parameters to the threshold policy includes a random set of hosts. However, a remote host that receives a probe message determines if transferring a job to the host would make its load greater than the threshold. In the real-time environment of Srimani et al [Srimani92] a probing node that is unable to find a suitable node sends the newly created process as a priority job to the last node probed. Equivalent receiver-initiated algorithms have also been developed. Such algorithms, referred to as the "reverse" policy, are activated every time a process completes [Mirchandaney89]. In the study conducted by Lin et al [Lin91a] an underloaded host will randomly probe remote sites to find those with job

queues above a fixed threshold. The located sites are subsequently probed and, those whose job queues continue to be above the threshold, will be sent requests for work.

3.2.3 Adaptive Algorithms

These algorithms are characterised by a variable threshold level governed by the current state of the system, and may combine two or more load balancing policies. For example, if a processor is overloaded, it will inform a fixed number of processors of its desire to off load work, and subsequently enter negotiation with one or more positive respondents regarding their status. Should no node respond positively, the initiator may presume that the threshold level is too low (therefore everyone is overloaded), and may adjust the level upwards. Likewise, should a processor's load fall below the threshold, then it may well inform other processors in the network of its willingness to receive work.

More complex adaptive algorithms may change their policy according to the dynamic workload characteristics of the system. For example, at light system loads, the strategy may be to adopt a sender-initiated load balancing policy. However, should the system become heavily loaded, the sender-initiated policy would be abandoned and a receiver-initiated approach adopted. Such algorithms can be extremely sophisticated, requiring complex communications protocols. The "symmetric" algorithms developed in [Lin91, Mirchandaney89] uses both sender and receiver-initiated strategies. In the implementations proposed load balancing is activated when the local load becomes more than the sender-threshold or less than the receiver-threshold.

The algorithm proposed by Dikshit et al [Dikshit89] is activated at regular intervals and computes the difference between its own load and the least busy host. A

difference greater than an acceptable bias initiates a job transfer. However, a time window, maintained by the load balancer which keep time histories for process transfers made, is used to avoid overloading the least loaded processor. An alternative approach, known as "broadcast idle" is used to resolve the problem of saturating underloaded hosts. The policy operates such that a local host whose load is more than one must wait for a time period equal to the dimension of the network divided by the local load. Thus, the more heavily loaded a host is, the less will be the duration of its wait . A broadcast 'reservation' can then be made to the idle host if one has not been made already.

In the State Collection Algorithm of Ammar et al [Ammar88] each node keeps a record of the global state in terms of the process queue length for each node. However, unlike the Global policy in which state information is broadcast periodically, their algorithm only collects state information as a by-product of communication between two or more processes. For example, whenever a process is migrated, the load of the sending host is packaged with the process itself. Likewise, the termination of a process remotely will result in packing the load of the remote host with the exit message sent to the originating host.

3.3 System Performance

The most common finding of the majority of research conducted thus far, is the superior performance exhibited by distributed systems with a load balancing policy over those without any policy at all. This was particularly evident in the case of static balancing algorithms, which one might have expected to only yield marginal performance improvements due to its complete lack of system state information on which to base its balancing decisions.

Researchers such as [Mirchandaney89, Hac91, Zhou88, [Simpson94] found that substantial improvements in system performance can be achieved over a wide range of workload intensities, even for very simple load balancing algorithms. The improvement in overall system performance at low system loading, although positive, was generally negligible. At low system loadings, the overheads in employing load balancing negates the benefits of a more balanced workload. Performance improvements are particularly marked in cases of moderate to heavy system loadings. [Zhou88] showed that load balancing for moderate system loads, reduced the average response time of a task by as much as 60%, and also made it much more predictable.

The effect of load balancing overheads on system performance is generally accepted as a limiting factor. Hac et al [Hac91] found that the delay caused by load balancing was in the region of 5 to 15 seconds, depending on the number of processes, and resulted in a five percent degradation of system performance. However, this overhead was found to be fairly consistent across system workloads consisting of a range of job mix.

Some researchers have explored the impact of network speed on the overall performance of distributed load balancing systems [Mirchandaney89, Ali92]. Generally, the slower the network speed becomes as a result of the physical medium of transfer, and the protocols effected during the transfer, then the smaller the benefits of load balancing. Ali et al [Ali92] found that the performance does deteriorate at high system loads, the greater the distance between communicating processors. This was due to job congestion in the network arising from the greater incidence of process migration. However, Mirchandaney et al [Mirchandaney89] found that, despite the deterioration in performance, their experiments demonstrated that load sharing

remained a worthwhile task, even when delays in the network were in excess of 10 times the run time of a migrated process.

The simulation study conducted by Blake [Blake92] investigated the performance of a variety of load balancing algorithms for application processes that were assumed to be highly independent. In their study, the system was saturated with processes, and the effectiveness of the algorithm was measured in terms of its ability to redistribute the workload such that an overall improvement in system performance resulted. Results were collected for an Ethernet bus-based architecture with 10 and 100 processors respectively. In the case where scheduling and migration overheads are negligible, Blake found that:

a) the no load balancing and random scheduling policy were ineffective for systems with moderate system loads;

b) the effectiveness of the sender-initiated threshold policy increased the greater the system load;

c) the receiver-initiated threshold policy strayed furthest away from the optimal at low system loads, but formulated near optimal schedules the greater the load.

A centralised scheduler, that collects state information at negligible costs constructed optimal schedules at all times [Blake92]. In the case where significant costs were attributed to task migration and the execution of the scheduler, Blake found the performance of the centralised scheduler degraded. In contrast, the receiver-initiated threshold policy was found to give the best performance overall. Blake therefore concluded that the receiver-initiated policy was the scheduling method of choice for both a 10 and 100 processor system [Blake92].

Whilst these findings are in line with the results of other researchers in the field, the model used by Blake omitted a number of important parameters. Firstly, the assumption is made that tasks communicate infrequently, and this directly implies a light load on the interconnection network making bus contention an insignificant issue. Secondly, Blake makes the presumption that bus-based interconnection is generally characterised by uniform message delivery time, resulting in the scalability of the scheduling algorithms for larger processor networks. Under moderate system load these assumptions are valid, but the load on most systems is highly variable. Therefore, what is of particular interest is the behaviour of such algorithms across a range of loads and, more importantly, their scalability in such cases.

It may be argued that the above results are not surprising for independent, CPU-intensive processes with little or no communication with other processes. Joosen et al [Joosen90], and Eager et al [Eager88] investigated the impact on system performance of dependency between a migrated process and the host from which it was migrated and concluded that remote execution is less expensive and preferable to migration even in cases where load balancing is advantageous. The above observation is not an argument against load balancing, but rather the unsuitability of process migration as the basic mechanism for load balancing.

However, the study of Hac et al [Hac91] examined the impact of a decentralised load balancing algorithm on system performance for a variety of job mixes ranging from CPU-intensive to I/O-intensive processes. Generally, they found the improvement of system performance to be proportional to the number of unloaded processors in the network across a range of job mixes. For example, in a network where only 25% of the processor nodes are overloaded, the results of Hac et al [Hac91] show that there was a threefold improvement in the mean response time of CPU-intensive processes in

the system which used dynamic load balancing. Furthermore, the mean response time for I/O-intensive processes showed a 25% to 45% improvement on systems without load balancing. Similarly, experiments conducted on a network in which 75% of the processor nodes were overloaded, returned a response time improvement of 40% for CPU-intensive processes for the load balancing case. However, a performance decrease of between two and eight percent was exhibited for I/O-intensive processes on systems with load balancing principally because of the transfer of very large files between local and remote processors. This led Hac et al to conclude that:

> "The overhead caused by the execution of the decentralised load balancing algorithm is not significant in comparison with the mean response time. The overhead caused by the transfer of large files and large processes is significant." [Hac92].

The work of [Johnson88] examined the impact of load balancing in a communicating process group model. In this study, load balancing was also found to be worthwhile at or above moderate system loadings. Other researchers have considered load balancing in a heterogeneous processor environment and have demonstrated similar findings. The study by [Harinarayan91] examined a range of limited access network topologies, where access to server nodes by one or more source nodes in the network was restricted according to locality. The results of this study demonstrated that "different interconnection patterns" gave similar performance measurements.

Given the above findings, recent research effort has endeavoured to identify the "perfect" or optimal balancing algorithm. Such an algorithm should perform well across a wide range of system loads, workload mixes, and configurations, achieving resource utilisation at or near the upper reaches of the performance boundary defined by the number of nodes in the network. More recently the primary issue has been whether more sophisticated algorithms are the means by which "perfection" in

performance can be attained compared to the less sophisticated group of algorithms. One might expect balancing algorithms which make use of current system state information to perform better than static algorithms depending on the completeness of the information collected. Thus, the more complete the state information, the better the location decisions, and thus the overall performance.

Eager et al [Eager86] argues that the optimal level of algorithmic complexity falls in the simple dynamic range. Further, Eager et al [Eager86] states that "..simple adaptive load sharing is of considerable practical value and that there is no firm evidence that the potential costs of collecting and using extensive state information are justified by the potential benefits". This latter observation is supported by Johnson [Johnson88] who found that in an environment consisting of short, cpu-intensive, and non-communicating processes, the more complex global average algorithm was unable to produce any further improvement over the simple dynamic algorithm represented by the threshold policy across a range of system load. In the case of a lightly loaded system one could envisage the situation where the communication cost involved in collecting information about the system state is significant relative to the execution time of a potential migratable process. Further, there is a greater likelihood of an overloaded processor finding a potential recipient without the need to collect a significant amount of state information. Similarly, in the case of heavy system loading, one might expect an overloaded processor to receive a greater number of rejections as most processors are generally overloaded themselves. Thus, the greater the amount of global state information available, the better the location decision, with a consequent reduction in the possibility of rejection. However, Philp [Philp90] argues that at high loadings, even a perfect balancing algorithm would not show increased performance because the additional state information collected by complex policies are counterproductive. The failure of

complex algorithms to deliver improved performance was explained by the increasing likelihood that other processors in the system have and are acting on the same information. For example, an underloaded processor in a network of mainly overloaded processors may well find itself to be a potential recipient of excess process loads from the other processors that have access to its state information. Consequently, instability will arise as the overloaded processors overwhelm the least-loaded processor with migrating processes.

In cases of high system loadings, the fact that demand for additional processing capacity outstrips the availability of least-loaded locations creates a requirement for additional mechanisms to avoid potential instability. For example, what may have been a relatively simple protocol for migrating a process could be extended to encompass a "Request, Acknowledge, Send, Acknowledge" sequence. Likewise, the processor load measure could be extended to also cover processes that are in transit to or from the node concerned. However, these mechanisms impose additional complexity and cost on an already complex algorithm. Philp [Philp90] argues that irrespective of such mechanisms, the problem of delayed information is one that affects complex algorithms the most. The problem is that such algorithms work well when the state information received is an accurate reflection of a node's state at the time the information was received and at the time that it was acted upon. However, a processor has no real way of knowing when, where from, and how many processes will be resident within the time frame between the announcement of its state and the action on that state information. Thus, the information sent out is often imprecise and incomplete. Philp [Philp90] therefore suggests that the best strategy for complex algorithms is to make decisions on information supplied by a small collection of respondents.

The study by Ali et al [Ali92] examined load balancing on a range of heterogeneous distributed system network topologies using an algorithm which limits the distance that a migratable process can travel. The distributed system would be split into a number of partitions of a given size. The size of the partition then becomes a parameter to control the flow of jobs through the communication network. That is, the partition to which a node belongs will dictate whether they are allowed to exchange processes. Thus, load balancing is limited to the nodes within a given partition, and, whenever a major state change occurs, each node should send its status to all the nodes in its partition. The load balancing technique is such that each node will use the received state information to calculate the difference between its load and the expected load of every node in the partition. Furthermore, "a node may choose not to send any job for remote processing if the estimated improvement in performance is below a certain threshold" [Ali92].

Ali et al [Ali92] found that the average utilisation in systems with and without load balancing was around 80% and 66% respectively. Furthermore, partitions with broadcast distance equal to two were found to give the best performance for all configurations studied. The greater the size of the partition, the greater the number of nodes that can communicate and share workload. However, greater partition size may result in less accurate information being exchanged due to communication delays. Ali et al [Ali92] therefore, conclude that "the best performance can be reached when the broadcast distance is around half the diameter of the graph that represents the interconnection network".

This particular study was limited in a number of ways. Firstly, a distributed system with only eight nodes was examined. The question remains as to whether the performance of the algorithm studied can be scaled given networks with a greater

number of nodes, and partition sizes. Secondly, the system load studied was confined to heavy system loads. Again, how the algorithm performs relative to other balancing algorithms across a range of system workloads and network sizes should be explored. This is particularly important as the proposed algorithm is fairly complex compared to others. For example, it attempts to predict system load and determine the cost-effectiveness of its process migration decisions.

Ahmad et al [Ahmad90] argues that a fully distributed load balancing scheme, in which all nodes collect state information about every other node, presents practical problems for large systems consisting of hundreds or thousands of processors. He therefore, proposes a semi-distributed load balancing scheme to overcome the increased communication overhead of fully-distributed schemes. The semi-distributed approach used was based on an interconnection structure partitioned into independent symmetric regions, where each region has a special processor known as the scheduler. The scheduler maintains state information for all the processor nodes within its allotted region. Load balancing operates at two levels: at the high level the schedulers cooperate in an attempt to balance workload across regions; whilst at the low level, an individual scheduler balances the workload amongst the nodes of a particular region. The resulting task migrations are carried out between the scheduler and its regional nodes, and the schedulers for other regions. The threshold level for inter-regional migration was one process whereas for intra-regional it was three processes.

Ahmad et al [Ahmad90] compared fully distributed load balancing schemes for multiprocessor systems with up to 1024 processors and found that for very large distributed systems, both semi and fully distributed load balancing scheme yielded a significant improvement in response time over the no load balancing case under low,

medium, and high system loads. At loadings of 90%, the fully distributed and semi-distributed schemes showed performance improvements of 62% and 82% respectively. Further, the semi-distributed scheme outperformed the fully distributed load balancer across both the system load range and size. The division of the system into regions consisting of 176 nodes increased the likelihood of finding an idle processor locally at low system loads. At high system load intra-region migration predominates. In both, the messaging overhead is lower compared to the fully distributed scheme. In the fully distributed scheme proposed by Ni et al [Ni85] each processor maintains the most recent state information of all the other processors in a broadcasting network but only that of its most immediate neighbours in the case of a point-to-point network. The main problem is that the former introduces substantial communication overheads whilst the latter limits the benefits achievable through load balancing to the most immediate neighbours. In the study of Ozden et al [Ozden93] the centralised load sharing algorithm would not scale to thousands of workstations, whilst large fully distributed systems yielded poorer performance. Therefore, the researchers proposed a semi-distributed algorithm, referred to as the *distributed clustering load sharing algorithm*, demonstrating good scalability properties. The key facet of the algorithm is that clusters of processors are formed where each cluster has a processor responsible for maintaining state information regarding the nodes in the cluster and, in order to locate resources, limited information about other clusters. Thus, the overheads of state updates are limited to a well-defined set of managers [Ozden93].

An alternative semi-distributed strategy adopted by Ryou et al [Ryou93] organised processors into a logical tree structure where the root node of each subtree acts as a central controller for the processors in its two sub-trees. Each host is assigned a *sending* and *receiving* set such that load information is only exchanged between

processors that are members of their reciprocating sets. Thus, "the sending set of a node contains all of its ancestors, and the receiving set contains all of its offsprings" [Ryou93]. Ryou et al demonstrated that the number of message required for each state update can be reduced to the square root of the number of processors in the system providing the sets are balanced (or of the same size). Further, the researchers found that there was no compromise in performance as the load balancing scheme with balanced sets outperformed other schemes where a greater amount of state information is collected.

Whilst the schemes suggested by Ahmad et al [Ahmad90], Ozden et al [Ozden93], and Ryou et al [Ryou93], gave a better overall performance by reducing communication overheads and making more accurate placement decisions than a fully distributed scheme, it does rely on special processors to collect state information and schedule tasks. Further, it is not only possible for a fully distributed scheme to generate good performance results and scalability properties, as detailed in this study, but to do so using simple load balancing policies. The study by Ali et al [Ali92] proposed a fully distributed load sharing policy that was based on simple asymmetric partitions arising from the distance between nodes. The researchers were also able to show performance improvements over a load sharing scheme in which all nodes collected state data from every other node [Ali92]. However, the study of Ahmad et al [Ahmad90] did not investigate the performance of the schemes proposed for large system configurations.

On the basis of the published findings of many researchers it would appear that the most effective policies are likely to be those in the simple dynamic range because they are based on the assumption that "most nodes will find their own distinct lightly-loaded partner to migrate jobs to" over a period of time. The results produced and

documented in the later chapters of this thesis, confirms such findings. The study by Harinarayan et al [Harinarayan91] of load balancing in a network of server and client nodes found that greater accessibility between nodes had the greatest impact on performance. This accessibility meant that, irrespective of the type of algorithm, performance was constrained during periods of high system loads by the ripple effect of poor load balancing decisions based on inaccurate information. Their measure of diversity is the total number of server nodes accessible to a client node which is not accessible to neighbouring clients. Thus, at high system loading, increasing diversity should result in improved performance. Harinarayan et al [Harinarayan91] conclude however, that "once there is some load sharing in the system, increasing diversity really does not help in improving performance as much as increasing the number of resources a source has access to".

Whilst studies such as Eager et al [Eager86], and Philp [Philp90] reaffirm the failings of complex algorithms in improving the overall performance of distributed systems characterised by light-weight processes, the results documented in this thesis, and alluded to in the Johnson [Johnson88] model show that complex algorithms can perform better in an environment consisting primarily of "heavy-weight" processes. Furthermore, in the case of a light-weight processing environment, if performance was to be judged by the general stability and balance of workload across all nodes, the success of such algorithms are undeniable.

CHAPTER 4

MODEL DESIGN AND IMPLEMENTATION

4.1 SYSTEM MODEL

In the development of a distributed system simulation model, there were three key requirements considered important to this study. These are: flexibility in the representation of a variety of system configurations; speed and efficiency in the production of statistical data and results for analysis; and comparable accuracy in modelling system behaviour when validated against other systems and models. Mathematical queuing models provide the opportunity to explore load sharing policies for very large network topologies. The models are generally characterised by speed and efficiency in the generation of analytical results for such networks. However, many simplifying assumptions are often made about communication patterns between processes and processors, workload characteristics, and the load balancing activity to make the mathematics tractable. For example, a common set of assumptions made include: negligible message-passing overheads; guaranteed message delivery; and tightly-coupled architectures where centralised process scheduling is effected with or without circuit-switched routing. Further refinements of such models often incorporate probabilistic reasoning about network, and process behaviour. Whilst mathematical models for distributed systems can act as a general yardstick for measuring the effectiveness of load balancing activity at a macroscopic level, their ability to provide insight into the dynamic relationship between the algorithms, network activity, and workload characteristics, is limited.

The two other approaches considered for modelling a loosely-coupled distributed system were a physical model, and a computer simulation model. In contrast to mathematical (stochastic) models a physical system can encapsulate with greater accuracy and realism the behavioural dynamics of load balancing in a loosely-coupled distributed system. However, the capital cost and technical constraints of building a LAN model of loosely-coupled workstations with distributed operating system kernels, would severely limit the size and diameter of the network that could be investigated. An alternative strategy would be to link a large number of cheap processors, such as the Inmos transputer chip, together. The transputer, with on-board memory and communication channels, and the ability to link directly to four other transputers, facilitates the development of a distributed system model of various sizes and topologies [Galletly90]. Models based on transputer-type technology allows one to focus on an architecturally independent distributed system model where each component of the system is identified and described in terms of its behaviour. That is, the emphasis changes to **what** a component does rather than **how** it does it. As the central theme of this study is concerned with the effect of load balancing components, the model design is more important than its physical realisation. Once the design is complete and representative of the important features of the system, each abstract idea or process can be configured or mapped to the transputers in the network.

Despite the practical advantages of transputers for building a physical model, the Transputer Development System (TDS) was a limiting factor both in terms of the transputer network size and the availability of usable program development tools. Consequently, this study adopted computer-based simulation modelling techniques using the superior facilities of powerful RISC-based workstations. A computer simulation model would allow larger network sizes to be studied with the added speed and flexibility for experimentation and network configuration.

Most tools used for describing a system either emphasise the structure of its components or their behaviour. For example, in the former case, a processor component may be described according to category, function, technology, and other factors such as speed, capacity, and size. In contrast, a behavioural description of a processor may highlight the fetch, decode, and execute cycle. However, according to Djordjevic et al,

> "Structure and behaviour go hand in hand and it is a measure of the power of a language that it is able to enhance one aspect over the other" [Djordjevic'85].

Object-oriented modelling allows one to combine both structural and behavioural information and emphasise aspects of either by hiding certain details. In this study, an object-based approach was used to design the system model. As the focus of the study was primarily concerned with distributed system behaviour, descriptions of system structure was at the system component level. All other information pertaining to processor organisation are hidden. As can be seen in Figure 4.1, the design stage of the experimental model identified three main objects, namely the Simulator, the System, and its Monitor. Again, it is important to note that these three objects can be mapped onto a transputer network of up to three processors. Thus, the Monitor object could be mapped onto a transputer for monitoring the System object, whilst the latter would be a collection of one or more CPU objects residing on one or more transputers. The category (or type description) to which the objects belong are commonly referred to as a class.

Another important aspect of the object-oriented paradigm is the inheritance of common behaviour and attributes by classes that are derived from one or more object class. In Figure 4.1, objects from the monitor and system class are derived from the

simulator class and therefore will inherit the properties of that class. For example, the simulator class defines the context of each simulation run, and each derived class will inherit this context, be it an interactive or batch-oriented simulation context. Further, each derived class will have properties that distinguishes objects of that class from other objects. Thus, the system class may represent the network and processor characteristics, whilst the monitor class is primarily concerned with collecting (or sampling) statistical information.

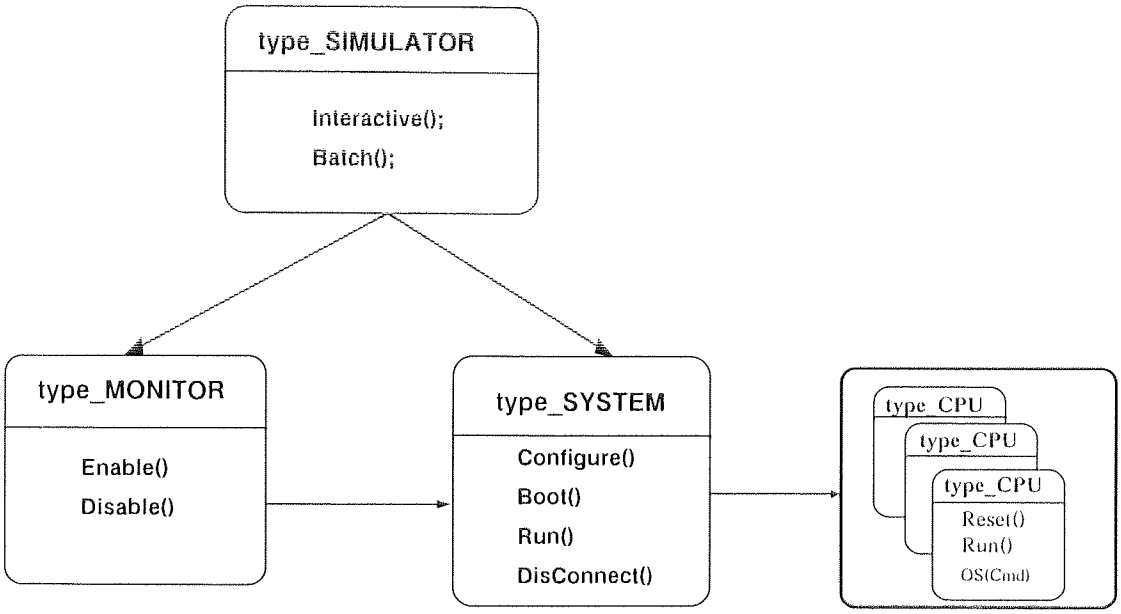


Figure 4.1: Simulation Model Object Interface

However, a common requirement in simulation models is for objects from different derived classes to communicate. In the above example, the monitor object should be able to issue synchronisation and data requests events to the system object using the facilities of the latter without the need to inherit such facilities from the simulator class. To do so would result in a complicated design for the simulator class and a loss in the quality of the abstractions made. These issues are discussed in more detail in Appendix C with reference to the implementation of the design.

The object-oriented approach allows one to describe and represent the functionality (or behaviour) of an object at two levels, commonly referred to as the public and private interfaces. The private interface represents the internal structures and operations specific to the object and is generally inaccessible to the users of that object. In contrast, the public interface can be equated to the list of services offered by an object to users of that object. For example, consider the following description for a monitor object:

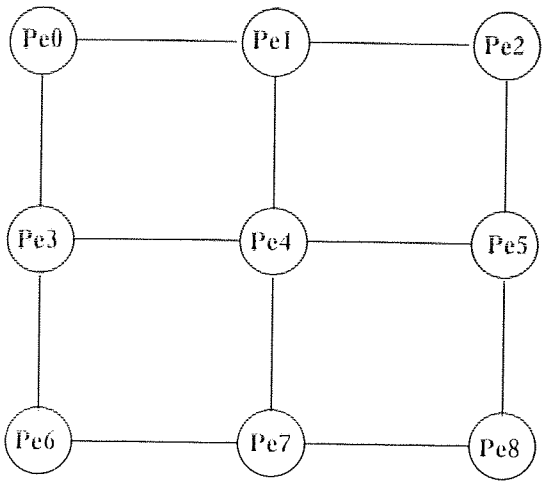
```
task type_Monitor : derived from type_Simulator
{
  private:
        Initialise_Nodes();
        Reset_Reached();
  public:
        Enable();
        Disable();
}
```

In this example, a monitor object provides explicit behavioural methods to Enable or Disable the monitoring of system behaviour. Additional behaviour for the monitor object, such as the initialisation and synchronisation of System objects, is implicitly performed (using private functions). Such low level details on how to model such behaviour is not given the opportunity to confuse, influence or complicate the nature of the public services that must be provided by the monitor object. Thus, as in Figure 4.1, the central focus is the aggregate behaviour of the three objects and the services they offer.

4.2 The Simulator Object

The services provided by the Simulator object include the specification of the system under study both in terms of the processor and the network architecture. In addition, this object also allows the user to select the mode of operation for the system. An

interactive mode of operation allows monitoring information to be displayed on the terminal, and the system parameters to be altered during execution. The definition of the simulated system will generate a configuration file which specifies the connections between the processors in the network. Thus, for a three-by-three square mesh topology illustrated in Figure 4.2, the configuration file specifies for every node in the network a set of nodes to which it is directly linked.



(a) 3-by-3 Mesh

Pe0	{Pe1, Pe3}
Pe2	{Pe1, Pe5}
Pe6	{Pe3, Pe7}
Pe8	{Pe5, Pe7}
Pe1	{Pe0, Pe2, Pe4}
Pe3	{Pe0, Pe4, Pe6}
Pe5	{Pe2, Pe4, Pe8}
Pe7	{Pe4, Pe6, Pe8}
Pe4	{Pe1, Pe3, Pe5, Pe7}

(b) Configuration File

Figure 4.2: A Typical Square Mesh System Configuration

The entries in the configuration file are arranged according to the number of direct links from the source node. For example, node Pe0 has two direct links (to nodes Pe1 and Pe3), whereas node Pe7 has three direct links to nodes Pe4, Pe6, and Pe8. The network configuration file can be generated automatically in the case of the square mesh topology or specified manually for other network interconnections. The configuration file may contain more than one set of network definitions to facilitate multiple simulation runs. Thus, a typical control program for the simulation model would take the following form:

```
PROGRAM Simulation_Model
{
  select
      Task1:
          Simulator.Interactive();
          Simulator.Configure();
          Task2.Entry
      end Task1;

   or Task2:
          Simulator.Batch();
        Entry:: While not end of (Configuration_File)
              System.Configure();
              System.Boot();
              System.Run()
          end While
      end Task2;
}
```

In this example, there are two tasks. The first task (Task1) involves the interactive generation of the configuration file using the Simulator object. In the second task (Task2), the simulator is run in batch service mode but is also characterised by an "entry" point (or rendezvous) through which Task1 may be synchronised. The configuration file created by Task1 is then used to provide the System object with parameters defining the operational context.

4.3 The System Object

The system is essentially a collection of processors and communication links organised in some way. The services provided by the system object include the configuration of both the processor and network architecture, and boot/re-boot, run, and disconnect (or power down) services. The configuration file created by the simulator object is used by the configuration service in three ways. Firstly, to build the distributed system model by identifying the processors and their attributes. Secondly, to set up the direct communication channels between pairs of nodes, and thirdly, to determine the routing table for all nodes in the network. Once the system is

configured, the Boot service will signal each processor, and initialise the links between them.

The dilemma presented in the design of a system object relates to the determination of its properties. The services provided by a system object must be delivered on one or more of the system's processor components. One could consider the introduction of a class of specialist processors that delivers the services required and to which all other processor objects are slaves. For a loosely-coupled system, these specialist processors are not considered to be part of the system specified in the configuration file but are merely private mechanisms for the control of the actual system components. However, an alternative viewpoint that removes the need for specialists is to regard the system's components as systems in their own right that inherits general system characteristics alongside their own specific properties. Consider the following outline class definition for a system object:

```
task type_System : derived from type_Simulator
{
  public:
        Configure();
        Boot();
        Run();
}
```

Each derived component of the system class will inherit and thus reflect all the facets of the system to configure, boot, and run itself. This applies to all the processors of the system and their respective communication channels. Thus, the components are able to deliver the range of services available to the system irrespective of the existence of specialist servers or drivers. Further, the latter approach ensures the integrity of the system model as one can model specialists as actual components of

the system by redefining, if necessary, any of the inherited properties. For example, given other system properties such as processing speed, memory capacity, and software portfolio, one can easily construct an heterogeneous system by redefining these global properties locally for each inherited component.

4.3.1 The Processor Object

As discussed in the previous section, the processor objects are themselves systems with hardware, software, and data characteristics. The hardware architecture encompasses the organisation of memory, processors and registers, timer devices, and the communication technology used. In contrast, the software architecture is primarily concerned with the organisation and operation of the operating system components such as the device drivers, memory manager, and process scheduler. The following outline class definition attempts to encapsulate the general characteristics of a processor object:

```
task type_LocalHost
{
  private:
        Clock(System, Timer)
        Processor (CPU, Graphics, IO, Message)
        Memory (Ram, Register)
        Comms(Data, Control :Bus)
         Host_Admin (Pid, Load, Threshold)
        Address_Table (Link, Route, Comm_Set)
        Event_Queues (Message, Process)
public:
        OS_Command_Interpreter(cmd)
}
```

In the class definition given for a processor object, the operating system command line interpreter (*OS_Command_Interpreter()*) is the only mechanism through which the resource components of a processor object can be utilised. However, it proved impractical to implement independent models of the internal hardware and software components and the interaction between them for a processor object. A more

practical solution would be to adopt a single model of behaviour such that the hardware components become an implicit element of an appropriately named software object, given the degree of interrelationship that exist between them. For example, memory , communication, and timer devices would become a facet of the memory manager, message handler, and the timeout handler software respectively.

The basic data held by each host processor object includes a unique name or identifier, and perhaps its current load and threshold values for load balancing purposes. However, further detailed architectural descriptions would necessitate choice between a proprietary or non-proprietary architecture, be it hardware or software. It is the author's belief that to build a complete model of an existing architecture would only provide a better understanding of that system and the manner in which a load balancing facility can be made an integral component. In this instance, the development of the facility would be better served by the use of actual instances of the proprietary hardware and software. However, as the primary concern of this study is the general scalability and relative ranking of load balancing algorithms, detailed proprietary hardware and software characteristics are of less significance. Therefore, the overall behaviour of the processor was defined specifically in terms of its performance rating. That is, the parameters for describing the processor architecture for each simulation run were primarily the processor and network hardware speed, and the operating system and protocol software processing overheads. Further assumptions, relating to the organisation and management of memory, timers, and user processes were considered in the design of the operating system kernel.

4.3.2 Network Interconnection

The communication medium both in terms of the cable and device types were encapsulated in the form of a message handling process. Again, an important parameter was considered to be the general speed of the communication device rather than the cabling used in connecting processors. In this study, general features of Ethernet communication technology in terms of its speed and reliability was assumed. By representing the communication device as a component of the local host, it is possible to represent its message-passing functionality by making use of the interprocess communication mechanism of the underlying UNIX development environment. For example, within UNIX messages can be exchanged between processes and systems using UDP or TCP/IP sockets. The former is an efficient way of sending messages but does not guarantee the sequence in which messages are sent or the delivery of such messages. In contrast, TCP sockets are slow but provide a guaranteed message delivery service. Therefore, one could use UDP sockets for sending load balancing messages, and TCP sockets for the migration of processes. However, this assumes that the loss of load balancing information is far less important than ensuring the successful delivery of a user process. From the point of view of load balancing algorithms that make use of varying degree of state information such loss may have significant performance implications. It may be argued that the impact on load balancing performance is significantly greater for algorithms that rely on relatively small amounts of state information. However, the timeliness of the information collected is also important. That is, the late arrival of a message may have the same impact as a lost message packet. For example, information that arrives late may result in the recipient continuing processing and ignoring the information as if it was never sent. Another algorithm may attempt to respond to all message arrivals irrespective of their time of arrival. Therefore, it seemed appropriate and practical to use TCP sockets for the load balancing protocol

between two sites at any one time, and UDP sockets for representing the broadcast mechanism for state information to two or more sites. No specific broadcast or multicast mechanism is assumed in the model, and message-passing is effected through point-to-point connection. That is, each message is individually addressed and routed accordingly. UDP sockets seemed an appropriate abstraction as it sends messages many times faster than the TCP channels.

In addition to the communication channels, the individual hosts of a mesh network topology also maintain tables with network addresses for their neighbouring processors (Links) or members of their communicating set (Comm_Set), and tables of paths (Routes) to all nodes in the network. Figure 4.3 shows some typical entries for the route tables for Node0 and Node3 in a three-by-three mesh network. Each host initialises all its address and routing tables with information gleaned from its configuration file.

With the exception of the communicating set table of addresses, the tables remained constant throughout the life of the system. In systems such as Wide Area Networks, where locations and routes are liable to change as a result of traffic volumes, and network or remote system failure, such tables would need to be altered dynamically for all affected sites. This would require additional protocol software to ensure the exchange and verification of host addresses and their routes. Therefore, it was assumed that the network was reliable and traffic contention minimal given the point-to-point architecture adopted. A discussion of routing strategies follows.

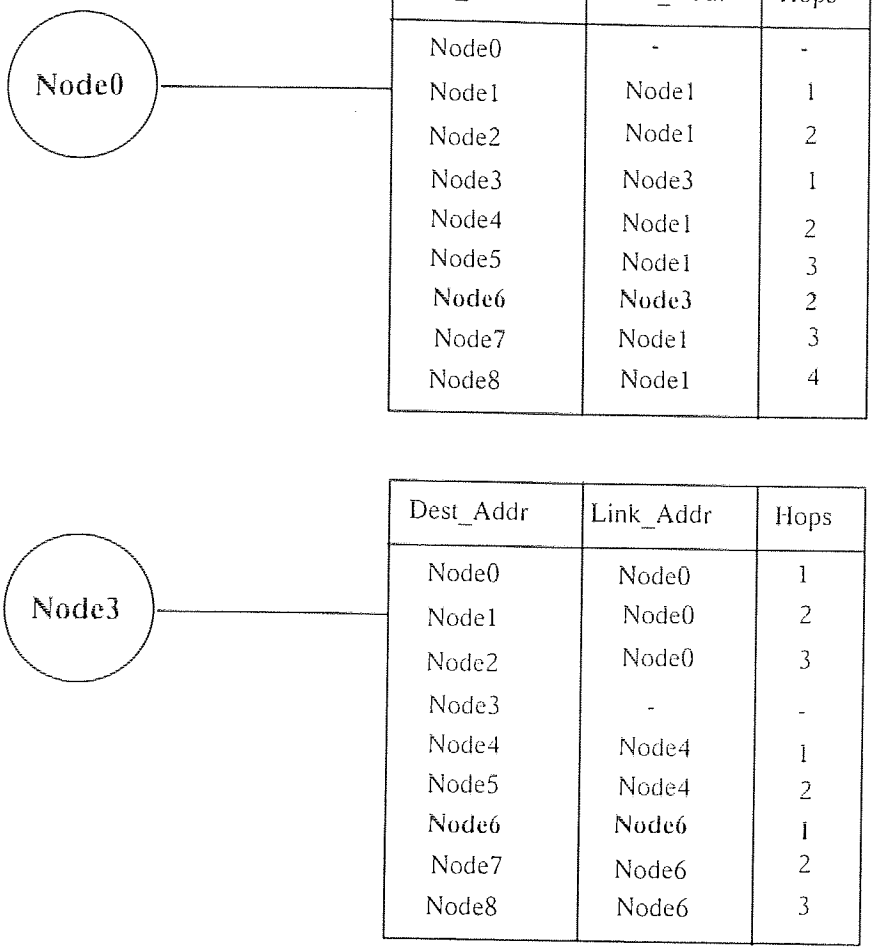


Figure 4.3: Route Table of Node0 and Node3

4.3.3 Routing Algorithm

In a large distributed network it is impractical to provide each network processor with complete and up-to-date knowledge of the current topology and processing characteristics of the whole network. An interval routing scheme, using limited knowledge of the network topology, would allow each host processor to communicate with all other hosts in the network via their immediate neighbours. Each host makes use of two tables to effect correct host addressing: the Link table maintains the physical address and operational state of its most immediate neighbours, and the Route table represents the preferred route and distance from the local host to all other hosts in the network. Normally, the size of the routing table at each node is

represented by O(N) where N represents the number of nodes in the network. Thus, the storage requirement for the routing table will limit the size of the network in terms of its node composition. Various other routing schemes have been suggested to eliminate or minimise the size of the routing table to O(D), where D is the degree (or the number of neighbours) of a node [Leeuwen87]. Whilst such strategies have the advantage of performing message routing for larger networks with insignificant increases in storage overheads at each node, the algorithms are complex and can impose significant processing overheads in attempting to resolve the routing requirement of a given message. As this study is primarily concerned with load balancing, a simple interval routing strategy using partially complete routing tables, is used.

The Route table created at system boot time represents the shortest distance to each host in the network. The path to a given host is generated from the Link table for each intermediate host. For example, given the two route tables in Figure 4.3, the route for a message addressed to Node3 from Node0 can be determined from the Route Table. In this case, the entry indicates that Node3 is an immediate neighbour (a distance of a single hop), and as Node0 maintains a set of addresses for its immediate neighbours, the message can be directly addressed to Node3. In contrast, if the message was to be sent to Node6 from Node0, the Route table entry indicates that it should be sent via Node3. When the message arrives at Node3, its Route table entry indicates that Node6 is an immediate neighbour and can be addressed directly. Thus, messages sent via Node3 to Node6 will always represent the shortest route for Node0. Alternative routes via Node1 will be at least one hop longer. In cases where two equally acceptable routes exist (in terms of distance), the route chosen will be based on the expected traffic intensity on the respective neighbour links. For example, Node0 can send messages to Node4 via Node1 or Node3. However, Node1 will be

selected if it is less of a bottleneck for message traffic on-route to other hosts when compared with Node3. Although the Link and Route tables are created and initialised once in this study, it is possible for a host to broadcast its new address to its immediate neighbours, who in turn will update their respective Link tables. Likewise, the policy of using fixed routes may mean that under extreme traffic intensity routes considered to be shortest may take the longest time to traverse. In such cases, dynamic routing based on traffic flow may be more efficient. Experiments conducted with dynamic routing of messages, based on the assumption that it imposes no additional overheads, was found to have an insignificant impact on overall system performance. This is primarily due to the balance attained in the initial Route tables generated for each host.

4.4 The Operating System Kernel

The operating system, via its command line interpreter (*OS_Command_Interpreter()*) is the public interface through which users can run their processes using the available resources of the system. The operating system is itself a process that is composed of a collection of interacting processes. Therefore, the design of a base class called *type_Aprocess* was first considered as the means by which kernel and user process objects can be built. The resulting object hierarchy is illustrated in Figure 4.4.

A Process object is characterised by information such as: an identifying name, a priority, state, and run-time history in terms of the originating and preferred hosts. In addition, methods are made available to access this information. From this general class, two distinct process sub-classes can be identified, namely user and kernel processes. The kernel processes in the system will include such things as the local scheduler, the load balancing algorithms, and the event and message handlers. Three

types of user processes were identified: io-bound (type_InOut), cpu-bound (type_Cpu), and communication-oriented (type_commset). Such class of processes operate within a different context to kernel processes and are discussed further in Appendix C.

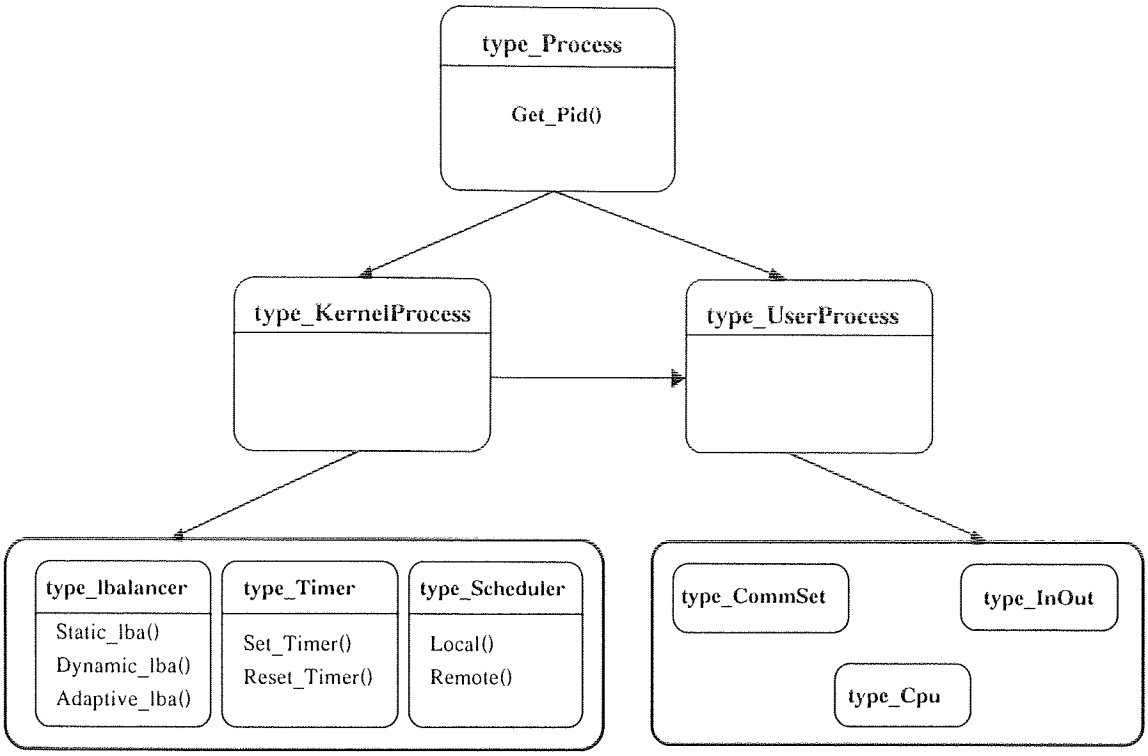


Figure 4.4: Process Object Hierarchy

4.5 WORKLOAD CHARACTERISTICS

The workload of a given host may be characterised according to the type of resources used and the duration and pattern of usage. Such resources include processors, and input-output channels to other devices and processes. A process may be classified according to its service time requirements. A process is considered to be light-weight if it only requires a small number of resource time-slices during its lifetime. Heavy-weight processes would require considerably more time-slices than light-weight

processes. Various researchers have considered workloads characterised by a mixture of light and heavy weight processes using specific resources, or in some cases, a variety of resources. In this study our focus is on both heavy-weight and light-weight, CPU-intensive processes.

The workload used in the Trace-Driven Simulation Model developed by Zhou et al[Zhou88] was based on trace dumps from actual systems. Whilst this may be useful in gaining insight into how load balancing affects performance within a specific user environment, the workload may be unrepresentative for other environments, and it presents difficulties in cases where the workload characteristics are variable. As this study is concerned with the sensitivity of load balancing algorithms to varying workload characteristics, like many other researchers in the field, a discrete-event simulation model was developed.

The UNIX linear congruential random number generator function was used to generate the inter-arrival times for each host, and this in turn was transformed to produce an exponential distribution. In using the queuing model, a system should be stable if processes arrive at a rate that is smaller than their service requirement [Lavenberg83]. In order to examine the sensitivity of load balancing algorithms to different load intensities, the following traffic intensity formula of Lavenberg et al [Lavenberg83] was used:

$$\varrho = \lambda E[S]/m\gamma \qquad (4.1)$$

where λ represents the load factor. Further details on the the queuing model used in this study can be found in Appendix B.

4.6 Load Balancing Protocols

In large loosely-coupled distributed systems, the general decision strategy that an operating system might pursue, could be represented as follows:

```
IF    [load Imbalance cost] >> [message cost]
THEN
      MAXIMISE (load balancing)
ELSE
      MINIMISE(load balancing).
```

That is, if the relative increase in load cost (in terms of average processor run-time) as a result of current workload, is very much greater than the relative increase in message traffic cost resulting from load sharing then one should attempt to maximise load balancing across the whole network. However, where such costs are greater whenever global balancing is attempted, then such activities should be minimised through local load sharing. The problem with the approach outlined above is that it requires that the control algorithm predict and anticipate what the likely load and message traffic cost will be for every change in workload, before it can decide whether to minimise or maximise load balancing. This is an extremely difficult task and the overheads involved in achieving this must be taken into account when developing models for the load balancing protocol. In the protocols discussed in the following sub-sections no a priori knowledge of job arrival times is assumed. Therefore, no job length information can be used when making load balancing decisions in the case of non-preemptive sender-initiated algorithms. However, the remaining service time of processes can be considered in relation to communication costs for the algorithms under study. For example, non-preemptive sender-initiated, and pre-emptive receiver-initiated algorithms will only consider for migration those processes with the longest remaining service times.

In this model, the process queue length is used to measure the load on each host processor. Researchers have shown that load indices based on process queue length provide a relatively accurate representation of system load compared to more complex methods. The primary advantages of queue length are: it is simple to compute, and it imposes negligible computational overheads. For example, by using a register to maintain a count of actual processes with hardware support for increment and decrement operations, the load index may be computed in a single machine cycle. Alternatively, the operation can be conveniently carried out as a by-product of larger system processes that are activated to handle the arrival and removal of processes from local process queues.

However, the decision-making policy of the load balancing algorithm, based on actual process queue length can lead to instability. For example, an underloaded host, on the basis of its actual load at time t, requests work from remote hosts at time t. Two or more remote hosts may respond by migrating one of their local processes. But, due to network delay the migrated processes are received by the local host at time $t+1$, $t+2$, $t+3$. Load instability may ensue as the requesting host generates further requests between times t and $t+1$, without regard for processes in transit. This potential instability in the decision-making policy can be reduced in a number of ways. A simple strategy is to operate a stop-and-go protocol such that no further requests can be made until replies are received from the remote hosts that have been addressed. However, the local host should not wait indefinitely for a reply as this imposes additional performance overheads. Therefore, a timer should be set at the start of each process migration dialogue and the local host can then decide whether or not to re-initiate the exchange of messages should the timer expire. The main problem with this method is selecting the optimum time-out period for maximising overall system performance. For maximum effectiveness the time-out period may

vary according to the communication delay imposed by the system load. Thus, one would expect the time out period to be longer under heavy system loads and shorter when the system is lightly loaded. Whilst it is possible through the use of timers to limit the number of successive requests made by a local host, and to allow the local load to stabilise as those processes in transit arrive at their destination, the problem remains that requests received from remote hosts may result in inaccurate information being transmitted by the local host about its present load state. The local host could delay its reply to a remote request until the time-out period has expired but success again depends on the time-out periods operated by local and remote hosts and the synchronisation of both.

Alternatively, the process queue length can be averaged over a period of time where the period selected takes into account the average duration of network delays and the system load. The main problems with this method are: the required computations are more involved and thus results in greater overheads, and the accuracy of the measure in response to changes in system loads is dependent on adaptive time period variables over which averages are computed.

In this model it was decided to supplement the process queue length with the number of potential processes that are expected to arrive at the local host. This is known as the virtual process load and is represented by the formula:

$$V(h) = Q(h) + \sum_{x}^{n} p_x \qquad (4.2)$$

where Q(h) is the process queue length for host h, and P_x has a value of one for processes destined for host h, and zero otherwise. In this model, hosts that actively negotiate a process transfer on the basis of available capacity, will increment their

virtual load, each time a remote host agrees on a process transfer. The virtual load is decremented (and the actual load incremented) if migration is completed successfully. But fixed period timers are also used to ensure that the virtual load does not remain artificially high should a remote host fail to complete the required transfer within a given time period. By using the virtual process count and process queue length a local host can reply to requests received from remote hosts between time period t and $t + 1$ according to the number of processes it has already agreed to negotiate with other remote hosts. The advantage of this measure is that it imposes fewer overheads compared to the other methods considered. Increment and decrement operations are supported in hardware and can be performed as a by-product of other event handling processes for events such as an expired reply-message timer or the arrival of a migrated process.

Based on the measured processor load, a local host can be in one of three states: Overloaded, Balanced, or Underloaded. The state table in Table 4.1 represents the transition between these states. One or more threshold values may be used to define the load state. A local host is said to be overloaded if its virtual load rises above the maximum threshold value, and underloaded if it falls below the minimum threshold value. The state of a local host is said to be balanced if its load remains within the minimum and maximum threshold values. A fixed threshold value of two was found to give better overall system performance across a wide range of system loads. For the dynamic algorithms implemented a host with three or more processes is considered to be overloaded, balanced with two processes, and lightly loaded with one or zero processes.

In Table 4.1 the ARRIVAL event represents both processes that are generated locally and arriving from a distant host. The REMOVAL event represents processes that

either complete on the local host or are due to be migrated to a remote host as a result of negotiation. Thus, if a local host is overloaded, the removal of a process will cause the local load to be decremented and a change of state will only ensue if the resulting load measure is less than or equal to the threshold value. Whilst it is possible for a local host in the overloaded state to move to the underloaded state as a result of instantaneous process completion and negotiated transfers, the effect of forcing such instantaneous events to occur in sequence means the local host would be in the balanced state if only for a transitory period before it became underloaded.

Event State	Process ARRIVAL	Process REMOVAL
OVERLOADED	Inc(Load); OVERLOADED.	Dec(Load); p2: BALANCED. **not** p2: OVERLOADED.
BALANCED	Inc(Load); p1: OVERLOADED. **not** p1: BALANCED	Dec(Load); p3: UNDERLOADED **not** p3: BALANCED
UNDERLOADED	Inc(Load); p3: UNDERLOADED **not** p3: BALANCED	Dec(Load); UNDERLOADED

p1: Load > [MAX(Threshold)].
p2: Load **in** [RANGE(Max, Min)].
p3: Load < [MIN(Threshold)].

Table 4.1: System State Transition Table

The significance of the load state for local hosts is governed by the granularity of load activation and process transfer decisions. Fine-grained decisions may require the local host to be operated systematically and sequentially through each available state, whilst coarse-grained decisions are not concerned with transitory states but only the final state which results from aggregate behaviour. The load balancing decisions in this study are coarse-grained in nature as such decisions are based on the load state at the time of reading. Thus, a sender-initiated load balancing policy will remain active whilst the local host is perceived to be overloaded and inactive otherwise.

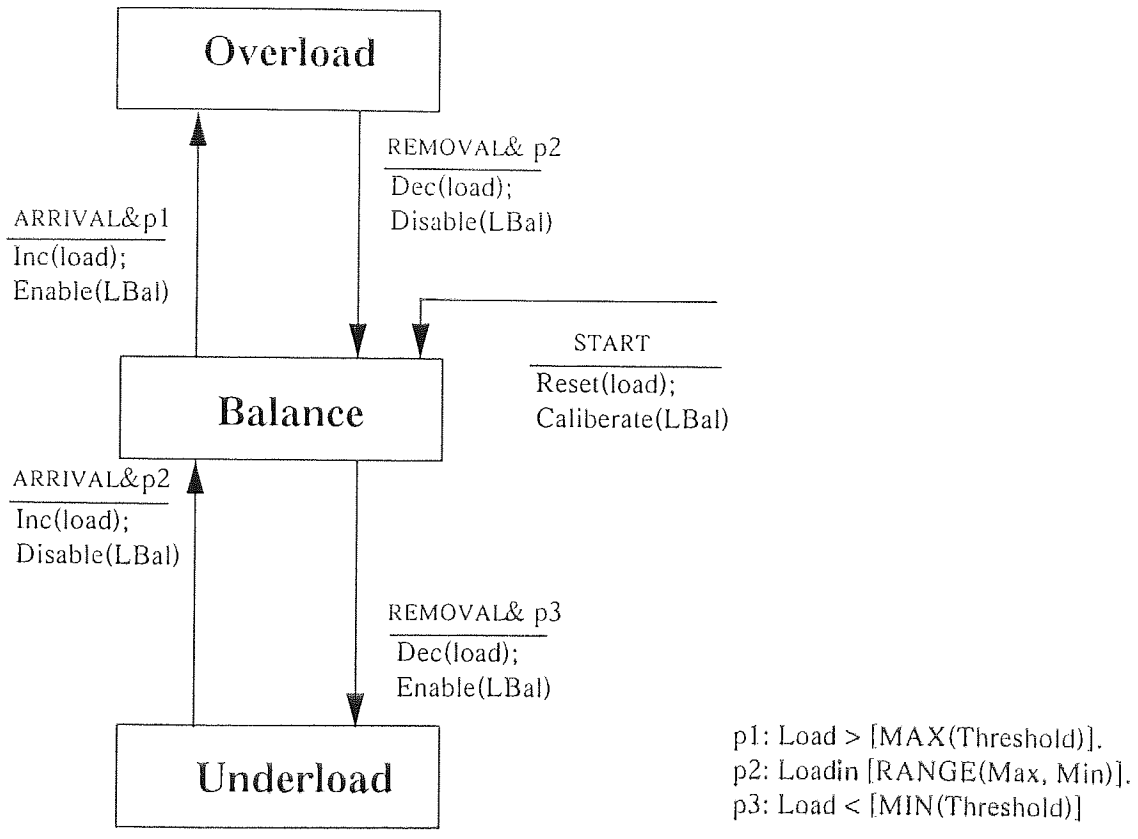


Figure 4.5 State Transition Diagram for Adaptive Load Sharing Policy

In contrast, a receiver-initiated policy remains active in the underloaded state and inactive in the overloaded or balanced state. Adaptive policies are only inactive in cases where the local host is perceived to have a balanced load. Figure 4.5 shows a

typical state transition diagram for an adaptive load sharing policy which is disabled if there are no imbalances in the workload when compared to the threshold.

The local host will continue in its present state providing the respective predicates p1, p2, and p3 of the available states remain true after successive process arrival and departure events. The following sub-sections discuss a range of load balancing algorithms used in this study and tailored to perform across a range of network diameters. In all cases, with the exception of the Random Policy, timeouts are used to avoid hosts waiting indefinitely for replies to requests made on behalf of user processes for suitable location and subsequent execution.

4.6.1 The Random Policy

The random load sharing policies are characterised by the absence of state information pertaining to the remote hosts in the system. However, a variety of implementations exist for the transfer and location policy. A pre-emptive transfer policy would permit the transfer of one or more runnable processes which may be selected at random. Without state information the local host operating a sender-initiated policy must select a remote location at random and initiate process migration. The remote site cannot refuse the transfer and must therefore enter a dialogue with the local host to ensure the successful transfer of the process and its context. This implementation makes use of only one type of Protocol Data Unit (PDU) for communication between the peer entity of the load balancing protocol, namely:

PRC_ARI_MSG *Migrating process in transit for host.*

The receiver-initiated policy would result in a lightly loaded host demanding work from one or more remote hosts. In this case two types of PDUs are required:

PROBE_MSG	*Probe message sent by underloaded host.*
PRC_ARI_MSG	*Migrating process in transit for host.*

The recipient of a probe message should in principle, migrate a process irrespective of its present load state or ignore the demand. A scalable random policy without system state information will need to avoid one or more of the following: process thrashing through multiple migrations of the same process; predominant transfers involving processes with the shortest remaining service time; and locations that have communication delays, due to distance, which is greater than the remaining service time of a migratable process. Further, a receiver-initiated policy should be avoided as performance degrades if lightly loaded hosts demand work from other lightly loaded hosts. In the case of a sender-initiated policy, an overloaded local host receiving processes from a remote host can opt to migrate one of its own processes to another host thus limiting performance degradation. A simplified random policy protocol is shown in Table 4.2.

	ENABLE	DISABLE	PRC_ARI_MSG
IDLE	Select (Process); Rnd_Select (dest); Migrate(Process, dest); Dec(Load). **IDLE**	**IDLE**	Enqueue (Process); Inc(Load). **IDLE**

Table 4.2: State Transition Table for Random Policy

The main events are Enable or Disable load balancing, and Process Arrival messages (PRC_ARI_MSG). The latter description embodies the execution of the process migration protocol. During the execution of the protocol the virtual load is incremented. This may result in a corresponding transfer of a local process should the

site become overloaded as a result of its virtual load. The arrival of a migrated process at the local host is placed in the queue of runnable processes, the virtual load is decremented, and the actual load incremented.

The transfer policy is non-preemptive as processes that have started execution are not eligible for transfer. Further, processes received from remote hosts cannot be re-migrated and must be executed locally. Therefore, only locally created processes waiting to be scheduled for execution are eligible for migration. Such processes will benefit most by being migrated to a lightly loaded remote host. Whilst processes for migration are selected on a first-come-first-serve basis, the location is randomly selected. It is important that all hosts, irrespective of distance, are considered to be potential destinations, thus increasing the likelihood of success in pairing heavily loaded and lightly loaded hosts. Such a policy is potentially scalable as only locally created processes with the longest remaining service time are candidates for migration; and a much larger pool of possible destinations is available.

4.6.2 The Threshold Policy

Unlike the previous protocol, the threshold policy negotiates each process transfer. The scalability of the sender-initiated implementation is supported by non-preemptive remote scheduling. That is, the transfer policy considers newly created local processes as potential candidates for migration to a remote host. The creation of a local process whilst in the overloaded state, initiates the location policy to negotiate the transfer. The policy on the local host randomly selects a remote site, sends it a probe message seeking permission to initiate a process transfer, then awaits a reply. The following pseudo-code illustrate the steps taken by the remote host once a probe message arrives:

PROBE_MSG: IF load < Current_Threshold
THEN Transmit($Site_i$, Acceptance);
Increment virtual load;
Set_Timer (Wait_For_Process);
ELSE
Transmit($Site_i$, Rejection);

If the remote site is also overloaded, process transfer will be refused and the local host must continue negotiation with another randomly selected site. An underloaded remote site will reply favourably to probing, increment its virtual load and set the timer to await the initiation of process transfer. Failure to initiate migration within the timeout period will cause the virtual load to be decremented.

The scalability of the algorithm imposes certain constraints. Firstly, the time given to negotiating a process transfer should be directly proportional to the service and delivery time for a migratable process. In terms of the number of negotiations, research studies have shown that a probe limit of three remote sites produced the best overall performance for algorithms of this nature[Johnson88], [Zhou88], [Eager86]. If a local host fails to find a suitor, having reached the probe limit, the candidate process will be executed locally. Further, the duration of each negotiation should be minimised resulting in a reduction in message traffic and subsequent message delays. In this study, the remote host packs the present load state with its reply to any probe message it receives. Furthermore, once the remote site has agreed to accept a process, it is committed and cannot continue to prolong the dialogue inorder to revoke the transfer because of a recent change in its own load state. However, providing the candidate process is still available to the remote site, the local host will initiate the transfer. If a reply is delayed the local host has no guarantee that the reply would have been favourable and must also use a timeout, after which it may commence negotiation with another site or run the process locally.

The problem with negotiating on behalf of individual processes is that the implementation requires memory space and timers proportional to the number of waiting processes. It is possible to reduce the resources required by means of a single set of resources consisting of a timer and memory space to keep track of those sites that have been probed. Candidate processes would be scheduled using a queue discipline and negotiation is conducted without specific reference to an individual process. The maximum probe possible is dictated by the length of the queue and needs to be recalculated each time a process is created or scheduled, and decremented for every refusal received. The location policy can choose to respond to delayed responses from underloaded sites as such replies are not specific to any of the candidate processes.

A location policy that negotiates on behalf of individual processes was selected for the following reasons. Firstly, the implementation was relatively simple, and offered greater flexibility without increasing its computational requirements. Furthermore, newly created processes can be started earlier. This is beneficial because the longer a process remains in a queue the less likely it is to benefit from being migrated on the third attempt. It is possible for a remote host to be the recipient of enquiries on behalf of two or more processes. However, the potential benefit of probing an underloaded host that can accept multiple process transfers, far outweighs the potential cost of sending multiple requests to an already overloaded host. Finally, the memory space required is a fraction of the space available to user processes and a lower level of tolerance is acceptable for timers implemented in a user's workspace.

The implementation of an equivalent receiver-initiated policy needs to make use of pre-emptive process scheduling. Therefore, in response to a work request, the transfer

policy of an overloaded host must determine which of the available processes have the longest remaining service time. Such a decision requires a priori knowledge of the processes, or the use of statistical or heuristic methods. Figure 4.6 shows two alternative implementation for the location protocol. In figure 4.6(a) fewer messages are exchanged compared to the sender-initiated protocol. That is, an overloaded remote host will initiate a process transfer on receipt of a "Work Request" message. Such a policy is effective, providing the local host awaits a response before initiating further requests. In Figure 4.6(b), a much lengthier dialogue is undertaken as the remote host awaits confirmation for the transfer to take place. Such a confirmation may be necessary if the local host has initiated multiple requests.

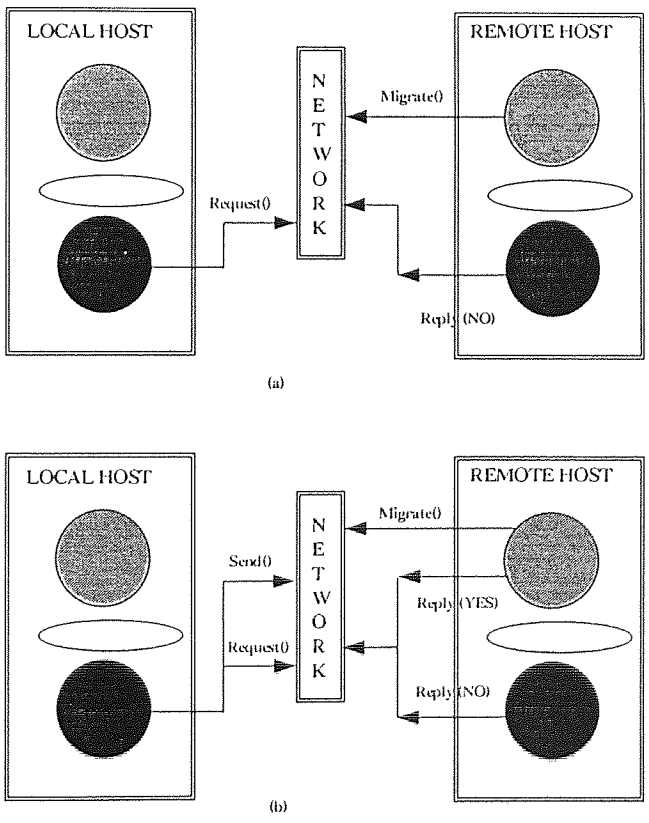


Figure 4.6: Message Exchange for Receiver-Initiated Threshold Policy

The implementation in Figure 4.6(b) requires three types of PDUs:

WRK_RQST_MSG	*Probe message sent by underloaded host.*
WRK_RPLY_MSG	*Reply/Acknowledgement sent by remote host*
PRC_ARI_MSG	*Migrating process in transit for host.*

In this study the reverse policy implemented, initiates multiple requests, using the location protocol illustrated in Figure 4.6(b). Table 4.3 represents the sender-initiated, non-preemptive implementation of the threshold policy used.

4.6.3 Threshold Neighbour Policy

This policy differs from the threshold probe by simultaneously sending probe messages to neighbouring hosts. Remote hosts are considered to be neighbours if their distance, in terms of the number of hops, from the local host are within a predefined range. Thus, remote hosts that are a single hop away from a local host are considered to be immediate neighbours of that host. Thus, probe messages can be sent directly without the need for routing via one or more intermediate remote sites. The selection of a site with which to conduct process migration can vary in complexity and effectiveness. In the case of probe messages sent to immediate neighbours, a local host may simply decide to initiate a migration dialogue with the first remote site to respond favourably despite the fact that other neighbours may exist that are in greater need of load sharing. A local host that awaits all responses before selecting a location, will delay the confirmation of process transfer with those sites that responded favourably and therefore runs the risk of increased competition from other hosts for such sites. Further, in the case of messages sent to sites that are two or more hops away from the local host, it may be necessary to base selection on the load and distance ratio of suitable respondents. For example, such a strategy may select remote sites that are furthest away from the local host if their need for load balancing is greater than more immediate neighbours according to their load and distance ratio.

	ENABLE	DISABLE	PRC_ARI_MSG	PRB_MSG	RPLY_MSG	WAITP_TOUT	PRB_TOUT
IDLE	Reset Probe(i); Rnd_Select(M/c); send(PRB_MSG); Set Timer(PRB). ACTIVE	IDLE	Enqueue (Proc); Inc(Load). **not** p4:Cancel(WAITP); IDLE	p3: send(RPLY_YES) Set Timer(WAITP) IDLE **not** p3: send(RPLY_NO); IDLE	IDLE	IDLE	IDLE

ACTIVE	Reset Probe(i);
	Block(Prc);
	Rnd_Select(M/c);
	send(PRB_MSG);
	Set Timer(PRB);
	
	ACTIVE
		Cancel
		(Timers);
		
		IDLE
			Enqueue(Prc);
			Inc(Load);
			
			ACTIVE
				send(RPLY_NO);
				
				ACTIVE.
					Cancel (PRB);
					p6:Select(prc);
					Migrate();
					Dec(Load);
					p5: IDLE
					not p5: ACTIVE
					not p6 **and** p7:
					Rnd_Select(M/c);
					send(PRB_MSG);
					Set Timer(PRB);
					ACTIVE
					not p6 **and not** p7:
					UnBlock(prc);
					p5: IDLE
					not p5: ACTIVE
						ACTIVE	p7:
							Rnd_Select(M/c);
							send(PRB_MSG);
							Set Timer(PRB);
							ACTIVE
							
							not p7:
							UnBlock(prc);
							p5: IDLE
							not p5: ACTIVE

p4: Queue(WAITP) is EMPTY
p5:Queue(PRB) is EMPTY
p6: Response is YES
p7: Probe(i) < LIMIT

Table 4.3: State Table for Sender-Initiated Threshold Policy

To attain scalability in terms of a range of network sizes and system workload, the threshold neighbour policy was restricted in a number of ways. Firstly, the location policy selected the first site to respond favourably to a probe message. This was justified on the basis that any delay in sending a confirmation message to respondents would be minimised. Further, a remote host whose reply arrives late as a result of a routing delay may cause a further propagation of delays if such a host was selected for process migration. Whilst the idea of pairing the most imbalanced hosts has many benefits in facilitating multiple process migration, its effectiveness relies on the use of preemptive process scheduling. With non-preemptive scheduling only a single process is considered for migration thus making the actual load capacity of all hosts within a specified distance of less significance.

Secondly, the geographical distance over which probe messages are sent should be limited especially where only a small set of migratable process exists. That is, the number of probe messages sent should be proportional to the number of processes available for migration. Thus, as the number of migratable processes increases, the distance considered for sending probe messages should increase. However, a distance no greater than two hops, resulting in the local host sending a maximum of five probe messages simultaneously, was considered scalable for this particular algorithm. Whilst it is conceivable that limiting the probe distance may result in imperfect pairing of light and heavy loaded hosts, over a period of time the load should balance through the propagation of any excess load from a local region (determined by distance) to a neighbouring region.

4.6.4 Communicating Set Policy

The Communicating Set Policy aims to minimise the message traffic between processors. In addition, the policy attempts to avoid instances in which many overloaded hosts send processes to the same lightly loaded site, causing the recipient host to be overloaded. Processor thrashing may arise as the recipient hosts also embark on fruitless load balancing

activity to find a host able to accept responsibility for process execution without further load instability. Based on the concept of a working set model developed by Denning[Denning80] to reduce the frequency of page faulting, the communicating set of a local host is an essential component in managing the load balancing activity. That is, a local host will not initiate load balancing unless it has adequate state information for the members of its communicating set.

In the working set model the reduction in page thrashing is based on the assumption that a process's behaviour is generally characterised by a predictable sequence of local page references over a given period of time. Similarly, it can be argued that processor thrashing can be reduced if one assumes the existence of a predictable set of processors with a tendency to overloading or underloading. Over a period of time such nodes should become members of one anothers communicating set. The consequence of applying the communicating set model is a reduction in the level of message traffic. In particular, at high system loads, the policy should also remove fruitless migrations.

In building the communicating set for a local host the load balancing policy needs to retain a historical record of remote sites that have participated in successful exchanges of processes. Such sites would then be targeted in any future requirements for load sharing. Each host maintains a list of preferred sites. Preference may be governed by available load capacity or distance from the host. The former lays emphasis on maximised load balancing whilst the latter is more concerned with minimal message traffic. Two approaches to updating the preference list were considered. Firstly, each host sends periodic state information to members of its preference (or communicating set) list. An additional PDU type is introduced to represent the state information of broadcasting hosts, namely,

LOAD_STATE *Load State of Remote Host*

The amount of messages generated is proportional to the duration period between the exchange of messages. Thus, although a shorter period results in more accurate state information, the side effect is greater message traffic. An alternative strategy is that the preference list is only updated when load balancing is activated. Message traffic will be further reduced by packaging state information with the request and reply messages sent by paired sites.

Based on the "stable marriage" assignment problem in [Pickard92], the suite of communicating set algorithms should in principle result in the optimal pairing of overloaded and underloaded processors. In the "stable marriage" problem nodes make proposals to other nodes. The recipient will continue to accept or reject one or more of the proposals received until all marriages are stable. That is, all nodes are paired with their most preferred partner. The two major facets of the algorithm are:

(i) given the initial assignments (or pairing) of nodes, no reassignment should be made that would leave both paired nodes worse off.

(ii) if paired nodes are both better off as a result of reassignment, the previous assignment is deemed to be unstable.

Therefore, the basic communicating set algorithm for the local host PE_i during an unstable load state will send proposal requests to each of its preferred sites until an acceptance reply is received or the preference list is exhausted:

IF PE_i detects(load imbalance) THEN
REPEAT
  $Site_j$ = Next_Member ($PreferenceList_i$)
  Transmit ($Site_j$, Proposal);
UNTIL Received($Site_j$, Acceptance) OR Empty($Preference\ List_i$)

On receiving a proposal message the remote host will send an acceptance message if it is the first or only proposal received. However, if the proposal is the worst received, the remote host will reply with a rejection message.

IF First_to_Arrive ($Site_i$, $proposal_i$)
Transmit ($Site_i$, Acceptance)
ELSE
$Site_t$ = Worst_Proposal ($proposal_i$, $Preference\ List_j$)
Transmit ($Site_t$, Rejection);

In this study, a proposal is judged according to the available load capacity of the proposer. As state information is packaged with the PDUs transmitted, the proposal is evaluated each time an enquiry or response is received. Thus, a sender-initiated threshold policy would execute the following steps:

```
CASE Message_Type OF:
    PROBE_MSG:      IF      OverLoaded
                    THEN AsCending_Order(Preference_List);
                    ELSE  Descending_Order(Preference_List);
                    .....
    PRB_RPLY_MSG:   AsCending_Order(Preference_List);
                     .....
End_Case;
```

If the local host receives a probe message in the overloaded state, the preference list is sorted in ascending order of underloaded hosts. A probe message in the underloaded state would result in reordering based on the most overloaded communicating remote host. It is important to note that re-ordering the preference list involves the update and re-evaluation of the state of existing members responsible for the probe message.

In an attempt to attain scalability and performance, the communicating set model was applied to the threshold policy using the preference list to target neighbouring or randomly

selected hosts. The communicating set model was also applied to the global average algorithm discussed below.

4.6.5 Global Average Policy

In contrast to the policies discussed earlier global average algorithms attempt to ensure that the load threshold used by each host is within range of the average system load. One strategy is for all hosts to collect state information from all the other hosts and compute the average load threshold. However, such a policy is susceptible to network delay resulting in inaccurate and incomplete state information. Furthermore, the algorithm is not scalable as any increase in network diameter results in a significant increase in message traffic. The implementation selected was based on the algorithm suggested by Johnson et al[Johnson88] which used a series of time-outs to control the global average maintained by each node. The algorithm is activated periodically and is represented by the following pseudo-code:

```
IF PE_i is Overloaded THEN
   BroadCast(HIGH_LD_MSG);
   Set_Timer(High_Load_TimeOut);
ELSE
   IF PE_i is Underloaded THEN
      Set_Timer(Low_Load_TimeOut);
```

If the local host (PE_i) is in the overloaded state a probe message is broadcast and a timer is set. However, an underloaded host merely sets a timer for its current state. There are four types of Protocol Data Units(PDUs) exchanged by the peer entity of the load balancing protocol. These are:

HIGH_LOAD_MSG	*Probe message sent by overloaded host.*
PRB_RPLY_MSG	*Reply/Acknowledgement sent by remote host*
NEW_LOAD_MSG	*New Load Threshold Value*
PRC_ARI_MSG	*Migrating process in transit for host.*

	ENABLE	HI_TOUT	PRC_ARI_MSG	HI_LD_MSG	RPLY_MSG	WAITP_TO UT	LOW_TOUT	NEW_LD_M SG
IDLE	p1 and not p10: BroadCast(HI_LD). Set High_t_set Timer(HI_TOUT). p3 **and not** p11: Set Low_t_set Timer(LO_TOUT).	Reset High_t_set p1: Inc(Threshold) BroadCast (Threshold)	Enqueue (proc); Inc(Load). not p4: Dec(Virtual_Load) Dec(pwait_count) Cancel(WAITP);	p3: Send(RPLY_MSG) Inc(Virtual_Load) Inc(pwait_count) Timer(WAITP) Reset Low_t_set.	p10 and p6: Cancel(HI_TOUT) Reset Hi_t_set Select (proc) migrate(proc) Dec(Load).	Dec(Virtual_Load) Dec(pwait_count)	Reset Low_t_set p3 and p12: Dec(Threshold) BroadCast (Threshold)	Reset Low_t_set Reset High_t_set Cancel(HI_TOUT) Cancel(LO_TOUT) Set Threshold

p1: Load > [MAX(Threshold)].
p2: Load in [RANGE(Max, Min)].
p3: Load < [MIN(Threshold)].

p4: Queue(WAITP) is EMPTY
p5:Queue(PRB) is EMPTY
p6: Response is YES
p10: High_t_set is TRUE
p11: Low_t_set is TRUE
p12: Threshold > 1

Table 4.4: State Table for Global Average Algorithm

The state transition table for handling the different message types as incoming events is shown in Table 4.4. Should the local host timeout before it receives a reply or an enquiry from an underloaded host, it assumes that this is due to a low system load threshold such that all hosts are in the overloaded state. The local host therefore, increments its current load threshold and broadcasts this new value to all remote hosts. Likewise, if the local host is found to be in the underloaded state, a timer is set within which it might reasonably expect to receive an enquiry from an overloaded remote host. If no enquiry is forthcoming and the timer expires, the policy assumes that this is because the system load is too high. Consequently, the current load threshold is decremented and the new load value is broadcast to all remote hosts. The steps described are represented by the following pseudo-code for handling load state timeout events:

```
CASE Signal_Type OF
    HIGH_TOUT:  IF load > Current_Threshold
                    Increment (Current_Threshold);
                    BroadCast(New_Load_Msg, Current_Threshold);

    LOW_TOUT:   IF load < Current_Threshold
                    Decrement (Current_Threshold);
                    BroadCast(New_Load_Msg, Current_Threshold);
```

The implementation discussed is referred to as a sender-initiated global average algorithm because load enquiry messages are only sent in the overloaded system state. In this study, a receiver-initiated global average algorithm was also implemented where only work request messages are transmitted by underloaded hosts. In a similar manner to the sender-initiated strategy, an overloaded host that does not receive a work request enquiry will assume that the threshold is too low, increment the current threshold, and broadcast this new value on the network.

To address the issue of scalability for such algorithms a number of constraints have been imposed. Firstly, the distance over which enquiries are made is restricted. Secondly, no host can make multiple broadcasts of enquires within the timeout period.

4.7 PROCESS IMPLEMENTATION

In implementing the system and process objects described, two possible representations were available. Each object in the system could be represented as a UNIX process, an ADA task or a relatively simple data structure of the implementation language. For example, each host entry in the configuration file could be configured as a UNIX process spawned by the System object by means of the configuration service. In addition, the communication links between hosts would be represented by I/O streams using the UNIX "pipe" or "socket" abstractions. This is represented in the following pseudo-code:

```
Procedure System::Configure()
{
    while next configuration incomplete(Configuration_File)
    Begin
        Host[i] = Read(Host_Id, Links, Configuration_File);
        SetUp_IO_Stream(Links);
        Host[i] = fork();
        if (Host[i] == CHILD)
          wait(SYNC);
          break;
    END
}
```

In the code extract above the system object maintains the table named Host that contains the UNIX process IDs for all the hosts in the simulation model. The system call to spawn a process (fork()) has the effect of creating another system object with its own separate data area but the same code segment as its parent. Thus, the only

means by which a host can identify itself is by checking its table, Host[i]. A host will then await a synchronisation signal to indicate completion of system configuration. The process spawning method of UNIX when used to model a distributed system can be unwieldy resulting in fairly complex code.

In comparison, the ADA tasking abstraction is more intuitive and easier to use. Thus, one can define process task types such as cpu-intensive, IO-intensive, or communicating groups, where each object task instantiation will behave according to its type definition. For example, a CPU-intensive task type can be defined as follows:

```
task type CpuIntense_Task is
   entry Restart_Signal (node_id : in integer);
   entry Suspend  (node_id : in integer);
end CpuIntense_Task;

task body CpuIntense_Task is
-- local declarations
begin
   loop
     select
          accept Restart_Signal (node_id: in integer) do
             exit when computation_complete;
              -- perform computation
          end restart_signal;
     or
          accept Suspend (node_id: in integer) do
              -- block process
          end restart_signal;
     or
          -- blocked
  end loop;
  Termination_Signal (My_Id);
end cpuIntensive_Task;
```

In this case, each cpu-intensive process object will only accept calls made to the Restart_Signal and Suspend entry points by other process objects. However, in a time-sliced scheduling environment, only the scheduler process is likely to make such calls. The body of the process task is to loop continually answering calls until the computation terminates. In the latter case the process makes a call to the scheduler

(via its entry point Termination_Signal) and sends its process identification number. On receiving a "restart signal" call, the computation will continue until it completes or a "suspend signal" call is received.

Whilst the use of ADA tasks directly or indirectly represented as spawned processes connected through the use of pipes and sockets would give the model a sense of realism, it does have some major drawbacks. Firstly, each process created occupies valuable memory space, and restrictions on the number of processes that can be spawned by another process may severely limit the total number of hosts and processes that can be created. Secondly, the host operating system (in this case UNIX) will spend considerable time managing such processes along with the other very real processes of other users. The model developed by Johnson [Johnson88] used spawned processes which resulted in a restricted simulation model limited to a three-by-three processor network. Furthermore, the model was found to run extremely slowly, taking anything from eight to 24 hours to complete a single simulation run [Johnson88]. Therefore, the approach adopted in this study was to implement host objects (and user processes) as internal data structures, such as a series of dynamic linked lists, where each entry represents a host (see Appendix C for further details). Further, the simulation of message-passing within the network is accomplished by time-stamping each message with the expected time of arrival at its intermediate/final destination node. These messages are then queued at the receiving processors where they will only be processed if they have reached or exceeded the time-stamped due date. This approach was found to deliver the required performance for a range of network diameters and, as demonstrated in the next chapter, produced similar results when given near-identical parameters to those used in models designed by other researchers.

CHAPTER 5

MODEL VALIDATION

5.1 VERIFICATION

The areas of the model that formed the major focus for verifying the simulation runs were the accuracy of the send-receive cycle for message passing, consistent time keeping, and the reliability of the results produced. A 16-processor mesh was generated and routing tables were automatically created using Node0 as the initial starting point. Figure 5.1 shows the potential message routing traffic bottlenecks based on the pre-specified shortest-route tables generated . It can be seen that nodes *5, 9, 10, 11, 13,* and *14* would be the most heavily used switching points in the network. Likewise, nodes at the edges of the mesh (*0, 3, 12, and 15*) represent potential areas of light message re-routing activity.

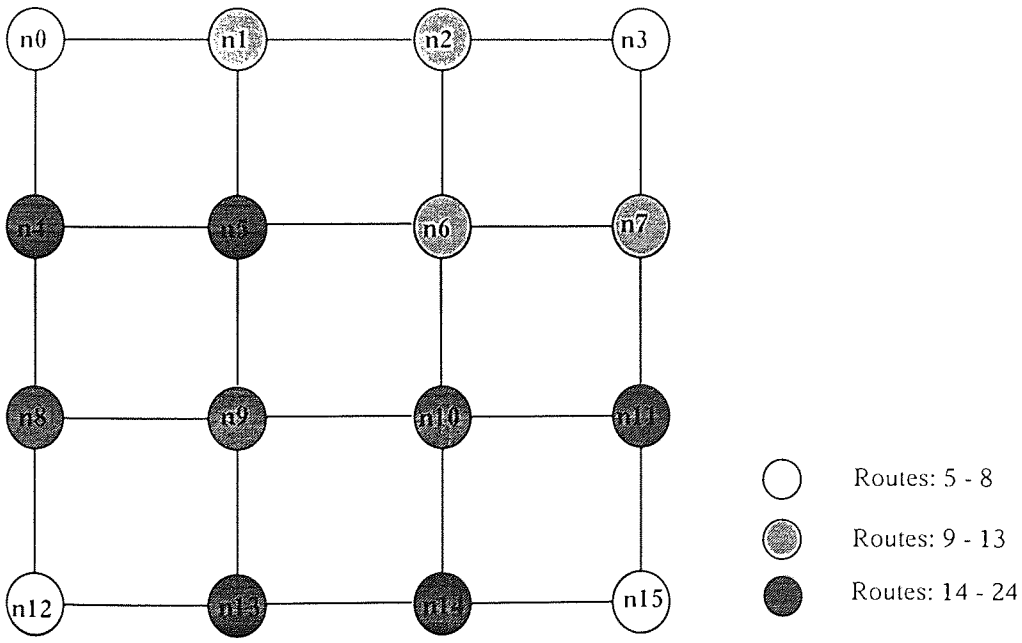


Figure 5.1. Automatic Distribution of Message Routing Load From Node0

Figure 5.2 represents the route distribution resulting from selecting node6 as the initial starting point for route table generation. Clearly a better distribution is attained as the more central nodes take more routes away from the edge nodes. Given that the variation in the distribution of routes is dependent on the node chosen to initiate the generation of route tables, a possible solution could be the use of a 3-dimensional Torus topology. Each node would have four connected links which in turn eliminates any routing imbalance caused by edge nodes with only two or three links in the case of a two-dimensional mesh. However, such a topology is more complex and would limit the variation in communication distance between all nodes in the network. The net effect is that the portability of load balancing algorithms may be limited to networks organised around equi-distant nodes.

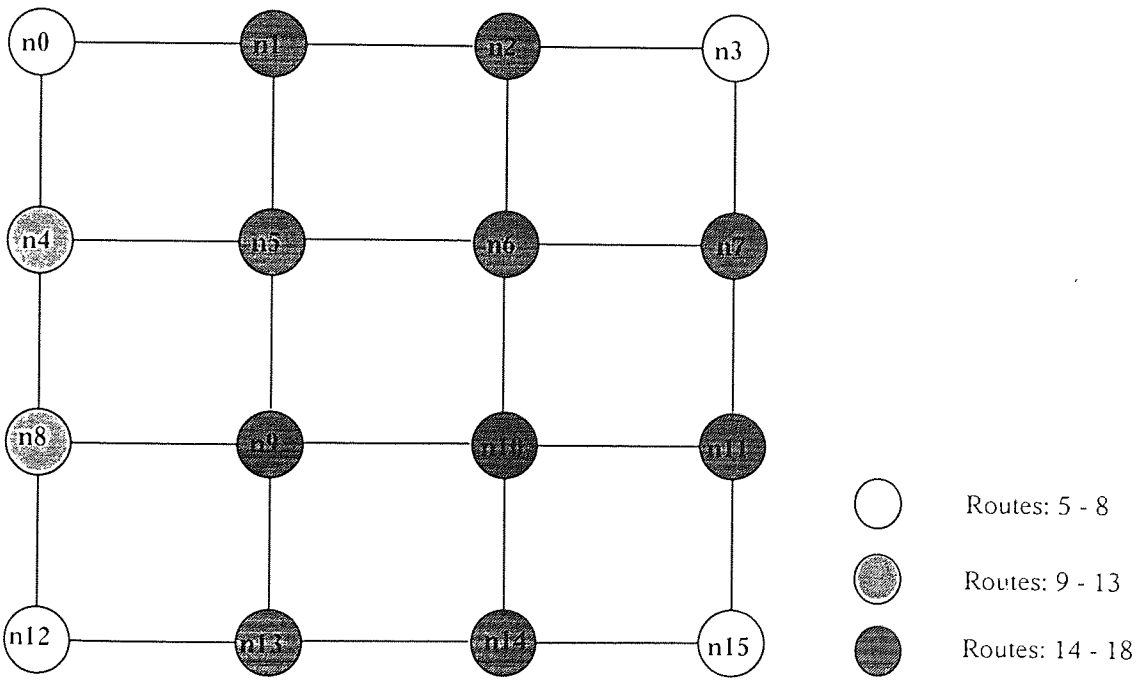


Figure 5.2. Automatic Distribution of Message Routing Load From Node6

5.1.1 System Parameters

In terms of this study it is important that the route distribution does not distort the performance of the host processors concerned. For example, it is possible that nodes

representing potential communication bottlenecks may have a poorer performance as more of the processor time is devoted to switching messages. Thus, the simulation model was run using the different routing tables generated against a group of load balancing algorithms. All the results produced were comparable and the relative ranking of the algorithms remained consistent.

The parameters of the model were also selected to aid in the validation of the model (discussed in the next section). Each processor's raw CPU speed was set to one MIPs. The speed of the communication medium was set to 10Mbits per second with protocol stack processing overheads of 1000 machine instructions per transmitted message. Thus, the timeout period for the transmission and receipt of a reply or acknowledgement message was primarily governed by the size of each message and the distance travelled. This was represented by the formula:

$$T_{xp} = M_{sz}\, D\, (T_{rx} + T_{tx}) + T_y \qquad (5.1)$$

where T_{xp} is the timeout period in milliseconds; M_{sz} is the message size in bytes; D represents the distance in terms of hops to the destination; and T_{rx} and T_{tx} represents the respective receive and transmit times for the protocol layers of the system. Given the point-to-point nature of the network topology, it is assumed that each intermediate host between the source and destination will receive the complete message and retransmit to the next node en route. Further, it is recognised that network loading will have a variable effect on the expected message arrival times. In this study a constant value T_y is used to represent the overall effect of such delays, and a fixed period of 200 milliseconds was found to be sufficient time for the completion of the send and receive cycles irrespective of the system load.

The workload consisted of independent, cpu-intensive processes with average response times of one second. Local scheduling was performed using the 'round-robin' scheduling discipline. A time-slice of 50 milliseconds was used with process context-switching overheads of 200 microseconds. The period 10 milliseconds between each simulation phase was found to be of sufficient resolution to capture and respond to messages exchanged between processors. Further, all processors were synchronised with the global simulation clock at 50 millisecond intervals. Given the parameters outlined, it is possible that the timeout period for a process on whose behalf load balancing negotiation is effected can represent as much as 50% of the service time. Therefore, in such cases it is imperative that any negotiation that takes place results in the migration of the waiting process to a remote site where its expected service time can be improved. Otherwise, the resulting delay may well degrade the performance of the system imposing overheads that are absent in cases where load balancing is not practised [Mirchandaney89].

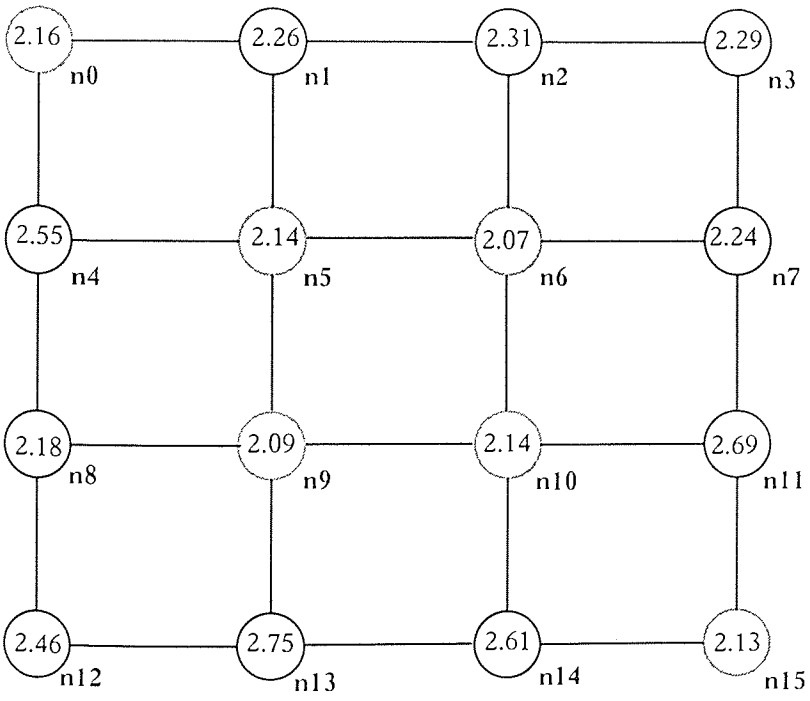


Figure 5.3. Processor Response Times After 500 Seconds

Figure 5.3 shows the average response time per host, after 500 seconds of simulation time, using the threshold load balancing policy under heavy system load. The nodes offering the best average run time performance were nodes *0, 5, 6, 8, 9, 10,* and *15*. The simulation was then run and sampled over a longer period of time, the result of which are shown in Figure 5.4. Whilst the average response time has increased overall, the run-time performance of nodes *4, 11, 13,* and *14* have improved. However, nodes *5, 6, 9,* and *10* continue to yield better than average performance despite the increase in response times.

It is clear from the results produced that there is little correlation between message bottlenecks and runtime performance. However, the 'edge effect' is evident as the central nodes, each connected to four processors, produced the better overall run-time performance. This is primarily due to their proximity to each other and to all the other nodes in the network.

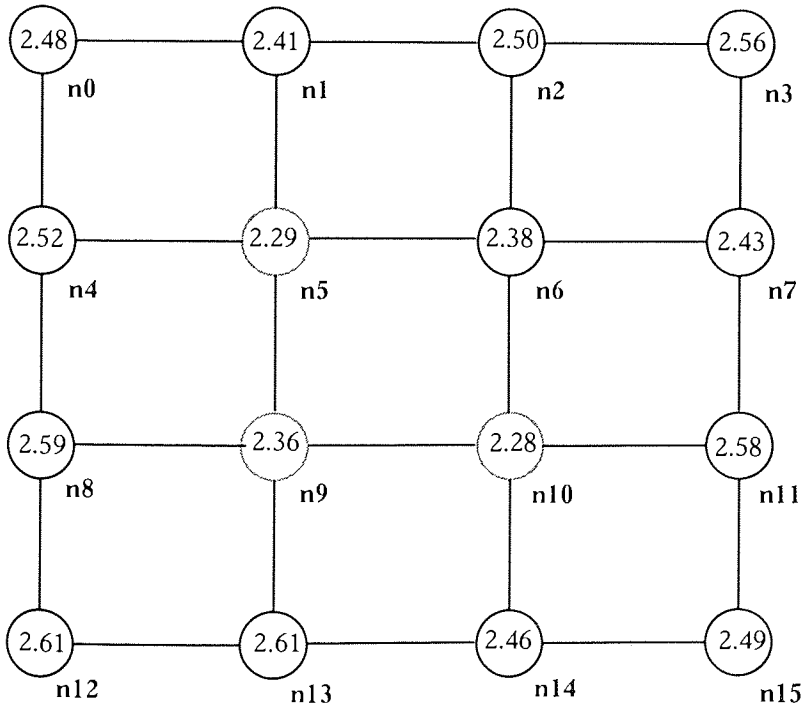


Figure 5.4. Processor Response Time After 6500 Seconds

5.1.2 Load Balancing Protocols

After identifying the possible effect of message bottlenecks resulting from route distribution, it was important to also examine its impact, if any on the operation of the load balancing protocols. Therefore, the simulation model was run for a range of policies with debugging information enabled. Statistics were obtained for the whole system and for each host for every 500 (simulated) seconds. The following extract represent the output from the model during the first period of the simulation run for the threshold load balancing policy.

```
SIMULATION no.1 STARTED ON Thu May 19 16:24:27 1994

*** erand48 seeds are: 17757, 22851, and 10394 ***

No. of Processors =     16   Load value      = 0.80
Total No. of jobs =  350000  Ld Balancing Alg. = Threshold L/B Policy

Threshold Level  =    2.00   Probe Limit     =      3
Convergence to < 2.00%

TIME        VAR.   LOAD  DIFF.   RTime   %Conv   MIG.     Tx    %JOB
 ----        ----   ----  -----   -----   -----   ----     --    ----
500.00      1.2915 2.9026 3.8620  2.2819    1.44  0.3155 7.3187  1.83

N:  0     Mg:170.00     L:  3   Rt: 2.16 Tx:  13  Imm: 141    Dth:  431
N:  1     Mg:175.00     L:  4   Rt: 2.26 Tx:  18  Imm: 146    Dth:  433
N:  2     Mg:138.00     L:  3   Rt: 2.31 Tx:  17  Imm: 144    Dth:  392
N:  3     Mg:165.00     L:  4   Rt: 2.29 Tx:  12  Imm: 162    Dth:  401
N:  4     Mg:152.00     L:  2   Rt: 2.55 Tx:  22  Imm: 141    Dth:  404
N:  5     Mg:150.00     L:  3   Rt: 2.14 Tx:  23  Imm: 171    Dth:  387
N:  6     Mg:151.00     L:  3   Rt: 2.07 Tx:  20  Imm: 182    Dth:  377
N:  7     Mg:158.00     L:  4   Rt: 2.24 Tx:  21  Imm: 160    Dth:  399
N:  8     Mg:128.00     L:  5   Rt: 2.18 Tx:  19  Imm: 182    Dth:  361
N:  9     Mg:158.00     L:  3   Rt: 2.09 Tx:  34  Imm: 164    Dth:  388
N: 10     Mg:162.00     L:  1   Rt: 2.14 Tx:  34  Imm: 164    Dth:  413
N: 11     Mg:159.00     L:  2   Rt: 2.69 Tx:  31  Imm: 141    Dth:  378
N: 12     Mg:176.00     L:  2   Rt: 2.46 Tx:  16  Imm: 159    Dth:  416
N: 13     Mg:155.00     L:  4   Rt: 2.75 Tx:  33  Imm: 153    Dth:  369
N: 14     Mg:161.00     L:  3   Rt: 2.61 Tx:  31  Imm: 158    Dth:  388
N: 15     Mg:166.00     L:  3   Rt: 2.13 Tx:  19  Imm: 155    Dth:  410
```

The output produced by the model for the purpose of statistical analysis include, the average load variance (*VAR*), average workload (*LOAD*), the maximum difference in

processor workload (*DIFF*), the average response time (*RTime*), the average number of migrations per second (MIG), and the average number of messages transmitted per second (*Tx*) by the time the system dump was made (*TIME* in seconds).

Using the output from the run trace, the average load per processor (L:) was examined to see whether improved run-time performance (Rt:) is the result of having a light system load. In the simulation run conducted using a fixed threshold, a host is considered to be overloaded if its process queue is at least four processes long. Thus, the actual load state of the hosts after 500 seconds and displayed in Figure 5.5, would indicate the overloaded hosts to be nodes *1, 3, 7, 8*, and *13*. Likewise, the underloaded hosts were nodes *4, 10, 11*, and *12* with process queue length of two processes or less. Only node *10* had a response time that was well below the average.

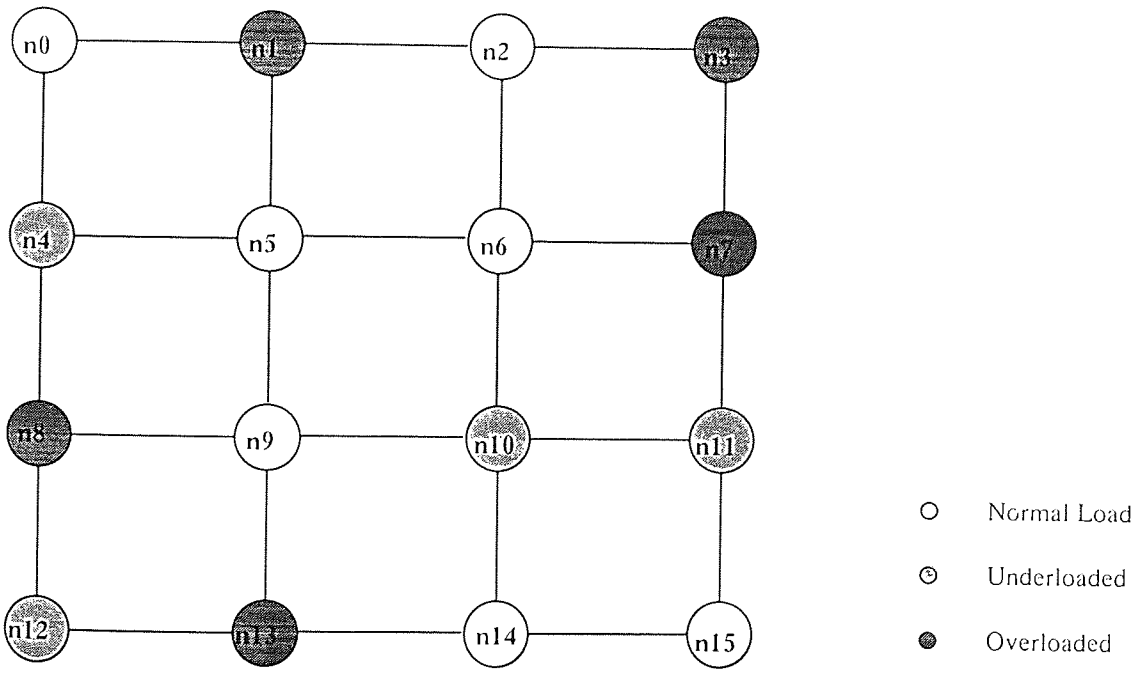


Figure 5.5. Average Process Queue Length Per Processor

However, a more indicative measure of a processor's load is its activity over a period of time rather than at a specific instance in time. Therefore, by examining the number

of process migrations relative to the number of migrants received by each host one can determine the load characteristics of a given host. That is, a local host will be considered as having a propensity for overloading if it has migrated many more processes (at least 10 in the case of hundreds of processes) than it has received. Thus, in Figure 5.6 nodes *0, 1, 4, 11, 12, 15* would appear to be the overloaded nodes, whilst nodes *5, 6,* and *8* are underloaded. These three underloaded hosts also had response times that were well below the average.

To gain insight into the threshold probe load balancing algorithm the message dialogue between nodes was output and examined. Given the size and volume of information and the level of activity in a 16-processor mesh the trace focused on the message dialogue of one node that is currently underloaded (node*10*) and one that is overloaded (node*8*).

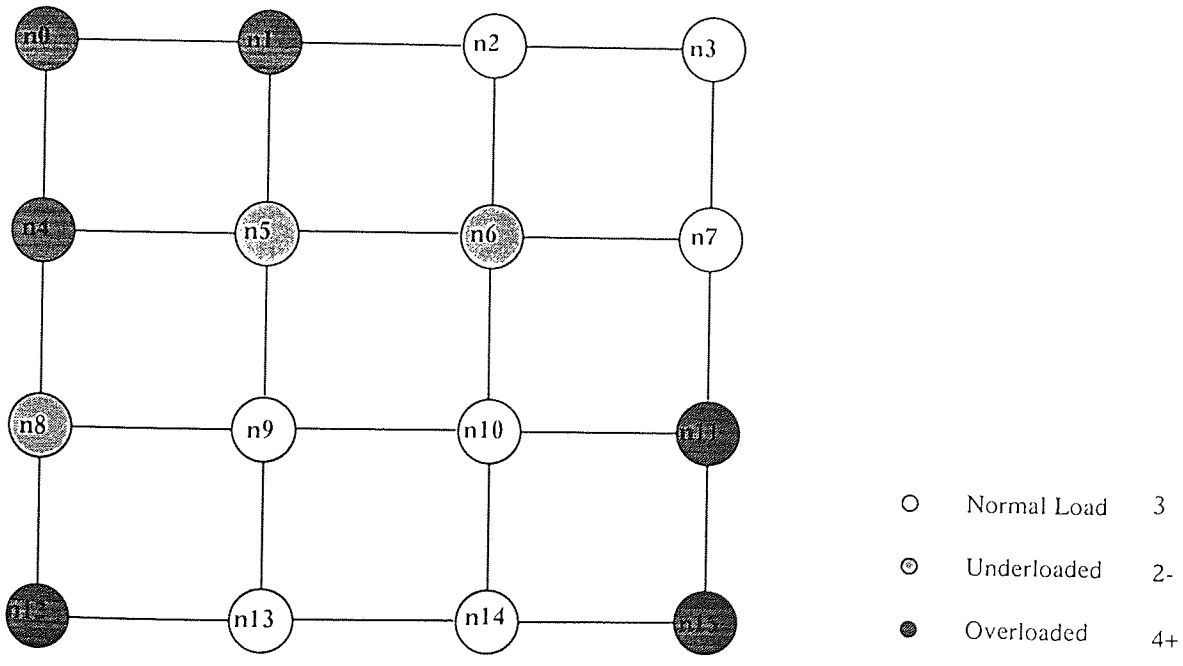


Figure 5.6. Workload Distribution Based On The Processor Send-Receive Ratio

The trace extracts used in this section represents the dialogue just before and after the first period of the simulation. Only the message end-points are output in order to simplify the descriptions given. The intermediate destinations are less important at this stage.

```
Node8 receives Packet N6-N8::   node 8 receives Exit msg [prc:: 6325 at node:6]

Node10 receives Packet N2-N10::        Node2 -overloaded- has asked node10 TO accept
Node10 GMT 497.497    TripT0.254     TIMER:WAITP_TOUT          Etime: 497.751
Node10 receives Packet N2-N10::        node 10 RECEIVES process6357 [load = 2] FROM node 2
```

In the above extract, *Node8* receives an exit message from *Node6* for process *6325*. This process originated at *Node8* and was migrated to *Node6* where its execution was completed. In the case of *Node10*, a probe message is received from an overloaded *Node2*. As *Node10* is underloaded, an accept message is returned, and the "wait for process" timeout is set. Subsequently, Process *6357* then arrives from *Node2* and the timer is cancelled.

```
Node8 receives Packet N15-N8:: node 8 receives Exit msg [prc:: 6344 at node:15]
Node8 receives Packet N2-N8::   Node2 -overloaded- has asked node8 TO accept
Node10 receives Packet N2-N10::        Node2 -overloaded- has asked node10 TO accept
Node10 receives Packet N5-N10::        node 10 receives Exit msg [prc:: 6343 at node:5]
Node10 receives Packet N9-N10::        Node9 -overloaded- has asked node10 TO accept

Node8 receives Packet N12-N8:: Node12 -overloaded- has asked node8 TO accept
Node8 GMT 498.771     TripT0.252     TIMER:WAITP_TOUT          Etime: 499.023
Node8 receives Packet N12-N8:: node 8 RECEIVES process6366 [load = 2] FROM node 12
```

In this extract *Node8* receives an exit message for process *6344* at *Node15* followed by a probe message from an overloaded *Node2*. *Node10* was also probed by the latter as well as by *Node9* and both were rejected. When *Node8* was probed by *Node12*, the former was found to be underloaded, the timer was set, which was later followed by the arrival of process *6366* from *Node12*.

```
Node8 receives Packet N14-N8:: Node14 -overloaded- has asked node8 TO accept
Node8 GMT 499.049    TripT0.256    TIMER:WAITP_TOUT        Etime: 499.305
Node8 receives Packet N14-N8:: node 8 RECEIVES process6369 [load = 3] FROM node 14

Node8 GMT 499.369    TripT0.254    TIMER:PRB_TOUT    Etime: 499.623
Node8 receives Packet N13-N8:: Node8::Time:499.475283      Node13's reply (prc6386) was  4
Node8 GMT 499.476    TripT0.260    TIMER:PRB_TOUT    Etime: 499.736
Node8 receives Packet N1-N8::  Node1 -overloaded- has asked node8 TO accept
Node8 GMT 499.479    TripT0.258    TIMER:PRB_TOUT    Etime: 499.737
Node8 GMT 499.583    TripT0.252    TIMER:PRB_TOUT    Etime: 499.835

Node8 receives Packet N9-N8::
Node8::Time:499.658963Node9's reply (prc6392) was  -2
node 8 sends process6392 [load = 6] to node 9

Node8 GMT 499.672    TripT0.252    TIMER:PRB_TOUT    Etime: 499.924
Node8 receives Packet N15-N8:: Node8::Time:499.723590      Node15's reply (prc6391) was  4
Node8 GMT 499.725    TripT0.252    TIMER:PRB_TOUT    Etime: 499.977
Node8 receives Packet N3-N8::  Node8::Time:499.725599      Node3's reply (prc6386) was  3
Node8 GMT 499.727    TripT0.256    TIMER:PRB_TOUT    Etime: 499.983
Node8 receives Packet N9-N8::  Node8::Time:499.780016      Node9's reply (prc6393) was  3
Node8 GMT 499.781    TripT0.256    TIMER:PRB_TOUT    Etime: 500.037
Node8 receives Packet N9-N8::  Node8::Time:499.782025      Node9's reply (prc6391) was  3
Node8 GMT 499.783    TripT0.256    TIMER:PRB_TOUT    Etime: 500.039

Node8 receives Packet N11-N8::
Node8::Time:499.959678Node11's reply (prc6393) was  -2
node 8 sends process6393 [load = 6] to node 11

Node8 receives Packet N14-N8:: Node8::Time:499.972901      Node14's reply (prc6386) was  3
Node8 receives Packet N1-N8::  Node8::Time:499.973906      Node1's reply (prc6391) was  4
```

In the final extract above, *Node8* is probed by *Node14* resulting in the migration of process *6369* from the latter to the former. However, subsequent to the migration, the creation of local process *6386* at *Node8* has resulted in the overloaded state. Therefore, *Node8* sends a probe message to *Node13* and sets the probe timer. As *Node13* is unable to accept the process, another node is selected and a probe message sent. During that time *Node8* also receives a probe message from *Node1*, and a further local process (*6392*) is created. The now overloaded site (*Node8*) probes *Node9* and sets the probe timer. As *Node9* is underloaded and willing to accept, process *6392* is migrated. In addition Nodes *15, 3,* and *9* were also probed on behalf of processes *6391*, *6386*, and *6393* respectively but all indicated their over capacity. Subsequently, *Node9* was probed on behalf of process *6391*, and *Node11* on behalf of

process *6393*. Only the latter was able to accept a process. The overloaded site also sent probe messages to *Node14* and *Node1* on behalf of process *6386* and *6391* respectively, but both were rejected. Therefore, the latter two processes were executed locally. After 6500 seconds of simulated time, the following output was produced by the model:

N: 0	Mg:1863.00	L: 4	Rt: 2.48	Tx: 14	Imm:1783	Dth: 5197
N: 1	Mg:2041.00	L: 3	Rt: 2.41	Tx: 18	Imm:2064	Dth: 5253
N: 2	Mg:2044.00	L: 3	Rt: 2.50	Tx: 18	Imm:2022	Dth: 5323
N: 3	Mg:1913.00	L: 3	Rt: 2.56	Tx: 14	Imm:1965	Dth: 5132
N: 4	Mg:1948.00	L: 3	Rt: 2.52	Tx: 20	Imm:2008	Dth: 5160
N: 5	Mg:2075.00	L: 3	Rt: 2.29	Tx: 25	Imm:2175	Dth: 5268
N: 6	Mg:2022.00	L: 3	Rt: 2.38	Tx: 25	Imm:2081	Dth: 5272
N: 7	Mg:2063.00	L: 3	Rt: 2.43	Tx: 22	Imm:2098	Dth: 5192
N: 8	Mg:2054.00	L: 5	Rt: 2.59	Tx: 24	Imm:1957	Dth: 5194
N: 9	Mg:2036.00	L: 2	Rt: 2.36	Tx: 36	Imm:2120	Dth: 5178
N: 10	Mg:1945.00	L: 3	Rt: 2.28	Tx: 36	Imm:2192	Dth: 5040
N: 11	Mg:2058.00	L: 2	Rt: 2.58	Tx: 31	Imm:1937	Dth: 5123
N: 12	Mg:2010.00	L: 2	Rt: 2.61	Tx: 18	Imm:1888	Dth: 5266
N: 13	Mg:2050.00	L: 2	Rt: 2.61	Tx: 35	Imm:1945	Dth: 5205
N: 14	Mg:2071.00	L: 3	Rt: 2.46	Tx: 34	Imm:1986	Dth: 5258
N: 15	Mg:2010.00	L: 5	Rt: 2.49	Tx: 20	Imm:1982	Dth: 5185

In terms of the ratio of processes sent to those received nodes *5, 9,* and *10* can be identified as having a propensity for the underloaded state, whilst nodes *0, 8, 11, 13*, and *14* to the overloaded state. In this instant, given that thousands of processes are being considered, a host is regarded as having a tendency to an unstable load state if the difference in migration activity is significantly greater than 50 processes. It is noticeable from Figure 5.7 that only nodes *0* and *11* are still members of the overloaded set, and only node *5* of the underloaded set. Further, members of the overloaded set have response times that are worse than average whilst underloaded set members have better than average response times. It is clear from the workload distribution sets developed after 500 and 6500 seconds of simulation time, that members of the overloaded and underloaded sets tend to be edge nodes and centre nodes respectively. Over a period of time, the underloaded edge nodes will become

overloaded as they experience difficulty in off loading their excess workload. Further, by using the threshold policy it is more likely that edge nodes will receive many more probe messages than the central nodes given their greater numbers in the network. Therefore, such a policy is unlikely to reduce to a satisfactory level the variation in response time between the different processors in a two-dimensional mesh topology.

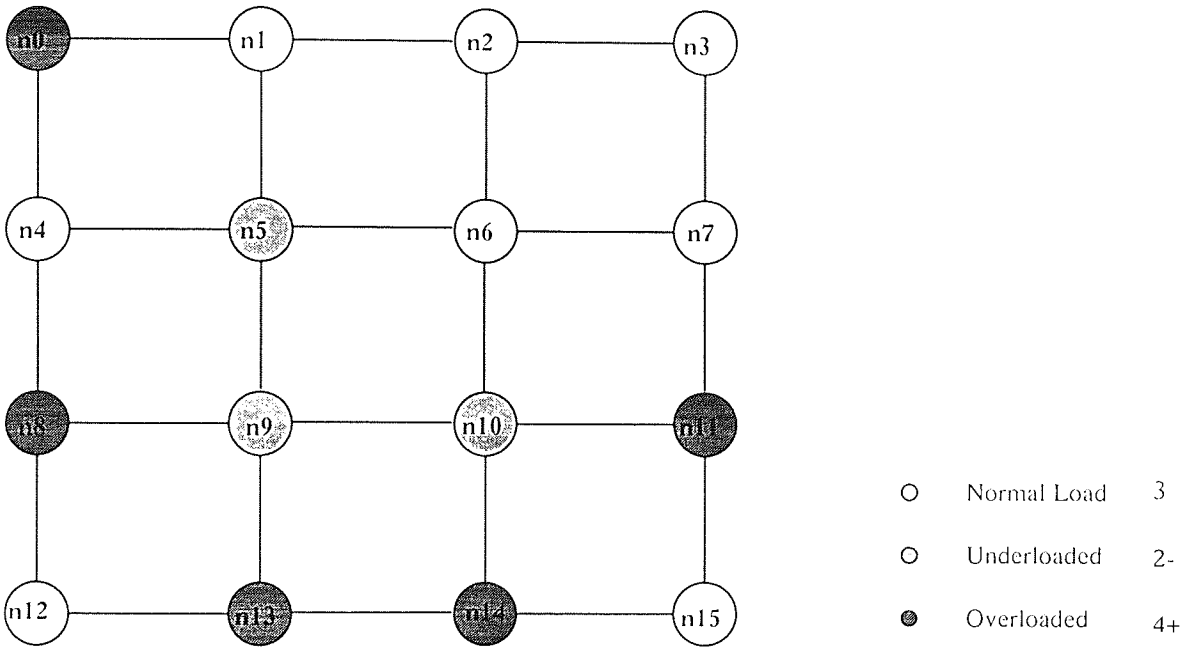


Figure 5.7. Workload Distribution Over An Extended Simulation Period

The detailed simulation traces were conducted and their output analysed for all the main load balancing protocols. Further message trace profiles for these algorithms can be found in Appendix A. All the algorithms exhibited the behaviour expected under various load conditions. Therefore, having resolved the issue of message routing and the integrity of the implemented load balancing protocols, the validation of the model could commence.

5.2 VALIDATION

In order to validate the system model, it is important that the model is able to reproduce the results obtained by other researchers in the field, such as Eager et al [Eager86], Zhou [Zhou88], and Johnson [Johnson88], given identical parameters for the system characteristics and the load sharing algorithms selected. The system considered consisted of nine loosely-coupled homogeneous processors configured in a mesh network. The performance measures of particular interest are the average variance in processor load and the average response time. The former will give an indication of the system stability and balance in the workload between sites, whilst the latter will indicate the effectiveness of the policy in pairing overloaded sites with underloaded sites. Thus, the successful pairing of sites should result in an overall improvement in system performance. The average processor load is also of interest as this may provide a context in which one can assess the accuracy of both fixed and variable threshold values.

The threshold values for the random and threshold algorithms were fixed at a level of two processes as this was found by other researchers, and confirmed in this study, to yield the best performance results over a range of system loads. In the case of the threshold algorithm, the probe limit was fixed at a maximum of three probe messages; thereafter a process considered for migration must run locally. The period for the global average algorithm (implemented by Johnson [Johnson88]) was fixed at 250 milliseconds, and the timeout period governing any alteration to the threshold level was based on the time required to complete the send-reply cycle discussed earlier. These values were found to yield the best overall performance.

Finally, each simulation run was terminated whenever the average response time for all sites converged to within less than two percent of each other. An average of 20 simulation runs was found to be sufficiently reliable for the purposes of analysis. The algorithms considered were threshold (*SndProbe*), random, global average broadcast (*GsndHop4*), and the no load balancing case (*NoBal*). The response time and load variance results produced by the model and illustrated in Figures 5.8 and 5.9 were similar to the findings of other studies, namely that systems undertaking load balancing activities outperformed those that do not across all system loads both in terms of the average response time and the equitable distribution of workload.

Even the random policy, widely regarded as the lowest common denominator for load balancing algorithms exhibited a better performance than the no load balancing case across all system loads. For example, the improvement in performance of the random policy over the no load balancing case at 50% and 80% system loading, varied between 25% and 45% respectively.

The results illustrated in Figures 5.8 and 5.9 also demonstrate a strong correlation between load variance and the run time performance of the algorithms considered. Thus, low load variance generally implies a corresponding improvement in response time and, therefore, the relative ranking of the algorithms are similar. It is clear from Figure 5.9 that at a 90% system loading, only the global average and threshold algorithms appear to retain a reasonable level of system stability whilst the random policy exhibits an exponential growth rate in load variance.

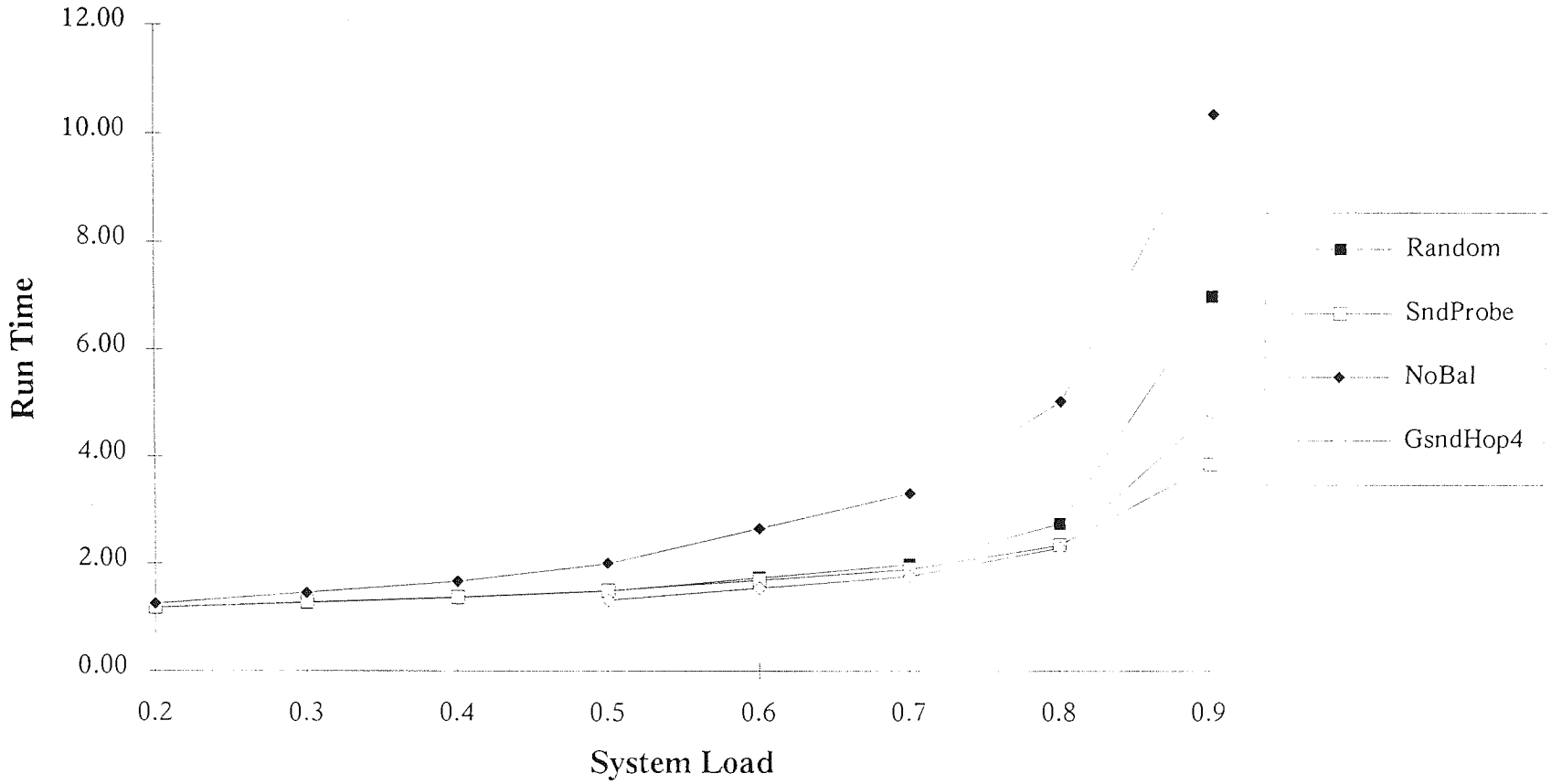


Figure 5.8. Average Response Time Using Independent Processes

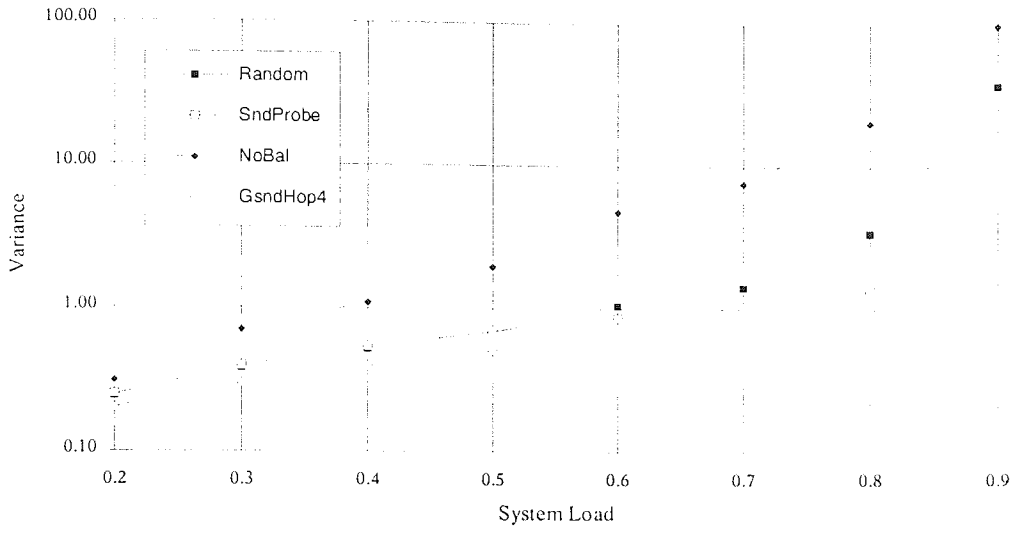


Figure 5.9. Average Load Variance Using Independent Processes

In Tables 5.1-5.3 the performance results of the model used in this study is represented alongside the published results (enclosed in parenthesis) found in the study conducted by Johnson [Johnson88] for a range of system loads. The tables show that for the three measures selected (mean, variance, and difference in workload) the results produced are of a similar magnitude and range to those recorded by Johnson with an almost identical ranking [Johnson88]. At very low system load, shown in Table 5.1, the more complex global average algorithm exhibited better overall load distribution profile than the much simpler random and threshold algorithms. Given that the resulting mean load for the algorithms considered is approximately one process, the simpler algorithms which make use of a fixed threshold value of two, will undertake load sharing less often as each host must have a minimum of two processes. In contrast, the variable threshold of the global average algorithm does mean that a host with two processes can share that workload with an idle host.

	Load Variance	Mean Load	Load Difference
Random	0.246 ± 0.004 *(0.219)*	1.236 ± 0.003 *(1.204)*	1.121 ± 0.010 *(1.010)*
Threshold	0.249 ± 0.004 *(0.214)*	1.236 ± 0.004 *(1.204)*	1.128 ± 0.013 *(1.01)*
NoBal	0.311 ± 0.01 *(0.441)*	1.250 ± 0.004 *(1.309)*	1.255 ± 0.021 *(1.160)*
Global Average	0.202 ± 0.01 *(0.227)*	1.220 ± 0.004 *(1.204)*	0.980 ± 0.02[6] *(1.09)*

Table 5.1 Overall System Behaviour for the 3-by-3 Mesh Model (Load 20%)

In Table 5.2, showing the results produced under moderate system loads, the performance improvement is more marked in systems where load balancing is active. However, it is noticeable that as in Table 5.1, the random policy has a marginally better performance than the threshold policy. This success is due to two factors: firstly, at low and moderate system load there is a high probability that a randomly selected remote host will be underloaded, therefore its location policy does not have the communication overheads that characterise the threshold policy; secondly, the selection policy implemented will only allow a single migration for any newly created process. Thus, the migration overhead is kept to a level similar to that of the threshold policy.

A mean load of around two processes allows a greater amount of load sharing for all the algorithms. However, as in Table 5.1, the ability of the global average algorithm to share processes during the peaks and troughs in workload, resulted in better overall performance. It has been argued by some researchers that the improvement in performance achieved by the more complex algorithms at low and moderate system load was marginal and more than outweighed by the overheads of its implementation.

	Load Variance	Mean Load	Load Difference
Random	0.687 ± 0.009 *(0.693)*	1.739 ± 0.008 *(1.769)*	2.034 ± 0.016 *(1.970)*
Threshold	0.696 ± 0.008 *(0.609)*	1.743 ± 0.007 *(1.742)*	2.065 ± 0.014 *(1.930)*
NoBal	1.966 ± 0.106 *(2.107)*	2.000 ± 0.019 *(2.091)*	3.553 ± 0.078 *(3.160)*
Global Average	0.506 ± 0.016 *(0.694)*	1.660 ± 0.006 *(1.750)*	1.720 ± 0.021 *(1.950)*

Table 5.2 Overall System Behaviour for the 3-by-3 Mesh Model (Load 50%)

However, at high system loads (shown in Table 5.3) the random policy yielded a significantly poorer performance as there is a greater likelihood of its location policy selecting sites that are already heavily loaded. In contrast, the performance of the threshold and global average policies are significantly better with the latter giving a consistently lower variation in workload between sites. Nevertheless, the relative success of the threshold policy is attributable to one major factor, namely the reduced level of process migration that takes place. No migration takes place unless a receiving host is willing to accept the process in response to a probe message. Furthermore, the processes that are eligible for migration are those with the longest remaining service times. Such processes, which have not been started on the local host are more likely to benefit from migration. In addition, as a process can only be migrated once, the process will have a "higher priority" than the newly created processes on the selected remote host.

In the case of the global average algorithm the variable threshold policy manifests inherent limitations on its performance potential at high system loads. Whilst the policy ensures that each host has a local workload in range of the average system workload, it promotes many more migrations especially of processes whose

service time may be far less than the sum of the process transfer and the run queue service time of the local host. The lack of real performance is even more marked (relative to the threshold policy) in cases where processes are selected randomly from the run queue without due regard for their remaining service times.

	Load Variance	Mean Load	Load Difference
Random	3.429 ± 0.376	3.187 ± 0.069	4.898 ± 0.213
	(4.188)	(3.138)	(4.320)
SndProbe	1.355 ± 0.052	2.866 ± 0.043	3.202 ± 0.057
	(1.449)	(2.817)	(3.260)
NoBal	20.356 ± 2.164	5.050 ± 0.157	12.163 ± 0.622
	(21.059)	(4.983)	(10.310)
Global Average	0.999 ± 0.02	2.826 ± 0.043	2.754 ± 0.028
	(1.208)	(2.893)	(2.980)

Table 5.3 Overall System Behaviour for the 3-by-3 Mesh Model (Load 80%)

The results presented so far lend support to the proposition of Eager et al [Eager86] that simple load balancing algorithms such as the random and threshold policies perform at least as well as more complex global average algorithms and should therefore be the algorithm of preference given the low overheads imposed and the simplicity of their implementation. However, whilst it may be impossible to justify complex algorithms for small distributed systems such as a three-by-three mesh, larger systems may present a plausible case for them as well as for newer and, perhaps less complex, algorithms. Furthermore, it is important that simple algorithms remain stable under extreme load conditions which, on the basis of the three-by-three mesh, is certainly not the case for the random policy at high system loads. The remaining sections of this study extends the analysis of the standard load balancing algorithms studied by other researchers both in terms of their scalability for systems consisting of 16 or more host processors, and their

relative ranking. The results presented will attempt to highlight some key issues in this regard.

CHAPTER 6

EXPERIMENTAL RESULTS

6.1 LIGHT-WEIGHT INDEPENDENT PROCESS MODEL

A flexible load sharing algorithm needs to be: adaptable and scalable across all system sizes, stable across all system loads, and induce minimum overhead and maximum performance from the system. These properties are very much dependent on the ability of the algorithm to avoid poor process allocation decisions given little or no state information on processing activity within the system. In addition to the average variance in processor load and the average response time, this study also considers the hit ratio suggested by Kremien et al [Kremien92], and defined as the ratio of remote execution requests concluded successfully. Further, the relative run-time performance improvement of a load balancing algorithm is represented as a percentage by the formula:

$$Speedup(x,k) = 100 * \frac{\mathrm{T}k - \mathrm{T}x}{\mathrm{T}k} \tag{6.1}$$

where x represent the load balancing algorithm under consideration, k represent the baseline (in this case no load balancing), and T_x and T_k are their respective average response times. Given that the performance differences between algorithms were minimal at very low system loads (20%), this study focused (in later sections) on their performance under moderate (50%), heavy (80%), and extreme (90%) load conditions.

The load balancing algorithms considered consisted of a range of sender (*Snd*), and receiver (*Rcv*) initiated implementations. These categories are then sub-divided according to the distance over which state information is collected, from the most

immediate neighbour (*Nbor*) to the complete network (*Hop4* in the case of the nine-processor topology). Algorithms such as threshold (*Tsnd*), threshold neighbour (*TsndNbor*), and threshold broadcast (*TsndBcast*) will accept and reply to the first positive reply received as they do not retain, neither maintain, state information for the nodes interrogated. In contrast, the more complex algorithms such as the global average (prefixed by the letter *G*) and the communicating set models (letter *C* prefix) attempt to keep track of system state. The sender-initiated and receiver-initiated global average algorithms will be referred to as *GsndBcast* and *GrcvBcast* respectively. The global average algorithms do not retain state information on any of the nodes in the network but attempt to ensure that all nodes are within range of the average system load using appropriate message *request*, *reply*, and *timeout* sequences. In the case of the communicating set model, state information is retained for a restricted set of nodes which may be updated either through the broadcast of periodic state information (*Cp*), or as a by-product of any information exchange events between hosts (*Ce*). Therefore, an event-driven communicating set implementation of the sender and receiver initiated policies are referred to as *Tsnd_Ce* and *Trcv_Ce* respectively.

6.1.1 Nine-Processor Mesh Topology

The experimental environment consisted of an interconnected mesh of nine homogenous processors with communication channel speeds of 10Mbits per second. Other parameters included the raw CPU speed for each processor of one MIPs, and protocol stack processing overheads of 1000 machine instructions for every transmitted message. The workload consisted of lightweight independent, cpu-intensive processes with average response times of one second. In all cases, each simulation run was terminated whenever the average response time for all sites converged to within less than two percent of each other.

The parameters of the fixed threshold policies were set to the optimum values given in the literature. This is typified by the value two for the threshold level, and an activation period of 250ms in the case of the global average algorithm. Further, an average of 20 simulation runs allowed a statistical analysis to be conducted.

The three-by-three mesh was the smallest configuration used in the experiments conducted. The results obtained for the algorithms are tabulated in tables 6.1 to 6.2. Table 6.1 shows the ranked performance of the algorithms for a lightly loaded system. Using lightweight processes with an average service time of one second, the system is said to be lightly loaded as the average arrival rate is one process per second. Thus, it is possible that at any instant in time one of the hosts will be busy whilst eight others are idle. In Table 6.1, it is clear that the global average algorithms (*Gsnd* and *Grcv*) in many instances produced the best performances both in terms of the overall response time and the redistribution of workload. This is primarily due to their ability to share workload below the fixed threshold level of two processes common to many of the other algorithms.

As evidenced in Table 6.1 the sender-initiated algorithms are more successful than their receiver-initiated counterparts. These include *GsndBCast*, *TsndBCast*, and *Tsnd_CeBCast*. In addition, algorithms which limit the distance over which transfers are negotiated tend to yield a better performance than those where distance is unrestricted. For example, algorithms such as *GsndNbor*, *GrcvNbor*, and *TsndNbor* produced better response times than *GsndHop2*, *GrcvBCast*, and *TsndBCast* respectively.

The success of the sender-initiated algorithms is mainly due to the minimisation of the communication overheads in negotiating process placement. Such negotiations only takes place when a new process arrives. Furthermore, given the greater number of

idle hosts in the system, there is a very high probability that such negotiations will end successfully. This also favours algorithms that restrict the distance over which negotiation can take place, as there is also a greater likelihood of finding an underloaded host locally. It is also true to say that, under light system loads, the communication costs involved in finding and transferring a process to a remote location will far outweigh the benefits.

The algorithm *GsndNbor* produced the best overall performance with the best response time and the lowest load variance values of *1.055* seconds and *0.181* processes respectively. In addition, its hit ratio of about two messages for every process migration is amongst the highest. This success is attributable to factors mentioned previously, namely sender-initiated, and constrained negotiations. In contrast, the response time and load variance figures for *GsndBCast* (broadcast implementation) are *1.087* seconds and *0.202* processes respectively. However, the performance difference is more marked in the case of the receiver-initiated implementation, *GrcvBCast*. Whilst the load variances of the global average algorithms are of a similar magnitude, only *GrcvBCast* (with average response time of *1.208* seconds) produced a poorer speedup improvement of approximately 3%.

It is clear from the results produced at low system load, that the underloaded hosts in a receiver-initiated load sharing strategy were less successful at finding overloaded sites. This is typified by the very low hit ratio of *GrcvBCast* which required around 526 messages for every successful migration. The problem is further compounded in the case of the global average algorithms where underloaded sites incorrectly assume that the lack of success may be the result of a high load threshold and therefore broadcast new threshold values. The reasons why the system does not become unstable in this case is due to the low level of process activity and the fact that the variable load threshold cannot fall below one process.

	Variance	MeanLd	Difference	RunTime	Mig/s	Tx/s	Hit Ratio	SpeedUp
GsndNbor	0.181 ± 0.003	1.213 ± 0.003	0.920 ± 0.011	1.055 ± 0.005	0.037 ± 0.002	0.302 ± 0.017	2.074 ± 0.084	15.600 ± 1.265
GsndHop2	0.191 ± 0.007	1.216 ± 0.004	0.948 ± 0.018	1.066 ± 0.006	0.034 ± 0.002	0.773 ± 0.041	5.624 ± 0.236	14.900 ± 1.370
GrcvNbor	0.184 ± 0.003	1.217 ± 0.004	0.931 ± 0.013	1.073 ± 0.005	0.037 ± 0.002	16.585 ± 0.086	112 ± 6.	14.286 ± 1.204
GsndBCast	0.202 ± 0.010	1.220 ± 0.004	0.980 ± 0.026	1.087 ± 0.015	0.031 ± 0.003	1.297 ± 0.114	10.391 ± 0.483	13.083 ± 1.730
GrcvHop2	0.185 ± 0.003	1.226 ± 0.004	0.931 ± 0.010	1.123 ± 0.002	0.042 ± 0.002	53.596 ± 0.294	326 ± 19	10.375 ± 1.204
Tsnd_CeProbe	0.251 ± 0.005	1.238 ± 0.003	1.135 ± 0.015	1.181 ± 0.004	0.007 ± 0.000	0.059 ± 0.005	2.080 ± 0.121	5.700 ± 1.160
Random	0.246 ± 0.004	1.236 ± 0.003	1.121 ± 0.010	1.177 ± 0.009	0.007 ± 0.001	0.028 ± 0.002	1.000	5.950 ± 0.686
TsndHop2	0.248 ± 0.005	1.241 ± 0.004	1.132 ± 0.015	1.177 ± 0.009	0.007 ± 0.001	0.139 ± 0.012	4.985 ± 0.182	5.850 ± 1.137
TsndNbor	0.249 ± 0.005	1.238 ± 0.004	1.132 ± 0.015	1.177 ± 0.009	0.007 ± 0.001	0.051 ± 0.004	1.834 ± 0.068	5.900 ± 1.071
TsndBCast	0.247 ± 0.005	1.243 ± 0.004	1.129 ± 0.015	1.178 ± 0.008	0.007 ± 0.001	0.239 ± 0.021	8.563 ± 0.286	5.800 ± 1.005
Tsnd_Probe	0.249 ± 0.004	1.236 ± 0.004	1.128 ± 0.013	1.180 ± 0.009	0.007 ± 0.000	0.057 ± 0.004	2.016 ± 0.095	5.700 ± 0.657
Tsnd_CeNbor	0.251 ± 0.006	1.240 ± 0.005	1.134 ± 0.018	1.184 ± 0.014	0.007 ± 0.001	0.040 ± 0.003	1.474 ± 0.095	5.200 ± 2.150
Tsnd_CpNbor	0.258 ± 0.016	1.252 ± 0.019	1.154 ± 0.043	1.203 ± 0.006	0.008 ± 0.001	10.428 ± 0.007	347 ± 36	3.800 ± 1.229
Tsnd_CeBCast	0.261 ± 0.028	1.262 ± 0.035	1.165 ± 0.077	1.206 ± 0.054	0.007 ± 0.001	0.158 ± 0.017	5.516 ± 0.325	3.700 ± 4.620
GrcvBCast	0.194 ± 0.003	1.243 ± 0.004	0.956 ± 0.011	1.208 ± 0.001	0.045 ± 0.002	93.601 ± 0.464	526 ± 33.	3.400 ± 1.506
Trcv_CeBCast	0.276 ± 0.007	1.246 ± 0.004	1.189 ± 0.020	1.223 ± 0.009	0.005 ± 0.000	5.980 ± 0.053	312. ± 23.	2.400 ± 1.506
Trcv_BCast	0.274 ± 0.005	1.245 ± 0.003	1.186 ± 0.010	1.225 ± 0.007	0.005 ± 0.000	6.099 ± 0.042	319 ± 26	2.200 ± 1.476
Trcv_CeNbor	0.290 ± 0.007	1.247 ± 0.004	1.218 ± 0.021	1.227 ± 0.010	0.003 ± 0.000	1.003 ± 0.008	87.870 ± 8.335	1.900 ± 1.595
TrcvNbor	0.286 ± 0.005	1.245 ± 0.003	1.210 ± 0.010	1.228 ± 0.008	0.003 ± 0.000	1.009 ± 0.008	88.5 ± 10.97	1.800 ± 1.476
NoBal	0.311 ± 0.010	1.250 ± 0.004	1.255 ± 0.021	1.250 ± 0.014				
Trcv_CpNbor	0.299 ± 0.006	1.255 ± 0.003	1.239 ± 0.015	1.261 ± 0.012	0.003 ± 0.000	11.380 ± 0.008	956 ± 95	-0.700 ± 1.889
Tsnd_CpProbe	0.281 ± 0.003	1.271 ± 0.003	1.217 ± 0.012	1.340 ± 0.007	0.009 ± 0.001	54.416 ± 0.011	1581 ± 93.	-7.100 ± 1.524
Tsnd_CpBCast	0.305 ± 0.038	1.320 ± 0.057	1.278 ± 0.091	1.386 ± 0.081	0.010 ± 0.001	61.241 ± 0.030	1615 ± 211	-10 ± 5.9

Table 6.1 Performance Results for the 3-by-3 Processor Mesh Model (Load 20%)

It is the more restricted algorithms *GrcvNbor*, and *GrcvHop2* which produced speedup performance comparable to the sender-initiated global average algorithms, namely 14% and 10% respectively. Furthermore, their load variances are comparable to the broadcast implementation and lend support to the proposition that, at low system load, any variation in workload tends to be localised.

In the case of the fixed threshold implementation, Table 6.1 shows that the communicating set implementations (*Ce* and *Cp*) performed worse than their equivalent non-communicating set versions. Further, with the exception of the nearest neighbour implementations, algorithms which rely on the periodic updating of the communicating set performed worse than the no load balancing case and are characterised by a greater amount of message traffic with a correspondingly poorer hit ratio. It is this message traffic overhead which has degraded performance. The period selected for broadcasting state information could be increased in order to reduce this communication overhead. However, the consequence of such a change are three-fold: firstly, the state information is likely to be less timely; secondly, the communication costs are less significant for the nearest neighbour implementation; and thirdly, the equivalent event-based communicating set algorithms (such as *Trcv_CeBCast* and *Tsnd_CeBCast*) represents the performance baseline on which a communicating set model could operate.

The speedup values for the algorithms *TsndNbor*, *Tsnd_CeNbor*, and *Tsnd_CpNbor* are around 6%, 5%, and 4% respectively. Although the difference in performance is quite marginal, the results produced by the communicating set implementations also highlights the possibility that at low system loads there is a high degree of unpredictability regarding the accuracy of the state information maintained. For example, as the overall system load is often below the threshold level for long periods, the information on individual sites may be inaccurate in the case of sender-initiated

algorithms and as such, these sites will alternate more frequently between load states. Alternatively, this state information may be highly accurate in the case of the receiver-initiated algorithms, but is characterised by communicating sets that are frequently empty. The net result is that more expensive message exchanges must take place in order to rebuild the sets.

The results in Table 6.1 also shows that the random policy performed as well as, and in some cases better than, many of the other fixed threshold algorithms. It has the best hit ratio given that no host can refuse to accept any process it wishes to send, and delivers a speedup performance of around 6%. Given the overheads of load balancing and preemptive process migrations, Kremien et al [Kremien92] argues that any policy that relies on negotiating the migration of processes can only be justified by showing at least a 10% improvement in performance over the random policy and the no load balancing case. At light system load, only the global average algorithms demonstrated performance of this magnitude and many of the other algorithms would not be considered. Further, it is notable that the preemptive global average algorithms produced an average of three migrations every 100 seconds compared to one migration for the fixed threshold receiver-initiated and non-preemptive sender-initiated algorithms. Given the greater overheads of the preemptive migrations, it is only reasonable to reject the other algorithms considered if they are consistently outperformed across a range of system loads.

The results of the experiments conducted at moderate system loads, where the average arrival rate is around four processes per second, is shown in Table 6.2. It is clear from these figures that all the algorithms demonstrated improved run-time performance over the no load balancing case with speedups varying between 6% and 37%.

	Variance	MeanLd	Difference	RunTime	Mig/s	Tx/s	Hit Ratio	SpeedUp
GsndHop2	0.438 ± 0.009	1.623 ± 0.007	1.594 ± 0.019	1.244 ± 0.010	0.210 ± 0.004	5.854 ± 0.098	6.968 ± 0.098	37.200 ± 1.033
GsndNbor	0.476 ± 0.009	1.633 ± 0.008	1.675 ± 0.022	1.261 ± 0.010	0.191 ± 0.003	1.955 ± 0.036	2.563 ± 0.029	36.300 ± 1.252
GrcvHop2	0.401 ± 0.005	1.630 ± 0.009	1.527 ± 0.013	1.267 ± 0.009	0.231 ± 0.005	34.055 ± 0.298	36.941 ± 1.145	36.200 ± 1.033
GrcvNbor	0.493 ± 0.009	1.647 ± 0.010	1.714 ± 0.020	1.299 ± 0.010	0.192 ± 0.002	12.275 ± 0.064	16.051 ± 0.325	34.600 ± 1.350
GrcvBCast	0.390 ± 0.007	1.649 ± 0.010	1.512 ± 0.021	1.307 ± 0.011	0.244 ± 0.005	54.778 ± 0.438	56.280 ± 1.657	34.300 ± 1.418
GsndBCast	0.506 ± 0.016	1.660 ± 0.006	1.720 ± 0.021	1.315 ± 0.011	0.196 ± 0.004	11.083 ± 0.257	14.147 ± 0.237	33.800 ± 1.229
TsndNbor	0.667 ± 0.017	1.751 ± 0.011	2.010 ± 0.024	1.435 ± 0.012	0.096 ± 0.003	0.723 ± 0.025	1.881 ± 0.018	27.850 ± 1.182
Tsnd_CeProbe	0.631 ± 0.007	1.720 ± 0.010	1.957 ± 0.014	1.447 ± 0.011	0.097 ± 0.003	0.900 ± 0.031	2.332 ± 0.032	26.889 ± 1.269
Tsnd_BCast	0.653 ± 0.009	1.824 ± 0.011	1.997 ± 0.013	1.448 ± 0.010	0.096 ± 0.003	3.347 ± 0.114	8.737 ± 0.084	27.200 ± 1.196
Tsnd_CpNbor	0.640 ± 0.010	1.734 ± 0.009	1.977 ± 0.015	1.465 ± 0.008	0.089 ± 0.003	10.889 ± 0.017	30.568 ± 0.837	26.100 ± 1.524
Tsnd_CeNbor	0.698 ± 0.017	1.736 ± 0.012	2.066 ± 0.024	1.477 ± 0.009	0.077 ± 0.002	0.504 ± 0.016	1.633 ± 0.028	25.700 ± 1.252
Tsnd_CeHop2	0.692 ± 0.015	1.735 ± 0.012	2.062 ± 0.023	1.477 ± 0.010	0.076 ± 0.002	1.151 ± 0.037	3.767 ± 0.083	25.600 ± 1.647
Random	0.687 ± 0.009	1.739 ± 0.008	2.034 ± 0.016	1.481 ± 0.009	0.104 ± 0.003	0.416 ± 0.011	1.000	25.737 ± 1.046
Tsnd_CeBCast	0.693 ± 0.015	1.737 ± 0.013	2.062 ± 0.023	1.482 ± 0.011	0.076 ± 0.002	1.843 ± 0.070	6.048 ± 0.091	25.400 ± 1.430
Tsnd_Probe	0.696 ± 0.008	1.743 ± 0.007	2.065 ± 0.014	1.485 ± 0.010	0.104 ± 0.003	0.846 ± 0.028	2.048 ± 0.018	25.300 ± 0.923
Trcv_CeBCast	0.775 ± 0.008	1.783 ± 0.011	2.199 ± 0.014	1.573 ± 0.012	0.085 ± 0.003	12.326 ± 0.087	36.616 ± 1.051	20.556 ± 1.590
Trcv_CeNbor	0.959 ± 0.017	1.812 ± 0.013	2.464 ± 0.029	1.630 ± 0.016	0.064 ± 0.002	2.285 ± 0.023	8.958 ± 0.282	17.778 ± 1.641
Trcv_CpNbor	0.972 ± 0.019	1.836 ± 0.009	2.492 ± 0.027	1.668 ± 0.018	0.068 ± 0.003	12.571 ± 0.016	46.159 ± 1.569	15.889 ± 1.616
Tsnd_CpProbe	0.690 ± 0.008	1.841 ± 0.010	2.078 ± 0.013	1.681 ± 0.008	0.119 ± 0.003	53.831 ± 0.024	112.911 ± 2.562	15.200 ± 1.476
Tsnd_CpBCast	0.724 ± 0.009	1.860 ± 0.011	2.147 ± 0.016	1.717 ± 0.017	0.104 ± 0.003	61.348 ± 0.041	148.148 ± 3.194	13.600 ± 1.713
Trcv_CpBCast	0.850 ± 0.011	1.932 ± 0.013	2.347 ± 0.017	1.863 ± 0.016	0.110 ± 0.004	69.901 ± 0.079	158.680 ± 5.164	6.100 ± 1.729
NoBal	1.966 ± 0.106	2.000 ± 0.019	3.553 ± 0.078	1.991 ± 0.027				

Table 6.2 Overall System Behaviour for the 3-by-3 Mesh Model (Load 50%)

Compared to the random policy, the relatively poorer performance figures for many of the periodic state gathering communicating set algorithms (in particular *Trcv_Cp*, and *Tsnd_Cp*) continues to reflect their excessive communication overheads. This is especially the case for the broadcast algorithms *Trcv_CpBCast* and *Tsnd_CpBCast* where, on the basis of the hit ratio, every successful remote execution required the exchange of at least 140 load-related messages. A brief examination of the threshold algorithms will provide further insight into the effectiveness of the communicating set algorithms. These algorithms are: *Tsnd_Probe* which randomly selects and probes a remote host; *Tsnd_CeProbe* which probes in turn, its set of remote sites with the least process loads, where the state information used is a by-product of the most recent load sharing dialogue between both sites; and *Tsnd_CpProbe* which only probes those sites with the least process loads at the time of their last periodic state broadcast

From Table 6.2, it is clear that the performance speedup of 25% and 27% for the threshold algorithm (*TsndProbe*) and its event-based communicating set counterpart (*Tsnd_CeProbe*) respectively, are significantly better than the 15% speedup for the equivalent periodic state collecting implementation (*Tsnd_CpProbe*). The performance of *Tsnd_CpProbe* is hindered by the resulting increase in message traffic. However, the much improved performance of *Tsnd_CeProbe* implies that an overloaded host is more likely to find success in sharing its workload amongst those lightly loaded remote hosts that have previously accepted some of its workload. Thus, in contrast to the results produced at light system loads, the communicating set for an overloaded host is less likely to be empty, and more likely to possess one or more members whose load state will have changed relatively slowly. It may be argued that for the algorithms *Tsnd_CeProbe* and *Tsnd_Probe*, the former is characterised by marginally fewer migrations, more messages per second, and a lower hit ratio which reflects failure rather than success. However, such results are also a reflection of the quality of the decisions made. That is, the migrations that takes place results in

genuine performance improvement as the load state of the selected location becomes relatively stable over a period of time.

Generally, the regionalised non-communicating set implementations (from *TsndNbor* to *TsndBCast*) exhibited comparatively better load stability and run-time performance than their communicating set counterpart. These algorithms are characterised by a greater number of process migrations, where algorithms such as *TsndNbor* results in 10 process migrations every 100 seconds compared to nine processes for *Tsnd_CpNbor*, and eight processes for *Tsnd_CeNbor*. In the case of the communicating set implementations the results produced are a product of the general accuracy of the state information collected, where the periodic information is more accurate and results in a greater number of successful migrations. However, the improved performance is expensive in communication overhead as indicated by the very low hit ratio of 31 load related messages for every migration compared to about 2 messages for *Tsnd_CeNbor*.

Only the immediate neighbour implementation (*Tsnd_CpNbor*) produced the best performance for algorithms of this category with a speedup value of 26%. This latter result lends further support to the proposition made for the performance achieved under light system loads, namely that state information collected from the most immediate neighbour is generally more accurate, and load variations tend to be localised. Therefore, during periods of moderate system loads, an overloaded host will be just as likely to find an underloaded host amongst its most immediate neighbours. However, even where communication overheads are negligible, the marginal improvement in performance for *Tsnd_CeProbe* compared to *Tsnd_CpNbor*, is the result of the greater choice of underloaded sites, and the greater number of successful migrations.

Whilst all algorithms studied exhibited performance results well above those systems where load balancing is absent, it is evident that only the global algorithms are anywhere near the target set by Kremien et al for outperforming the random policy by as much as 10%. Clearly, the ability of the global average algorithms to adapt its threshold level according to prevailing load conditions has a beneficial effect on performance. In comparison with algorithms that makes use of fixed threshold values, the threshold level of the global average implementation varied between two and four processes, resulting in more equitable load distribution, and better response times. Further, algorithms based on locality (*GsndNbor* and *GsndHop2*) were characterised by a greater threshold range, a higher hit ratio, and better overall performance. At moderate system load, the communication overheads of the global broadcast algorithms (*GRcvBCast* and *GsndBCast*) clearly had an effect on their performance as the majority of load imbalance problems and solutions that occurs between two or more hosts are likely to be successfully identified and resolved locally.

In the case of the receiver-initiated implementations of global average, a by-product of their relatively low response time and load variance were the significantly higher amount of message traffic and the relatively poorer hit ratio. For example, *GrcvHop2* generated around 37 messages per remote execution compared to 7 messages for *GsndHop2*. However, the hit ratio has improved significantly on the results obtained when the system was lightly loaded. Again, these results are due to many underloaded sites sending requests for work to remote sites which are likely to be common to other sites, and receiving a reject-without-explanation message or no reply at all. Subsequently, an underloaded local site will time out, and assume the community threshold to be too high and, as a consequence, broadcast a lower threshold value. However, this activity occurs less frequently compared to the results produced at light system loads.

	Variance	MeanLd	Difference	RunTime	Mig/s	Tx/s	Hit Ratio	SpeedUp
GsndHop2	0.979 ± 0.023	2.682 ± 0.040	2.716 ± 0.036	2.106 ± 0.044	0.435 ± 0.005	13.609 ± 0.140	7.802 ± 0.059	58.700 ± 1.829
GsndNbor	1.376 ± 0.046	2.700 ± 0.038	3.154 ± 0.049	2.119 ± 0.041	0.410 ± 0.004	4.777 ± 0.070	2.907 ± 0.022	58.400 ± 1.897
Tsnd_CeProbe	1.376 ± 0.075	2.715 ± 0.028	3.163 ± 0.060	2.141 ± 0.036	0.304 ± 0.005	4.946 ± 0.082	4.065 ± 0.086	58.000 ± 1.333
GrcvHop2	0.955 ± 0.014	2.787 ± 0.039	2.699 ± 0.020	2.227 ± 0.043	0.447 ± 0.004	26.520 ± 0.084	14.832 ± 0.169	55.800 ± 1.476
GrcvNbor	1.431 ± 0.042	2.797 ± 0.034	3.244 ± 0.043	2.238 ± 0.050	0.414 ± 0.003	10.793 ± 0.041	6.510 ± 0.055	55.600 ± 1.506
GsndBCast	0.999 ± 0.020	2.826 ± 0.043	2.754 ± 0.028	2.284 ± 0.054	0.390 ± 0.003	22.256 ± 0.172	14.242 ± 0.098	55.000 ± 1.886
Trcv_CeHop2	1.727 ± 0.071	2.863 ± 0.027	3.607 ± 0.060	2.322 ± 0.037	0.289 ± 0.004	7.870 ± 0.115	6.807 ± 0.086	54.300 ± 1.418
Tsnd_Probe	1.355 ± 0.052	2.866 ± 0.043	3.202 ± 0.057	2.330 ± 0.045	0.429 ± 0.006	4.643 ± 0.125	2.704 ± 0.063	53.500 ± 1.433
Trcv_CeBCast	1.661 ± 0.071	2.883 ± 0.027	3.531 ± 0.065	2.344 ± 0.034	0.302 ± 0.005	12.668 ± 0.138	10.475 ± 0.179	53.800 ± 1.398
Tsnd_CpNbor	2.267 ± 0.188	2.890 ± 0.048	4.022 ± 0.126	2.356 ± 0.045	0.238 ± 0.003	12.432 ± 0.045	13.026 ± 0.177	53.800 ± 1.814
TsndNbor	2.677 ± 0.983	3.041 ± 0.106	4.181 ± 0.665	2.364 ± 0.151	0.352 ± 0.023	3.175 ± 0.096	2.258 ± 0.118	52.750 ± 3.210
Tsnd_CeNbor	2.699 ± 0.184	2.950 ± 0.038	4.417 ± 0.124	2.424 ± 0.047	0.190 ± 0.002	2.140 ± 0.026	2.806 ± 0.051	52.000 ± 1.563
GrcvBCast	0.853 ± 0.013	2.971 ± 0.058	2.581 ± 0.022	2.455 ± 0.069	0.426 ± 0.003	40.778 ± 0.183	23.907 ± 0.247	51.300 ± 1.829
Trcv_CeNbor	2.354 ± 0.116	2.977 ± 0.030	4.235 ± 0.075	2.463 ± 0.043	0.237 ± 0.003	2.863 ± 0.027	3.021 ± 0.030	51.600 ± 1.578
Tsnd_CeHop2	2.611 ± 0.150	2.986 ± 0.039	4.355 ± 0.107	2.473 ± 0.052	0.182 ± 0.002	5.375 ± 0.110	7.382 ± 0.196	51.000 ± 1.764
Trcv_CpNbor	2.172 ± 0.123	2.994 ± 0.044	4.059 ± 0.100	2.493 ± 0.049	0.264 ± 0.004	13.174 ± 0.017	12.456 ± 0.187	50.600 ± 2.221
Tsnd_CeBCast	2.929 ± 0.194	3.086 ± 0.047	4.590 ± 0.127	2.590 ± 0.056	0.173 ± 0.002	9.114 ± 0.177	13.164 ± 0.357	48.600 ± 2.119
Random	3.429 ± 0.376	3.187 ± 0.069	4.898 ± 0.213	2.732 ± 0.074	0.484 ± 0.009	1.935 ± 0.032	1.000	45.500 ± 1.792
Tsnd_BCast	5.022 ± 1.865	3.971 ± 0.137	5.514 ± 0.988	2.927 ± 0.274	0.340 ± 0.044	16.638 ± 0.581	12.403 ± 1.502	41.650 ± 5.678
Trcv_CpBCast	3.159 ± 0.402	3.834 ± 0.116	4.545 ± 0.238	3.541 ± 0.126	0.311 ± 0.006	65.994 ± 0.230	53.050 ± 0.919	30.500 ± 2.991
Tsnd_CpProbe	4.490 ± 0.800	3.849 ± 0.166	5.354 ± 0.399	3.550 ± 0.206	0.256 ± 0.008	57.498 ± 0.165	56.209 ± 1.933	29.700 ± 3.917
NoBal	20.356 ± 2.164	5.051 ± 0.157	12.163 ± 0.622	5.014 ± 0.177				
Tsnd_CpBCast	20.286 ± 7.894	5.831 ± 0.660	10.650 ± 1.669	5.851 ± 0.669	0.158 ± 0.012	70.099 ± 0.906	111.627 ± 9.817	-15.00 ± 13.275

Table 6.3 Overall System Behaviour for the 3-by-3 Mesh Model (Load 80%)

The performance of the algorithms under heavy load conditions is shown in Table 6.3. The majority of algorithms produced performance speedup results between 30% and 59%. Even the random policy produced speedup results of around 45%. The periodic state algorithm *Tsnd_CpBCast* was the only algorithm to yield a performance inferior to systems where load balancing is inoperative. Given the increased process load and general shortage of suitable sites, the load balancing algorithms need to minimise the state information collected and the number of migrations that takes place. Certainly, *Tsnd_CpBCast* lacks the ability to adapt to the increased workload, and its general performance reflects this weakness.

Of particular note is that, with the exception of the majority of periodic communicating set algorithms, other complex state collecting algorithms such as the event-based communicating set and global average algorithms exhibited significant performance improvement. The relative ranking of the three threshold algorithms *Tsnd_CeProbe, Tsnd_Probe* and *Tsnd_Cprobe*, remains the same although the performance differences are even more marked with approximate speedup values of 58%, 54%, and 28% respectively. Likewise, the communicating set implementations are characterised by a lower migration rate of three processes every ten seconds compared to four processes for *Tsnd_Probe*. The lower migration rate is welcome given the load conditions, but the message traffic overheads of *Tsnd_Cprobe* accounts for its poor performance. Further, given that many more sites are overloaded, thus increasing the level of load negotiation activity in the system, the state information of *Tsnd_CeProbe* is likely to be at least as accurate as *Tsnd_Cprobe*.

In terms of the sender-initiated, fixed threshold regional algorithms such as *Tsnd_CpNbor, Tsnd_CeHop2* and *TsndNbor*, only the nearest neighbour implementations continue to display a similar performance pattern to that produced under moderate system load. Multiple requests over greater distance under heavy load

conditions would seem to result in poorer decision-making, evidenced by the relatively low hit-ratio, which in turn is the consequence of state information that is out of date. Furthermore, when compared to their nearest-neighbour counterpart, it is clear that such algorithms are grossly inefficient in cases where a local host with only one process above the load threshold has multiple requests sent on its behalf to remote sites outside their immediate vicinity. This is in marked contrast to the regional global average algorithms where the variable threshold level is a significant factor in their general performance.

The global average algorithms continue to perform well under heavy load conditions both in terms of load redistribution and response time. In Table 6.3 it can be seen that for the regional algorithms, a regional distance of two hops produced better results than for those over greater distances or implementations based on the nearest neighbour. As mentioned earlier greater distances imply better stability but longer delays for a decision-making local host. However, shorter distances results in greater load variance as a result of a more incomplete picture of system load, although it is possible to optimise the response times within the general locality. Thus, a distance of two hops is an appropriate compromise that allows a local host to view and respond to the perceived system state, and be influential during the establishment of common goals such as load targets.

It is interesting to note that the sender-initiated global average algorithms (*GsndNbor* and *GsndHop2*) performed better than their receiver-initiated counterparts (*GrcvNbor* and *GrcvHop2*) with speedup up performances of 58% and 55% respectively. The hit ratio of the receiver initiated global average algorithms would imply that at high loadings they are twice as likely to fail in their load balancing decisions. At high system loads there are many more overloaded sites and, with the receiver-initiated implementation, such sites are more likely to timeout if a request for work had not

been received resulting in the broadcast of a higher threshold value. It is unlikely that, given the contention amongst overloaded hosts during high system loads, that underloaded hosts will have the opportunity to influence the global state other than by responding directly to individual requests. Thus, the sender-initiated implementations are more likely to prosper in this environment.

The results also re-emphasise the relative success of the simple fixed threshold algorithm *Tsnd_Ceprobe* which performed as well as, and in some cases better than, the global average algorithms. The performance of *Tsnd_CeProbe* can also be explained in terms of the following: the greater success in filling the communicating set and pairing overloaded and underloaded sites; and the propensity of overloaded sites to remain in the overloaded state over a period of time. This further emphasises the earlier finding that an overloaded host when paired with an underloaded remote site will be successful in sharing their workload over a period of time rather than at any instant in time.

In Table 6.4 the performance of the algorithms are examined under extreme load conditions, where the average arrival rate is about eight processes per second. The average speedup in performance ranged from 19% for *TsndBCast* to 67% for *Trcv_CeHop2*. It is under such load conditions that the random policy can be seen for what it is, namely the lowest common denominator of performance attainable through load balancing. Although the average speedup is around 32%, the distribution of workload is significantly unstable with load variance of about 37 processes. Many more of the algorithms implemented exhibited better load stability and exceeded the performance speedup of the random policy by at least 10%.

	Variance	MeanLd	Difference	RunTime	Mig/s	Tx/s	Hit Ratio	SpeedUp
Trcv_CeHop2	4.867 ± 0.901	4.043 ± 0.133	5.713 ± 0.365	3.392 ± 0.155	0.307 ± 0.005	6.318 ± 0.111	5.153 ± 0.090	67.200 ± 3.584
Trcv_CeBCast	4.934 ± 1.012	4.124 ± 0.146	5.738 ± 0.395	3.476 ± 0.172	0.312 ± 0.006	9.826 ± 0.227	7.888 ± 0.136	66.200 ± 3.584
Trcv_CeNbor	6.235 ± 1.057	4.202 ± 0.137	6.587 ± 0.419	3.566 ± 0.161	0.258 ± 0.004	2.420 ± 0.045	2.350 ± 0.028	65.300 ± 3.592
Tsnd_CeProbe	6.250 ± 1.396	4.227 ± 0.211	6.228 ± 0.559	3.584 ± 0.233	0.268 ± 0.008	8.064 ± 0.175	7.525 ± 0.381	65.000 ± 4.110
GrcvNbor	2.371 ± 0.087	4.396 ± 0.220	4.228 ± 0.078	3.771 ± 0.251	0.497 ± 0.005	10.823 ± 0.052	5.451 ± 0.046	63.500 ± 4.552
Tsnd_Probe	4.404 ± 0.890	4.469 ± 0.201	5.289 ± 0.387	3.839 ± 0.182	0.439 ± 0.013	8.203 ± 0.220	4.679 ± 0.245	62.800 ± 2.331
TsndNbor	11.094 ± 4.106	4.774 ± 0.302	8.185 ± 1.225	3.982 ± 0.325	0.370 ± 0.014	4.546 ± 0.057	3.082 ± 0.156	61.500 ± 2.877
GsndNbor	11.094 ± 4.106	4.774 ± 0.302	8.185 ± 1.225	3.982 ± 0.325	0.370 ± 0.014	4.546 ± 0.057	3.082 ± 0.156	61.500 ± 2.877
Tsnd_CeNbor	11.546 ± 1.883	4.754 ± 0.211	8.801 ± 0.616	4.147 ± 0.207	0.169 ± 0.005	3.298 ± 0.058	4.883 ± 0.209	59.800 ± 4.104
GsndHop2	2.183 ± 0.099	5.130 ± 0.328	4.187 ± 0.101	4.373 ± 0.319	0.423 ± 0.006	10.871 ± 0.069	6.424 ± 0.086	57.600 ± 4.766
Tsnd_CeHop2	14.891 ± 3.520	5.293 ± 0.374	9.820 ± 1.004	4.702 ± 0.346	0.148 ± 0.007	9.455 ± 0.361	16.042 ± 1.327	54.400 ± 5.317
GsndBCast	1.635 ± 0.059	5.590 ± 0.425	3.584 ± 0.067	4.818 ± 0.426	0.399 ± 0.007	15.438 ± 0.063	9.670 ± 0.147	53.400 ± 6.041
TsndHop2	17.706 ± 3.732	6.180 ± 0.374	10.500 ± 1.100	5.256 ± 0.395	0.346 ± 0.017	14.405 ± 0.215	10.441 ± 0.647	49.300 ± 4.001
GrcvHop2	1.422 ± 0.047	6.087 ± 1.004	3.395 ± 0.059	5.527 ± 0.899	0.497 ± 0.003	25.891 ± 0.051	13.033 ± 0.091	46.800 ± 8.626
Tsnd_CeBCast	33.483 ± 16.93	6.970 ± 0.914	13.973 ± 2.676	6.422 ± 0.815	0.117 ± 0.008	17.956 ± 0.874	38.661 ± 4.738	38.000 ± 8.420
Random	36.910 ± 11.08	7.318 ± 0.635	15.265 ± 1.908	6.976 ± 0.613	0.761 ± 0.017	3.043 ± 0.070	1.000	32.450 ± 4.740
Tsnd_BCast	51.612 ± 24.07	9.104 ± 1.123	17.568 ± 3.563	8.406 ± 1.215	0.256 ± 0.025	26.877 ± 0.362	26.486 ± 2.748	18.700 ± 12.6
NoBal	98.431 ± 21.44	10.442 ± 0.749	26.871 ± 2.905	10.344 ± 0.728				

Table 6.4 Overall System Behaviour for the 3-by-3 Mesh Model (Load 0.9)

The results in Table 6.4 also illustrate the predominance of receiver-initiated, fixed threshold algorithms with *Trcv_CeHop2* and *Trcv_CeBCast* producing speedup performances of 67% and 66% respectively. These algorithms dominate as load balancing activity is initiated and controlled by a small number of underloaded sites. The communicating set is updated each time a local host is probed by an overloaded host. Given that excessive load conditions prevail, the local host will possess a complete communicating set. Thus, whenever the load of the local host falls below the fixed threshold, it will send a request for work to all the members of its communicating set.

The success of this strategy is confirmed by improved response time, but extreme load conditions will mean that a fixed threshold value that is too low will result in marked inequality in workload between sites of at least four processes. In particular, the fixed threshold value of two is well below the average system load and this results in fewer migrations, such as three processes per second. The poor load distribution of *Trcv_CeNbor* is due to a very limited set (nearest-neighbour) of potential load sharing partners, compared to *Trcv_CeBCast* which makes use of message broadcast over a much greater distance.

The ranking of the threshold algorithms *Tsnd_CeProbe*, *Tsnd_Probe*, and *Tsnd_Cp* remained the same at 90% system loading. However, this is purely on the basis of response times. Whilst it was the case that *Tsnd_CeProbe* delivered better load variance and response time than *Tsnd_Probe* at 80% loading, only the response time is better under extreme load conditions. In this case it is likely that the majority of overloaded sites have near-identical communicating set members. Thus, the likelihood of many local hosts reaching their maximum limits for rejections is high in comparison with *Tsnd_Probe*. This factor is borne out in the hit ratio for both algorithms of about eight and five messages respectively. Furthermore, the focus of

Tsnd_CeProbe on the most underloaded sites is likely to facilitate further load instability as such sites become the focal point of dialogue with the majority of overloaded sites.

The fixed threshold, receiver-initiated communicating set algorithms delivered the best response times but, with the exception of *GsndNbor*, the global average algorithms exhibited much better workload distribution. In particular, the load variances for *GrcvNbor*, *GsndHop2*, and *GsndBCast* were in the region of two processes. Unlike *GsndBCast*, the race condition developed in *GrcvBCast* (not shown). As a consequence of excessive network delay, the greater number of overloaded hosts will time out, update and re-broadcast a new load state. These frequent changes in the load state by different overloaded sites makes for an extremely unstable environment resulting in system saturation. This is particularly the case for message broadcast beyond the immediate neighbours of the local host. It is possible to minimise the race condition over longer distances by increasing the timeout period. However, the management of an adaptive timeout period would introduce further complexity to an already complex algorithm.

The race condition was not evident in *GsndBCast* as the smaller number of underloaded sites were regularly probed and had a lower threshold limit of one process. What is clear from the results produced so far for moderate and heavy system loads is the durability of a global average algorithm based on a regionalised load sharing strategy where the send-receive cycle delay is small and changes in the load threshold have a ripple effect through the network as each local host shares its workload with neighbouring hosts.

6.1.2 16-Processor Mesh Topology

The larger 16-processor mesh topology may well affect the performance of some algorithms in a number of ways and thus alter the relative ranking obtained for smaller mesh networks. For example, implementations that rely on broadcasting requests to all remote sites will almost double the message traffic as each local host will now target 15 (as oppose to eight) remote hosts. In the case of the threshold probe algorithm (*Tsnd_Probe*), over 50% of remote sites will be at least two hops away from the requesting local host. Similarly, for the nearest-neighbour implementations, a greater proportion of hosts possess three or more links, whereas the opposite was true for the smaller mesh. The communicating set algorithms, given the size of the network may need to maintain larger sets and thus impose greater storage and processing overheads. The results presented for the 16-processor mesh, should provide further insight into the scalability of load balancing algorithms and the impact of network size on their performance and relative ranking.

The results at moderate system load for a 16-processor mesh is presented in Table 6.5 and exhibit a similar ranking to that of the nine-processor mesh. The relative ranking of the algorithms selected shows the global average implementations continuing to display a much better overall performance. Similarly, the threshold neighbour algorithms performed better than their communicating set counterparts both in terms of response time and load variance. It is notable that the receiver-initiated threshold algorithms, omitted in the nine-processor model, performed worse than the no load balancing case and significantly worse than the random policy. The explanation for such poor performance rests in the inability of these algorithms to locate overloaded sites. Further, their load balancing dialogue tends to be rather lengthy as an underloaded site, having found an overloaded remote host, must then request the transfer of the process. However, given the time delay between the request being

made and the process transfer being confirmed, the process on the remote host may no longer be available and the overloaded state cleared. The nature of the dialogue also affects the nearest-neighbour implementation as an overloaded host will be common to many underloaded sites. In contrast, the communicating set implementation of the receiver-initiated policy produced speedup factors of at least 12% over the no load balancing case. This clearly reflects the success of the communicating set policy in ensuring that an underloaded site successfully receives work from sites that have been overloaded in the recent past.

	Variance	MeanLd	RunTime	Hit Ratio	SpeedUp
GrcvHop2	0.404 ± 0.003	1.631 ± 0.006	1.260 ± 0.015	32.207 ± 0.828	36.800 ± 1.619
GrcvNbor	0.492 ± 0.006	1.643 ± 0.007	1.279 ± 0.019	12.845 ± 0.269	35.700 ± 1.567
GrcvHop4	0.395 ± 0.006	1.691 ± 0.010	1.388 ± 0.015	60.746 ± 2.169	30.300 ± 1.567
TsndNbor	0.688 ± 0.005	1.762 ± 0.006	1.424 ± 0.005	1.517 ± 0.011	28.350 ± 1.694
TsndHop2	0.777 ± 0.009	1.880 ± 0.008	1.432 ± 0.008	4.807 ± 0.037	27.950 ± 1.605
Tsnd_CeProbe	0.656 ± 0.008	1.727 ± 0.008	1.451 ± 0.018	2.324 ± 0.022	27.100 ± 1.969
SndHop4	0.818 ± 0.023	2.098 ± 0.017	1.461 ± 0.008	13.416 ± 0.098	26.550 ± 1.638
Tsnd_CpNbor	0.673 ± 0.008	1.732 ± 0.006	1.463 ± 0.009	13.771 ± 0.371	26.550 ± 1.538
Tsnd_CeNbor	0.711 ± 0.013	1.739 ± 0.010	1.470 ± 0.026	1.269 ± 0.011	26.400 ± 1.897
Tsnd_CeHop2	0.706 ± 0.010	1.739 ± 0.009	1.471 ± 0.019	3.205 ± 0.037	26.300 ± 1.829
Random	0.690 ± 0.009	1.738 ± 0.006	1.477 ± 0.008	1.000	25.800 ± 1.196
Ce_SndHop4	0.723 ± 0.012	1.747 ± 0.009	1.488 ± 0.024	7.541 ± 0.096	25.500 ± 2.068
Tsnd_Probe	0.726 ± 0.009	1.761 ± 0.005	1.492 ± 0.008	2.035 ± 0.022	25.000 ± 1.298
cP_TsndHop2	0.686 ± 0.007	1.748 ± 0.006	1.498 ± 0.009	31.059 ± 0.604	24.850 ± 1.461
Trcv_CeHop2	0.802 ± 0.011	1.783 ± 0.010	1.562 ± 0.020	19.903 ± 0.829	21.600 ± 2.066
Ce_RcvHop4	0.763 ± 0.008	1.798 ± 0.009	1.593 ± 0.018	46.940 ± 1.722	20.000 ± 2.055
Trcv_CeNbor	0.953 ± 0.019	1.811 ± 0.012	1.614 ± 0.033	7.072 ± 0.234	18.900 ± 2.283
Tsnd_CpProbe	0.700 ± 0.006	1.807 ± 0.008	1.617 ± 0.008	68.734 ± 1.478	18.700 ± 1.689
cP_RcvNbor	0.948 ± 0.012	1.819 ± 0.007	1.638 ± 0.009	29.244 ± 0.717	17.600 ± 1.536
cP_RcvHop 2	0.816 ± 0.008	1.825 ± 0.007	1.653 ± 0.010	68.230 ± 1.753	17.000 ± 1.686
cP_RcvBCast	0.797 ± 0.007	1.870 ± 0.007	1.745 ± 0.006	97.320 ± 2.597	12.400 ± 1.729
NoBal	1.982 ± 0.110	2.002 ± 0.015	1.991 ± 0.038		
RcvHop2	1.183 ± 0.023	2.797 ± 0.015	2.155 ± 0.022	8.231 ± 0.107	-8.300 ± 3.234
RcvNbor	1.875 ± 0.058	2.873 ± 0.014	2.283 ± 0.022	2.833 ± 0.035	-14.600 ± 3.273
RcvHop4	1.098 ± 0.048	3.017 ± 0.021	2.392 ± 0.031	19.199 ± 0.249	-20.200 ± 3.882
RcvBCast	1.163 ± 0.043	3.104 ± 0.022	2.498 ± 0.03	21.866 ± 0.263	-25.400 ± 3.978

Table 6.5 : Overall Performance for the 4-by-4 Mesh Model (Load at 50%)

The global average algorithms produced the lowest load variance and average response times. In particular, the load variance improves over much greater distances (see *GrcvHop4*) but the increased communication overheads results in a comparatively poorer response time compared to *GrcvNbor* and *GrcvHop2*. Whilst the results in Table 6.5 might re-emphasise the effectiveness of the global average implementations at moderate system loads, it also supports the argument for more regionalised load sharing strategies, given larger distributed systems. Thus, the results produced at this stage lends support to the proposition of Kremien et al that:

> "Activities related to remote execution should be bounded and restricted to a small proportion of the activity in the system"[Kremien92].

Under heavy system loads, the results in Table 6.6 are also similar to the nine-processor mesh and shows that the best response times were attained by the regionalised global average algorithms, with the nearest neighbour implementation *GsndNbor* delivering a speedup performance of around 59%. The global broadcast algorithms *GsndBCast* and *GrcvBCast* produced better load distribution but a speedup performance of only 44% and 35% respectively. It is also notable that the receiver-initiated threshold algorithms have an improved performance under heavy loads and *TrcvHop2* is amongst the top three algorithms yielding a performance speedup of 57% and a smaller variation in workload when compared with *GsndNbor*. Thus, in contrast to its performance under moderate system load, *RcvHop2* is characterised by a higher degree of success in the negotiation of workload. Given the greater number of overloaded sites, underloaded hosts are likely to have a steady stream of work.

In comparison with the results obtained for the smaller mesh size, the communicating set implementations performed worse than the standard threshold implementations. For example, the performance difference between the threshold neighbour algorithm (*TsndNbor*) and its communicating set counterpart (*Tsnd_CeNbor*) has become more

marked with the *TsndNbor* implementation exhibiting much better load distribution and response times. These results imply that by using the communicating set poorer decisions were being made which invoked migrations that increased load imbalances and overall system response times. A key factor in the improvement of *TsndNbor* is the greater number of hosts with links to three or more immediate neighbours. Thus, under heavy load conditions, a broadcast to all neighbours is likely to result in at least one positive response. In contrast, the communicating set implementation may have a more restricted set of underloaded neighbour sites to which it sends requests, and it is likely that other overloaded neighbours will have that site in common. Thus, the load state of the targeted node will fluctuate more erratically in response to requests made by many of its neighbours. The consequence is a greater likelihood of rejection for the local host and the need to broadcast the request to all its neighbours. In addition, it can also be seen that the overall performance of *TsndNbor* is within striking distance, if not comparable to that of the receiver-initiated global average implementations.

In order to illustrate the problem faced by a communicating set model, a simulation trace, based on the communicating set implementation of the threshold probe policy (*Tsnd_CeProbe*), is described. In this implementation the set is built up as a by-product of load balancing events such as the receipt of, or reply to, a probe message. For example, after 500 seconds of simulation time, *Node10*'s view of the network in its overloaded state only identifies nodes *9* as an underloaded site for fruitful load sharing activities. In the case of *Node8,* its communicating set is empty as it is unable to find an underloaded site.

```
N: 10      Mg:173.00      L: 3    Rt: 2.26 Tx:  35 Imm: 176 Dth:  412
CommSet:[9->2 ]
N: 8       Mg:120.00      L: 4    Rt: 2.18 Tx:  23 Imm: 163 Dth:  361
CommSet:[]
```

	Variance	MeanLd	RunTime	Hit Ratio	SpeedUp
GsndNbor	1.369 ± 0.049	2.771 ± 0.039	2.057 ± 0.039	2.349 ± 0.020	59.000 ± 2.211
GsndHop2	0.961 ± 0.022	2.946 ± 0.041	2.088 ± 0.043	7.120 ± 0.044	58.300 ± 2.406
TrcvHop2	1.183 ± 0.023	2.797 ± 0.015	2.155 ± 0.022	8.213 ± 0.077	57.100 ± 1.595
GrcvNbor	1.481 ± 0.027	2.762 ± 0.051	2.160 ± 0.032	4.926 ± 0.195	57.000 ± 2.052
TsndNbor	1.749 ± 0.156	2.903 ± 0.036	2.144 ± 0.048	1.687 ± 0.017	56.800 ± 1.576
Trcv_CeHop2	1.599 ± 0.034	2.824 ± 0.044	2.250 ± 0.027	5.692 ± 0.111	55.150 ± 1.927
GrcvHop2	1.107 ± 0.035	2.827 ± 0.063	2.220 ± 0.058	12.103 ± 0.587	55.700 ± 2.342
TrcvNbor	1.875 ± 0.058	2.873 ± 0.014	2.283 ± 0.022	2.827 ± 0.026	54.600 ± 2.011
Tsnd_CpNbor	2.293 ± 0.118	2.856 ± 0.029	2.305 ± 0.032	5.760 ± 0.053	54.000 ± 1.944
Trcv_CpHop2	1.507 ± 0.051	2.861 ± 0.022	2.316 ± 0.026	12.796 ± 0.174	53.800 ± 2.098
Tsnd_CeNbor	2.431 ± 0.127	2.899 ± 0.047	2.332 ± 0.045	2.096 ± 0.040	53.550 ± 2.305
Tsnd_Probe	1.413 ± 0.020	2.883 ± 0.026	2.353 ± 0.033	2.596 ± 0.030	52.633 ± 1.542
Trcv_CeHop4	1.567 ± 0.061	2.922 ± 0.050	2.362 ± 0.028	12.275 ± 0.232	52.900 ± 2.100
TsndHop2	3.210 ± 0.188	3.649 ± 0.044	2.347 ± 0.056	5.639 ± 0.063	52.895 ± 1.487
Trcv_CpNbor	2.063 ± 0.058	2.909 ± 0.025	2.373 ± 0.034	6.073 ± 0.063	52.600 ± 2.171
TrcvHop4	1.098 ± 0.048	3.017 ± 0.021	2.392 ± 0.031	19.158 ± 0.168	52.300 ± 1.947
Trcv_CeNbor	2.283 ± 0.080	2.935 ± 0.039	2.394 ± 0.033	2.323 ± 0.042	52.150 ± 2.231
Tsnd_CeHop2	2.364 ± 0.162	2.993 ± 0.096	2.399 ± 0.049	6.103 ± 0.205	52.050 ± 2.585
Tsnd_CpHop2	2.494 ± 0.155	2.988 ± 0.038	2.458 ± 0.045	14.209 ± 0.156	50.900 ± 2.079
TrcvBCast	1.163 ± 0.043	3.104 ± 0.022	2.498 ± 0.031	21.820 ± 0.249	50.200 ± 2.044
Tsnd_CpProbe	1.890 ± 0.078	3.036 ± 0.030	2.530 ± 0.038	19.332 ± 0.214	49.600 ± 2.319
GsndHop4	0.755 ± 0.016	3.576 ± 0.069	2.641 ± 0.087	20.366 ± 0.138	47.100 ± 3.695
Trcv_CpBCast	2.016 ± 0.129	3.189 ± 0.041	2.724 ± 0.062	22.672 ± 0.185	45.600 ± 2.271
Random	3.284 ± 0.212	3.176 ± 0.051	2.725 ± 0.073	1.000	45.133 ± 1.925
GsndBCast	0.759 ± 0.015	3.777 ± 0.079	2.833 ± 0.091	23.951 ± 0.151	43.500 ± 3.629
GrcvHop4	0.867 ± 0.019	3.418 ± 0.106	2.847 ± 0.126	25.948 ± 0.169	43.200 ± 4.367
GrcvBCast	0.853 ± 0.013	3.742 ± 0.169	3.239 ± 0.218	29.417 ± 0.212	35.200 ± 6.125
TsndHop4	12.638 ± 0.709	5.857 ± 0.090	3.403 ± 0.140	18.599 ± 0.410	31.684 ± 3.215
NoBal	20.081 ± 2.065	5.0 ± 0.13640	4.974 ± 0.170		

Table 6.6 : Overall Performance with System Load at 80%

The following trace focused on the message dialogue of node*10* and node*8* just before and after the first period of the simulation run.

```
Node8 receives Packet N6-N8::   Node8::Time:496.556486     Node6's reply (prc6344) was  3
Node8 GMT 496.557   TripT0.252          TIMER:PRB_TOUT          Etime: 496.809
Node8 receives Packet N4-N8::   Node8::Time:496.659295     Node4's reply (prc6344) was  3
Node8 GMT 496.660   TripT0.256          TIMER:PRB_TOUT          Etime: 496.916

Node10 receives Packet N0-N10::          Node0 -overloaded- has asked node10 TO accept
Node10 GMT 496.759  TripT0.258           TIMER:WAITP_TOUT        Etime: 497.017

Node8 receives Packet N14-N8::
Node8::Time:496.812504          Node14's reply (prc6344) was  -2
node 8 sends process6344 [load = 4] to node 14
```

As the communicating set for *Node8* is empty, *node6* and *node4* were probed at random. Both nodes were found to be overloaded, and another remote site (*node14*) was selected at random. The probe was successful as *node14* was found to be underloaded and therefore it automatically became a member of the communicating set for *node8*. Process migration then takes place as *node8* sends prc6344 to *node14*. Similarly, when *Node10* is probed by *Node0*, it was also found to be underloaded, and initiated the "wait timer" for the remote process to arrive.

```
Node10 receives Packet N7-N10::      node 10 receives Exit msg [prc:: 6318 at node:7]
Node10 receives Packet N0-N10::      node 10 RECEIVES process6347 [load = 3] FROM node 0
Node10 receives Packet N0-N10::      Node0 -overloaded- has asked node10 TO accept
Node10 GMT 497.333 TripT0.258        TIMER:WAITP_TOUT        Etime: 497.591
Node10 receives Packet N7-N10::      node 10 receives Exit msg [prc:: 6320 at node:7]
Node10 receives Packet N0-N10::      node 10 RECEIVES process6354 [load = 3] FROM node 0
Node10 receives Packet N14-N10::     Node14 -overloaded- has asked node10 TO accept
```

In the protocol fragment above, an exit message is received by *node10* from *node7* for one of its original processes (*prc6318*) followed by the arrival of *prc6347* from *node0*. As *node10* is a member of the communicating set for node0 a further probe message is sent by the latter resulting in the subsequent migration of *prc6354* to *node10*. A further probe message by *node14* was rejected as *node10* is no longer underloaded.

```
Node8 receives Packet N9-N8::   Node9 -overloaded- has asked node8 TO accept
Node8 receives Packet N14-N8:: node 8 receives Exit msg [prc:: 6344 at node:14]
Node8 receives Packet N9-N8::   Node9 -overloaded- has asked node8 TO accept
Node8 GMT 499.383   TripT0.256          TIMER:PRB_TOUT          Etime: 499.639
Node8 GMT 499.487   TripT0.256          TIMER:PRB_TOUT          Etime: 499.743
Node8 receives Packet N14-N8:: Node8::Time:499.588688    Node14's reply (prc6386) was  4
Node8 GMT 499.590   TripT0.252          TIMER:PRB_TOUT          Etime: 499.842
Node8 GMT 499.593   TripT0.254          TIMER:PRB_TOUT          Etime: 499.847

Node8 receives Packet N4-N8::
Node8::Time:499.645715          Node4's reply (prc6386) was  -2
node 8 sends process6386 [load = 5] to node 4
```

In its previous overloaded state *node8* performed a successful migration (prc6344) to *node14*. Therefore, given its current state two probe messages are sent to *node14* on behalf of the excess local processes. However, these requests are rejected and another two consecutive probe messages were sent to randomly selected hosts. *Node4* was found to have available capacity and *prc6386* migrated to it.

```
Node8 GMT 499.659   TripT0.252          TIMER:PRB_TOUT          Etime: 499.911
Node8 receives Packet N14-N8:: Node8::Time:499.710342    Node14's reply (prc6391) was  4
Node8 GMT 499.711   TripT0.252          TIMER:PRB_TOUT          Etime: 499.963
Node8 receives Packet N0-N8::   Node8::Time:499.712351    Node0's reply (prc6392) was  3

Node8 GMT 499.713   TripT0.252          TIMER:PRB_TOUT          Etime: 499.965
Node8 receives Packet N4-N8::   Node8::Time:499.766364    Node4's reply (prc6393) was  3
Node8 GMT 499.767   TripT0.256          TIMER:PRB_TOUT          Etime: 500.023
Node8 receives Packet N4-N8::   Node8::Time:499.818773    Node4's reply (prc6391) was  3
Node8 GMT 499.820   TripT0.256          TIMER:PRB_TOUT          Etime: 500.076
Node8 receives Packet N4-N8::   Node8::Time:499.820782    Node4's reply (prc6392) was  3
Node8 GMT 499.822   TripT0.258          TIMER:PRB_TOUT          Etime: 500.080

Node8 receives Packet N14-N8:: Node8::Time:499.923591    Node14's reply (prc6393) was  3
Node8 GMT 499.925   TripT0.258          TIMER:PRB_TOUT          Etime: 500.183

Node8 receives Packet N15-N8:: Node8::Time:500.026400    Node15's reply (prc6392) was  4
Node8 receives Packet N1-N8::   Node8::Time:500.027405    Node1's reply (prc6391) was  3
Node8 receives Packet N2-N8::   Node8::Time:500.179610    Node2's reply (prc6393) was  3
Node8 GMT 500.281   TripT0.254          TIMER:PRB_TOUT          Etime: 500.535
```

In the example above *node8* sends consecutive requests to the only member of its communicating set (*node4*) on behalf of processes *6391* to *6393*, but receives in turn, consecutive rejections. This is also the case for requests sent to nodes *14, 0, 15, 1,*

and *2*. These results highlight a key weakness of the communicating set implementation under heavy system loads where consecutive process arrivals occur. In such cases, probe messages would have been sent to members of the communicating set before the first reply was received and the local host is able to update the set. A possible resolution of this problem is to increment the load entry for the member of the communicating set that is probed. Potentially, this may result in the load entry being inaccurate and may also require the introduction of additional mechanisms to test for continued membership. However, it has the advantage of significantly reducing the number of consecutive messages sent to a remote site which may only be in a transitory underloaded state. An alternative method is to simply remove the node entry from the set. If the probed site is still underloaded as a result of the state information returned, an up-to-date entry can be made in the communicating set. Consecutive process arrivals will be forced to seek alternative sites until the reply is received for a previous probe message from a member of the communicating set for the local host.

The results in Table 6.7 are also similar to those achieved under extreme load conditions for a nine processor mesh topology. The results show that the better response times were attained by the regionalised, receiver-initiated algorithms with performance speedup of up to 72%. However, in the case of the random policy, the speedup improvement was only 16% compared to 32% for the smaller mesh system. The run time performance of the regionalised global average algorithms were generally poorer than the receiver-initiated algorithms but continue to exhibit a greater degree of load stability. This is particularly true for *GrcvNbor* achieving a workload variation of about three processes between sites; whilst algorithms such as *TrcvNbor* produced the better run time performance. Only *TrcvHop2* produced the best response time and comparable variations in workload to that of the global average

algorithms. The sender-initiated algorithm *TsndNbor* produced a performance speedup of 67% but the level of load imbalance was significantly worse at around seven processes between sites.

	Variance	MeanLd	RunTime	Hit Ratio	SpeedUp
TrcvHop2	2.866 ± 0.357	3.684 ± 0.077	2.926 ± 0.091	6.580 ± 0.046	71.800 ± 1.751
TrcvNbor	4.034 ± 0.249	3.790 ± 0.072	3.067 ± 0.090	2.299 ± 0.017	70.500 ± 1.900
Trcv_CeHop2	4.202 ± 0.326	3.923 ± 0.073	3.209 ± 0.085	4.175 ± 0.068	69.000 ± 2.055
GsndNbor	2.526 ± 0.097	4.083 ± 0.114	3.266 ± 0.100	2.409 ± 0.014	68.400 ± 2.591
Trcv_CpNbor	5.090 ± 0.435	4.006 ± 0.074	3.302 ± 0.091	4.885 ± 0.071	68.286 ± 1.799
Trcv_CeNbor	5.604 ± 0.488	4.092 ± 0.083	3.402 ± 0.096	1.765 ± 0.025	67.300 ± 2.627
Trcv_CpHop 2	5.139 ± 0.772	4.164 ± 0.114	3.444 ± 0.115	10.337 ± 0.169	67.000 ± 2.000
TsndNbor	7.399 ± 1.543	4.126 ± 0.166	3.396 ± 0.167	3.126 ± 0.127	67.000 ± 2.631
Trcv_CeHop4	5.028 ± 0.685	4.228 ± 0.098	3.533 ± 0.120	9.149 ± 0.127	66.400 ± 2.547
GrcvNbor	2.834 ± 0.175	4.285 ± 0.134	3.594 ± 0.168	3.994 ± 0.043	65.900 ± 2.644
TrcvHop4	4.884 ± 0.505	4.346 ± 0.115	3.666 ± 0.146	15.527 ± 0.088	64.500 ± 2.224
Tsnd_CeNbor	9.814 ± 0.943	4.559 ± 0.128	3.905 ± 0.146	3.630 ± 0.151	62.900 ± 2.998
Tsnd_Probe	3.985 ± 0.530	4.530 ± 0.139	3.909 ± 0.144	4.727 ± 0.192	62.071 ± 2.129
TrcvBCast	5.828 ± 0.667	4.582 ± 0.134	3.936 ± 0.158	17.836 ± 0.116	62.100 ± 2.644
Tsnd_CpNbor	11.690 ± 3.261	4.710 ± 0.167	3.982 ± 0.149	9.050 ± 0.299	61.857 ± 2.116
GsndHop2	1.597 ± 0.060	4.933 ± 0.269	3.987 ± 0.259	7.161 ± 0.023	61.300 ± 3.368
Tsnd_CeHop2	14.227 ± 1.994	5.306 ± 0.221	4.670 ± 0.210	14.399 ± 1.027	55.200 ± 3.706
GrcvHop2	2.082 ± 0.162	7.158 ± 1.377	6.647 ± 1.303	10.245 ± 0.050	45.600 ± 7.152
Random	58.628 ± 18.131	8.779 ± 0.801	8.566 ± 0.856	1.000	16.538 ± 6.827
NoBal	97.999 ± 17.230	10.430 ± 0.595	10.321 ± 0.627		

Table 6.7: Mesh16 Overall Performance with System Load at 90%

Under extreme load conditions, the global average algorithms which operate beyond a distance of two hops were extremely unstable. Again, in contrast to the smaller nine processor mesh size, the excess message traffic and communication delays rendered these global algorithms completely ineffective. Thus, *GrcvBCast* and *GsndBCast* were equally susceptible to the race condition.

It is noticeable that the regionalised sender-initiated communicating set algorithms performed significantly worse than their receiver-initiated counterparts in terms of response time and load distribution. One of the major problems for such algorithms is the inaccuracy of the state information collected as a result of excessive delays in the network. Again, many overloaded sites will have a small set of underloaded processors in common. As a result of excessive requests to the same underloaded site, it is more likely for the state information received to be outdated or replies failing to arrive within a given time frame. Whilst the sender-initiated implementation may result in many overloaded hosts sending requests to the same underloaded site, the receiver-initiated algorithm allows a few underloaded sites to send requests to many and often mutually exclusive overloaded hosts.

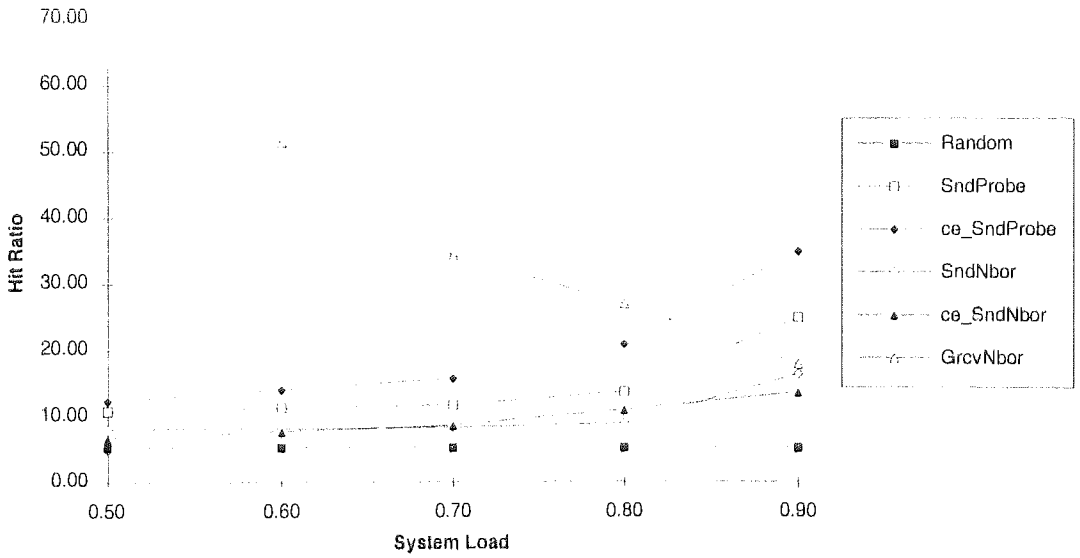


Figure 6.1 Hit Ratio for Load Balancing Algorithms

Figure 6.1 shows how the hit-ratio of a selection of algorithms varies with system load. In particular, the hit ratio of the receiver-initiated global average algorithm improves the greater the system load and continues to rise dramatically under extreme

load conditions. In contrast, the sender-initiated algorithms exhibited a poorer hit ratio at around 70% loading which continues to fall the greater the load.

The results produced by the threshold policy up until now have been primarily due to the removal of message timers, as implemented in other studies. Whilst this strategy is simple, its consequence is that an overloaded host will wait for an unspecified period of time for a reply to be returned from a remote host when probed. The main problem is that a newly created user process may have to wait for long periods before being started locally or remotely. But more importantly, its susceptibility to network failure may result in suspending the user process indefinitely. Therefore, a timer-based version of the communicating set model was also implemented and the results are presented in Table 6.8. It is clear from the table that the real difference in performance occurs under heavy system load. Under those circumstances, message delay is significant. The timer-based implementation times out more frequently and has to make decisions without the information it requires. Thus, at 80% loading the performance speedup for Threshold and Threshold Timer are 56% and 52% respectively, and at 90% system load, a speedup of 65% and 47% was attained.

Furthermore, under extreme system load the timer-based implementation is extremely unstable with variance in workload of around 18 processes. The communicating set model implemented without timers produced a speedup of 65% compared to 62% for the standard threshold algorithm, but the latter exhibited better load stability. However, given that it is generally unacceptable to make user processes wait indefinitely, the timer-based implementation reflects the achievable performance of threshold policies.

	Variance	MeanLd	Difference	RunTime	Mig/s	Tx/s	Hit Ratio	SpeedUp
Threshold_Ce	0.253 ± 0.004	1.239 ± 0.003	1.408 ± 0.014	1.183 ± 0.005	0.007 ± 0.000	0.078 ± 0.007	2.078 ± 0.210	5.700 ± 0.823
Timer-based	0.253 ± 0.006	1.239 ± 0.004	1.406 ± 0.021	1.180 ± 0.008	0.007 ± 0.000	0.080 ± 0.012	2.114 ± 0.222	5.800 ± 1.033

(a) 20% System Loading

	Variance	MeanLd	Difference	RunTime	Mig/s	Tx/s	Hit Ratio	SpeedUp
Timer-based	0.680 ± 0.007	1.756 ± 0.008	2.268 ± 0.013	1.481 ± 0.016	0.102 ± 0.002	1.294 ± 0.030	2.385 ± 0.034	25.700 ± 2.406
Threshold_Ce	0.676 ± 0.006	1.747 ± 0.007	2.261 ± 0.014	1.468 ± 0.005	0.101 ± 0.002	1.278 ± 0.039	2.385 ± 0.037	26.100 ± 2.079

(b) 50% System Loading

	Variance	MeanLd	Difference	RunTime	Mig/s	Tx/s	Hit Ratio	SpeedUp
Timer-based	1.442 ± 0.080	3.006 ± 0.046	4.073 ± 0.107	2.408 ± 0.047	0.308 ± 0.003	7.707 ± 0.173	4.691 ± 0.130	51.900 ± 1.792
Threshold_Ce	1.324 ± 0.082	2.792 ± 0.034	3.798 ± 0.091	2.176 ± 0.035	0.323 ± 0.003	6.919 ± 0.136	4.023 ± 0.093	56.400 ± 2.066

(c) 80% System Loading

	Variance	MeanLd	Difference	RunTime	Mig/s	Tx/s	Hit Ratio	SpeedUp
Timer-based	17.753 ± 2.116	6.122 ± 0.203	13.271 ± 0.769	5.526 ± 0.193	0.152 ± 0.007	12.811 ± 0.111	15.828 ± 0.833	46.600 ± 4.526
Threshold_Ce	5.899 ± 0.891	4.323 ± 0.144	7.667 ± 0.511	3.627 ± 0.158	0.269 ± 0.007	11.319 ± 0.210	7.904 ± 0.334	65.100 ± 2.726

(d) 90% System Loading

Table 6.8 Comparative Performance of Communicating Set Threshold Using Timers

6.1.3 25-Processor Mesh Topology

In a 25 processor mesh the maximum possible distance that a message can travel in order to reach all nodes is 10 hops. Thus, given the average service time for user processes, the relative communication costs will have a significant impact on performance for the unrestricted broadcast algorithms. It is in networks of this size that the fixed threshold broadcast algorithms are at their weakest, as it is no longer cost effective to send requests to all hosts on behalf of a single process. The relative ranking of the algorithms are presented in Table 6.9 to 6.11 for 50%, 80% and 90% loading respectively.

At moderate system load (see Table 6.9) the algorithms exhibited similar ranking to the previous mesh topologies studied. In particular, the regionalised global average algorithms continue to produce the lowest load variance and the better response times. However, there is also a moderate increase in speedup for most algorithms, given the greater proportion of idle nodes in the system. Whilst attempting to maintain the global average over greater distances would tend to result in better load distribution, the results indicate that, even at moderate system load, the impact of communication delay places an upper limit on the network diameter considered by a load balancing algorithm. For example, the algorithms *GsndHop2*, *GsndHop4*, and *GsndHop6* produced load variance results of *0.39*, *0.38*, and *0.4* respectively with corresponding performance speedup of 39%, 31%, and 23%. Further, the difference in the process migration rate is marginal with 21 processes migrated every 100 seconds for *GsndHop6* compared to 22 processes for *GsndHop2*, and *GsndHop4*. Although the receiver-initiated algorithm *GrcvHop6* produced the lowest variation in workload, its speedup performance was only 22% with a resulting process migration rate of 30 processes every 100 seconds.

	Variance	MeanLd	Difference	RunTime	Mig/s	Tx/s	Hit Ratio	SpeedUp
GsndHop2	0.391 ± 0.003	1.777 ± 0.004	2.050 ± 0.011	1.217 ± 0.005	0.222 ± 0.002	7.956 ± 0.062	5.386 ± 0.036	38.900 ± 0.738
GsndNbor	0.480 ± 0.004	1.687 ± 0.003	2.187 ± 0.010	1.239 ± 0.005	0.196 ± 0.001	2.239 ± 0.018	1.718 ± 0.007	37.700 ± 0.949
GrcvHop2	0.408 ± 0.010	1.657 ± 0.031	1.996 ± 0.055	1.255 ± 0.004	0.241 ± 0.005	38.922 ± 6.746	24.207 ± 3.764	36.950 ± 0.759
GrcvNbor	0.497 ± 0.014	1.652 ± 0.020	2.216 ± 0.044	1.272 ± 0.010	0.199 ± 0.003	12.879 ± 1.444	9.725 ± 1.013	36.150 ± 1.040
GsndHop4	0.381 ± 0.004	2.003 ± 0.008	2.164 ± 0.011	1.365 ± 0.008	0.222 ± 0.002	30.934 ± 0.255	20.952 ± 0.145	31.400 ± 0.699
TsndNbor	0.704 ± 0.005	1.765 ± 0.006	2.478 ± 0.013	1.418 ± 0.004	0.095 ± 0.002	0.805 ± 0.014	1.270 ± 0.008	28.600 ± 0.995
GrcvHop4	0.377 ± 0.017	1.762 ± 0.050	2.029 ± 0.035	1.426 ± 0.026	0.286 ± 0.004	100.481 ± 7.484	52.771 ± 3.226	28.350 ± 1.565
TsndHop2	0.861 ± 0.011	1.904 ± 0.010	2.783 ± 0.020	1.429 ± 0.004	0.095 ± 0.002	2.740 ± 0.050	4.330 ± 0.035	28.050 ± 0.945
Tsnd_CeNbor	0.712 ± 0.008	1.745 ± 0.014	2.544 ± 0.018	1.461 ± 0.008	0.078 ± 0.001	0.539 ± 0.008	1.033 ± 0.009	26.750 ± 0.967
Tsnd_CeProbe	0.673 ± 0.007	1.741 ± 0.017	2.432 ± 0.018	1.463 ± 0.013	0.099 ± 0.002	1.527 ± 0.027	2.325 ± 0.024	26.350 ± 0.988
Tsnd_CeHop2	0.715 ± 0.010	1.771 ± 0.039	2.566 ± 0.026	1.475 ± 0.015	0.078 ± 0.001	1.411 ± 0.028	2.720 ± 0.033	26.000 ± 1.026
TsndHop4	1.175 ± 0.032	2.284 ± 0.017	3.244 ± 0.036	1.478 ± 0.007	0.097 ± 0.002	10.009 ± 0.169	15.559 ± 0.099	25.750 ± 0.786
Random	0.691 ± 0.006	1.736 ± 0.005	2.452 ± 0.018	1.479 ± 0.007	0.103 ± 0.002	0.687 ± 0.015	1.000	25.700 ± 0.801
Tsnd_CeHop4	0.732 ± 0.007	1.744 ± 0.005	2.620 ± 0.014	1.488 ± 0.004	0.075 ± 0.001	3.533 ± 0.076	7.035 ± 0.121	25.200 ± 1.135
TsndProbe	0.726 ± 0.007	1.747 ± 0.005	2.581 ± 0.018	1.497 ± 0.009	0.103 ± 0.002	1.392 ± 0.028	2.032 ± 0.017	24.750 ± 0.716
Tsnd_CeHop4	0.729 ± 0.007	1.887 ± 0.008	2.671 ± 0.016	1.527 ± 0.008	0.076 ± 0.001	3.749 ± 0.086	7.361 ± 0.098	23.300 ± 0.675
GsndHop6	0.401 ± 0.003	2.168 ± 0.007	2.239 ± 0.010	1.537 ± 0.011	0.205 ± 0.001	51.506 ± 0.461	37.746 ± 0.357	22.800 ± 0.919
GrcvHop6	0.358 ± 0.003	1.884 ± 0.004	2.150 ± 0.014	1.546 ± 0.010	0.302 ± 0.001	124.022 ± 0.521	61.621 ± 0.451	22.300 ± 1.160
Trcv_CeHop2	0.796 ± 0.008	1.785 ± 0.011	2.770 ± 0.017	1.552 ± 0.007	0.085 ± 0.001	9.875 ± 0.084	17.545 ± 0.307	22.050 ± 0.887
Trcv_CeNbor	0.941 ± 0.010	1.807 ± 0.008	3.152 ± 0.027	1.602 ± 0.008	0.068 ± 0.001	2.659 ± 0.021	5.860 ± 0.106	19.600 ± 0.940
Trcv_CeHop4	0.767 ± 0.008	1.824 ± 0.018	2.652 ± 0.017	1.616 ± 0.006	0.099 ± 0.002	31.716 ± 0.259	48.217 ± 1.044	18.750 ± 0.910
NoBal	1.968 ± 0.055	1.999 ± 0.011	4.972 ± 0.080	1.988 ± 0.022				

Table 6.9 System Behaviour for a 25 Processor Mesh Model at 50% Loading

The lower variation of *Grcv* over greater distances can be accounted for by the accuracy of the system state information maintained by the collective efforts of a significant number of underloaded sites and the lengthier load balancing dialogue under moderate load conditions. However, the higher migration rate does not equate with improved response time because the remaining service time of the greater number of migrated processes may be far less than the time taken to complete the transfer. Therefore, a sender-initiated strategy is preferable under low to moderate system load conditions.

The performance of the algorithms under heavy load conditions is shown in Table 6.10. Again, the pattern of performance is consistent with that of the previous mesh sizes examined as the relative ranking and overall performance of the algorithms are similar. The performance of restricted algorithms such as *TsndNbor*, *GrcvNbor*, and *Tsnd_CeProbe* continue to improve with the increased mesh size over less restricted implementations.

The performance of *TsndNbor* is commendable and comparable to the more complex global average algorithm *GsndNbor*. In particular, it exhibits very low load variance and a high hit ratio of around one message for every migration that takes place compared to two messages for *GsndNbor*. Algorithms that attempt to operate the load balancing policy over distances of more than two hops performed badly. Thus, *TsndHop4* produced an average speedup of 41% but exhibited load instability variations of up to 17 processes. In contrast, *GrcvHop4* produced the lowest variation in workload, but one of the poorest response times. In terms of their process migration rate, both algorithms migrate about four processes every ten seconds but the message traffic is three times as great for *GrcvHop4*. The network delays arising from the reduced bandwidth seriously impaired the average process response time especially where pre-emptive migrations are involved.

	Variance	MeanLd	Difference	RunTime	Mig/s	Tx/s	Hit Ratio	SpeedUp
GsndNbor	1.406 ± 0.030	2.764 ± 0.029	4.175 ± 0.057	2.043 ± 0.031	0.419 ± 0.004	5.427 ± 0.056	1.946 ± 0.008	59.500 ± 1.080
TsndNbor	1.765 ± 0.100	2.894 ± 0.020	4.931 ± 0.157	2.102 ± 0.030	0.392 ± 0.004	3.627 ± 0.043	1.389 ± 0.008	58.250 ± 1.209
GsndHop2	1.047 ± 0.027	2.976 ± 0.033	3.692 ± 0.050	2.105 ± 0.040	0.449 ± 0.004	18.578 ± 0.119	6.219 ± 0.033	58.300 ± 1.160
GrcvNbor	1.511 ± 0.038	2.750 ± 0.062	4.324 ± 0.070	2.139 ± 0.040	0.440 ± 0.005	11.692 ± 0.572	3.993 ± 0.173	57.333 ± 0.970
Trcv_CeHop2	1.552 ± 0.031	2.808 ± 0.047	4.648 ± 0.079	2.218 ± 0.026	0.320 ± 0.003	10.234 ± 0.195	4.816 ± 0.107	55.889 ± 0.832
GrcvHop2	1.195 ± 0.034	2.852 ± 0.066	3.944 ± 0.054	2.234 ± 0.039	0.487 ± 0.004	33.088 ± 1.814	10.217 ± 0.510	55.611 ± 1.243
Tsnd_CeNbor	2.391 ± 0.100	2.883 ± 0.054	5.835 ± 0.155	2.303 ± 0.035	0.203 ± 0.002	2.288 ± 0.043	1.698 ± 0.033	54.056 ± 1.056
TsndHop3	4.273 ± 0.193	3.845 ± 0.038	7.939 ± 0.208	2.343 ± 0.053	0.393 ± 0.004	13.319 ± 0.183	5.087 ± 0.052	53.300 ± 1.380
Trcv_CeNbor	2.239 ± 0.068	2.914 ± 0.041	5.606 ± 0.121	2.366 ± 0.031	0.258 ± 0.003	3.239 ± 0.052	1.891 ± 0.036	52.944 ± 0.998
Tsnd_CeHop2	2.359 ± 0.096	2.999 ± 0.104	5.829 ± 0.177	2.388 ± 0.041	0.188 ± 0.003	6.523 ± 0.156	5.234 ± 0.190	52.556 ± 1.042
Trcv_CeHop4	1.563 ± 0.043	2.964 ± 0.052	4.742 ± 0.052	2.399 ± 0.029	0.350 ± 0.004	28.084 ± 0.417	12.058 ± 0.280	52.333 ± 0.907
Tsnd_CeProbe	1.562 ± 0.134	2.979 ± 0.228	4.708 ± 0.362	2.419 ± 0.230	0.298 ± 0.010	9.578 ± 0.849	4.870 ± 0.577	51.889 ± 4.484
TsndProbe	1.519 ± 0.024	2.956 ± 0.031	4.354 ± 0.052	2.445 ± 0.032	0.471 ± 0.006	8.180 ± 0.184	2.608 ± 0.036	51.467 ± 1.042
Random	3.529 ± 0.194	3.246 ± 0.055	7.013 ± 0.192	2.809 ± 0.061	0.504 ± 0.008	3.352 ± 0.056	1.000	44.167 ± 1.234
TsndHop4	16.644 ± 0.572	5.719 ± 0.059	14.569 ± 0.275	2.987 ± 0.083	0.404 ± 0.003	32.834 ± 0.383	12.229 ± 0.163	40.500 ± 1.933
Tsnd_CeHop4	4.276 ± 0.317	3.405 ± 0.062	7.620 ± 0.335	2.989 ± 0.070	0.153 ± 0.001	22.464 ± 0.604	22.075 ± 0.841	40.875 ± 1.642
NoBal	20.943 ± 1.480	5.093 ± 0.100	16.984 ± 0.603	5.032 ± 0.126				
GrcvHop4	1.025 ± 0.807	13.584 ± 16.458	3.597 ± 1.821	10.909 ± 12.423	0.375 ± 0.108	91.592 ± 2.313	41.300 ± 11.32	-118.11 ± 248.99

Table 6.10 System Behaviour for a 25 Processor Mesh Model at 80% Loading

The communicating set algorithms operating at a two hop radius produced lower variation in workload, but marginally poorer response times, than their setless counterparts. The results for *Tsnd_CeHop4* show workload variation of about four processes compared to the more unstable *TsndHop4*. In terms of the process migration and message transmit rate the communication model had 50% fewer migrations per second (two processes per second) and 30% less transmits. Thus, although its hit rate might suggest poor decision making activity in *Tsnd_CeHop4*, the level of stability in workload would suggest otherwise.

It is under extreme load conditions of 90% (see Table 6.11) that the random policy truly becomes inferior to the no load balancing case. So far, as the mesh size has increased the performance of the random policy has been relatively consistent between system loads of up to 80%. It is under chaotic load conditions that this strategy has demonstrated progressive increases in load instability and poor run time performance for every increase in the mesh size.

The migration rate for the random policy is around eight processes per second compared to a maximum of five processes for the other algorithms. The restricted global average algorithms still exhibit better load stability but the fixed threshold receiver-initiated algorithms produced better overall response times. If one examines the results for the top two algorithms it is noticeable that the migration rate for *Trcv_CeHop2* was under four processes every ten seconds compared to five processes for *GsndNbor*. At extreme load, the fixed threshold nearest neighbour algorithm *TsndNbor* is less stable but, nevertheless, is able to deliver a performance speedup of 66%. The sender-initiated communicating set implementation of threshold and nearest neighbour performed worse than their setless counterpart. The poorer performance is attributable to untimely state information for *Tsnd_Ceprobe*, and a very limited choice of sites in the case of *Tsnd_CeNbor*.

	Variance	MeanLd	Difference	RunTime	Mig/s	Tx/s	Hit Ratio	SpeedUp
Trcv_CeHop2	3.871 ± 0.251	3.842 ± 0.074	7.580 ± 0.238	3.132 ± 0.093	0.351 ± 0.005	7.694 ± 1.820	3.473 ± 0.041	70.000 ± 1.556
GsndNbor	2.823 ± 0.132	4.096 ± 0.087	5.922 ± 0.125	3.276 ± 0.096	0.522 ± 0.005	6.910 ± 0.066	1.992 ± 0.004	68.700 ± 1.703
Trcv_CeNbor	5.269 ± 0.425	3.998 ± 0.080	8.869 ± 0.337	3.316 ± 0.111	0.291 ± 0.003	2.618 ± 0.618	1.426 ± 0.014	68.200 ± 1.673
TsndNbor	8.053 ± 1.366	4.395 ± 0.122	11.315 ± 0.941	3.512 ± 0.129	0.434 ± 0.007	5.384 ± 0.053	1.867 ± 0.039	66.300 ± 1.593
GrcvNbor	3.055 ± 0.159	4.217 ± 0.114	6.221 ± 0.132	3.535 ± 0.148	0.543 ± 0.004	11.463 ± 2.729	3.341 ± 0.104	65.950 ± 1.986
Trcv_CeHop4	5.276 ± 0.766	4.315 ± 0.093	8.699 ± 0.541	3.664 ± 0.157	0.326 ± 0.006	18.032 ± 4.271	8.771 ± 0.189	64.750 ± 2.124
Tsnd_CeNbor	9.615 ± 0.984	4.473 ± 0.128	11.733 ± 0.525	3.828 ± 0.165	0.186 ± 0.004	3.641 ± 0.059	2.957 ± 0.098	63.150 ± 2.159
TsndProbe	4.612 ± 0.589	4.873 ± 0.164	7.976 ± 0.402	4.290 ± 0.187	0.435 ± 0.015	15.307 ± 0.255	5.300 ± 0.270	58.767 ± 1.501
GsndHop2	2.205 ± 0.065	5.260 ± 0.239	5.290 ± 0.078	4.339 ± 0.266	0.540 ± 0.004	22.735 ± 0.239	6.328 ± 0.033	58.700 ± 2.627
Tsnd_CeHop2	14.315 ± 1.956	5.270 ± 0.244	13.990 ± 0.886	4.634 ± 0.288	0.150 ± 0.006	11.654 ± 2.764	12.400 ± 0.772	55.400 ± 3.299
TsndHop2	21.288 ± 3.589	6.655 ± 0.271	17.678 ± 1.387	5.468 ± 0.310	0.382 ± 0.012	21.848 ± 0.236	8.613 ± 0.361	47.400 ± 3.515
Tsnd_CeProbe	23.372 ± 5.104	6.670 ± 0.399	17.753 ± 1.798	6.182 ± 0.470	0.124 ± 0.008	16.168 ± 0.192	19.658 ± 1.496	41.100 ± 5.343
GrcvHop2	2.945 ± 0.240	9.530 ± 3.449	6.281 ± 0.334	8.534 ± 3.174	0.569 ± 0.004	34.603 ± 1.261	9.139 ± 0.347	18.632 ± 32.838
NoBal	102.748 ± 10.82	10.582 ± 0.340	38.307 ± 2.364	10.420 ± 0.356				
Random	233.251 ± 111.2	14.579 ± 2.168	54.446 ± 12.443	14.735 ± 2.320	0.836 ± 0.010	5.560 ± 0.066	1.000	-41.133 ± 19.046

Table 6.11 System Behaviour for a 25 Processor Mesh Model at 90% Loading

6.1.4 36-Processor Mesh Topology

The results produced for the 36-processor model are summarised in tables 6.12-6.14. The relative performance ranking of the algorithms was similar to that of the 25-processor mesh. However, at moderate system load (see Table 6.12) the effect of network delay on the performance of the global average algorithms operating outside a four hop radius is evident. For both the sender and receiver-initiated implementations the load variance and performance speedup is worse. In particular, the speedup performances for *GsndBCast* and *GrcvBCast* were 14% and 12% respectively. These algorithms are also characterised by very low hit rates. The sender-initiated broadcast algorithms operating beyond a radius of two hops exhibited significant variation in workload although a speedup of approximately 25% was achieved. However, the scale of the variation in workload under moderate system load is unacceptable. This algorithm can only be justified if its activation is restricted to multiple process placements. But, to do so under moderate system load would result in fewer migrations and a reduction in system performance. Therefore, the nearest neighbour implementation is preferable.

Under heavy system load (see Table 6.13) the results are similar to those for the 25 processor mesh where the nearest neighbour implementations produced the best overall performance. It is interesting to note that *Tsnd_CeHop2* exhibit a significantly greater degree of load stability than *TsndHop2* but a marginally poorer response time. Further, the former produced 50% fewer migrations per second but with a comparable hit ratio. Again, the stability of the communicating set implementation is to be preferred given the minor differences in their respective average response time.

Under extreme load conditions (see Table 6.14) the fixed threshold sender-initiated algorithms are extremely unstable whilst the restricted domain adaptive algorithms

such as global average are relatively stable. The exceptional case however, is *GrcvHop2*. Whilst it produced a speedup performance of 18% for a 25 processor mesh, this has fallen to less than 3%. What is particularly noticeable is the very high load average compared to the other algorithms even though its process migration rate and hit ratio are comparable to *GsndHop2*. This indicates that many of the dominant overloaded sites are not being probed regularly by underloaded sites. Therefore, an underloaded site will assume the threshold to be too high, which may not be case, decrement the average, resulting in a further increase in the average number of processes per host. In contrast, the relatively small number of underloaded sites for *GsndHop2* are probed regularly by overloaded sites which ensures that the global average being used is relatively accurate.

	Variance	MeanLd	Difference	RunTime	Mig/s	Tx/s	Hit Ratio	SpeedUp
GsndHop2	0.393 ± 0.002	1.782 ± 0.003	2.198 ± 0.009	1.215 ± 0.003	0.223 ± 0.001	8.483 ± 0.061	4.751 ± 0.037	38.700 ± 0.823
GsndNbor	0.476 ± 0.002	1.685 ± 0.003	2.330 ± 0.009	1.234 ± 0.004	0.197 ± 0.002	2.301 ± 0.020	1.458 ± 0.009	37.700 ± 0.823
GrcvHop2	0.412 ± 0.002	1.685 ± 0.003	2.176 ± 0.012	1.251 ± 0.004	0.238 ± 0.002	34.529 ± 0.074	18.114 ± 0.21	36.900 ± 0.994
GrcvNbor	0.509 ± 0.003	1.669 ± 0.003	2.415 ± 0.010	1.279 ± 0.004	0.198 ± 0.002	11.685 ± 0.013	7.378 ± 0.086	35.400 ± 0.843
GsndHop4	0.398 ± 0.004	2.035 ± 0.008	2.393 ± 0.014	1.384 ± 0.007	0.229 ± 0.002	36.673 ± 0.302	20.064 ± 0.15	30.200 ± 0.789
TsndNbor	0.711 ± 0.006	1.765 ± 0.004	2.633 ± 0.016	1.415 ± 0.005	0.094 ± 0.002	0.823 ± 0.015	1.089 ± 0.006	28.650 ± 0.813
TsndHop2	0.920 ± 0.014	1.920 ± 0.007	3.063 ± 0.029	1.429 ± 0.005	0.095 ± 0.001	2.947 ± 0.046	3.886 ± 0.026	27.950 ± 0.605
TsndHop3	1.223 ± 0.032	2.140 ± 0.011	3.526 ± 0.041	1.453 ± 0.006	0.096 ± 0.002	6.809 ± 0.114	8.915 ± 0.066	26.750 ± 0.786
GrcvHop4	0.361 ± 0.003	1.840 ± 0.004	2.223 ± 0.011	1.459 ± 0.008	0.292 ± 0.002	107.149 ± 0.470	45.949 ± 0.41	26.300 ± 0.823
Tsnd_CeNbor	0.718 ± 0.004	1.757 ± 0.004	2.708 ± 0.014	1.462 ± 0.006	0.078 ± 0.001	0.547 ± 0.005	0.874 ± 0.008	26.200 ± 0.789
Tsnd_CeProbe	0.679 ± 0.005	1.756 ± 0.005	2.444 ± 0.014	1.473 ± 0.010	0.100 ± 0.001	1.536 ± 0.026	1.923 ± 0.021	25.600 ± 0.699
Random	0.693 ± 0.007	1.737 ± 0.006	2.596 ± 0.022	1.481 ± 0.006	0.104 ± 0.002	0.831 ± 0.016	1.000	25.400 ± 0.770
Tsnd_CeHop2	0.729 ± 0.004	1.809 ± 0.005	2.747 ± 0.018	1.487 ± 0.008	0.078 ± 0.001	1.468 ± 0.017	2.351 ± 0.026	24.900 ± 0.994
TsndHop4	1.694 ± 0.061	2.437 ± 0.017	4.064 ± 0.059	1.490 ± 0.007	0.097 ± 0.002	12.255 ± 0.215	15.856 ± 0.09	24.900 ± 0.718
TsndProbe	0.756 ± 0.007	1.772 ± 0.006	2.808 ± 0.019	1.505 ± 0.008	0.104 ± 0.002	1.687 ± 0.033	2.032 ± 0.014	24.267 ± 0.691
Tsnd_CeHop4	0.729 ± 0.007	1.887 ± 0.008	2.671 ± 0.016	1.527 ± 0.008	0.076 ± 0.001	3.749 ± 0.086	6.130 ± 0.094	23.000 ± 1.054
Trcv_CeHop2	0.798 ± 0.003	1.792 ± 0.003	2.953 ± 0.013	1.553 ± 0.006	0.085 ± 0.001	10.511 ± 0.046	15.402 ± 0.17	21.500 ± 1.080
Trcv_CeNbor	0.938 ± 0.008	1.809 ± 0.003	3.391 ± 0.022	1.599 ± 0.008	0.069 ± 0.001	2.737 ± 0.013	4.959 ± 0.059	19.200 ± 1.135
Trcv_CeHop4	0.777 ± 0.003	1.851 ± 0.004	2.822 ± 0.007	1.637 ± 0.006	0.101 ± 0.001	36.724 ± 0.107	45.356 ± 0.50	17.300 ± 1.059
GsndBCast	0.437 ± 0.006	2.325 ± 0.010	2.599 ± 0.014	1.704 ± 0.016	0.216 ± 0.002	75.965 ± 0.739	43.913 ± 0.34	14.000 ± 1.633
GrcvBCast	0.381 ± 0.003	1.976 ± 0.006	2.475 ± 0.013	1.744 ± 0.016	0.324 ± 0.002	154.778 ± 0.476	59.679 ± 0.56	11.900 ± 1.197
NoBal	1.990 ± 0.058	2.002 ± 0.012	5.527 ± 0.102	1.985 ± 0.023				

Table 6.12 System Behaviour for a 36 Processor Mesh Model at 50% Loading

	Variance	MeanLd	Difference	RunTime	Mig/s	Tx/s	Hit Ratio	SpeedUp
GrcvNbor	1.388 ± 0.029	2.737 ± 0.023	4.457 ± 0.052	2.007 ± 0.031	0.424 ± 0.002	5.597 ± 0.033	1.655 ± 0.005	59.600 ± 1.174
GsndNbor	1.406 ± 0.030	2.764 ± 0.029	4.175 ± 0.057	2.043 ± 0.031	0.419 ± 0.004	5.427 ± 0.056	1.622 ± 0.005	58.600 ± 0.699
TsndNbor	1.781 ± 0.093	2.891 ± 0.017	5.444 ± 0.192	2.088 ± 0.022	0.396 ± 0.003	3.736 ± 0.037	1.183 ± 0.004	58.150 ± 0.671
GsndHop2	1.071 ± 0.021	2.976 ± 0.029	4.041 ± 0.034	2.093 ± 0.041	0.453 ± 0.003	19.809 ± 0.128	5.476 ± 0.016	58.000 ± 1.333
Trcv_CeHop2	1.530 ± 0.031	2.824 ± 0.019	5.029 ± 0.066	2.211 ± 0.029	0.326 ± 0.003	10.775 ± 0.046	4.140 ± 0.053	55.400 ± 0.966
GrcvHop2	1.215 ± 0.031	2.892 ± 0.033	4.304 ± 0.046	2.225 ± 0.046	0.492 ± 0.003	33.455 ± 0.136	8.521 ± 0.039	55.200 ± 1.135
Tsnd_CeNbor	2.309 ± 0.097	2.889 ± 0.025	6.360 ± 0.162	2.281 ± 0.035	0.204 ± 0.001	2.313 ± 0.030	1.420 ± 0.015	54.100 ± 1.287
TsndHop2	5.069 ± 0.120	3.978 ± 0.030	9.612 ± 0.119	2.339 ± 0.024	0.396 ± 0.005	14.384 ± 0.163	4.549 ± 0.032	53.100 ± 0.912
Trcv_CeNbor	2.196 ± 0.043	2.913 ± 0.020	6.058 ± 0.089	2.354 ± 0.029	0.262 ± 0.003	3.325 ± 0.014	1.590 ± 0.016	52.500 ± 1.080
Tsnd_CeHop2	2.283 ± 0.079	3.071 ± 0.026	6.439 ± 0.112	2.389 ± 0.038	0.186 ± 0.001	6.822 ± 0.117	4.599 ± 0.072	51.800 ± 1.398
Trcv_CeHop4	1.537 ± 0.038	3.022 ± 0.023	5.200 ± 0.078	2.432 ± 0.030	0.360 ± 0.003	31.549 ± 0.202	10.975 ± 0.131	51.100 ± 1.287
TsndHop2/t3	3.894 ± 0.126	3.765 ± 0.033	7.938 ± 0.164	2.503 ± 0.028	0.267 ± 0.004	9.215 ± 0.161	4.323 ± 0.026	49.800 ± 1.056
TsndProbe	1.580 ± 0.016	3.155 ± 0.027	4.815 ± 0.046	2.560 ± 0.039	0.484 ± 0.004	10.251 ± 0.178	2.652 ± 0.028	48.538 ± 1.140
Tsnd_CeProbe	1.841 ± 0.087	3.247 ± 0.042	5.835 ± 0.134	2.700 ± 0.059	0.267 ± 0.003	12.408 ± 0.256	5.823 ± 0.186	45.500 ± 1.434
Random	3.652 ± 0.195	3.304 ± 0.044	7.819 ± 0.208	2.881 ± 0.050	0.514 ± 0.006	4.102 ± 0.049	1.000	42.280 ± 1.061
TsndHop3	30.529 ± 0.890	6.578 ± 0.079	22.725 ± 0.438	3.212 ± 0.115	0.409 ± 0.004	38.847 ± 0.621	11.904 ± 0.180	35.450 ± 2.743
NoBal	20.591 ± 1.041	5.061 ± 0.065	18.506 ± 0.662	4.988 ± 0.085				

Table 6.13 System Behaviour for a 36 Processor Mesh Model at 80% Loading

	Variance	MeanLd	Difference	RunTime	Mig/s	Tx/s	Hit Ratio	SpeedUp
Trcv_CeHop2	3.644 ± 0.176	3.824 ± 0.045	8.277 ± 0.190	3.085 ± 0.071	0.358 ± 0.003	8.460 ± 0.091	2.961 ± 0.016	70.300 ± 1.252
GsndNbor	2.751 ± 0.069	3.996 ± 0.065	6.339 ± 0.073	3.167 ± 0.090	0.526 ± 0.004	7.115 ± 0.049	1.694 ± 0.004	69.600 ± 1.578
Trcv_CeNbor	5.113 ± 0.166	3.988 ± 0.032	9.739 ± 0.210	3.314 ± 0.033	0.296 ± 0.003	2.813 ± 0.032	1.192 ± 0.006	68.250 ± 0.886
TsndNbor	7.474 ± 0.830	4.319 ± 0.092	12.415 ± 0.776	3.424 ± 0.101	0.449 ± 0.006	5.579 ± 0.048	12.415 ± 0.237	67.154 ± 1.625
GrcvNbor	2.997 ± 0.092	4.160 ± 0.074	6.701 ± 0.098	3.451 ± 0.097	0.548 ± 0.004	11.998 ± 0.046	2.743 ± 0.018	67.000 ± 1.491
Trcv_CeHop4	5.043 ± 0.343	4.331 ± 0.061	9.573 ± 0.312	3.667 ± 0.093	0.327 ± 0.005	20.275 ± 0.461	7.777 ± 0.094	64.900 ± 1.595
Tsnd_CeNbor	8.949 ± 0.873	4.413 ± 0.100	12.769 ± 0.578	3.764 ± 0.121	0.187 ± 0.003	3.697 ± 0.053	2.474 ± 0.074	64.100 ± 1.729
GsndHop2	2.595 ± 0.164	5.352 ± 0.196	6.120 ± 0.149	4.436 ± 0.251	0.550 ± 0.003	24.432 ± 0.235	5.565 ± 0.030	57.400 ± 3.239
TsndProbe	11.316 ± 0.864	5.195 ± 0.144	13.788 ± 0.459	4.619 ± 0.165	0.193 ± 0.005	18.206 ± 0.156	94.329 ± 3.392	55.923 ± 1.754
Tsnd_CeHop2	15.591 ± 2.121	5.472 ± 0.165	16.333 ± 0.932	4.899 ± 0.144	0.144 ± 0.003	13.161 ± 0.227	11.452 ± 0.398	53.000 ± 1.897
TsndHop2	21.649 ± 2.753	6.741 ± 0.203	20.012 ± 1.341	5.523 ± 0.264	0.392 ± 0.011	23.735 ± 0.201	60.625 ± 2.079	47.077 ± 3.378
Tsnd_CeProbe	31.833 ± 1.973	7.290 ± 0.056	22.985 ± 0.653	6.849 ± 0.381	0.107 ± 0.001	19.476 ± 0.044	22.746 ± 0.257	33.500 ± 2.121
GrcvHop2	3.822 ± 0.217	10.823 ± 2.130	7.504 ± 0.218	10.175 ± 2.508	0.585 ± 0.004	36.147 ± 0.172	7.738 ± 0.049	2.400 ± 24.771
NoBal	106.151 ± 9.871	10.653 ± 0.255	42.476 ± 2.304	10.465 ± 0.272				

Table 6.14 System Behaviour for a 36 Processor Mesh Model at 90% Loading

6.2 Heavy-Weight Independent Process Model

In previous sections, the performance and relative ranking of the load balancing algorithms were considered with light-weight processes. One of the primary factors in this case is the effect of communication distance on average response time. Clearly, for the larger mesh models, a migration undertaken by an unrestricted algorithm may be successful in terms of correctly placing a process at the most underloaded site, but unsuccessful as a result of the communication cost outweighing the performance advantage. Therefore, it was considered important to examine the relative rankings of such algorithms against more restricted implementations in cases where the communication cost is of less significance. Whilst the communication parameters remained the same, the average run time of a process was increased to 10 seconds. Tables 6.15, 6.16, and 6.17 shows the performance results obtained for a 16-processor mesh at moderate, heavy, and extreme system loads respectively.

Under moderate system load (see Table 6.15) the speedup was significantly greater compared to light weight processes with increases in performance ranging between 20% and 47%. In addition, the results show the continued dominance of the global average algorithms but also the resurgence of the receiver-initiated global average implementation over greater distances in terms of load stability and run time performance. In contrast, the sender initiated implementations were most effective over shorter distances. The fixed threshold receiver-initiated algorithms were less effective than the sender initiated implementations and the event-based communicating set policies were marginally worse than their setless counterparts. However, it is the periodic communicating set algorithms which produced the worse performance in terms of their overall hit ratio and response times.

Whilst the results at moderate system loading show similarity in ranking between the algorithms for light-weight processes in terms of fixed threshold policies, there are differences under heavy load conditions. In Table 6.16 the improvement in performance for all algorithms is significantly greater varying between 32% and 73% compared to a maximum 60% for lightweight processes. Furthermore, the performances of the nearest-neighbour algorithms were inferior to the majority of regionalised implementations operating within a radius of two hops or more.

In terms of the algorithms *TsndNbor*, *Tsnd_CeNbor*, and *Tsnd_CpNbor*, the performances speedup were 57%, 55%, and 58% respectively with a correspondingly low variation in workload. The improved performance of *Tsnd_CpNbor* is due to the greater accuracy of the state information used which also results in a moderately higher process migration rate.

Under extreme load conditions (see Table 6.17) the speedup performance for all algorithms was in the range 44% to 80%. The global average algorithms exhibited a high level of stability but the nearest neighbour implementations were surpassed by the simple threshold probe algorithm and the fixed threshold receiver-initiated algorithms over a distance of two hops. A particular characteristic of the top ranking algorithms is their higher process migration rates of around four processes per second compared to between two and three processes for other algorithms.

	Variance	MeanLd	Difference	RunTime	Mig/s	Tx/s	Hit Ratio	SpeedUp
GrcvHop4	0.265 ± 0.002	1.523 ± 0.006	1.204 ± 0.011	10.489 ± 0.027	0.024 ± 0.001	14.33 ± 0.577	111.12 ± 3.992	46.800 ± 1.398
GrcvHop2	0.284 ± 0.003	1.525 ± 0.006	1.296 ± 0.019	10.525 ± 0.047	0.024 ± 0.001	6.031 ± 0.065	48.156 ± 1.310	46.700 ± 1.418
GrcvNbor	0.372 ± 0.012	1.564 ± 0.010	1.621 ± 0.044	11.265 ± 0.146	0.021 ± 0.000	1.851 ± 0.015	16.606 ± 0.298	42.800 ± 1.874
GsndNbor	0.416 ± 0.019	1.577 ± 0.021	1.764 ± 0.040	11.504 ± 0.168	0.020 ± 0.001	0.231 ± 0.057	2.157 ± 6.788	41.500 ± 1.764
GsndHop2	0.469 ± 0.015	1.581 ± 0.021	1.874 ± 0.037	11.626 ± 0.132	0.020 ± 0.001	0.860 ± 0.060	7.884 ± 1.545	41.000 ± 1.703
GsndHop4	0.488 ± 0.079	1.603 ± 0.019	1.910 ± 0.145	12.110 ± 0.265	0.019 ± 0.001	2.871 ± 0.037	28.129 ± 0.297	38.500 ± 1.633
Tsnd_CpNbor	0.594 ± 0.013	1.691 ± 0.011	2.051 ± 0.029	13.861 ± 0.109	0.008 ± 0.000	5.956 ± 0.003	133.07 ± 6.825	29.500 ± 2.224
Tsnd_CeProbe	0.592 ± 0.013	1.701 ± 0.023	1.999 ± 0.026	13.850 ± 0.164	0.010 ± 0.001	0.120 ± 0.013	2.302 ± 0.092	29.600 ± 1.647
TsndProbe	0.593 ± 0.032	1.701 ± 0.023	1.998 ± 0.076	13.821 ± 0.194	0.010 ± 0.001	0.121 ± 0.022	2.293 ± 0.235	30.300 ± 1.969
TsndHop2	0.641 ± 0.021	1.705 ± 0.022	2.193 ± 0.036	13.894 ± 0.157	0.008 ± 0.001	0.233 ± 0.021	5.579 ± 1.500	29.900 ± 2.898
TsndNbor	0.635 ± 0.027	1.703 ± 0.021	2.176 ± 0.055	13.907 ± 0.186	0.008 ± 0.000	0.073 ± 0.006	1.698 ± 0.333	29.600 ± 3.978
Tsnd_CeHop4	0.656 ± 0.021	1.717 ± 0.024	2.218 ± 0.050	14.118 ± 0.214	0.008 ± 0.000	0.309 ± 0.029	7.328 ± 0.467	28.300 ± 1.703
Tsnd_CeHop2	0.669 ± 0.038	1.720 ± 0.029	2.246 ± 0.078	14.152 ± 0.306	0.008 ± 0.001	0.138 ± 0.014	3.247 ± 0.136	28.100 ± 1.853
Tsnd_CpHop2	0.597 ± 0.014	1.715 ± 0.011	2.058 ± 0.031	14.368 ± 0.123	0.009 ± 0.000	22.41 ± 0.005	486.54 ± 20.232	26.900 ± 2.283
Tsnd_CeNbor	0.685 ± 0.029	1.727 ± 0.025	2.281 ± 0.066	14.263 ± 0.231	0.008 ± 0.001	0.054 ± 0.005	1.282 ± 0.028	27.600 ± 1.713
Random	0.685 ± 0.025	1.723 ± 0.024	2.256 ± 0.052	14.419 ± 0.186	0.010 ± 0.001	0.053 ± 0.022	1.000	27.200 ± 2.263
Trcv_CeHop4	0.684 ± 0.015	1.736 ± 0.022	2.248 ± 0.029	14.505 ± 0.156	0.009 ± 0.001	2.613 ± 0.044	53.117 ± 4.015	26.400 ± 1.955
TrcvHop2	0.728 ± 0.017	1.738 ± 0.010	2.359 ± 0.057	14.727 ± 0.090	0.008 ± 0.000	1.077 ± 0.007	23.930 ± 0.052	25.800 ± 1.780
Trcv_CeHop2	0.754 ± 0.021	1.752 ± 0.022	2.431 ± 0.041	14.865 ± 0.139	0.009 ± 0.001	0.923 ± 0.018	20.304 ± 1.344	24.500 ± 1.900
Trcv_CpHop2	0.753 ± 0.013	1.781 ± 0.014	2.406 ± 0.027	15.552 ± 0.113	0.009 ± 0.000	23.13 ± 0.010	480.85 ± 15.825	21.100 ± 1.792
TrcvNbor	0.877 ± 0.014	1.774 ± 0.016	2.709 ± 0.028	15.459 ± 0.216	0.007 ± 0.001	0.280 ± 0.003	7.578 ± 0.089	22.600 ± 2.201
Trcv_CeNbor	0.925 ± 0.039	1.795 ± 0.030	2.802 ± 0.064	15.675 ± 0.265	0.007 ± 0.001	0.254 ± 0.007	6.849 ± 0.533	20.300 ± 2.214
Trcv_CpNbor	0.902 ± 0.040	1.790 ± 0.017	2.743 ± 0.076	15.723 ± 0.226	0.007 ± 0.000	6.149 ± 0.004	166.72 ± 5.646	20.200 ± 2.573
Tsnd_CpHop4	0.641 ± 0.010	1.810 ± 0.011	2.160 ± 0.028	16.205 ± 0.100	0.010 ± 0.000	64.96 ± 0.021	1233.04 ± 44.8	17.600 ± 2.319
NoBal	2.022 ± 0.068	2.005 ± 0.020	4.398 ± 0.130	19.706 ± 0.523				

Table 6.15 Heavyweight Processes: 16-Processor Mesh Model at 50% Loading

	Variance	MeanLd	Difference	RunTime	Mig/s	Tx/s	Hit Ratio	SpeedUp
GrcvHop4	0.325 ± 0.010	2.050 ± 0.034	1.789 ± 0.037	13.011 ± 0.419	0.060 ± 0.001	11.201 ± 0.120	34.916 ± 1.510	72.900 ± 1.595
GrcvHop2	0.477 ± 0.019	2.146 ± 0.032	2.169 ± 0.038	14.185 ± 0.368	0.062 ± 0.002	5.027 ± 0.025	15.377 ± 0.554	70.200 ± 1.398
GsndHop2	0.716 ± 0.016	2.302 ± 0.036	2.655 ± 0.030	16.123 ± 0.368	0.053 ± 0.001	2.304 ± 0.029	8.133 ± 0.634	66.300 ± 1.430
GsndHop4	0.706 ± 0.023	2.369 ± 0.035	2.585 ± 0.050	16.951 ± 0.425	0.044 ± 0.001	6.117 ± 0.050	25.982 ± 0.312	64.500 ± 1.494
GrcvNbor	0.897 ± 0.020	2.378 ± 0.035	2.961 ± 0.036	17.069 ± 0.410	0.053 ± 0.001	1.688 ± 0.009	5.965 ± 0.253	64.300 ± 1.494
GsndNbor	0.964 ± 0.035	2.403 ± 0.054	3.090 ± 0.064	17.362 ± 0.504	0.049 ± 0.001	0.626 ± 0.077	2.417 ± 2.744	63.700 ± 1.663
TsndProbe	0.860 ± 0.289	2.419 ± 0.077	2.843 ± 0.356	17.887 ± 0.659	0.032 ± 0.000	0.567 ± 0.046	3.334 ± 0.465	62.700 ± 2.221
Tsnd_CeProbe	0.899 ± 0.128	2.458 ± 0.064	2.928 ± 0.177	18.079 ± 0.662	0.033 ± 0.001	0.591 ± 0.037	3.402 ± 0.210	62.200 ± 1.317
Trcv_CeHop4	0.973 ± 0.040	2.542 ± 0.046	3.125 ± 0.075	19.125 ± 0.341	0.036 ± 0.001	2.959 ± 0.031	15.560 ± 0.811	60.000 ± 1.155
TrcvHop2	0.927 ± 0.042	2.531 ± 0.042	3.078 ± 0.068	19.083 ± 0.391	0.037 ± 0.001	1.747 ± 0.013	8.949 ± 0.089	60.000 ± 1.337
Trcv_CeHop2	1.185 ± 0.066	2.601 ± 0.053	3.557 ± 0.116	19.836 ± 0.431	0.033 ± 0.001	1.102 ± 0.017	6.325 ± 0.408	58.500 ± 1.179
Tsnd_CpNbor	1.430 ± 0.096	2.603 ± 0.044	3.903 ± 0.153	19.832 ± 0.449	0.027 ± 0.001	6.101 ± 0.007	42.270 ± 1.878	58.400 ± 1.838
Tsnd_CpHop2	1.231 ± 0.156	2.647 ± 0.065	3.619 ± 0.214	20.385 ± 0.579	0.028 ± 0.001	22.454 ± 0.014	149.089 ± 6.026	57.400 ± 1.955
TsndNbor	1.891 ± 0.126	2.647 ± 0.055	4.504 ± 0.179	20.564 ± 0.533	0.020 ± 0.001	0.275 ± 0.007	2.536 ± 0.099	56.800 ± 1.912
Trcv_CpHop2	0.974 ± 0.031	2.699 ± 0.029	3.223 ± 0.048	21.073 ± 0.426	0.039 ± 0.001	22.944 ± 0.016	110.451 ± 4.419	55.900 ± 1.969
TsndHop2	2.203 ± 0.020	2.707 ± 0.040	4.854 ± 0.040	21.291 ± 0.359	0.018 ± 0.001	0.942 ± 0.029	9.617 ± 0.386	55.400 ± 1.333
Tsnd_CeHop2	1.875 ± 0.278	2.705 ± 0.084	4.453 ± 0.258	20.998 ± 0.807	0.020 ± 0.001	0.562 ± 0.029	5.324 ± 0.293	56.200 ± 2.044
TrcvNbor	1.577 ± 0.163	2.722 ± 0.038	4.161 ± 0.191	21.458 ± 0.431	0.030 ± 0.001	0.457 ± 0.006	2.911 ± 0.206	55.100 ± 2.068
Tsnd_CeNbor	2.112 ± 0.226	2.750 ± 0.075	4.790 ± 0.225	21.553 ± 0.711	0.020 ± 0.000	0.212 ± 0.009	1.985 ± 0.082	54.900 ± 1.370
Trcv_CpNbor	1.602 ± 0.101	2.774 ± 0.043	4.205 ± 0.133	21.901 ± 0.403	0.030 ± 0.001	6.231 ± 0.004	38.851 ± 1.740	54.200 ± 1.476
Tsnd_CeHop4	2.048 ± 0.233	2.750 ± 0.074	4.654 ± 0.218	21.629 ± 0.713	0.018 ± 0.000	1.450 ± 0.077	14.913 ± 0.775	54.800 ± 1.751
Trcv_CeNbor	1.975 ± 0.219	2.808 ± 0.070	4.639 ± 0.208	22.289 ± 0.638	0.026 ± 0.001	0.332 ± 0.005	2.381 ± 0.114	53.500 ± 1.716
Random	2.274 ± 0.061	2.860 ± 0.053	4.913 ± 0.087	23.136 ± 0.564	0.044 ± 0.001	0.237 ± 0.015	1.000	51.500 ± 1.969
Tsnd_CpProbe	1.596 ± 0.176	2.991 ± 0.074	4.096 ± 0.241	24.655 ± 0.775	0.033 ± 0.001	45.321 ± 0.015	260.107 ± 9.255	48.500 ± 2.321
NoBal	18.565 ± 0.018	4.901 ± 0.034	14.178 ± 0.029	47.818 ± 0.375				

Table 6.16 Heavyweight Processes: 16 Processor Mesh Model at 80% Loading

	Variance	MeanLd	Difference	RunTime	Mig/s	Tx/s	Hit Ratio	SpeedUp
GrcvHop4	0.462 ± 0.009	2.764 ± 0.106	2.250 ± 0.023	19.483 ± 1.154	0.069 ± 0.001	9.602 ± 0.090	26.153 ± 1.065	80.000 ± 2.563
GrcvHop2	0.761 ± 0.043	2.867 ± 0.114	2.794 ± 0.072	20.543 ± 1.259	0.075 ± 0.001	4.496 ± 0.069	11.215 ± 0.562	78.875 ± 2.532
TrcvHop2	1.328 ± 0.121	3.101 ± 0.110	3.764 ± 0.136	23.438 ± 1.003	0.049 ± 0.001	1.844 ± 0.020	7.094 ± 0.091	76.300 ± 2.696
GsndHop4	0.924 ± 0.053	3.148 ± 0.087	3.093 ± 0.082	23.729 ± 0.832	0.051 ± 0.001	5.876 ± 0.046	21.346 ± 0.317	75.750 ± 2.357
GrcvNbor	1.378 ± 0.104	3.158 ± 0.092	3.756 ± 0.123	23.737 ± 0.937	0.066 ± 0.001	1.584 ± 0.011	4.491 ± 0.172	75.875 ± 2.696
Trcv_CeHop4	1.708 ± 0.447	3.239 ± 0.171	4.268 ± 0.422	24.657 ± 1.455	0.045 ± 0.001	2.598 ± 0.113	10.854 ± 0.517	74.900 ± 2.132
GsndHop2	0.983 ± 0.220	3.018 ± 0.537	3.212 ± 0.488	22.219 ± 4.320	0.064 ± 0.010	2.507 ± 0.429	7.374 ± 3.793	77.125 ± 5.401
Trcv_CpHop2	1.869 ± 0.654	3.326 ± 0.125	4.168 ± 0.433	25.526 ± 1.348	0.047 ± 0.001	12.192 ± 0.027	48.192 ± 3.012	73.750 ± 2.816
Tsnd_CeProbe	2.343 ± 0.684	3.287 ± 0.219	4.875 ± 0.578	25.045 ± 2.071	0.035 ± 0.001	0.984 ± 0.059	5.268 ± 0.537	74.400 ± 2.413
Trcv_CeHop2	2.338 ± 0.645	3.374 ± 0.168	5.035 ± 0.487	26.129 ± 1.418	0.041 ± 0.001	0.988 ± 0.030	4.534 ± 0.242	73.600 ± 2.633
Trcv_CpNbor	3.137 ± 0.567	3.521 ± 0.088	5.865 ± 0.341	27.862 ± 0.856	0.037 ± 0.001	3.304 ± 0.006	16.744 ± 0.897	71.625 ± 2.925
GsndNbor	1.578 ± 0.200	3.234 ± 0.591	4.009 ± 0.443	24.555 ± 4.882	0.059 ± 0.008	0.761 ± 0.638	2.414 ± 17.042	74.875 ± 6.064
Tsnd_CpNbor	5.263 ± 0.728	3.627 ± 0.128	7.279 ± 0.297	28.722 ± 1.361	0.028 ± 0.001	3.280 ± 0.009	22.141 ± 1.485	70.286 ± 3.094
TrcvNbor	2.909 ± 1.978	3.489 ± 0.261	5.733 ± 0.929	27.621 ± 2.619	0.038 ± 0.002	0.469 ± 0.009	2.336 ± 0.190	71.900 ± 2.974
Trcv_CeNbor	4.026 ± 0.929	3.735 ± 0.189	6.763 ± 0.636	30.044 ± 1.777	0.031 ± 0.001	0.302 ± 0.009	1.801 ± 0.076	69.400 ± 3.098
TsndProbe	2.574 ± 4.290	3.291 ± 0.893	5.002 ± 2.835	25.290 ± 7.876	0.035 ± 0.003	0.967 ± 0.404	5.256 ± 3.961	74.300 ± 8.859
TsndNbor	7.603 ± 0.580	3.881 ± 0.149	8.638 ± 0.475	32.249 ± 1.666	0.018 ± 0.001	0.381 ± 0.011	3.902 ± 0.093	67.200 ± 2.558
Tsnd_CeHop2	5.971 ± 1.780	3.930 ± 0.310	7.977 ± 1.061	31.969 ± 3.013	0.019 ± 0.001	0.934 ± 0.060	9.142 ± 0.987	67.500 ± 2.759
Tsnd_CeNbor	7.149 ± 2.362	4.065 ± 0.355	8.789 ± 1.212	33.471 ± 3.535	0.020 ± 0.001	0.331 ± 0.017	3.176 ± 0.327	66.100 ± 3.665
Tsnd_CpProbe	5.887 ± 1.303	4.291 ± 0.245	7.458 ± 0.543	35.958 ± 2.571	0.028 ± 0.001	23.983 ± 0.027	160.857 ± 10.102	63.125 ± 5.055
GeTsndNbor	3.773 ± 1.103	3.847 ± 0.765	6.237 ± 1.241	32.096 ± 6.516	0.023 ± 0.006	0.444 ± 0.091	3.657 ± 0.424	67.100 ± 7.852
Random	9.405 ± 0.448	4.604 ± 0.606	9.974 ± 0.729	39.303 ± 4.876	0.068 ± 0.009	0.365 ± 0.169	5.345 ± 0.999	60.200 ± 6.223
Tsnd_CpHop4	18.441 ± 9.502	5.946 ± 0.856	12.652 ± 2.623	54.052 ± 9.154	0.019 ± 0.002	35.539 ± 0.294	356.984 ± 37.769	43.875 ± 15.292
NoBal	90.864 ± 0.031	10.078 ± 0.105	31.138 ± 0.052	98.821 ± 1.013				

Table 6.17 Heavyweight Processes: 16-Processor Mesh Model at 90% Loading

The run time performance for light weight processes are summarised in Figures 6.2 and 6.3 for a selection of algorithms against mesh sizes under low and extreme load conditions respectively. The graph in Figure 6.2 show that the performance of all algorithms, even the random policy, are scalable and produced a speedup factor of at least 20% for all network sizes. Further, the performance of the nearest-neighbour algorithms for both the restricted global and threshold policies were consistently high and unaffected by network size.

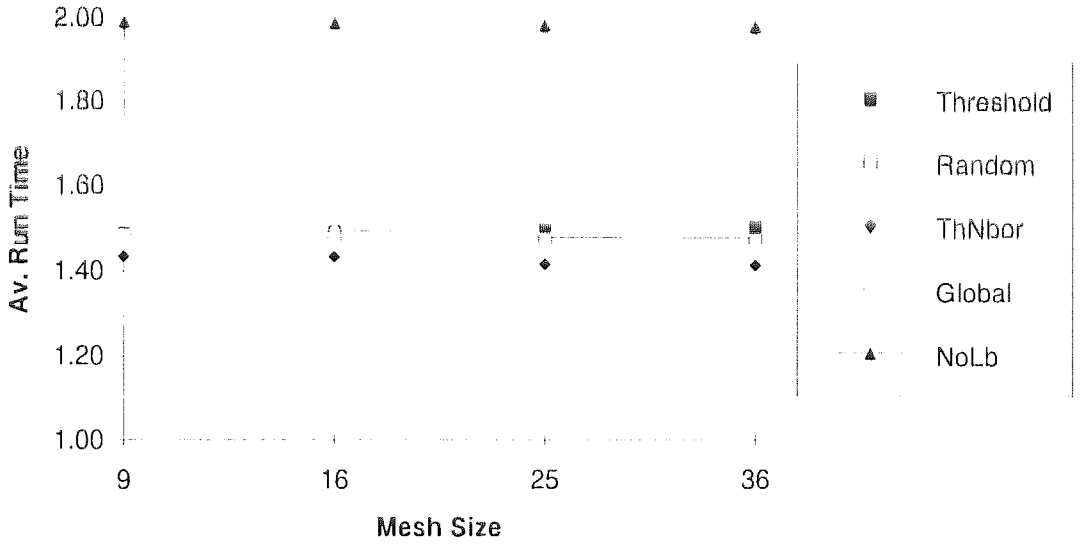


Figure 6.2: The Scalability of Algorithms at 50% Loading

Under extreme system loads (see Figure 6.3), the effectiveness of the random policy is clearly affected by system size. Its performance proved to be unscalable with a response time that exhibit exponential growth against network size. Further, its performance is significantly worse than the no load balancing case for network sizes greater than 16 processors. In contrast, the algorithms that engage in negotiation

within a limited network domain were scalable and produced consistently high speedup performance of at least 50%.

The broadcast algorithms were quite effective at low to moderate system load, but became increasingly unstable at high system load due to the greater intensity of the communication traffic generated. This is a problem that is further compounded with larger network sizes. It is concluded therefore, that the performance of global broadcast algorithms are not scalable across different network sizes especially in environments where light-weight processes are dominant. Further, with the exception of nearest-neighbour implementations, the periodic communicating set model proved to be unscalable also, because of the excessive message traffic it generates.

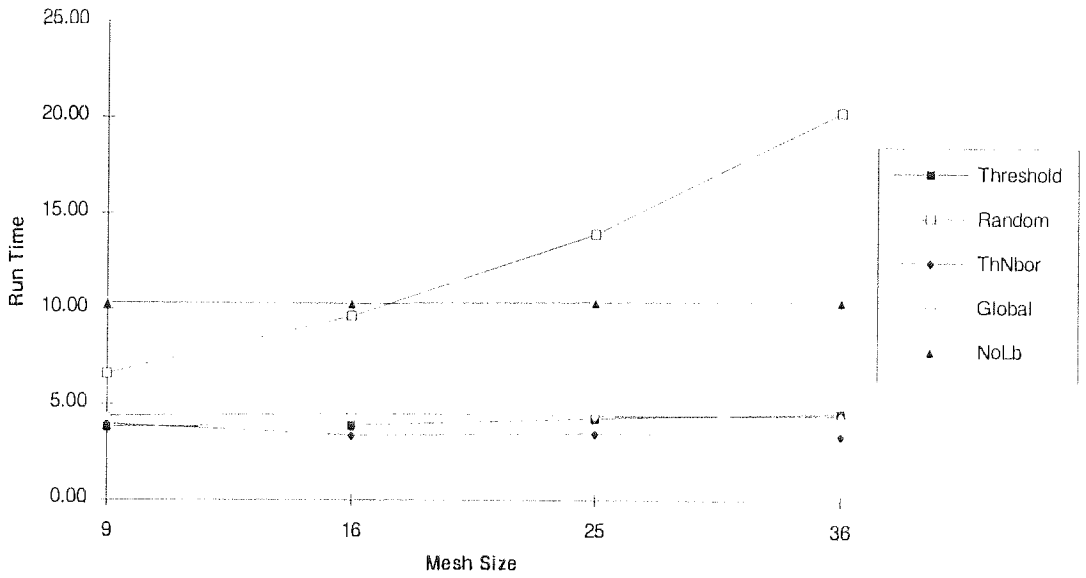


Figure 6.3: The Scalability of Algorithms at 90% loading.

The event-based communicating set model proved to be scalable across network sizes exhibiting better load stability than the non-communicating set model, but a poorer response time. Further, the sender-initiated algorithms were more effective under

light system load whilst the receiver-initiated model performed well at high system load. Thus, an adaptive communicating set model able to take advantage of this fact would be beneficial. However, it may be necessary for a host to maintain sets for underloaded and overloaded hosts. The load balancing mode could then be selected on the basis of the larger set size. The main disadvantage of this method is its additional complexity. Firstly, the storage spaced required for the sets would increase and additional effort invested in ensuring that the two sets are up to date.

On the basis of the results produced for heavy-weight processes, where the communication delay relative to the process service time is of less significance, the performance of all algorithms are exceptional. In particular, the performance of policies operating over greater distances is expected to scale with network size. However, one would expect the critical mass for such policies to be reached with network sizes of hundreds and thousands of processors. Therefore, one would expect algorithms that operate within a prespecified boundary to be more effective.

The global average algorithm is scalable both in terms of the network size, the load conditions, and the workload characteristics of this study. The restricted algorithms have demonstrated consistently low load variance and good response times. It may be argued, that the results of the global policy represent the most optimistic performance achievable given its reliance on identifying processes with the longest remaining service time, preemptive migrations, and reliable timers. Therefore, the performance of simple fixed threshold policies over different network sizes are more representative of the realistic level of optimisation achievable through load balancing. Certainly, the response times of such algorithms are within range of the global policies, but there is no denying the fact that system stability is enhanced considerably using the global average algorithm. Its performance in this respect cannot be improved on by even the

relatively simple communicating set model developed in this study. It is generally the case that by selecting short intensive CPU processes for migration using a "round robin" discipline, the response time of the global average algorithm did not significantly improve on that of the simple threshold algorithm. Nevertheless, the variance in workload was much lower.

CHAPTER 7

SUMMARY AND CONCLUSIONS

7.1 SUMMARY

In this study a range of load balancing algorithms have been studied with particular emphasis on the scalability of performance across various system sizes and the relative behavioural stability across a range of workload intensity. The distributed system size ranged from a small nine processor mesh to a much larger 36 processor network. The algorithms considered included a selection of sender-initiated and receiver-initiated policies of varying complexity. Algorithms from the simple threshold policy to the more complex global average algorithms were examined and adapted to enhance their useability and scalability. The experiments conducted covered light to moderate, and heavy to extreme system load conditions. The majority of experiments used short CPU-intensive processes, but "heavy-weight" processes of ten seconds duration were also used in the 16 processor network model.

Clearly, from the results obtained, real performance benefits are attainable from the simplest of algorithms to the more complex. The speedup performance was typically in the range of six percent at light system load to 72% under extreme system loading. However, near perfect speedup was unattainable given the resulting communication overheads associated with the load balancing activity. The sender-initiated algorithms were most effective in instances where the system was lightly loaded, whilst receiver-initiated policies produced their best response times under heavy system loads. The sender-initiated random policy, which represented the most basic load sharing

strategy, produced highly respectable performance results at relatively low and moderate system load, but was found to be inferior to algorithms that engaged in negotiation with potential hosts in the transfer of processes under heavy and extreme load conditions.

The load sharing policies that adopted a variable load threshold value exhibited better load distribution and system stability across all system loads. The more complex global average policies attempted to combine both receiver and sender initiated strategies by means of a variable threshold value. However, those implementations that rely on conducting process negotiation, and maintaining state information across the complete network did not scale particularly well even under moderate system load. For example, the global average algorithm exhibited good load distribution characteristics and matching response time performances across a range of network sizes. But, in using the broadcast mechanism under heavy system load, significant message traffic was generated and the resulting network delay had a tendency to initiate the "race condition" and system instability, especially under extreme load conditions. Thus, to minimise the risk of instability and, at the same time maximise performance, the regionalised global average algorithms were developed. Such policies were found to be most effective across all system load and system sizes within a radius distance of two hops.

It could be argued that the consistent performance attainment of the global average policy was the result of using a preemptive process migration strategy. Therefore, its performance would have been achieved at considerable expense in terms of managing any potential incidence of residual dependency. In this study only independent, CPU intensive processes were considered, rendering residual dependency of less significance. Given more I/O intensive, and cooperative process groups one can

expect the performance of the global average algorithm to degrade across all system loads, and in some cases reach unacceptable levels. In this regard, simple non-preemptive algorithms that are able to compete with the more complex global average policies are worthy of consideration. In this study, simple fixed threshold algorithms that make use of "local" state information were considered. It is true to say that of the simplest "negotiating" algorithms, *SndNbor* was consistently in the top three in terms of its overall response time performance for all system loads.

Whilst these results may support the view that the response time performance of simple algorithms are not generally bettered by more complex algorithms, one should not underestimate the importance of ensuring load stability across all hosts. It is in areas such as this that the global average algorithms have proved successful. The research has demonstrated that regionalised algorithms are scalable using partial but relatively accurate information without sacrificing overall performance . However, global policies based on state information from immediate neighbours represent the minimum level of state information required. Preliminary work conducted to date indicates that state information collected from remote hosts that are at most two hops away from the local host exhibit the best performance.

To attain minimum communication overheads, the load balancing algorithm should only be executed if there is a propensity for successful pairing of underloaded and overloaded sites relative to their difference in workload. That is, load balancing is invoked more frequently, the greater the difference in loading between various sites, and less often in cases of marginal workload differences. The communicating set algorithms attempted to keep track of successful pairings, but their performance was limited by the level of membership commonality between the sets for different sites. Thus, at high system loads, an underloaded site will tend to be common amongst the

sets of many overloaded sites. It is possible to limit the frequency of activation such that the policy operates only when the complete communicating set has been constructed for an overloaded or underloaded site. The problem with this method is that set membership and set size are very difficult to predict given the workload characteristics, the communication patterns, and the load conditions. Further, the load state for the members of the set must be accurate and stable over time. Therefore, the implementation used invoked the policy whenever an imbalanced load state is detected, even if this results in merely updating the load state for members of the current set. Performance maximisation may require all underloaded nodes to be paired with overloaded nodes and migration effected between them. At low system load each host would need to retain set membership capacity for the complete network. Thus, the tables required for such sets will grow arithmetically with the network size. In this study, the average set size was typically four processes although a limit of nine was set for the regionalised algorithms. In the majority of experiments conducted the communicating set implementation performed better than their equivalent non-communicating set algorithms.

7.2 Conclusions

Load balancing is an important resource management activity for any modern distributed system. The results of this study reaffirms the findings of other researchers in the field highlighting the performance benefits of such policies to system users. Modern distributed operating systems have mechanisms in place for facilitating load sharing. However, the implementation of policies has been rather sparse and, in some cases, limited to merely utilising spare processing capacity for limited periods, via a centralised scheduler. Such a cautious approach reflects the limited and incomplete knowledge and information currently available to system

developers about resource management in a loosely-coupled distributed system environment. Further research is required in areas such as: the range and characteristics of load sharing algorithms and user processes; process group interaction and thread-based scheduling; and data management and residual dependency in distributed environments.

This study has attempted to highlight the behavioural characteristics of a selection of load sharing policies in loosely-coupled environments. In particular, special emphasis was given to their scalability, on the basis that the distributed operating systems of the future will have responsibility for systems consisting of tens of thousands of processors. In the words of Kriemen et al,

> "Algorithm stability, which is a precondition to scalability, is an indication of the ability of the algorithm to avoid poor allocation decisions." [Kriemen92].

The results of this study demonstrate that regionalised load balancing algorithms are scalable and relatively stable using a variable load threshold parameter bounded by a multicast distance of up to two hops. In addition, amongst the simple fixed threshold algorithms, the communicating set implementations performed well and further improvements are expected in environments where inequality in the workload generated by users is a persistent problem. The algorithm could be extended to conform to the ideal of the working set model [Denning82], used to manage the incidence of page faults, by suspending any load balancing activity until a requesting host is in possession of its complete set of remote hosts. However, given the high level of unpredictability in workload patterns for each host, size and composition of the set are related to the size of the network and the pattern of process migrations elsewhere in the network. On this basis, the results produced indicate that sets with a maximum size of nine members was adequate.

The study is limited to distributed systems using a mesh network configuration and does highlight some of the issues pertaining to message routing between hosts. Experiments have not been conducted for other topologies such as the hypercube, tree, or bus. It is anticipated that the algorithms developed in this study are generally portable and should produce results of a similar magnitude in other architectures. In terms of the single bus topology such as Ethernet, the cornerstone of many Commercial LANs, the level of network traffic generated through load balancing may result in more moderate improvements in performance. Further, the network model developed can be extended to explore issues in load sharing for Wide Area Networks (WANs) where dynamic routing, internetworking, and transmit-and-send reliability are important issues. These factors could become additional parameters for the location, selection, and transfer components of a load sharing strategy.

The workload characterisation in this study has focused on independent CPU intensive processes with minimal residual dependency problems. The study can be extended by also giving consideration to load sharing policies on systems whose workload is dominated by dynamic sets of interacting processes. Whilst some researchers have examined static member sets, where the set itself is the unit of distribution (for load sharing purposes), the development of process thread schedulers in modern operating systems creates new areas for load balancing research. For example, the effect of using set membership as the unit of distribution may well highlight issues on the merit and ranking of load balancing algorithms. The continued growth in object-oriented development and applications consisting of objects with static and dynamic sets of methods may speed up developments in this area.

This study sought to explore the relative ranking of simple and complex load balancing algorithms in terms of their scalability across a range of network sizes. The findings support the general observation made by Eager et al [Eager86], regarding the effectiveness of simple dynamic policies compared to more complex global average strategies. However, it is clear from the algorithms developed in this study that the simple fixed threshold policy is ineffective at high system load where network delay is significant and timers are necessary. Further, a nearest-neighbour implementation which relies on the propagation of the average through the network performed consistently well across all system sizes. Finally, the performance of the simple communicating set model is proportional to the level of workload inequality generated at individual sites. Thus, the greater the inequality the better the load distribution and expected response times.

One cannot underestimate the significant degradation in performance that can occur where load balancing is absent. Under heavy system load, instability is high and load variation can fluctuate between 20 and 100 processes with response times to match. Any load balancing strategy able to keep load variation below three processes, and demonstrate performance improvements of at least 30%, across all system sizes and system loads is worthy of serious consideration.

REFERENCES

AND

BIBLIOGRAPHY

REFERENCES

[Acetta86] Acetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., Young, M., "Mach: A New Kernel Foundation for UNIX Development", Proc. Summer 1986 USENIX Conference, 1986, pp93-112.

[Ahmad90] Ahmad, I., Ghafoor, A., "A Semi Distributed Load Balancing Scheme for Large Multicomputer Systems", 2nd IEEE Symposium on Parallel and Distributed Processing, IEEE Computer Society Press 1990, pp562-9.

[Ammar88] Ammar, H., Deng, S., "A Simple Dynamic Load Balancing Algorithm for Homogeneous Distributed Systems", Proceedings 1988 ACM 16th Annual Computer Science Conference, 23-25th Feb. 1988, pp314-319.

[Ali92] Ali, H., El-Rewini H, Khalil, K. "Controlled Job Migration in Load Balanced Distributed Systems", Proc. of the 25th Hawaii Int. Conf. on System Sciences, Vol 1, IEEE 92, pp585-92.

[APM93] "The ANSA Comuptational Model", AR.001.01, Architecture Projects Management Ltd, Feb.1993.

[Artsy89] Artsy, Y., Finkel, R., "Designing a Process Migration Facility: The Charlotte Experience", Computer, Sept.89, IEEE Publications, pp47-56.

[Barman89] Barman, C., Faruqui, M. N., "Dynamic Load Balancing Algorithm in Homogeneously Distributed Systems Using Donor-Acceptor Policy", Fourth IEEE Region 10 Int. Conf., 22-24 Nov.89, Vol.22, pp610-613.

[Blake92] Blake, B. A., "Assignment of Independent Tasks to Minimise Completion Time", Software Practice and Experience, Vol.22, No.9, Sept.1992, pp723-734.

[Byron94] Byron, D., "Object Management Group Common Object Request Broker Architecture", DATAPRO Unix & Open Systems Service, McGraw-Hill Inc., Feb. 1994, pp101-109.

[Cheriton88] Cheriton, D., R., "The V Distributed System", Communications of the ACM, Vol.31, No.3, March 1988, pp314-333.

[Ciciani88] Ciciani, B., Dias D. M., YU, P., "Load Sharing in Hybrid Distributed Centralized Database Systems", Proc. IEEE 8th Int. Conf. on Distributed Computing Systems, Computer Society Press, 1988, pp274-281.

[Ciciani92] Ciciani, B., Dias D. M., YU, P., "Dynamic and Static Load Sharing in Hybrid Distributed Centralized Database Systems", Computer Systems Science and Engineering, Vol.7, No.1, Jan. 1992, pp25-41.

[Cocke88] Cocke, J., "The Search For Performance in Scientific Processors: Turing Award Lecture", Communications of the ACM, Vol.31, No.3, March 1988, pp250-253.

[Comer88] Comer, D., "Internetworking With TCP/IP: Principles, Protocols, and Architecture", 1988, Prentice-Hall.

[Comer91] Comer, D., Stevens, D., "Internetworking With TCP/IP Volume II: Design, Implementation, and Internals", 1991, Prentice-Hall.

[Cybenko89] Cybenko, G., "Dynamic Load Balancing for Distributed Memory Multiprocessors", Journal of Parallel and Distributed Computing, Vol.7 1989. pp 279-301.

[Denning82] Denning, P.J., "Working Sets Past and Present", IEEE Transactions on Software Engineering, 8(9), July 1982., pp 401 - 412.

[Dennis80] Dennis, J.B., "Data Flow Supercomputers", IEEE Trans. Computers, Nov.1980.

[Douglis91a] Douglis, f., Ousterhout, J.K., Kaashoek, M.F., Tanenbaum, A.S., "A Comparison of Two Distributed Systems: Amoeba and Sprite", Computing Systems, Vol. 4, Fall 1991, pp353-384.

[Douglis91b] Douglis, F., Ousterhout, J.K., "Transparent Process Migration: Design Alternatives and the Sprite Inplementation", Software Practice and Experience, Vol.21, Aug.1991, pp757-785.

[Dennis80] Dennis, J. B., "Data Flow Supercomputers", IEEE Computer, 13(11), Nov.1980, pp48-56.

[Djordjevic85] Djordjevic, J., Barbacci. M., Hosler, B., "A PMS Level Notation for the Description and Simulation of Digital Systems", The Computer Journal, Vol.28, No.4, August 1985, pp357-365.

[Dikshit89] Dikshit, P., Tripathi, S.K., Jaloti, P., "SAHAYOG: A test bed for Evaluating Dynamic Load Sharing Policies", Software Practice and Experience, 19(5), 1989. pp411-435.

[Eager86] Eager, D. L., Lazowska, E. D., Zahorjan, J., "Adaptive Load Sharing in Homogeneous Distributed Systems", IEEE Transactions on Software Engineering, Vol.SE-12, No.5, 1986, pp 662-675.

[Eager88] Eager, D. L., Lazowska, E. D., Zahorjan, J., "The Limited Performance Benefit of Migrating Active Processes for Load Sharing", Proc. ACM Sigmetrics,, May.1988, pp63-72.

[Fallmyr91] Fallmyr, T., Holden D., Anshus O., "Capturing the Behaviour of Distributed Systems", Proc. 1st EUUG Conf., EurOpen 91, pp167-83.

[Ferrari87] Ferrari, D., Zhou, S., "An Empirical Investigation of Load Indices for Load Balancing Applications", Technical Report UCB/CSD 87/353, University of California at Berkeley, May 1987.

[Finkel90] Finkel, D., "Modelling Dynamic Load Sharing in a Distributed Computing System", Computer Systems Science and Engineering, Vol.5, No.2, April 1990, pp89-94.

[Flynn72] Flynn, M.J., "Some Computer Organizations and Their Effectiveness.", IEEE Transactions on Computers, C-21, Sept.1972, pp948-960.

[Galletly90] Galletly, J., "Occam 2", Pitman Publishing, 1990.

[Gopinath91] Gopinath, P., Gupta, R., "A Hybrid Approach to Load Balancing in Distributed Systems", SEDMS 2: Symposium on Experiences with Distributed and Multiprocessor Systems., March 91, USENIX Association, pp133-147.

[Goscinski90] Goscinski A, Bearman, M., "Resource Management in Large Distributed Systems", Operating Systems Review, 24(4), Oct.1990, pp7-33.

[Gurd85] Gurd, J. R., Watson, I., Kirkham C., "The Manchester Prototype Data Flow Computer", Communications of the ACM, 28(1), Jan.85, pp34-52.

[Hac90] Hac' A., Johnson, T., "Sensitivity Study of the Load Balancing Algorithm in a Distributed System", Journal of Parallel and Distributed Computing, 10(1), 1990, pp85-89.

[Hac91] Hac' A., Jin, X., "A Decentralised Algorithm for Dynamic Load Balancing with File Transfer", Journal of Systems and Software, 16(1), Sept.91, pp37-52.

[Harinarayan91] Harinarayan, V., Kleinrock L., "Load Sharing in Limited Access Distributed Systems", Performance Evaluation Review, 19(1), May 91. pp 21-30.

[Hatch90] Hatch D, Finkel, D., "Load Indices for Load Sharing in Heterogeneous Distributed Computing Systems", Proc. of the 1990 UKSC Conf. on Computer Simulation, UK Simulation Council, Sept.90, pp202-6.

[Hockney88] Hockney, R.W., and Jesshope, C.R., "Parallel Computers 2: Architecture, Programming and Algorithms", 1988, IOP Publishing Ltd.

[Johnson88] Johnson, I.D., "A Study of Adaptive Load Balancing Algorithms for Distributed Systems", Ph.D Thesis, Aston University, 1988.

[Johnson89a] Johnson, I.D., and Harget, A.J., "On The Performance Of Load Balancing Algorithms in Distributed Systems", Proc. Information Processing 89, IFIP, 1989. pp175- 180.

[Joosen90] Joosen, W., Verbaeten P., "On the Use of Process Migration in Distributed Systems", Microprocessing and Microprogramming, 28(1-5), March 90, pp49 - 52.

[Juang89] Juang, J. -Y., Wah, B. W., "Load Balancing and Ordered Selections in a Computer System with Multiple Contention Buses", Journal of Parallel and Distributed Computing, Vol.7, Part 3, Dec. 1989, pp391-415.

[Kremien92b] Kremien, O., Kramer, J., "Methodical Analysis of Adaptive Load Sharing Algorithms", IEEE Transactions on Parallel and Distributed Systems, Vol.3, No.6, Nov.1992, pp747-760.

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

[Lavenburg83] Lavenburg, S., "Computer Performance Modelling Handbook", New York, USA, Academic Press, 1983.

[Levelt92] Levelt, W. G., et al "A Comparison of Two Paradigms for Distributed Shared Memory", Software Practice and Experience, Vol.22, No.11, Nov. 1992, pp985-1010.

[Leeuwen87] Leeuwen, J., V., Tan, R., B., "Interval Routing", The Computer Journal, Vol.30, No.4, August 1987, pp298-307.

[Lewis92] Lewis, T., G., El-rewini, H., "Introduction to Parallel Computing", Prentice Hall, 1992.

[Lin91a] Lin, H. -C ., Chiu, G -M., "Performance Study of Dynamic Load Balancing Policies for Distributed Systems With Service Interruptions", IEEE INFOCOM '91 Conference on Computer Communications, Vol.2, 1991, pp797-805.

[Lin91b] Lin, H. -C ., Raghavendra, C. S., "A Dynamic Load Balancing Policy with a Central Job Dispatcher", Proceedings 11th Int. Conf. on Distributed Computing Systems, 1991, pp264-271.

[Mirchandaney89] Mirchandaney, R., Towsley, D., Stankovic, J.A., "Analysis of the Effects of Delays on Load Sharing", IEEE Transactions on Computers, 38(11), Nov. 1989, pp1513 - 1525.

[Mittal89a] Mittal V.O., Singhal M, "SCATTERBRAIN: An Experiment in Distributed Problem Solving applied to Load Balancing", 13th Annual Int. Computer Software and Applications, IEEE Computer Soc. Press, 1989. pp760-766.

[Mittal89b] Mittal V.O., Singhal M, "An Expert System Based Load Monitoring and Scheduling System for Load Balancing in Distributed Systems", IEEE Int. Symposium on Intelligent Control, IEEE Computer Soc. Press, Aug.1989, pp215-220.

[Mullender93] Mullender, S., J., "Distributed Systems", 2nd Edition, Addison-Wesley, 1993.

[Nam92] Nam, S., Un, C., "Performance Analysis of Broadcast Star Network with Collision Avoidance Switch", The Int. Journal of Computer and Telecoms Networking, Vol. 25, No.2, August 92., pp169-182.

[Ni85] Ni, L. M., Xu, C-W., Gendreau, T.B., "A Distributed Drafting Algorithm for Load Balancing", IEEE Transactions in Software Engineering, Vol.SE-11, No.10, Oct.1985, pp1153-1161.

[OSF92] Open Software Foundation, "Introduction to DCE Rev 1.1", Prentice-Hall, 1992.

[OSF93] Open Software Foundation, "Design of the OSF/1 Operating System", Prentice-Hall, 1993.

[Ozden93] Ozden, B., Goldberg, A. J., Silberschatz, A., "Scalable and Non-Intrusive Load Sharing in Owner-Based Distributed Systems", Proc. 5th IEEE Symposium on Parallel and Distributed Processing, Dec 1-4, Texas, IEEE Computer Society Press, 1993, pp690-699.

[PC93] Personal communication.

[Philp90] Philp, I. R.., "Dynamic Load Balancing in Distributed Systems", IEEE Proceedings Southeastcon, part 1, April 90. pp 304-7.

[Pickard92] Pickard, R., H., "Getting Together By Numbers", .EXE Magazine, Vol 7, Issue 1, June 1992, pp14-18.

[Reed87] Reed, D. A., Fujimoto, R. M., "Multicomputer Networks: Message-Based Parallel Processing", The MIT Press, 1987.

[Ryou93] Ryou, J-C., Juang, J-Y., "An Efficient Load Balancing Algorithm in Distributed Computing Systems", Proc. 5th IEEE Symposium on Parallel and Distributed Processing, Dec 1-4, Texas, IEEE Computer Society Press, 1993, pp233-240.

[Shaw88] Shaw, A., Guest Editorial, Communications of the ACM, Vol.31, No.3, March 1988.

[Simpson94] Simpson, R., A., and Harget, A.J., "A Simulation Study to Determine the Importance of Load Balancing Algorithms for Loosely-Coupled Distributed Systems", to be presented at the 13th IFIP World Congress in Hamburg 1994.

[Srimani92] Srimani, P. K., Reddy, R. L., "Load Sharing in Soft Real-Time Distributed Systems", Int. Journal of Systems Science, Vol.23, Pt.7, July 1992, pp1115-1130.

[Singhal94] Singhal, M., Shivaratri, N., "Advanced Concepts in Operating Systems", McGraw-Hill, 1994.

[Stumm88] Stumm, M., "The Design and Implementation of a Decentralised Scheduling Facility for a Workstation Cluster", Proceedings of the 2nd Conference on Computer Workstations, March 1988, pp12-22.

[Tanenbaum92] Tanenbaum, A.S., "Modern Operating Systems", Prentice-Hall, 1992.

[Tanenbaum90] Tanenbaum, A.S. et al, "Experiences With The Amoeba Distributed Operating System", Communications of the ACM, Vol.33, No.12, Dec. 1990.

[Treleaven82] Treleaven, P. C., Brownbridge, D.R., et al "Data-Driven and Demand-Driven Architecture", Computing Surveys, 14(1), March 82, pp93-143.

[Tusch92] Tusch, J., "Performance Measurement in Token Ring Networks", The Int. Journal of Computer and Telecoms Networking, Vol. 25, No.2, August 92., pp159-168.

[Wah85] Wah, B. W., Juang, J. Y., "Resource Scheduling for Local Computer Systems with Multiaccess Networks", IEEE Trans. Computers, Vol. C-34, , Dec. 1985, pp1144-1157.

[Watson82] Watson, I., Gurd, J., "A Practical Data Flow Computer", IEEE Trans. Computers, Feb. 1982.

[Williams91] Williams, R. D., "Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations", Concurrency: Practice and Experience, Vol.3, No.5, Oct. 1991, pp457-481.

[Zhou88] Zhou, S., "A Trace-Driven Simulation Study of Dynamic Load Balancing", IEEE Transactions on Software Engineering, 14(9), Sept.1988, pp1327-1341.

BIBLIOGRAPHY

[Alonso88] Alonso, R., Cova, L. L., "Sharing Jobs Among Independently Owned Processors", Proc. 8th Int. Conf. DCS, IEEE, 1988, pp282-287.

[Anderson92] Anderson, T., Bershad, B., "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", IEEE Transactions on Computer Systems, Vol.10, No.1, Feb.1992, pp53-79.

[Athas88] Athas, W. C., Seitz, C. L., "Multicomputers: Message-Passing Concurrent Computers", Computer, Aug.88, IEEE Publications, pp9-24.

[Boglaev92] Boglaev, Y. P., "Exact Dynamic Load Balancing of MIMD Architectures with Linear Programming Algorithms", Parallel Computing, June 1992, Vol.18, No.6, pp615-623.

[Bokhari93] Bokhari, S. H., "A Network Flow Model for Load Balancing in Circuit-Switched Multicomputers", IEEE Transactions on Parallel and Distributed Systems, Vol.4, No.6, June 1993, pp649-657.

[Carling88] Carling, A., "Parallel Processing, Occam and the Transputer", Sigma Press, John Wiley & Sons Ltd, 1988.

[Casvant88] Casvant, T. L., Kuhl, J. G., "A taxonomy of Scheduling in General-Purpose Distributed Computing Systems", IEEE Transactions in Software Engineering, Vol.14, No.2, Feb.1988, pp141-154.

[Casvant88] Casvant, T. L., Kuhl, J. G., "Effects of Response and Stability on Scheduling in Distributed Systems", IEEE Transactions in Software Engineering, Vol.14, No.11, Nov.1988, pp1578-1588.

[Chow79] Chow, Y. -C., "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System", IEEE Transactions on Computers, Vol.c-28, No.5, May 1979, pp354-361.

[Coulouris89] Coulouris, G., and Dollimore, J., "Distributed Systems: Concepts and Design", 1989, Addison-Wesley.

[DeSouza91] De Souza e Silva, E., Gerla, M., "Queueing Network Models for Load Balancing in Distributed Systems", Journal of Parallel and Distributed Computing, Vol. 12, Part 1, May 1991, pp24-38.

[Douglis87] Douglis, F., Ousterhout, J.K., "Process Migration in the Sprite Operating System", Proc. 7th Int. Conf. DCS, IEEE, 1987, pp18-25.

[Eager89] Eager, D. L., Zahorjan, J., Lazowska, E. D., "Speedup Versus Efficiency in Parasllel Systems", IEEE Transactions on Computers, Vol.38, No.3, 1989. pp 408-423.

[Ferguson88] Ferguson, D., Yemini, Y., Nikolaou, C., "Microeconomic Algorithms for Load Balancing in Distributed Computer Systems", Proc. IEEE 8th Int. Conf. on Distributed Computing Systems, Computer Society Press, 1988, pp491-499.

[Gait90] Gait, J., "Scheduling and Process Migration in Partitioned Multiprocessors", Journal of Parallel and Distributed Computing, Vol.8, 9, 1990, pp274-279.

[Glazer93] Glazer, D., Tropper, C., "On Process Migration and Load Balancing in Time Warp", IEEE Transactions on Parallel and Distributed Systems, Vol.4, No.3, March 1993, pp318-327.

[Goswami93] Goswami, K., Devarakonda, M., "Prediction-Based Dynamic Load Sharing Heuristics", IEEE Transactions on Parallel and Distributed Systems", Vol,4, No.6,, June 1993, pp638-648.

[Huang93] Huang, J. -H., Kleinrock, L., "Performance Evaluation of Dynamic Sharing of Processors in Two-Stage Parallel Processing Systems", IEEE Transactions on Parallel and Distributed Systems", Vol,4, March 1993, No.3, pp306-317.

[Inmos87] Inmos Limited., "Transputer Development System 2.0: Programming Interface", Beta 2 documentation, April 1987.

[Inmos88] Inmos Limited., "Transputer Development System", Prentice Hall International, 1988.

[Kaashoek93] Kaashoek, F., Renesse, R. V., et al "FLIP: An Internetwork Protocol for Supporting Distributed Systems", IEEE Transactions on Computer Systems, Vol.11, No.1, Feb.93, pp73-106.

[Kleinrock93] Kleinrock, L., Korfhage, W., "Collecting Unused Processing Capacity: An Analysis of Transient Distributed Systems", IEEE Transactions on Parallel and Distributed Systems", Vol,4, No.5, May 1993, pp535-546.

[Kleinrock92] Kleinrock, L, Huany, J., "On Parallel Processing Systems: Amdahl's Law Generalized and Some Results on Optimal Design", IEEE Transactions on Software Engineering, Vol.18, No.5, 1992, pp434-447.

[Litzkow88] Litzkow, M. J., Livny, M., Mutka, M., "A Hunter of Idle Workstations", Proc. IEEE 8th Int. Conf. on Distributed Computing Systems, Computer Society Press, 1988, pp104-111.

[McCann93] McCann, C., Vaswani, R., et al "A Dynamic Processor Allocation Policy for Multi-Programmed Shared Memory Multiprocessors", IEEE Transactions on Computer Systems, Vol.11, No.2, May.1993, pp146-178.

[Mogul92] Mogul, J. C., "Network Locality at the Scale of Processes", IEEE Transactions on Computer Systems, Vol.10, No.2, May.1992, pp81-109.

[Pulidas88] Pulidas, S., Towsley, D., Stankovic, J. A., "Imbedding Gradient Estimators in Load Balancing Algorithms", Proc. IEEE 8th Int. Conf. on Distributed Computing Systems, Computer Society Press, 1988, pp482-490.

[Rommel91] Rommel, C. G., "The Probability of Load Balancing Success in a Homogeneous Network", IEEE Transactions on Software Engineering, Vol.17, No.9, Sept. 1991, pp922-933.

[Ryou89] Ryou, J-C., Wong, J. S., "A Task Migration Algorithm for Load Balancing in a Distributed System", Proc.22nd Annual Hawaii Int. Conf. Syst. Sci., Vol. II, Jan.89, pp641-1948.

[Magee89] Kagee, J., Kramer, J., "Constructing Distributed Systems in Conic", IEEE Trans. in Software Engineering, Vol.15, No.6, June 1989, pp663-725.

[Theimer88] Theimer, M. M., Lantz, K. A., "Finding Idle Machines in a Workstation-based Distributed System", Proc. IEEE 8th Int. Conf. on Distributed Computing Systems, Computer Society Press, 1988, pp112-122.

[Theimer89] Theimer, M. M., Lantz, K. A., "Finding Idle Machines in a Workstation-based Distributed System", IEEE Transactions in Software Engineering, Vol.15, No.11, Nov.1989, pp1444-1458.

[Shivaratri90] Shivaratri, N., G., Krueger, P., "Two Adaptive Location Policies for Global Scheduling Algorithms", Proc. IEEE 10th Int. Conf. on Distributed Computing Systems, Computer Society Press, 1990, pp502-509.

[Sih93] Sih, G., Lee, E., "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures", IEEE Transactions on Parallel and Distributed Systems", Vol.4, No.2, Feb. 1993, pp175-187.

[Smith88] Smith, J. M., "A Survey of Process Migration Mechanisms", ACM Operating Systems Review., Vol.22, No.3, July 1988, pp28-40.

[Efe89] Efe, K., Groselj, B., "Minimizing Control Overheads in Adaptive Load Sharing", Proc. 9th Int. Conf. DCS, IEEE, 1989, pp307-315.

[Barak85] Barak, A., Shiloh, A., "A Distributed Load Balancing Policy for a Multicomputer", Software Practice and Experience, Vol.15, No.9, Sept. 1985, pp901-913.

[Cabrera86] Cabrera, L. -F, "The Influence of Workload on Load Balancing Strategies", Proceedings of the 1986 Summer USENIX Conference, Atlanta, GA, USA, June 1986, pp446-458.

[Fox88] Fox, G., Johnson, M., Lyzenga, G., "Solving Problems On Concurrent Processors", Vol 1, Prentice-Hall, 1988.

[Johnson89b] Johnson, S.L., and Ho, C.T., "Optimum Broadcasting and Personalised Communication in Hypercubes", IEEE Transactions on Computers, 38(9), Sep. 1989, pp1249 - 1267.

[Joshi93] Joshi, B. S., Hosseini, S. H., Vairavan, K., "A Methodology for Evaluating Load Balancing Algorithms", Proc. 2nd Int. Symposium on High Performance Distributed Computing, July 20-23, Washington, IEEE Computer Society Press, 1993, pp216-223.

[Kremien92a] Kremien, O., Kramer, J., "Flexible Load Sharing in Configurable Distributed Systems", Proc. IEE Int. Workshop Configurable Distributed Syst., March 1992, pp224-236.

[Kremien92b] Kremien, O., Kramer, J., "Methodical Analysis of Adaptive Load Sharing Algorithms", IEEE Transactions on Parallel and Distributed Systems, Vol.3, No.6, Nov.1992, pp747-760.

[Krueger88] Krueger, P., Livny, M., "A Comparison of Preemptive and Non-Preemptive Load Distributing", Proc. IEEE 8th Int. Conf. on Distributed Computing Systems, Computer Society Press, 1988, pp123-130.

[Krueger87] Krueger, P., Livny, M., "The Diverse Objectives of Distributed Scheduling Policies", Proc. IEEE Int. Conf. DCS, IEEE, 1987, pp242-249.

[Kumar89] Kumar, A., "Adaptive Load Control of the Central Processor in a Distributed System with a Star Topology", IEEE Transactions on Computers, Vol.38, No.11, Nov. 1989, pp1502-1512.

[Kurose89] Kurose, J.F., and Simha R., "A Microeconomic Approach to Optimal Resource Allocation in Distributed Computer Systems", IEEE Transactions on Computers, 38(5), May. 1989, pp705 - 717.

[Lee91] Lee, C., "Load Balancing Using Semi-Private Memory", Proc. 10th Annual International Phoenix Conf. on Computers and Communications, March 1991, IEEE Comp.Soc.Press, pp167-173 .

[Li93] Li, J., Kameda, ., "Optimal Load Balancing in Tree Networks with Two-way Traffic", Computer Networks and ISDN Systems, Vol.25, No.12, pp1335-1348, Jul. 1993.

[Mullender90] Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R., van Staveren, J. M., "Amoeba - A Distributed Operating System for the 1990s", IEEE Computer, Vol.23, No.5, 1990.

[Rieken92] Rieken, B., Weiman, L., "Adventures in UNIX Network Applications Programming", John Wiley & Sons, 1992.

[Shen88] Shen, S., "Co-operative Distributed Dynamic Load Balancing", Acta Informatica, 25, 1988, pp 663 - 676.

[Shivaratri90] Shivaratri, N., G., Krueger, P., "Two Adaptive Location Policies for Global Scheduling Algorithms", Proc. IEEE 10th Int. Conf. on Distributed Computing Systems, Computer Society Press, pp502-509, 1990.

[Smith91] de Smith, N., "LAT Dynamic Rating: The Truth Behind The Rumors", VAX Professional, 1991.

[Suen92] Suen, T., Wong, J., "Efficient Task Migration Algorithm for Distributed Systems", IEEE Transactions on Parallel and Distributed Systems", Vol.13,, No.1, pp10-12, Feb. 1992.

[Svensson90] Svensson, A., "History, An Intelligent Load Sharing Filter", IEEE Proc. 10th Int. Conf. DCS, Computer Society Press, 1990, pp546-553.

[Woodside93] Woodside, Manforton, G., "Fast Allocation of Processes in Distributed and Parallel Systems", IEEE Transactions on Parallel and Distributed Systems, Vol,4, No.2, Feb. 1993, pp164-174.

[Yank91] Yank, C. -Q., Finkel, R., "Utility Servers in Charlotte", Software Practice and Experience, Vol.21, No.5, May 1991, pp429-441.