Abstract

Distributed systems offer the ability to execute a job at other nodes than the originating one. Load sharing algorithms use this ability to distribute work around the system in order to achieve greater efficiency. This is reflected in substantially reduced response times. In the majority of studies the systems on which load sharing has been evaluated have been homogeneous in nature. This thesis considers load sharing in heterogeneous systems, in which the heterogeneity is exhibited in the processing power of the constituent nodes.

Existing algorithms areevaluated and improved ones proposed most of the performance analysis is done through simulation. A model of diskless workstations communicating and transferring jobs by Remote Procedure Call is used. All assumptions about the overheads of inter-node communication are based upon measurements made on the university networks.

The comparison of algorithms identifies those characteristics that offer improved performance in heterogeneous systems. The level of system information required for transfer is investigated and an optimum found. Judicious use of the collected information via algorithm design is shown to account for much of the improvement. However detailed examination of algorithm behaviour compared with that of a 'optimum' load sharing scenario reveals that there are occasions when full use of all the information available is not beneficial. Investigations are carried out on the most promising algorithms to assess their adaptability, scalability and stability under a variety of differing conditions. The standard definitions of load balancing and load sharing are shown not to apply when considering heterogeneous systems.

To validate the assumptions in the simulation model a load sharing scenario was implemented on a network of Sun workstations at the University. While the scope of the implementation was somewhat limited by lack of resources, it does demonstrate the relative ease with which the algorithms can be implemented without alteration of the operating system code or modification at the kernel level.

Acknowledgements

During the years of my PhD programme I have had assistance and encouragement from many quarters and in a multitude of different forms. However it is without doubt that I single out my supervisor Dr Sati McKenzie as the individual to whom I am in the greatest debt. For her guidance, suggestions and powers of motivation I offer my warmest gratitude.

I would also like to thank my long suffering fiancée Fiona Newman for her understanding throughout the last few years. Combined with her material help in allowing me to use her car and laptop, both invaluable aids in the latter stages of my work. Peter Logie is another to whom I owe thanks, as he sacrificed his surfing so that I could use his modem.

At the University of Greenwich I have received technical help, advice and support from many sources. In particular Dr Chris Woollard my second supervisor who allowed me to access his knowledge of Distributed Systems. For technical help concerning the Universities computing resources I thank Ian Lee. Former research students who gave me the benefit of their knowledge and experience were Wasim Naqvi, Garrett Kearney and Peter Smith.

This work was conducted with the aid of a research studentship from the Engineering and Physical Sciences Research Council and help from the University of Greenwich. I would also like to mention my current employers Tertio Ltd, who have been very understanding in my first few months of employment.

Finally I'd like to thank the members of my family who have done so much to help me while I've been conducting my research. My parents for their constant support and encouragement. Plus my brother and sister who gave me the final impetus to finish.

Table Of Contents

List of Figures

List of Tables

1. Introduction

1.1 Distributed Systems

1.1.1 What is a Distributed System ?

The history of distributed systems (in this work the terms *distributed system* and *distributed computing system* are assumed to be analogous) began in the 1970's and was enabled by two parallel developments. The arrival of VLSI technology saw a move from the mainframe computer through mini and micro computing to the workstation/PC environment so common today. This change could not have occurred in isolation but was coupled with the improvement in communication technology that enabled the establishment of local and wide area networks (LANs & WANs). This combination proved an economic means of providing users with an independent computing resource at geographically distinct locations but still giving access to a wide range of facilities. Whilst not ending the reign of the mainframe, distributed systems have evolved to meet the changing demands of the user.

All distributed systems should display the same core characteristics of transparency, modularity, scalability, reliability and availability to varying degrees. This will be determined by the individual state of the system and design decisions taken to handle the tasks presented to it. Unfortunately these are about the only points on which a general definition is applicable. Such a definition under which all distributed systems could be clustered is given in [Cou94], "A distributed system consists of a collection of autonomous computers linked by a computer network and equipped with distributed system software". The distinction between different types of distributed system is made by Tanenbaum [Tan85] with the use of the terms, *distributed operating system* and *network operating system.* A distributed operating system is defined as one that performs like a conventional one but runs over multiple computers. Conversely a network operating system is constructed of computers all running their own independent operating system and co-operating together to utilise the resources available in the system.

A distributed operating system would need to employ extensive system software in order to function whereas a network operating system may only employ it in some areas of resource allocation. In this study the emphasis will be on network operating systems, although load sharing in distributed operating systems will also be discussed, as many of the ideas developed for use with the latter are still applicable to the more loosely coupled environment. The focus on network operating systems is because they are more readily available for research purposes, have established communication protocols and are becoming ever more popular. The term distributed system will be used to refer to all systems unless a distinction is deemed necessary. However a modified version of the quoted definition is proposed, "A distributed system consists of a collection of autonomous computing resources linked by a communications network and equipped with some distributed system software at least part of which operates transparently". The proviso of some element of transparency is needed as the benefits of load sharing can easily be negated if system users are involved with its operation.

1.1.2 Performance Improvements Via Load Sharing.

In a distributed system there is a high probability that at any point in time some of its constituent computing resources (nodes) will be highly utilised whilst others will be idle or lightly loaded, [Liv82, The89, Muk91]. By using the ability of distributed systems to execute jobs at other than their originating node, work can be transferred from one node to another in order to achieve an improvement in overall system performance. This approach can be referred to as *load sharing* or *load balancing*[Eag86a, Kru87, Zho87]. Load balancing has been used to refer to algorithms that attempt to equalise workload amongst the nodes, whilst load sharing algorithms attempt to ensure no node is idle. In this work the term load sharing has been adopted but it will be used in a broader sense, namely attempts to improve system performance by re-distributing some of the workload.

The granularity of the workload will influence its possible re-distribution from one node to another and any possibility for parallelism in the system [Kle85]. At the

The service times of jobs are exponentially distributed about an average of 1. Job interarrival times are also exponentially distributed, but are varied to give different system loads. Possible improvement is shown in Figure 1, as the area between the curves.

1.2 The Evolution of Load Sharing Algorithms - A Summary.

Load sharing has its origins in the task allocation algorithms of early distributed systems. These systems bore little resemblance to the workstation based ones of today. One example [Cho79] has all jobs arriving at one central dispatcher for allocation to the various nodes comprising the rest of the system. However a basic differentiation between the two classes of load sharing is made. The first class uses the simplest algorithms to implement, whose operation is based solely on past system performance. The second group is more sophisticated being based on the current state of the system These classes are respectively referred to by the terms static and dynamic. Occasionally dynamic algorithms are referred to as adaptive. With the increasing flexibility of distributed systems in the 1980's it became acknowledged that static algorithms would be of limited use [Tan85a, Wan85,Eag86a] as they could not react to changes in system state.

The task of load sharing became accepted as the re-distribution of work in a system, where work could arrive at any node. The initial placement of tasks from a central point has become a separate art, although the fields of interest will occasionally overlap. The division of dynamic algorithms into separate policies [Eag86a] can be seen as a milestone in their development, enabling the concentration of effort into investigating particular characteristics and more concise descriptions of results. Initially only a transfer and location policy were thought necessary. With time the use of three policies became commonly accepted, the transfer, location and information policies. Question addressed by these policies are shown below:

- Transfer policy - when should a job be considered eligible for transfer.
- Location policy - where should an eligible job be transferred.
- Information policy - when and how is information on the system state gathered.

Over the last decade a multitude of possible algorithms have been suggested and evaluated. Some of the relevant questions are:

- Source or server initiation : Whether an overloaded node should seek an under utilised one to which the job could be transferred or vice versa [Mir89a, Kru94].
- Load indices : Which is the best means for measuring the load at a node [Fer87, Kun91].
- Decision making : Should decisions be made in a distributed or centralised manner [Zho88, The89].

In general most of the algorithms suggested have been evaluated on homogeneous systems. Where heterogeneity is considered, it is often only in the workload offered to each node rather than the system composition. Until the 1990's systems combining heterogeneous but co-operative nodes were quite rare. This is reflected in the lack of work tackling this aspect of the load sharing. More recently heterogeneity has become of far greater concern with the rapid development in workstation technology leading to a proliferation of different types on the same communications network. Obviously load sharing, by simple job transfer, is not possible in cases of architectural or operating system heterogeneity. However by far the most common type is configurational where the technique is applicable. In some studies all aspects of diversity, CPU speed, I/O capabilities, memory are taken into account [Bak92, Ald93]. Others use just server rate or processing power and uses this solely to differentiate between nodes [Mah93, Wan94].

There may still be much debate about the details of implementing load sharing schemes, but there is general consensus about the properties required. An algorithm should be adaptable, scaleable, stable, fault tolerant and transparent to the system [Kre92], whilst still enhancing system performance. These are of course a set of ideal requirements and have yet to be met.

1.3 The Problem.

1.3.1 Unanswered Questions.

The history of load sharing algorithms, is almost as long as that of the distributed systems on which they are implemented. As the design, capabilities and expectations of the systems have evolved so have the techniques for optimal load sharing. The vast majority of algorithms are aimed at and adapted to systems of homogeneous nodes.

These algorithms when applied to heterogeneous systems exhibit several weaknesses leading to sub-optimal performance improvement. The algorithms specifically designed for a heterogeneous environment are still heavily influenced by the ideas pervasive in early work. An investigation is needed to establish if the assumption made in these established algorithms are all still applicable.

Heterogeneity in a system may be exhibited in a number of ways, configurational, architectural and operating system [Zho93]. With architectural and operating system heterogeneity the possibilities for load sharing are extremely limited if available at all. Differences in machine architecture will make the execution of the same code impossible and differences in operating systems may mean the same services, i.e. systems calls, are not available on all machines. Configurational heterogeneity offers more scope for load sharing as the machines involved will be fundamentally similar. They will differ in CPU speed, memory availability and other factors contributing to total processing power.

The introduction of standards in the 1980's has seen the interoperability of different machines increase. Of particular importance have been the attempts at establishing a portable operating system through the POSIX standards [IEEE90], which have been used by the X/OPEN organisation in the construction of their Common Application Environment (CAE). As a CAE becomes more globally accepted the portability it offers will increase the scope of configurational heterogeneity [Gra92]. Hence the increasing importance of heterogeneity while sharing computational resources with the use of load sharing algorithms. Table 1.1 shows the different UNIX based machines on one of the LAN's at the University of Greenwich. All the machines on this network originate from the same manufactuer, Sun Microsystems. Their processing power is indicated by results from the set of benchmarks used by the System Performance Evaluation Co-operative [SPE96] that measure multi-tasking throughput for integer code (SPEC*int*) and floating point code (SPEC*fp*). Ratings in each category are relative to the performance of a VAX11/780 , given a nominal rating of 24. The results shown are those achieved with the SPEC92 benchmark set. A new set of benchmarks SPEC95 is now in use by the organisation but results for all the machines on the LAN are not available for this newer group of tests. An anonymous quote sums up the usefulness of these figures, "While no benchmark can fully characterise overall system performance, the results of a variety of realistic benchmarks can give valuable insight into expected real performance".

MODEL	Processor Elements	Occurances on LAN	Clock Speed MHz	SPEC *int*	SPEC *fp*
SS/IPC	1	3	25	327	263
SS/ELC	1	1	33	432	425
SS2	1	1	40	517	541
SS/IPX	1	7	40	517	510
SS10/41	1	1	40	1264	1607
SS10/402	2	2	40	2112	2378
Classic, LX	1	10	50	626	498
SS10/51	1	3	50	1546	1969
SS20/514	4	1	50	7072	7341
SS5/70	1	10	70	1352	1122
SS4/70	1	5	70	1414	1110
SS20/71	1	5	75	2984	2875
SS20/HS14	4	1	100	8124	8906
SS4/110	1	2	110	1864	1549
Ultra1/140	1	2	140	5107	7175

Table 1.1 Configurational Heterogeneity in a Distributed System.

Previous studies have used many different means of assessing proposed algorithms, examples of which are: queuing network analysis, simulation and implementation. Of these simulation is the most flexible but may still leave doubts about the practicality and validity of any assumptions. Some factors are impractical to simulate on a large scale, one in particular being the underlying effect of any traffic generated by the implementing of the load sharing algorithm itself. Implementation can provide the answer to such questions but can be hampered through a lack of resources available for the project. Not many researchers are fortunate enough to have a network to themselves.

1.3.2 Aims

The aim of this work is to find answers to some of the questions raised in the previous section. This is accomplished as follows:

- Existing load sharing algorithms are investigated by simulation modelling. The simulation model is made as realistic as possible. Model assumptions such as communication overheads are based on experimental measurements.

- Based on the above studies, new algorithms are proposed which are effective in a heterogeneous environment. These are evaluated by simulation.

- An implementation of the simulated system is carried out. This will aid in validating the model and facilitate examination of factors which cannot be readily simulated, such as algorithm overhead and the effect of the extra communication traffic generated. The building of a working implementation will also ensure that any algorithms proposed are inherently practical.

1.4 Contribution of the Thesis

- An investigation of current load sharing algorithms when applied to heterogeneous systems. Heterogeneity is exhibited in the relative processing power of the nodes. This has led to the identification of characteristics that were responsible for the sub-optimal performance of the algorithms. The investigation was carried out with the use of a simulation model, which was constructed using communication overheads based upon measurements made over the university's local area networks.

- New algorithms are proposed which are better suited to a heterogeneous environment. The performance of the algorithms is evaluated using the simulation model. All algorithms take into account the restrictions imposed by the normal operating conditions of an existing distributed system.

- Validation of the simulation is accomplished through building an implementation of the simulation model on the university networks. The implementation is also used to test the underlying behaviour of the communication network and overheads of the algorithms that it is not feasible to simulate.

1.5 Layout of the Thesis

Chapter 1, Introduction:

Presents a background to the work covered in the thesis, indicating the problem that is to be tackled and possible solutions A general statement of the contribution of this thesis is given.

Chapter 2. Survey of related research:

The current research in the load sharing field can be divided into three principal sections. First the algorithms that control the manner in which load sharing is performed. Secondly the type of system on which the algorithms are implemented and investigated. Finally, the means by which the algorithms are evaluated.

Chapter 3. Scope of the present work:

Describes the approach to load sharing adopted in this work. The main emphasis is on heterogeneous systems and the way in which heterogeneity will influence algorithm design. The algorithms investigated are described in full as are the various system models used. Both simulation and measurement are presented as means of evaluating the algorithms.

Chapter 4. Discrete event simulation:

The chief method of investigating the load sharing algorithms presented is through a simulation model. The translation of a real system into a practical simulation model is described, with particular emphasis on the design decisions taken. Full implementation details are also presented, based on the object oriented simulation facilities offered by the MODSIM language used.

Chapter 5 Implementing the load sharing scenario:

The load sharing scenario was constructed as a means of validating both the assumptions made in developing the simulation model and the results it provided. The system was implemented on a network of workstations. Both network and system programming had to be used and the routines used are described in full. Particular attention is given to problems raised by the physical environment as opposed to the simulation model.

Chapter 6. Experimental Results:

The performance of the load sharing algorithms described in Chapter 3 over a variety of heterogeneous systems is evaluated using the simulation model. The charcteristics of each are described and analyzed. Those algorithms that are most suited to the heterogeneous environment are subjected to further investigation to discover their properties in the areas of adaptability, scalability and stability. Validation of the simulation assumptions and its subsequent results is performed via the implementation scenario.

Chapter 7. Final Remarks:

This chapter presents a summary of the experimental results and the conclusions that can be drawn from them. The conclusions cover both a comparison of algorithms for heterogeneous distributed systems and the validation of these algorithms. Ideas for furthering the work reported conclude the chapter. They have been suggested during the course of the research or prompted by recent technlogical devlopments.

2. Survey of Related Research

2.1 Qualitative Analysis - The Taxonomical Approach

The system of classification proposed in Casavant's taxonomy [Cas88], is, as the title suggests aimed at a broad range of distributed systems. Of the scheduling tasks considered load sharing is only one of many. The taxonomy must therefore be refined in order to describe succinctly the area in question. Most of the classification groups are still applicable and are used in the scheme shown in Figure 2.

Load sharing algorithms can be static or dynamic in operation. The static variety employs historical system performance data whereas dynamic algorithms can use information on the current system state in decision making. A distributed algorithm is implemented at every node in the system. A centralised one is only fully implemented on one node. The centralisation can encompass the full decision making process or just the gathering of information on system state. Co-operation implies that system state information is exchanged between the nodes. An optimal algorithm attempts to use all available information in its decision making. However as this is often impossible or computationally difficult the sub-optimal class covers those algorithms using only enough information to give an acceptable degree of performance improvement.

Static algorithms, as their name implies do not change whilst the system is running. All load sharing decisions are made using *a priori* information based upon relevant system data, examples of which are: average loading statistics, node processing power and network communication speed. Therefore they cannot be centralised as this would imply that nodes were exchanging information with a central node which would make decisions based on the information gathered there. The most rudimentary static algorithm is the allocation of machines to staff in any organisation. The most powerful machines would be allocated to those persons with greatest computing demands indicated by previous workload statistics. Unfortunately powerful machines can still lie under-utilised all summer on professorial tables and so a more sophisticated solution is called for. Random splitting algorithms [Ni81] distribute jobs according to a given probability distribution. A variation on this is the cyclic splitting [Yum81] algorithm that distributes jobs on a cyclic schedule in an attempt to avoid temporary congestion. An alternative example, "the optimal static load balancing algorithm" was proposed by Tantawi & Towsley [Tan85] and simplified by Kim & Kameda [Kim92a]. OR techniques are used to calculate an optimum load for each node dependent upon processing power and communication rates in the system.

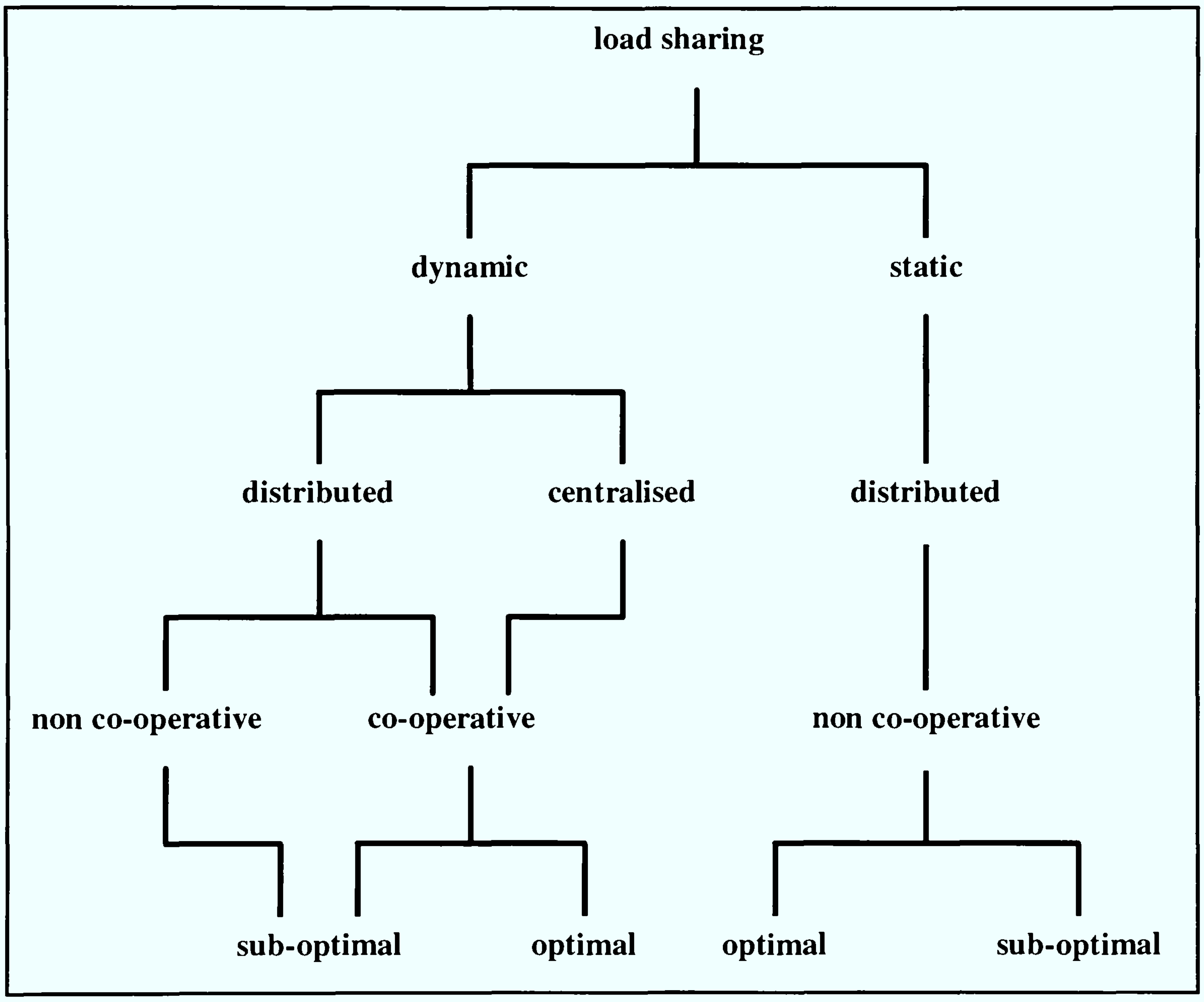

Figure 2.1 A Taxonomy of Load Sharing Algorithms.

Although these algorithms have achieved improvements over the no load sharing case, they are limited in their effectiveness as they cannot react to changes in the system state, in particular short term fluctuations in system load. Nor do they exhibit any scalability in respect of system size or constitution. For these reasons work over the last decade has been concentrated in the field of dynamic algorithms.

The first branch in the taxonomy of dynamic algorithms separates them into distributed and centralised classes. Two centralised algorithms were proposed and evaluated by Zhou [Zho87]. CENTRAL had both centralised information gathering and decision making. GLOBAL centralised the information and periodically broadcast it to all nodes allowing them to make a decision as to any transfer of jobs. Of the two centralised algorithms, CENTRAL was considered the best, although its performance was not dramatically better than that of comparable distributed algorithms. A comparison of CENTRAL and an equivalent distributed algorithm [The89] indicated that the simplicity of implementation of the latter can be an advantage. Other work has highlighted further potential weaknesses of centralised algorithms, notably bottlenecks forming at the central node and the vulnerability of the systems load sharing capabilities if this node fails [Ald92, Ber93]. These factors have lead to the conclusion that centralised solutions are better suited to multi-processor configurations, rather than distributed systems.

The suitability of distributed dynamic algorithms to the load sharing problem is reflected in the large body of work in this field. These algorithms and the techniques used for their evaluation will be described in the rest of this chapter. The three policies and question of initiation raised in section 1.3 will provide a discussion framework

2.1.1 Initiation

The concept of load sharing can be viewed from two opposite directions. The first is from the perspective of an over-loaded node, which will seek to send some of its work for processing elsewhere. The second is that of an under-loaded or idle node, which can advertise its services or actively seek more work. Therefore an algorithm can be initiated at the sender, receiver or both. The terms source and server initiated are sometimes used to represent the same concepts. Initiation will occur on change of state, i.e. a job arrives or finishes.

The assumption here is that all the nodes involved operate in a multiprogrammed mode, which is *de rigueur* in modern workstations. What will limit the initiation options are the job migration facilities available. Job migration is the ability to stop an executing job and move its whole context to enable continuing execution at another site. This is by no means a trivial task [Art89], but is an essential requirement for receiver initiated schemes. These are invoked when the completion of a job puts a node in a state that it is ready to receive more work from a heavily loaded one. It is highly unlikely this event will correspond with the arrival of a job at another node, hence only jobs that have already begun execution will be candidates for transfer.

In several distributed operating systems the ability to migrate processes is available [Bis95]. For the network operating system environment with which we are concerned, the Condor system [Epe95, Tan95] does offer migration facilities outside the kernel. Unfortunately this system has limitations and cannot deal with all types of process, in particular communicating processes. The lack of ability to deal with a job that spawns new processes places severe restrictions on any form of receiver initiated load sharing algorithm. Sender initiation, prompted by the arrival of a new job, relies on initial job placement occurring before the start of execution. This type of operation can be supported by any distributed system worthy of the name.

Studies have been performed to compare sender and receiver initiated policies. Simulation and network analysis techniques are used, where the effect of job migration can conveniently be represented by a time delay. The results are inconclusive with some [Eag88, Dan95] preferring sender initiated algorithms. Others [Kru88, Mir89a] conclude that receiver initiated algorithms perform best at high system loads, with the reservation that their performance is highly dependent upon the costs of migration. The RESERVATION algorithm [Eag86b] is receiver initiated but does not involve job migration as lightly loaded nodes reserve the next job arriving at a heavily loaded one. This approach was not successful with the algorithm being out performed by simple sender initiated ones. Intuitively one would expect receiver initiated algorithms to perform best at high loads as the chance of finding a heavily loaded node is high. A combination of initiation policies is used in the "Symmetrically Initiated" algorithm [Kru94], where lightly loaded nodes use receiver initiation and heavily loaded ones sender initiation.

With the difficulty in implementing full process migration and lack of evidence that receiver initiated algorithms offer a significant performance improvement, analysis will focus on sender initiated solutions.

2.1.2 Transfer Policy

In order to describe and facilitate the comparison of load sharing algorithms, they are separated into component parts or policies. The use of policies was introduced by

Eager et al [Eag86a], who used two: transfer and location. The trend now is to use three: transfer, information and location [Zho88, Gha90, Bak92, Ber93, Mah93, Kru94, Ben95].

In many ways transfer policy can be thought of as the first stage of an algorithm. It is the transfer policy which decides whether a job should be executed locally or made available to be transferred to another node for execution. The type of transfer policy varies in the literature, but the most widely used is the Threshold, based upon local queue length. As a new job arrives at a node, the CPU queue length at that node is examined. If accepting the new job for processing would cause the set threshold to be exceeded then the job is eligible for transfer. Eligibility for transfer does not imply that the job must be transferred only that the other policies of the algorithm will be invoked. The problem with use of a fixed threshold is that the optimum value changes with system load [Eag86a]. As the system load increases, chances of finding a lightly loaded machine decrease and therefore a higher threshold would be more appropriate. However this is not necessarily the case in heterogeneous systems which Eager did not investigate.

As an alternative to a fixed threshold a dynamic one was suggested in [Gha90]. The load at neighbouring nodes is used in calculating the transfer threshold when load sharing is initiated. Another alternative is a form of global threshold [Sta84], where each node asseses the loading across the system by exchanging information with its neighbours. If system loading is below or above predefined levels then no attempt is made to transfer any jobs. In both cases the communications network envisaged was based upon point to point links. This type of fixed structure allowed neighbours to be clearly defined and limited broadcasts to a small subset of the network involved. In the fully connected LAN's prevalent today broadcasting load statistics can be performed simply but each node in the system will incur overhead on receipt of the data. Even in systems where multicasting is considered [WIL95]the thresholds used in transfer policy have been fixed. Therefore in this work the use of a dynamic threshold is considered to be impractical. It would require each node to possess the ability to estimate overall system load in the short term at an economic cost.

The performance of a good transfer policy is dependent on a reliable measure of workload at a node. An accurate estimate would be obtained if the service time of each job at a node were known. Unless the work on a system was of a repetitive batch variety this is not possible. The load index should be simple, instantaneously available and enable comparison between nodes. Several possible indices have been investigated [Fer87, Kun91] that are generally available on UNIX based machines:

- Ready to run queue length
- 60 second load average
- CPU utilisation, 10 seconds and 60 seconds average
- 5 seconds system call rate
- CPU context switch rate
- Available memory

Of these, the ready to run queue length consistently outperformed the rest. No improvement was achieved by using an index that combined any two of these indices [Kun91] even when the best two were used. Although these results are for homogeneous systems they can be extrapolated to heterogeneous systems, when attention is paid to relative processing powers.

Stability is an important property of any load sharing algorithm [Sta85], and it can be adversely effected if processor thrashing is allowed. This phenomenon occurs at high system utilisation, when jobs are continually transferred and never executed. A simple cure is to put a limit on the number of transfers a job can experience. This has become known as the transfer limit [Eag86a].

One further procedure can be included in the transfer policy of an algorithm, that is to filter out jobs ineligible for transfer. This is normally done on the grounds that jobs of short duration should not be transferred. A simple enough task when using a simulation model [Zhou88]. Without the ability to assess the service time of a job this is impossible to accomplish in a transparent manner and so in the majority of studies it has been ignored.

2.1.3 Information Policy

Eager's definition of location policy, the policy which decides where a job eligible for transfer should be transferred to, included the means of acquiring the information on which to base the decision. Now the norm is to divide this into location and information policies, the latter concerning the acquisition of information upon which to base decisions.

Two strategies are possible, broadcast and probing. Broadcast can be periodic[Sta84, Ald92], with each node broadcasting its load to all the other nodes in the system at regular intervals,. Alternatively it can be event driven, by a node state change. The state change could be the arrival of a job eligible for transfer, upon which the source node will broadcast a request for state information from other nodes in the system.[Cas87]. Or any change in loading at any node may be broadcast [Sta84].

The most obvious problem with any broadcast based policy is the large amount of communication traffic that will be generated. A periodic broadcast will create extra traffic with no guarantee that the information is needed, but increasing the time interval between broadcasts may lead to inaccurate placement decisions based upon out of date information. Source initiated broadcasting although furnishing more accurate state information will lead to periods of intense activity on the communication network as all nodes try to respond concurrently. The advantage of using broadcast techniques are that an image of the whole system can be formed and idle nodes located, assuming that the state information used is still accurate. How great an advantage this is depends on the demands of stability. If distinct nodes make decisions based on the same information they will all come to the same conclusion. Underloaded nodes can become swamped with jobs transferred from many different overloaded ones, leading to performance degeneration.

Probing or polling, is event driven and so all information gathered will be as current as possible. A communication delay will be unavoidable but will be tiny in comparison to job service time and so it is unlikely that state information will be obsolete. It is normal for only a small subset of the available nodes to be probed, referred to as the probe limit. These are picked at random by the instigating node. Whether all the nodes up to the probe limit are probed is at the discretion of the location policy. Research into systems of homogeneous nodes has shown that probing 10% - 15% of the total system provides optimum results [Phi90, Ben94], even if communication costs are assumed to be negligible [Eag86a]. In reality these costs cannot be ignored, and the relatively small number of probes has the advantage of much lower communication overhead than broadcast.

General comparisons of these two means of information dissemination have been made. Probing has been shown to be the most efficient at low to moderate system loads and broadcast at high loads [Mah93].

2.1.4 Location Policy

The final task for a load sharing algorithm is to use available state information in deciding the destination of an eligible job The possibility of a node rejecting a transferred job is not discussed as the mechanism to allow this type of negotiation would add considerable overhead, which is better invested in making the best possible initial placement.

The simplest location policy is one which uses no state information at all, randomly selecting another node to accept the job, such as RANDOM [Eag86a, Zho88]. Although very simple, this form of "blind" [Ber93] location policy can exhibit substantial performance improvement over the no load sharing case at all levels of system load when implemented on homogeneous systems. Performance on heterogeneous systems is discussed in later chapters of this thesis.

A strategy common in early work is to identify the lowest loaded node and move jobs there from an overloaded one. [Sta84]. In a homogeneous system this can easily be identified as the node with shortest queue length. This is simple enough to determine if a global picture of the system is available, as with a broadcast information policy. However if probing is used a measure is needed to determine if a particular node is suitable. As selecting the lightest loaded is impossible unless all nodes are probed. Two methods are available, incorporated as the location policies of the THRESHOLD and SHORTEST algorithms [Eag86a, Phi90].

The first as its name implies is based upon a threshold, often of the same value as that used in the transfer policy. For example a threshold of 2 may be used, so that a node will only consider a job eligible for transfer if its own load is greater than 2 and will consider another node a possible recipient if it has a load of less than 2, in the knowledge that transfer will not degrade the response time of the job in question. The number of nodes probed is limited by a set probing limit. On detecting a suitable node transfer will occur immediately. If the probe limit is reached before a suitable candidate is discovered then the job in question is executed locally.

The second strategy also uses a threshold but rather than transferring to the first suitable node discovered attempts to find the node with the shortest run queue. So even if a suitable node is discovered, probing continues up to the probe limit in search of a more lightly loaded destination.

In either of the two location policies if an idle node is probed then the job can be immediately transferred, as no more suitable node could possibly be found. Of the two policies SHORTEST has been shown to have a slight edge in performance. Figure 2.2 shows how this algorithm works.

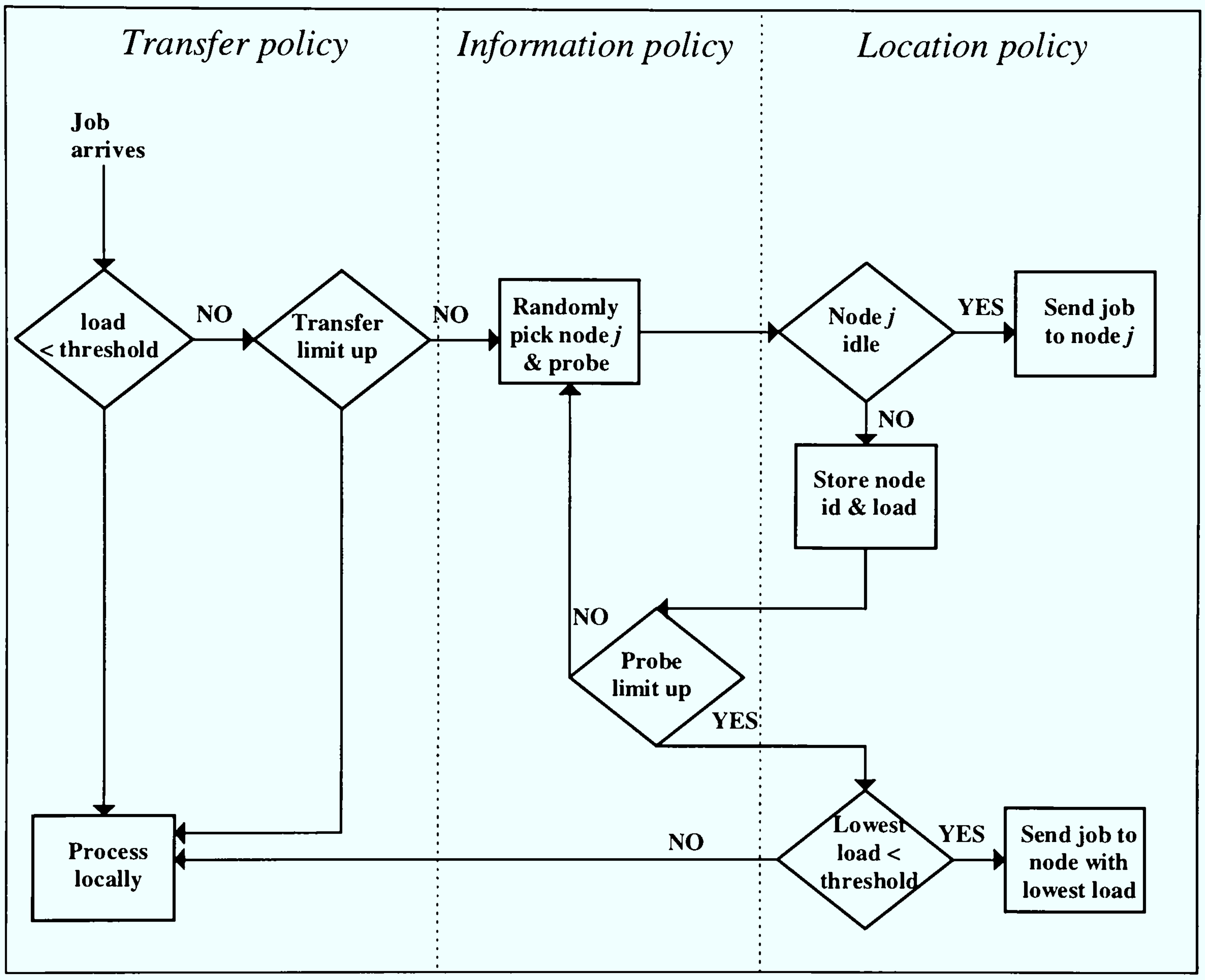

Figure 2.2 The SHORTEST algorithm in three policies.

As an alternative to a fixed threshold a bias can be employed. A suitable node will be one whose load is less than the overloaded one by the set bias [Sta84, Cas87]. The size of the bias may reflect the cost of job transfer, a large bias reflecting a high transfer cost [Rom91].

In a heterogeneous environment the use of queue length alone can still be effective [Bau89] but the majority of current work has attempted to show sensitivity to the differing service rates at nodes. To accomplish this some form of rating must be assigned to each node. If a mixture of CPU queue length, memory capabilities and I/O speed is used [Ald92, Zho93, Shi94] then prior knowledge of job requirements is needed in order to assess the relative merits of each factor. To avoid this requirement an overall measure of server rate or processing speed can be used [Mir89b, Bak92, Wan94] with which a number of different location policies have been proposed. All of these will in some way attempt to account for the inequality in processing speed by making job transfer easier from slow nodes to more powerful ones.

Mirchandey [Mir89b] uses a set of pre-determined thresholds. A node will only respond positively to a probe from an overloaded machine if its local load is currently less than its own threshold. Fast nodes will have high thresholds and slow nodes low ones. These are the same thresholds used in the transfer policy. A similar scheme is used by Baker [Bak92] although there is more differentiation between nodes. Set thresholds are used but the load value returned by a probed node is its local queue length divided by its threshold. If the product is less than unity transfer can take place. The advantage is that comparison of prospective destinations is allowed. While exhibiting some sensitivity to system heterogeneity, there can be problems due to a lack of load sharing between group of fast nodes all of the same power as they all have high transfer thresholds. Also there is little adaptability in these policies. If new nodes are introduced to the system, the ratio of thresholds may need altering which cannot be done dynamically.

A more flexible method is to use the ratio of relative processing powers. Wang [Wan94] suggests that a powerful node will accept work if its local load is less than a threshold based on proportional processing power of the two communicating nodes. No mechanism is provided to compare two nodes both capable of accepting a job.

Wang's algorithm does not allow the transfer of jobs from fast to slow nodes. While this avoids the problem of selecting idle but slow nodes it may lead to missed opportunities for load sharing. Zhou [Zho93] uses various load indices in the location decision one of which is ready to run queue length. The load at a remote node is scaled according to its relative CPU speed (cycles per second) rather than processing power (MIPs). But before scaling the remote load is incremented to account for the effect of the job if it was transferred. This has the effect of stopping inefficient transfers to idle but slow nodes. Fixed thresholds are still used for comparison purposes once the remote load has been scaled, combined with use of the other indices.

2.2 System Model

Once a load sharing algorithm has been developed, it can be evaluated by studying its performance on a given system model. The system model used will naturally have a great influence on perceived performance. In cases where algorithms have been studied through implementation, this is normally used in conjunction with, and as an aid to constructing a valid model. Unfortunately no standard model is available and those used in previous studies have varied enormously. The differences fall into the following categories:

- Network topology
- Heterogeneity of nodes
- System load
- Overheads

2.2.1 Network Topology

All distributed systems will use a communications network through which to function. The size of the network can vary from a localised environment to national or international proportions. This study will concentrate on the former and the related Local Area Networks (LAN's). Load sharing is possible over a much larger scale [Epe95] but only in a limited form, as the lengthy communications delay inherent in WAN's will add a significant overhead.

Algorithms have been evaluated on networks that were not fully connected [Sta84, Cas87], and this was reflected in their design. The LAN's in general use today have bus and ring topologies. These can all be considered as fully connected in that the average communication time between any pair of nodes will be the same. Due to this fact the design and evaluation of load sharing algorithms is not normally effected by the lower level (MAC) operation of the LAN in use. A rare exception [Kim92b] was developed specifically for a network using the CSMA/CD protocol. With this is mind any system model used will only need to consider differences in communication speeds. As mentioned in **1.2.2**, the effect of the extra traffic due to load sharing algorithms, on data transfer rates, can only be investigated through implementation and measurement..

The issue of inter-net load sharing between LAN's was addressed in [Ban89], assuming that inter-net communications has a considerably higher cost than intra-net communications. It concludes that no advantage is to be gained by inter-net load sharing. Another factor against inter-net sharing is the use of common data stored on file-servers within individual LAN's Transferring the job to another LAN would incur considerable extra cost.

2.2.2 Heterogeneity of nodes

As noted in **2.1.4** configurational heterogeneity can be exhibited in many ways. If all of these factors are implemented in the model it becomes very complex and limits soon arise to its scalability. A more practical method of expressing heterogeneity in a node is to use just one parameter, processing speed. Although jobs may have a variety of requirements in terms of CPU usage, memory and disk I/O, these cannot easily be estimated at run-time. It is a reasonable assumption that in general relative CPU speeds and memory capability of workstations will be comparable. It is unlikely that a manufacturer will supply a fast CPU with slow or insufficient memory. With regard to disk I/O, the diskless workstation is becoming more popular in networked systems due to ease of management of a central file server.

The model should be flexible enough to allow the evaluation of any algorithm over systems with differing configurational heterogeneity. If systems can exhibit different levels of heterogeneity the question arises as to what metric to use in characterising it. This question is not often tackled, but a simple ratio of processing power has been suggested [Mah93]. This approach cannot cover all cases, for instance when relative processing power is unchanged, but proportions of nodes with different speeds is, or when more than two types of nodes are concerned. A more sophisticated measure using skewness and variance of distribution of processing power can be devised. This is based on recent work by Sarraf [Sar95] in which a means of describing offered workload on a LAN is presented.

2.2.3 System Load

In order to assess the scalability of any algorithm it must be evaluated on a system with variable system load (overall utilisation). The question of interest is how the system load should be spread amongst the individual nodes and how it will be represented in service and interarrival time distributions.

When examining performance on a homogeneous system the load at each node can be the same. This is a reasonable assumption and is the scheme used in many system models proposed in the literature [Eag86a, Zho88]. Early work tended to consider only homogeneous loading as it was felt adequate to test the basic characteristics of an algorithm. However in order to meet rudimentary adaptability requirements an algorithm should be able to cope with some degree of heterogeneity in loading at the nodes [Kru94, Kar95].

When considering heterogeneous systems there are three possible loading representations. The homogeneous case, where each node experiences the same offered load, holds less water, although it is still used [Mah93]. Another possibility is that of proportional loading [Mir89b]. The offered load at a node is proportional to the processing power of that node. This is the natural extension of the loading patterns used in most studies of homogeneous systems. Lastly the heterogeneous situation where the offered load at a node bears no relation to its processing power is a possible scenario but as yet has not been explored in any depth.

Another characteristic of the load originating at each node is the distribution of interarrival times and job service times. In the majority of cases, where a workload must be created the use of an exponential distribution has sufficed for the interarrival time. A trace driven workload is used by Zhou [Zho88] in an attempt to reproduce true system conditions. This idea has not been followed in any later work as it is considered too restrictive, being based on the characteristics of just one machine. Use of a hyper-exponential interarrival time distributions has been investigated by Dandamudi [Dan95]. The algorithms investigated showed little relative sensitivity to the increase in job arrival clustering although response times did increase, not an unexpected result.

With regard to the distribution of job service times, there has been a little variation in the literature. Kruger and Livny [Kru87, Kru88] expound the virtues of a hyper-exponential distribution in accurately representing true service rates. But in a later paper [Kru94] returned to use of the exponential distribution. The hyper exponential case has also been explored more recently [Ben93, Dan95] in both instances it was reported that the relative performance of the algorithms studied was unaffected in comparison to the situation when using an exponential service time distribution. The bulk of system models use an exponential distribution.

2.2.4 Overheads

No dynamic algorithm can operate without imposing an extra overhead on the system, as state information must be collected and used in the chosen algorithm. There is also the cost of transferring a job, in whatever context, to consider. The only algorithms that are assessed with no regard for overhead are those aiming to give a lower bound on performance, with which to correlate other results. Examples of these are LB2 [Sta84] and NoCost [Zhou88].

Job transfer cost will depend upon the file service implemented. In a networked UNIX based workstation (often diskless) environment it is common for files to be stored remotely on a dedicated file server. Therefore on transferring a job only a command line need be passed between nodes, which can be represented by a fixed cost[Bak92, Kru94, Dan95]. If files are stored locally then the cost of transferring a job will be increased as these files will consequently be accessed remotely rather than locally. This extra cost is normally represented as a percentage of job service time [Eag86a, Mir89b, Phi90]. When this is the case and transfer costs can be very high the cost of information dissemination is considered negligible and ignored. Otherwise a fixed cost will be allocated to each probe or broadcast, depending on the information policy used.

All the costs associated with extra communication due to algorithm operation are modelled as delay at the CPU. In more sophisticated system models the costs to both sending and receiving nodes are taken into account, whereas earlier ones assumed all the overhead was borne at the sender. As dynamic load sharing algorithms are very simple in operation, the CPU cycles used by the algorithm for non-communication related activities are ignored in all but a very few cases.

2.3 Algorithm Evaluation

The first two sections of this chapter have described different types of load sharing algorithm and the system models on which they can be evaluated. There remains the question of.which techniques can be used to perform the evaluation and what metric should be used to judge performance.

2.3.1 Evaluation Techniques

The three standard techniques [Kan92] for studying system performance have all been applied in the evaluation of load sharing algorithms: analytical modelling, simulation and measurement. Analytical modelling in the form of queuing network analysis has been used in the past but always in conjunction with simulation, in that results have been checked against those achieved by simulation. The advantages offered are simplicity and speed. These were particularly useful when the processing power available for simulation purposes was at a premium. Generally the mathematical approach has been used in evaluating general algorithm performance on simple system models [Eag86a, Mir89a], or where the load sharing algorithm is based upon the underlying network protocol and so is too complicated to simulate [Kim92b]. Approximations will always be made in an analytical model to ensure it remains tractable and this can lead to unreliable results in some situations. One common assumption made is that each node is independent of others, a method of decompostion that is asympotically exact as the number of nodes tends to infinity. In general a system of less than fifteen nodes is considered too small. A comparison [Eag86b] of simulation and analytical results showed discrepancies at high system loads.

With the understanding gained of the general behaviour of algorithms over homogeneous systems, more complex models were introduced to represent the distributed systems involved more accurately. Factors previously considered negligible were now included, in particular the overhead associated with inter node communication. These considerations along with the introduction of heterogeneity, in both offered load and processing power, made analytical models ever more intractable.

It is arguable that the growth in system model complexity was prompted by the rapid increase in computing power available to researchers. This in turn led to the increased use of simulation as an evaluation method. Whatever the motivation simulation has become the most popular technique for the evaluation of load sharing algorithms. Unfortunately there are still practical limits to system size and complexity. The simulation of systems of over 20 nodes is rare. Zhou had a system of a maximum 49 nodes but only conducted short runs using systems of this size [Zho88]. Ghafor studied a

35 node system but it was not fully connected [Gha90]. Aldy [Ald92] considers many different parameters in algorithm operation and system model but restricts his studies to a network of 3 nodes.

Measurement is thought of as the most fundamental technique in performance evaluation. It is needed to some extent for both analytical modelling and simulation, as a means of establishing initial parameters such as communication overheads. For this purpose a full scale implementation is not needed as the required details may be obtained from an existing system. Measurement of algorithm performance will need a full implementation. The greatest problem here is the availability of resources and so implementation is often on a small scale, 3 and 11 nodes [Bau89], 6 nodes [Zho87].

2.3.2 Performance Metrics

To arrive at the best metric of performance, the purpose of the system must be examined. Should it deal with a large number of real time jobs then meeting deadlines will be of utmost importance. The primary goal of a load sharing algorithm in such an environment would be to minimise the rate of job loss due to deadline expiry [Sri92, Hou94].

A typical network operating system with different workstations will normally handle a wide variety of jobs but their completion time is not ultimately crucial. For systems without such restrictions Kleinrock [Kle76] suggests, "The average response time for a job requiring *X* seconds of processing is the single most important performance measure". The response time of a job is the time from when it enters the system for processing to when it leaves the system with all its associated tasks completed.This is the metric adopted in all previous load sharing studies not involving real time jobs.

Other metrics have been suggested, Kruger and Livny [Kru87] proposed a measure of fairness, Wait-Ratio. Which is the waiting time of a job relative to its service demands. The aim in a "fair" system was that all jobs should experience the same wait ratio. While this metric may be of some value in sequential FCFS systems it is less applicable in the multiprogramming systems that have become the norm [Tan87] and has not been adopted in later work.

3. Scope of the Present Work

3.1 Introduction

For the purpose of algorithm evaluation a system model is required. The structure of the model and rationale behind its construction are described in this chapter. Particular attention is paid to establishing differing levels of heterogeneity in the model in order to provide a wide variety of operating conditions. A number of loading conditions are possible with the model, varying both in overall system utilisation and loading patterns across the system. Construction of any accurate model of a distributed system is not possible without knowledge of the overhead involved in the operation of the system. An investigation into the costs of Remote Procedure Calls (RPC's) is presented. These costs are used as the basis of system overhead as RPC's are used for performing many of the functions underlying load sharing activities.

One of the aims of this work is to investigate the effects of heterogeneity on the performance of load sharing algorithms. But as the survey in Chapter 2 has shown there is a large choice of algorithms. Even if the area of study is restricted to dynamic distributed algorithms, it is not practical or desirable to evaluate them all. So criteria have to be established, to select suitable algorithms or individual policies. The primary rule that will be used is that implementation of the algorithms should be possible on a standard network of workstations. This will exclude the use of pre-emptive strategies that involve process migration. A process in this sense is a job which has begun execution. Concentrating on just non pre-emptive sender initiated algorithms is not felt to be unduly restructive. They are the same type used by Eager [Eag86a] and Zhou [Zho88] in their work on homogeneous systems, and their contribution to the field is still held in high regard.

3.2 System Model

The system model adopted for this study is based upon a network of workstations on a LAN. The use of LAN's implies that the nodes are on a fully connected network. All the workstations on the LAN are assumed to be diskless, with all files stored on a central file server. The file server is used solely as a central repository for data. None of the system's workload originates or executes on the file server. Therefore the transfer of a job that has not begun execution will entail no overhead due to the movement of job related data.

The bulk of algorithm evaluation is carried out on a system of 20 nodes. Systems of this size have been used in many previous studies [Eag86a, Mir89b, Ben93, Kru94] and are assumed to be an adequate testbed for load sharing algorithms. A larger system of 40 nodes will be considered in order to assess the scalability of algorithms. Due to limited resources validation and verification through implementation was not possible for systems any larger than 20 nodes.

The client-server model is often used to describe a distributed system and is adopted here. A busy node can be thought of as a prospective client and an idle or lightly loaded node as a prospective server. The objective of a load sharing algorithm to identify the latter to the former and facilitate any subsequent job transfer.

In the UNIX workstation environment considered in this study the client and server will both be processes running on distinct machines. In order to communicate with each other some form of inter-process communication (IPC) must be used. IPC across a network is by no means a trivial matter but it can be greatly simplified with the use of the remote procedure call (RPC). RPC facilities are now widely available on distributed systems and easily accommodate the needs of a load sharing algorithm, by offering a machine independent communication mechanism [Blo92].

3.2.1 Aspects of Heterogeneity

The main direction of this work is in investigating the effects of system heterogeneity on load sharing algorithms. In order to evaluate several systems there must be a means of ordering them. A possible means is to use the squared Coefficient of Variance (CV) of processing powers of the nodes. The larger the CV the greater the degree of system heterogeneity. A homogeneous system will have a CV of zero.

n = number of classes in system, f_i = number of nodes in class i, x_i = power of nodes in class i

$$\boxed{CV = \frac{(M_2 - \mu^2)}{\mu^2}} \quad \text{where } M_2 = \frac{\sum_{i=1}^{n} f_i x_i^2}{\sum_{i-1}^{n} f_i} \quad \text{and} \quad \mu = \frac{\sum_{i=1}^{n} f_i x_i}{\sum_{i=1}^{n} f_i}$$

Figure 3.1 Squared Coefficient of Variance of System Processing Power

However its is possible for two different systems to have the same CV. Consider 2 systems of 20 nodes with the same total processing power, A3/B3 and A7/B7 in Table 3.1. The nodes in these systems are split into two groups, with 12 in one group and 8 in the other. In one system the larger group of nodes has 30% less than the processing power it would possess in a homogeneous system, whilst in the other system the same group has 30% more. The CV will be the same for two different systems.

To differentiate the between the two examples and give a better measure of degree of heterogeneity the skewness of processing power can be used in combination with CV. A positive skew will indicate that the less powerful nodes (less powerful than the average for the system) are in the majority. Conversely a negative skew will indicate that the powerful nodes form the majority in the system.

$$\boxed{SKEW = \sum_{i=1}^{n} \left(\frac{(x_i - \mu)^3}{\mu^3} * \frac{f_i}{\sum_{i=1}^{n} f_i} \right)}$$

Figure 3.2 Skewness of System Processing Power

In this study both the CV and skewness will be used to characterise the degree of heterogeneity of a system. All of the systems investigated will have the same total processing power but this will be distributed in a variety of ways. If overall processing power in not maintained at the same level, comparison of results from different system is not valid. The systems nodes will be split into two groups of 12 and 8 nodes, as illustrated in the previous example. For ease of reference the majority group will be known as group A and the minority group B. Total power of the system is set at 20. In total 10 systems will be used. The composition of each is shown in Table 3.1. The division of processing power in this manner gives a broad spectrum of systems on which evaluation is made. Relative processing power of the nodes is varied between 1 : 1.5 and 1 : 66.

Systems in which the group sizes are very different give less variation. Consider a system in which the groups of nodes are split 18 : 2. Negative skew values are possible but not to any great degree. Even if the majority group has 99% of total processing power the skew is slight. When the minority group has the lions share the degree of heterogeneity rises rapidly. This configuration is used but only to assess algorithm adaptability.

While the present study was restricted to the systems with two types of node predominantly those defined in Table 3.1, the measure of heterogeneity adopted here can be used in the more general situation where there is more variety in node power. The present study was limited to the 12:8 and 18:2 split only due to restrictions of time and resources.

	Power	Fraction of total power		Power	Fraction of total power	skew	CV
A1	0.350	0.21	B1	1.975	0.79	0.206	0.634
A2	0.417	0.25	B2	1.875	0.75	0.149	0.510
A3	0.500	0.30	B3	1.750	0.7	0.094	0.375
A4	0.667	0.40	B4	1.500	0.6	0.028	0.167
A5	0.830	0.50	B5	1.250	0.5	0.004	0.042
A6	1.167	0.70	B6	0.750	0.3	-0.004	0.042
A7	1.330	0.80	B7	0.500	0.2	-0.028	0.167
A8	1.500	0.90	B8	0.250	0.1	-0.094	0.375
A9	1.583	0.95	B9	0.125	0.05	-0.149	0.510
A10	1.650	0.99	B10	0.025	0.01	-0.206	0.634

Table 3.1 System Composition With nodes divided 12 : 8

3.2.2 System Loading Conditions

The commonest method of load distribution in previous work has been a homogeneous distribution across the system. In a heterogeneous system this is not a safe assumption. It is highly unlikely that a powerful workstation will experience the same offered workload as a much slower counterpart. Even if workstations are office based and so accessible by only specified users the ability to logon remotely and execute work on other machines on the same system is widely available. In fact any system in which these activities were not allowed would not lend itself to load sharing anyway. Another possibility is that of remote users, gaining access via modem connections, i.e. researchers working from home. They are most likely to concentrate their efforts on the powerful machines in the system. These ideas do not contradict the principle of transparency, for it is not possible to hide the relative capabilities of machines from any user group.

Assuming that more powerful nodes do experience a heavier workload then the further assumption that load may be in proportion to processing power seems fair and has been adopted in other studies [Mir89b]. This is really just an extension of the principle used in homogeneous studies. Proportional loading will be used in the main in this study with job interarrival time being inversely proportional to processing power. Other cases are included for the purpose of judging algorithm adaptability in coping with more random loading patterns. In some cases a proportion of the nodes will experience no offered load at all.

The average service time of all jobs is 10 seconds on a node of processing power equal to 1. The actual service time will of course vary depending upon the executing node. In other work the trend has been to use anonymous "time units" rather than seconds, but the overheads in this study are based upon measurements of RPC timings where the relevant units are seconds. Some attempts at measurement of service times have been made [Zhou87, Zhou88, Kara95] and these range from 1.5 to 7.5 seconds.

Three levels of overall system utilisation are used in the evaluation. These are 50%, 70% and 90%. Corresponding to light, medium and high loading conditions [Kar95]. Load sharing at system loads of less than 50% gives little performance improvement over the no load sharing case except in cases of extreme loading patterns.

The system loading level is modified by changing the job interarrival time. Job service time is the same for all levels of system load and across all types of node.

3.2.3 Overheads Due to Remote Procedure Calls

The overhead incurred due to load sharing activity can be divided into three parts:

- The cost of information dissemination.
- The cost of transferring a job from one node to another.
- The CPU cost of algorithm decisions.

A primary requirement of any evaluation study that does not use measurement on a real implementation is that these overheads are accurately estimated. All communication between nodes will take place with the use of RPC's and so the cost of executing these is the basis for the estimates used in this study. Job transfer is also achieved by the use of an RPC, with no other costs, as the use of diskless workstations is assumed. As the algorithms proposed are simple in operation requiring very few instructions to be performed outside of those connected with the RPC mechanism the CPU cost of implementing them will be ignored.

Figure 3.3 shows the sequence of operations connected with a RPC. The diagram is not to scale but it does illustrate the delays that are inherent in any RPC. There is an initial delay on the client side as the client stub marshals the arguments of the local procedure call into a network message, followed by a network delay in transmitting the message. On the server side a server stub converts the arguments from the network message and makes a local procedure call to execute the server function. After the server function has been completed the return values are converted into another network message and sent back to the client stub which converts them back. Again network and processing delays are incurred in the course of these actions. There is the possibility that the client and server can both be on the same node in which case no network delay would be experienced, but as this will not occur in the load sharing environment it will not be discussed any further.

for the different delays according to machine pair would entail extra processing for each probe made and so hamper simulation studies. As the probes are random a balanced combination can be expected so one set of values for RPC overhead are used. The overhead estimates used are:

Probing:	10 ms to client node
	10 ms to server node
	30 ms per job
Job Transfer	10 ms to client node
	10 ms to server node
	30 ms per job

An assumption inherent in the above timings is that the operations involved in a RPC are evenly divided between the client and server as they perform symmetrical operations. The delay due to the server procedure when probing is performed is negligible, measurable in microseconds rather than milliseconds. The server procedure delay in job transfer is separately accounted for when job processing starts.

The effects of varying load are shown in Figure 3.5. RPC's are sent from Westar (SS10) to Terry (IPX) in the same manner as for the 5 machine test reported earlier. Results were gathered over a week but during this time the load on both machines was varied from an idle state to a utilisation as reported by the UNIX system call *uptime* of over 6, i.e. 6 jobs were in the ready to run queue. The changes in loading were not observed to have any effect on the RPC response time. The peaks shown are caused by the heavy network traffic during system backup which is conducted during the small hours every night.

The independence of RPC response time from loading conditions can be explained by the scheduling policy implemented on the workstations. Any new process or in the case of the server stub one that has only used the CPU lightly will obtain a higher scheduling priority and so rapid access to processing facilities [Sun90]. Therefore the timings proposed will be used at all levels of system utilisation.

load sharing inefficient. Unfortunately there is no way of knowing service time in advance. The relative performance of the algorithms will not be effected by this decision, except for the IDEAL algorithm which is used as an upper bound on performance.

A Threshold is used to determine if a new job should be considered eligible for transfer. The threshold will be based solely on local load at the time of job arrival. The metric by which local load is judged will be the number of jobs currently executing locally or in the ready to run queue. A simple but effective measure for workstations such as, Sun 2 [Fer87], and Sun 3/50 [Kun91]. The optimum threshold length is investigated in the course of the study.

3.3.2 Information Policy

Apart from a version of the RANDOM algorithm [Eag86a] which operates without any system state information except local loading, the dissemination of system state information will be accomplished with the use of probes (polling individual nodes). The alternative broadcast has been discussed in section 2.1.3. Use of broadcast has been limited and it is not a popular choice when considering fully connected networks, due to the associated high overhead with little perceived benefit. All recent load sharing algorithms use probing of some form.

Selection of the nodes to be probed will be made on a random basis as jobs eligible for transfer are identified. This will ensure that the information collected will be as current as possible. The use of prior information in the selection of nodes to be probed has been investigated [Shi92]. Increased performance was noted at system loads of greater than 85%, due to a greater efficiency of probing. However the transfer policy used was somewhat questionable with the threshold not varying with load. Whilst noting the potential of intelligent probes a random policy is considered adequate for this study.

3.3.3 Location Policy

A variety of location policies will be investigated in the algorithms evaluated. This is the area in which they display the greatest diversity. The simplest policy is that of blind location, where a suitable node is selected at random. This strategy has been used as a benchmark in many studies of homogeneous systems [Eag86a, Zhou88, Kre92].

Thresholds based on remote loading have been widely used in previous work such as the SHORTEST and THRESHOLD algorithms [Eag86a]. Proposed for use on homogeneous systems they are tested on the heterogeneous systems used in this work. Of more relevance are the location policies primarily designed for use where the processing speeds of nodes differ.

All the algorithms proposed in this study use a load index which is the ready to run queue length weighted by the relative processing power of the remote node. The use of fixed thresholds is investigated as well as that of flexible thresholds, where the remote load is compared with the local load in deciding whether to select the node probed. The mechanics behind these variations in location policy, which are kept simple to avoid the imposition of excessive overhead, are detailed in the next section, where all the algorithms evaluated are described.

3.3.4 Description of the Algorithms

The five algorithms on which this study concentrates are described below. Their descriptions are divided into Transfer, Information and Location policies (TP, IP, LP). With the exception of threshold levels, the values of parameters such as probe and transfer limits are postponed until later chapters.

The measures used to give upper and lower bounds on response times are the M/M1 and IDEAL scenarios respectively. The M/M/1 or no load sharing case was illustrated in Figure 1.1. The only complication for heterogeneous systems is that a response time must be calculated for each type of server and the weighted average computed. The IDEAL case used to reach a lower bound is based on the simulation of an idealised load sharing scheme, in which complete knowledge of queue length and job sizes at all nodes is assumed available and each job is sent to the node where it will be completed in the least possible time. Once a job has been sent to a node it cannot be migrated. Transfer and information costs are assumed to be zero. This is the same principle as the M/M/K scheme shown in Figure 1, but by utilising knowledge of job service times a truly optimal solution can be reached. The results of simulation of the IDEAL algorithm provide interesting information on the optimum distribution of workload.

RANDOM:

TP - A fixed threshold is used. If the arrival of a job causes the local load to reach or exceed the set threshold and the job has not been transferred more times than its transfer limit, then that job is considered eligible for transfer.

IP - No information policy is needed as no system state information is used in the location policy.

LP - A node is picked at random and the current eligible job is transferred to it.

SHORTEST:

TP - A fixed threshold is used, set at 1 for system utilisations up to 70% and 2 for higher. If the arrival of a job causes the local load to reach or exceed the set threshold and the job has not been transferred more times than its transfer limit, then that job is considered eligible for transfer.

IP - Nodes are selected at random and probed, in response to which they return their load, the total number of jobs in the ready to run queue. Probing continues until the number of nodes probed reaches the probe limit, unless an idle node is located.

LP - If an idle node is located, the current eligible job is transferred to it immediately. Otherwise when the probe limit is reached, the job is sent to the node with the lowest load, provided that load is less than the threshold used in the transfer policy.

HETRO: (Attempts to account for system heterogeneity)

TP - A fixed threshold is used, set at 1 for system utilisations up to 70% and 2 for higher. If the arrival of a job causes the local load to reach or exceed the set threshold and the job has not been transferred more times than its transfer limit, then that job is considered eligible for transfer.

IP - Uses a weighted load in its Location policy, this entails the Information policy gathering details of a remote node's load and processing power. Probing continues up to the probe limit unless an idle node is located. The weighted load is calculated as:

$$\boxed{\mathit{weighted\_\ load} = \frac{\mathit{local\_\ power}}{\mathit{remote\_\ power}} * \mathit{remote\_\ load}}$$

LP - If an idle node is located the current eligible job is transferred to it immediately. Otherwise when the probe limit is reached the job is sent to the node with the lowest weighted load provided that load is less than the threshold used in the transfer policy.

HETQL: (Accounts for heterogeneity and uses local queue length in location policy)

TP - All jobs that have not exceeded their transfer limit are considered eligible for transfer if the local node is busy, i.e. has a load of one, no matter what the system utilisation.

IP - Uses a weighted load in its Location policy, this entails the Information policy gathering details of a remote nodes load and processing power. Probing continues up to the probe limit unless an idle node is located. The weighted load is calculated as:

$$\boxed{\mathit{weighted\_load} = \frac{\mathit{local\_power}}{\mathit{remote\_power}} * \mathit{remote\_load}}$$

LP - If an idle node is located the current eligible job is transferred to it immediately. Otherwise when the probe limit is reached the job is sent to the node with the lowest weighted load provided that load is less than the local load as measured by ready to run queue length

HQNIT: (Accounts for heterogeneity, uses queue length and no immediate idle transfer)

TP - All jobs that have not exceeded their transfer limit are considered eligible for transfer if the local node is busy, i.e. has a load of one, no matter what the system utilisation.

IP - Uses a weighted load that takes into account the effect of possible job transferral in its location policy, this entails the information policy gathering details of a remote nodes load and processing power. Probing continues up to the probe limit. The weighted load is calculated as:

$$\boxed{\mathit{weighted\_load} = \frac{\mathit{local\_power}}{\mathit{remote\_power}} * (\mathit{remote\_load} + 1)}$$

LP - The newly arrived job is used in calculation of the local load. Transfer will not occur until the probe limit is reached, as no node will have a weighted load of zero. The eligible job will be transferred to the node with the lowest weighted load if :

$$\boxed{\mathbf{local\_\ load} + 1 > \mathbf{weighted\_\ load}}$$

This ensures that jobs will only be transfered to less powerful nodes if they will complete more quickly.

3.5 Simulation

The simulation of systems can be divided into two categories, continuous simulation and discrete event simulation. The approach taken is normally determined by the nature of the system to be evaluated. Continuous simulation is normally applied to systems in which state variables are continuously changing with respect to time. This is not the case in a distributed computer system where the state of the system will only change at discrete points in time on the occurrence of an event. Hence the type of simulation used in the evaluation of load sharing algorithms will be of the discrete event variety.

The simulation model used in this study is constructed using the MODSIM II programming language released by the CACI Products Company. This is an object oriented programming language that provides direct support for discrete event simulation. There are two approaches to discrete event simulation, the event oriented approach requires each event to be a separately coded activity. However MODSIM adopts the process approach with groups of related activities grouped together and the possibility of the process suspending execution when needed. The uses of processes eases the construction of larger models by simplifying the logical flow of the program.

Most simulations will attempt to discover the steady state behaviour of the systems investigated. Initial conditions will correspond to that of an idle system and so an initialisation ‘warming up’ phase is included and only after this has expired are results collected. Total run length is at least 10 times that of the initialisation phase, depending upon the level of system utilisation. A higher utilisation will give an effectively longer run as more jobs will be processed. Although the results collected are in the form of discrete time data, i.e. average job response time, the simulation runs are stopped at specified clock times as determining total system job output during simulation is easily accomplished. However the length of the simulations is such that any discrepancy between the total number of jobs offered in different replications is considered negligible. Standard error for all results is less than 5% at the 95% confidence level.

3.6 Measurement

Measurement is carried out on a working implementation of the same system as that modelled by simulation. This provides a means of verifying the model by ensuring that the features used in the simulation can actually be implemented on a real system. The results of the measurement are used in validating the simulation assumptions and results. In particular the assumptions about network behaviour and the effect of the added traffic due to load sharing activity.

The machines used were a mixture of Sun workstations all running the Solaris 2 operating system. All the machines were located on the same LAN. RPC's were used as the only means of communication between the machines. Two server procedures were needed for each machine one to handle probes, the other to handle transferred jobs. The code implemented operated outside the kernel, as any other approach would have necessitated full super-user control of each machine. This was not possible as the machines used were part of the general computing resource of the university.

The workload offered was all of the same type varying only in execution time. Although entirely CPU based this was not seen to be a handicap as the object was merely to affect the processing speed of other jobs currently executing. No discernible overhead was experienced due to collecting results, as they were only written to file at the end of each measurement period.

4. Discrete Event Simulation

4.1 System Model

The same system model is used as the basis for all the simulations performed. It consists of a collection of nodes communicating across a network. Any degree of heterogeneity in the system is exhibited solely in the processing power of the nodes The relative processing power of each node is known and does not vary during operation. All inter-node communication is performed through the use of RPC's, the durations of which are known and are independent of individual node processing power.

The functionality of each node is identical and based around a set of core operations. A stream of jobs is generated locally to represent the offered workload. As each job is generated a decision is made as to whether it should be executed locally (added to the local quue) or made available for possible transfer to another node for execution. Information on processing power and current load is passed between nodes on request. Transfer decisions can then be made based upon the information gathered. Each node has the facility to send jobs to and receive jobs from others in the system. On reception of a job a node adds it to the local queue for subsequent execution. Jobs in the local queue are executed on a First Come First Served (FCFS) basis.

4.1.1 Processes at a Node

The functionality of the nodes can be divided into more specific processes than the general description above. These are detailed below:

- Generation of offered load - The jobs generated at each node have an exponentially distributed interarrival and service time. All nodes generate jobs with the same average servicetime. However, in order to ensure each node has the same initial utilisation the average interarrival time is inversely proportional to its processing power.

- Transfer policy - As jobs are generated at a node they must be assigned for local execution or allocated for possible transfer. This decision will be based upon the current load at the node concerned.
- Information policy - Once a job has been allocated for possible transfer, information on the system state is gathered to use in the location decision. Only partial knowledge of the system state is needed and this is gathered through the use of probes to randomly selected nodes.
- Probe response - Complementary to the information policy is a mechanism to answer incoming probes.
- Location policy - Using the information gathered via probes around the system a load sharing decision is made as to the execution location of the job.
- Job transfer - When selected due to the operation of a load sharing algorithm a job will be transferred to another node.
- Job reception - On reception of an incoming job the destination node adds it to the local queuefor execution locally.
- Job execution - Jobs in the local queue are executed immediately on arrival in the queue. When the local queue is empty the execution process will wait for a signal indicating a new arrival.

4.1.2 Inter Process Communication

Communication between processes takes place on both an intra and inter node basis. Inter node communication is based on message passing, implemented entirely with the use of RPC's. Although it is possible to use RPC's as a means of intra-node communication they are too expensive, in terms of overhead to be of practical in this model. Two methods of intra-node communication are employed. Shared memory allows two or more processes to access the same information. Software interrupts in the form of signals allow processes to co-ordinate activities between each other.

4.1.3 Additional Functions Required

In addition to the core processes described in section 4.1.1 some extra functions are needed for a model from which useful results can be derived.

- Input parameters - A means of inputting variable parameters is needed. This allows the model to be flexible enough to handle a wide variety of possible scenarios.

- Initialise and start - All of the pre-built constructs used in the model must be initialised to the correct value before the commencement of any system activity. In the case of the model used in this study where the activities of several separate entities are interwoven, it is essential that all entities are also fully initialised before system activity starts.
- Statistics - Routines are provided for the collection of a number of different statistics. The most important is the average response time at each node. That is the time from a job's arrival in the system until the end of execution. Other statistics must also be collected not only to allow a greater understanding of the effect of differing input parameters and load sharing algorithms, but to aid in verifying that the simulation model is operating in the manner intended.
- Termination - at the end of the a pre-determined period the model must be halted. This has to be an orderly operation not just to ensure that the simulation period is strictly observed, but also to prevent any data being lost by the uncoordinated termination of any objects.

4.2 MODSIM

MODSIM is a high level special purpose simulation language. Although it can be used as a general purpose computing language, it is aimed at the construction of simulation models. There are many similarities between MODSIM and Modula-2, in syntax, data types and control structures. The differences are most apparent when considering the **object oriented features** and **simulation utilities** that are provided by MODSIM. All the standard object oriented properties are supported, such as inheritance, encapsulation and polymorphism. These are combined with extensive **library modules** which provide a large number of constructed objects to help in the writing of discrete event oriented simulations. Using object oriented techniques to develop these types of simulations has a history of over 30 years. One of the first object oriented programming languages to be developed for discrete event simulations was SIMULA, which became available in the 1960's.

As befits a modular language modules can be separately compiled. Compilation in all forms is handled by MSCOMP, MODSIM's compilation manager [CAC93a]. MSCOMP first uses the MODSIM compiler to produce a 'C' code version of the original MODSIM source code. This is then compiled using the standard 'C' compiler available. In this case it was the Sun UNIX compiler. Should more than one module be used MSCOMP automatically performs any linking that is needed to give the final executable code.

4.2.1 Object Oriented Features

The objects around which object oriented programming is based are the combination of data structures, and operations which can manipulate that data. Different categories of object are referred to as classes or types and individual examples as instances or objects. A definition of these concepts is offered by Booch, "An object has state, behaviour and identity: the structure and behaviour of similar objects are defined in their common class: the terms instance and object are interchangeable" [Boo91].

MODSIM uses the terms fields and methods for the two properties that define an object's type. These terms are synonymous with attribute and operation [Gra94, Rum91] The fields are used to represent the state an object is in and the methods are a means of describing the behaviour of an object. The packaging together of state and behaviour in this manner is known as encapsulation. The object becomes self-contained and immune from corruption from outside sources as only its own methods are permitted to alter its fields. MODSIM does allow an object to access the fields of another. A field may be any permissible variable including an object.

MODSIM supports the idea of polymorphism, where operations of the same name may perform different actions when performed by different objects. The term method ties an operation to a particular object. The ability for an object to be based on a another previously defined object and then inherit all of the earlier objects properties is available. This sharing of fields and methods is known as inheritance. Although the properties of polymorphism and inheritance are not utilised in the model developed they are noted here as they do enhance the language.

Communication between objects is possible through the use of message passing. This is a means by which one object can request to invoke the methods of another. Invoking an object's methods can only be performed in MODSIM by message passing. The message passed is a request for an object to perform a method. If parameters are expected by the requested method then these are passed in the message as well.

The modular design of MODSIM allows the construction of models using constructs from various different sources. However it is possible to have all the code in one MAIN module although this is only advisable for relatively simple programs. The MAIN module can import various constructs from the supplied library modules or these constructs can be used in creation of user defined constructs, as is the case in non-trivial programming. These new constructs can be defined in the MAIN program but the norm is to create new library modules.

A library module is comprised of two separate parts, the DEFINITION and IMPLEMENTATION modules. The DEFINITION module contains a declaration of all the constants, types, procedures and variables that are importable by any other module, but no executable code. This is the public section of the module providing adequate information for any future user. The actual implementation details of all procedures and objects are included in the IMPLEMENTATION module. These details are considered private as knowledge of them is not necessary for users of the modules facilities. Each part of the same library module will have the same identifier but a different prefix, D or I, for DEFINITION or IMPLEMENTATION module respectively. A MAIN module is prefixed by M and all modules have the extension '.mod'.

4.2.2 Simulation Utilities

MODSIM takes a process oriented approach to discrete event simulation as opposed to an event oriented approach. In an event oriented system each event is considered as a single activity during which no time can pass This can lead to problems with larger models as the flow of logic becomes more complex. Whereas in a process oriented model the process is a sequence of events or activities all pertaining to a particular entity. The processes are implemented as routines in which time can elapse. This simplifies matters by allowing the behaviour of an object to be described via the routines. In MODSIM these routines are known as the methods, introduced in the previous chapter.

Three different types are available to describe an objects behaviour : ASK, TELL and WAITFOR. The ASK method is used to perform a synchronous operation such as obtain the value of a state variable contained in an objects fields. No simulation time can be associated with an ASK method, i.e. performing an ASK method occurs instantaneously in terms of the overall simulation. To pass time a TELL method must be used, during the execution of which the simulation clock can be advanced. Because there is no guarantee that a TELL method will ever return it must be used asychronously and so no TELL method can return a value. The third method WAITFOR does provide for both passing simulation time and returning variables, but as it is not implemented in the model used for this study it will not be discussed further.

The processes around which a MODSIM simulation is based must have the ability to interact with each other. This is provided in two ways. First a method can wait for an event to occur as signalled by a trigger object (TriggerObj). Alternatively an executing method can be explicitly interrupted by another causing the "ON INTERRUPT" clause of the method to be executed.

Interrupts of the form provided by a TriggerObj are essential in a system involving queues. Without them any method waiting for an empty queue to receive a new member would have to be constantly checking the queue's contents. This would lead to a tremendous waste of CPU time and extend considerably the time to complete any simulation. The Fire method of a TriggerObj is constructed so that it only has effect if the object it is directed at is actually waiting for it. So there is no danger of queued signals negating method synchronisation.

In order to keep track of all the existing objects and ensure their activities are scheduled correctly MODSIM keeps a "pending list" of object instances. This list contains all objects with scheduled activities and is ordered by the imminency of those activities. Should an object have more than one scheduled activity this is shown in its own activity list. This leads to the formation of a two-dimensional list an example of which is shown in Figure 4.1.

As activities in the list are executed the list is resorted so that the most imminent activity is at the head. Only TELL activities are put in the pending list. ASK methods are executed immediately. After the completion of an activity simulation time is advanced to the time of the next scheduled activity. The timing procedure finishes when either the pending list is empty or on the execution of the "StopSimulation" command.

Another simulation oriented problem dealt with by MODSIM is the collection of statistics. A set of monitor objects are specifically provided for this task. Depending on how they are declared monitor objects either invoke specified methods on being referenced (right monitored) or when modified (left monitored). All the statistical monitors are left monitored, recalculating a set of standard statistical values (count, mean, standard deviation , ..etc.) every time the monitored variable is modified. The set of statistical monitors allows real or integer variables to be weighted against time, or not, as the situation demands.

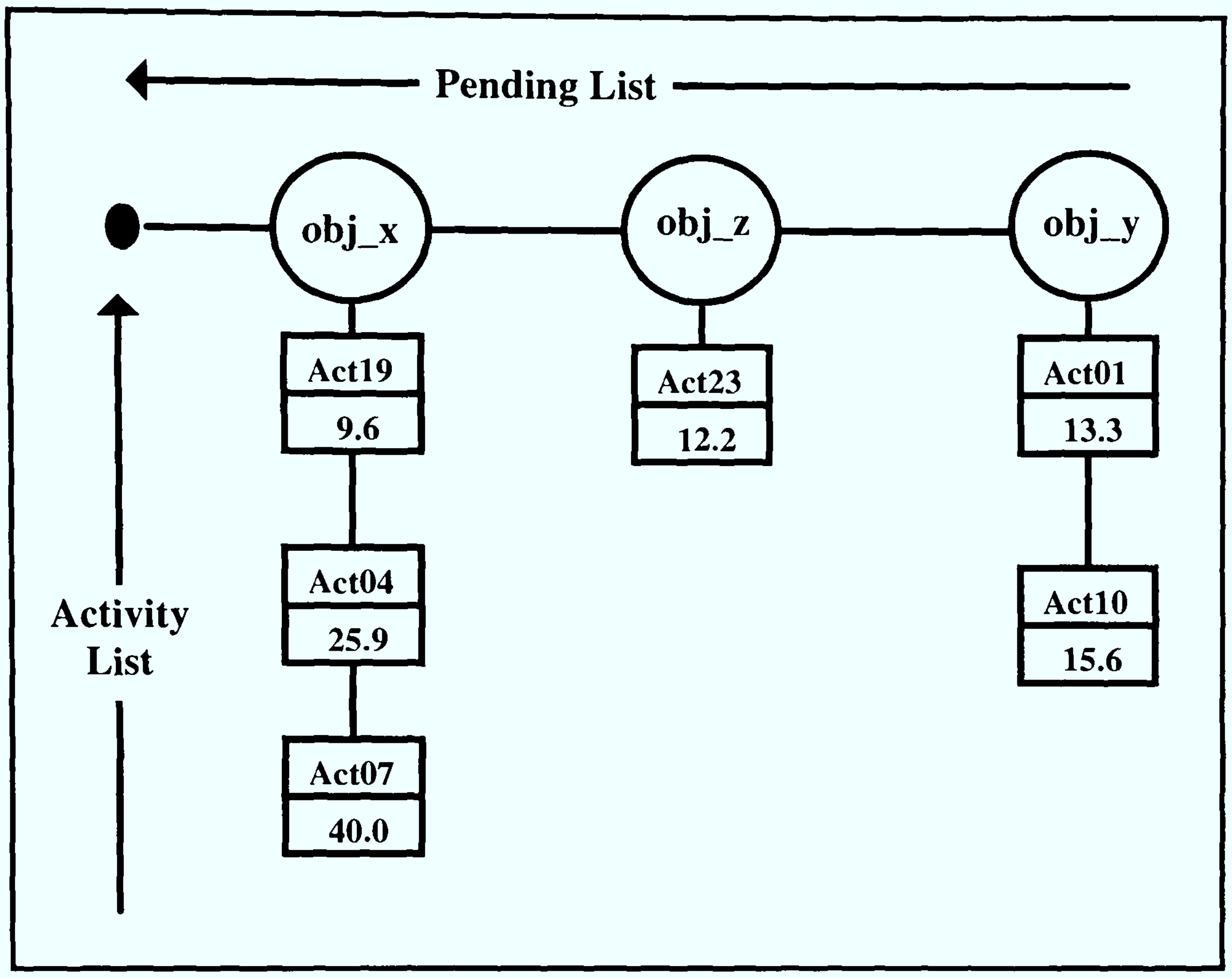

Figure 4.1 MODSIM Pending List Structure

Random number generation is also catered for by MODSIM. The RandObj object can be imported from library and will provide a series of randomly generated numbers. There are a number of possible probability distributions available, including the uniform and exponential distributions used in the model for this work. A means of varying the initial seed provided to the random number generators is available. This allows any number of objects to access independent and discrete sets of random variables.

4.2.3 Standard Libraries

MODSIM provides a number of built in procedures to cover many of the requirements of a simulation model. But these represent only a small portion of the available set. The others along with extra constants, types and all the pre-defined objects are available via the standard libraries, fully catalogued in the reference manual [CACc93].

There are ten standard libraries available with MODSIM II. Four were utilised in the construction of the model described in this chapter. A brief summary of these is presented below:

- ListMod : This library contains the objects and composite objects that can be used to group records and objects. The data structures that can be imported include lists, stacks and queues. Ample means to manipulate these data structures are provided by the methods of the relevant object.
- RandMod : As its name suggests this library's facilities are concerned with random number generation. It only contains one object, RandomObj. With many methods to allow different distributions of random numbers to be sampled. There are also some procedures available i.e. FetchSeed, which provides pre-defined seeds.
- SimMod : Time dependent features are the area for which this library's contents are intended. Without the SimTime procedure which returns the current simulation time it would be impossible to gather any meaningful statistics from a model. Procedures to start, stop and change the flow of simulations are also available, as is the TriggerObj vital in co-ordinating activities.
- StatMod : All the objects that can be used as monitored variables for the collection of statistics are defined in this library.

4.3 Simulation Model

Techniques such as object oriented analysis have been widely used in developing discrete event simulation models, as they provide a natural way to map the real world system onto a simulation model. This fact may appear obvious when the model is to be constructed in an object oriented language like MODSIM. However the outcome of any analysis should be tempered by the goals of the simulation. A detailed analysis may provide an exact mapping but implementation may not be possible or desirable.

Any system can be viewed at various levels of abstraction, the degree of granularity increasing until every process occurring in the real system is modelled. This should be avoided if possible. Only the features that are relevant to the simulation objectives need be incorporated. Once identified they should be implemented at as high a level as possible without losing any of their functionality. In addition the simulation model will need to include a number of extra features intrinsic to the task of simulation. These will provide for initialisation, reporting and termination.

To describe the design stages and implementation of the simulation model adopted in this study the object oriented notation associated with the Object Modelling Technique (OMT) [Rum91] is used.

Viewing the system to be simulated at the highest level of abstraction it can be seen as an aggregation of its constituent sub-systems or objects, shown in Figure 4.2. Each object is represented as having a multiplicity of association of one, except for the node object of which there must be at least 2 to form a distributed system. The offered load in this view represents the total workload experienced by the system in question.

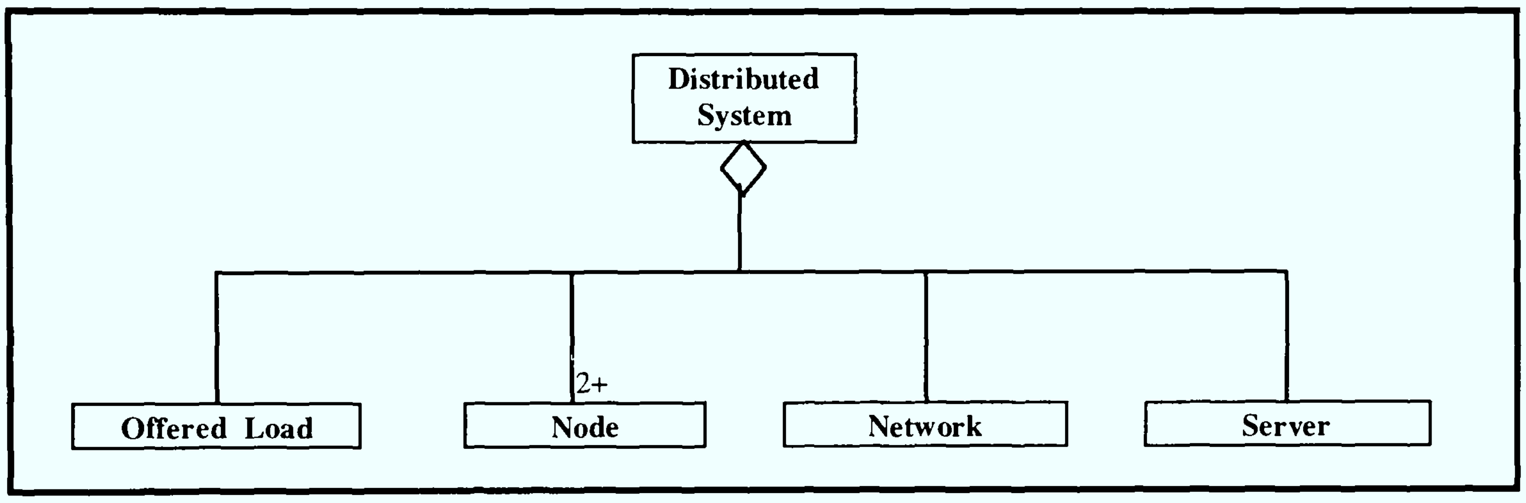

Figure 4.2 A Distributed System as an Aggregated Object.

Although all of the objects in Figure 4.2 are present in the system model it is not necessary to include them all in the simulation model. The server is needed as it has been assumed that all the nodes are diskless workstations and job transfer is simply a matter of sending an command line instruction. Modelling it would be pointless and any delay in retrieving data can be assumed to be part of the total job service time. Similarly explicit modelling of the underlying communication network can be avoided by representing its effect with fixed communication delays. The impact of the extra traffic due to load sharing activities is of interest but modelling the network at the required level to examine it is too complex to incorporate into any useful simulation. The offered load is made up of jobs originating at individual nodes. It is therefore more appropriate to consider it at a lower level of abstraction as a component of the node object.

The system's nodes can therefore become the basis of the model. they in turn can be visualised as aggregated objects. The significant components are shown in Figure 4.3

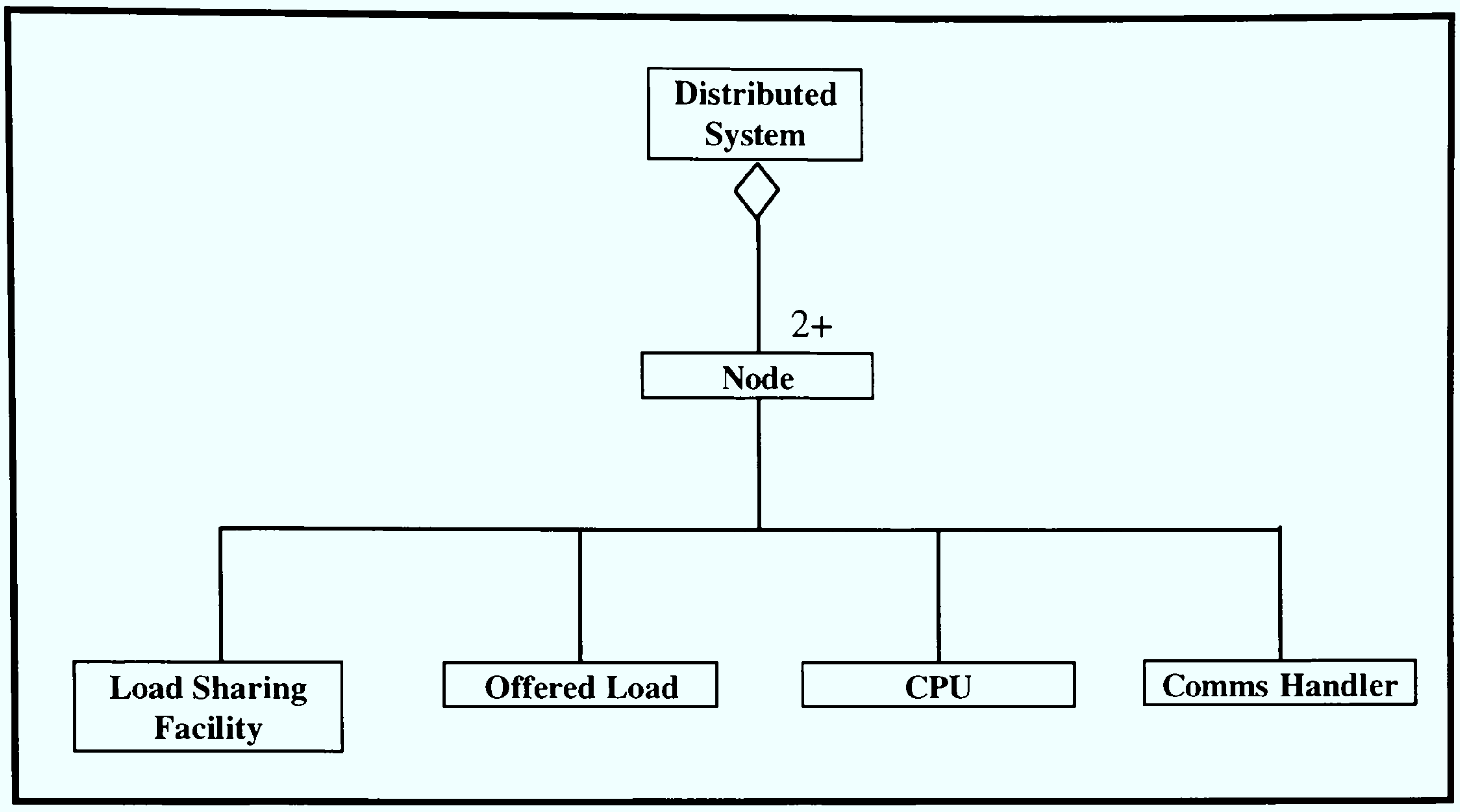

Figure 4.3 Component Analysis of an Aggregated Node.

Each component could be modelled as a separate object. However since the functions they will perform are not going to be simulated in detail it was felt they could be more simply implemented as methods of the node object.

Whilst the node object is the most important element of the simulation model, some additional objects had to be defined to provide the added functionality required to administer the simulation environment. The added features allow initialisation, data collection and orderly shutdown of a simulation. A brief summary of the object types used is given in Table 4.1.

The simulation program itself consists of a MAIN module, loadshare and a library module, Hetrodelaylib. The latter is in two parts. Dhetrodelaylib is the DEFINITION module of the library which contains all the type, variable and object definitions, with the IMPLEMENTATION module, Ihetrodelaylib containing all the object implementation details. All global variables are declared in the DEFINITION module as this makes them available to the other two modules.

Object Type	Functions performed
NodeObj	Generate local load, invoke load sharing algorithms, transfer jobs, execute jobs, compile local statistics, remove local data structures.
GenesisObj	Initialise system and individual nodes, activate individual nodes, collate batch statistics, collate overall statistics at simulation end, remove global data stuctures
StopAllObj	Perform orderly shutdown of activities on individual nodes, stop simulation.

Table 4.1 Summary of Object Functions

4.3.1 The MAIN Module : loadshare

Jain states that a discrete event simulation needs a component that co-ordinates the routines constituting it [Jai91]. He even refers to it as the main program. This is the role of the loadshare program. Figure 4.4 portrays the operation of the program via pseudo code.

The initial purpose of the loadshare module is to allow all necessary variables, types and objects to be imported, followed by the input of variable parameters, normally via a batch file. Each set of parameter values is iterated over a number of repetitions. Every repetition uses a different seed. The iterations are typically of duration 60,000 seconds, split into batches of 5,000seconds. Statistics are gathered after each batch and at the end of each run.

At the end of each batch, the average response time for all jobs completed in that batch is calculated. For this study average response time is the duration between the point at which a job arrives in the system to it being executed and the result being communicated to the original node. Batch statistics are used primarily in the verification and validation of the simulation. A more comprehensive set of statistics is gathered at the end of each full run. These include:

- Overall system average response time.
- Individual node average response time.
- Number of jobs originating at each node.
- Number and average length of jobs not eligible for transfer executed at origin.
- Number and average length of eligible jobs refused transfer and executed at origin.
- Number and average length of jobs transferred.

```
START
    Import global variables from Hetrodelaylib
    Import modsim utilities from standard libraries
    Set global constants
    Input variable parameters (runtime, batchtime, probelimits, threshold, algorithm)

    LOOP (system utilisation varies according to input parameter )
        LOOP (Probelimit min to max)
            LOOP (desired repetitions each with a different seed)
                Calculate E(ta)
                Create new instances of GenesisObj & StopAllObj
                Invoke initialisation and activation of nodes, passing necessary parameters to
                    instance of GenesisObj
                Invoke instance of StopAllObj to cease simulation after runtime
                Invoke instance of GenesisObj to collect and output simulation statistics
                Remove instances of GenesisObj & StopAllObj
            END LOOP
        END LOOP
    END LOOP
END
```

Figure 4.4 Mloadshare.mod (Pseudo code)

The three objects that comprise the simulation model are: GenesisObj, NodeObj and StopAllObj. These are the objects that loadshare co-ordinates. They will be described in the following sections.

4.3.2 The GenesisObj Object

The GenesisObj object as its name suggests, creates the simulation system and initialises activities at all the constituent nodes. To accomplish this it is passed details of the system constitution and an initial seed by loadshare. As GenesisObj creates all the nodes it makes an ideal candidate for collating performance statistics when the simulation finishes. The system nodes form an array that is resident in its own address space. The full structure of the GenesisObj, with methods and fields is shown in Figure 4.5.

InitialiseNodes handles the creation of all the nodes, performing a separate FOR loop for each type of node to be implemented. Once a new node instance is created, the node is assigned a processing power and ID. By using global array element numbers the individual nodes can easily identify each other with the minimum simulation overhead. After creating a random seed the method then initialises the various NodeObj methods that will run continuously for the length of the simulation. The loop at the end of this method and the Batchresults method were used in compiling the ensemble averages needed for simulation output analysis. They have no effect upon the results gathered or operation of the simulation model. Neither passes any simulation time or affects any of the statistical counters used in the compilation of run end results.

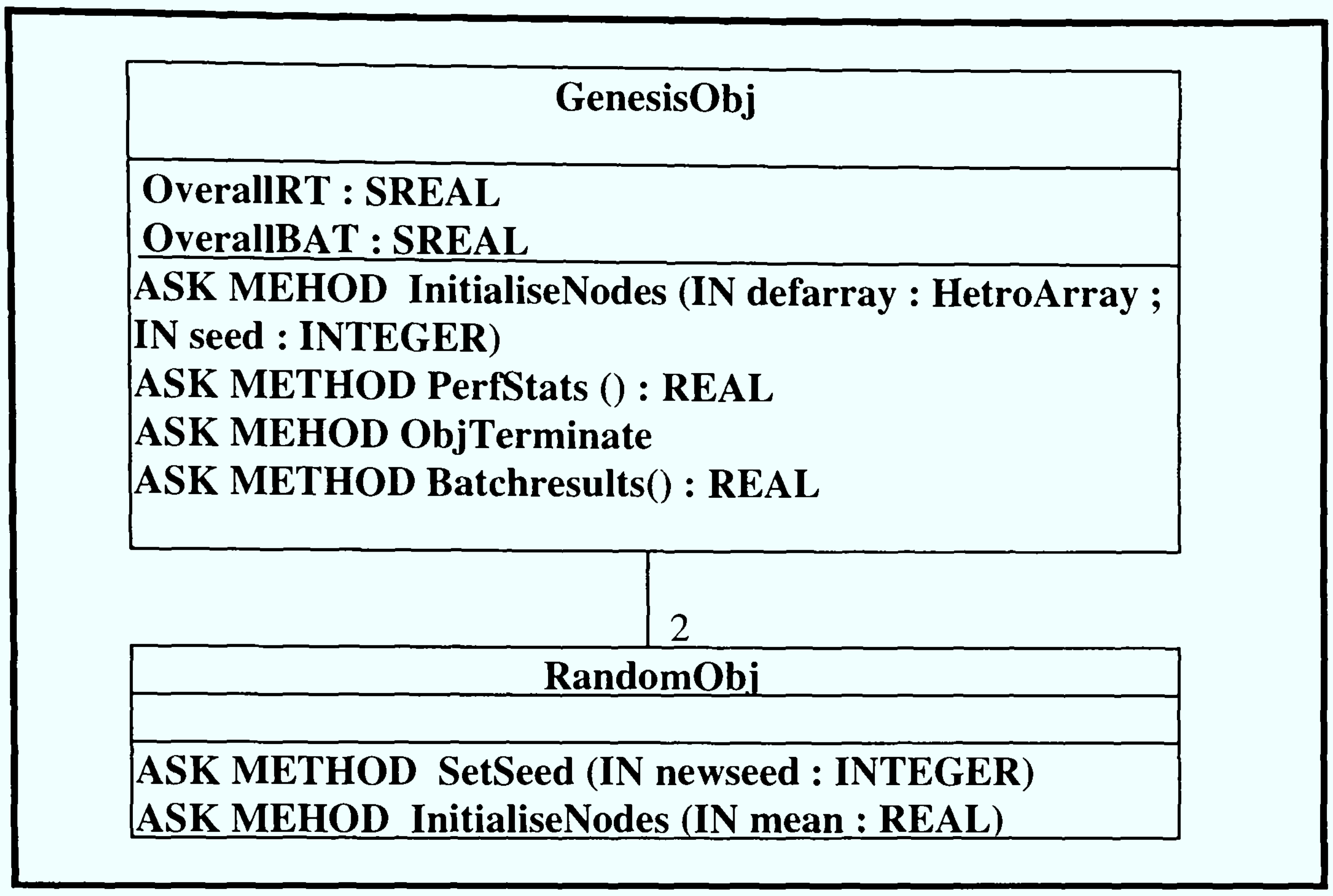
GenesisObj
OverallRT : SREAL
OverallBAT : SREAL
ASK MEHOD InitialiseNodes (IN defarray : HetroArray ; IN seed : INTEGER)
ASK METHOD PerfStats () : REAL
ASK MEHOD ObjTerminate
ASK METHOD Batchresults() : REAL
2
RandomObj
ASK METHOD SetSeed (IN newseed : INTEGER)
ASK MEHOD InitialiseNodes (IN mean : REAL)

Figure 4.5 Full Structure of GenesisObj Object

The Perfstats method operates only after simulation activity has ceased, but it is of considerable importance as it compiles the final simulation statistics. The overall average response time for the system simulated is returned to the MAIN module. In its calculation several other useful metrics, pertaining to each individual node, are arrived at. These metrics are printed to stdout, which is then redirected to a file. All the metrics calculated by this method were listed in the previous section.

Last of GenesisObj's methods is ObjTerminate. This method is a special feature of MODSIM. If it exists it is called before an object instance is deallocated. Thus allowing any 'cleanup' operations to be performed. In the case of GenesisObj this method disposes of all components that will consume memory. This action is essential if batches of simulation are performed, otherwise there is a danger of available memory running short if it is not de-allocated as simulation runs finish.

4.3.3 The NodeObj Object

Four areas of activity are needed in each node. These were identified in Figure 4.3. Three can be fully contained in single methods. However the constraints of object oriented programming forced the communications facilities to be spread across several methods. The association between the analysis results and methods used in the actual model is as follows :

• Offered Load	-	GenerateJobs
• Load Sharing Facility	-	Process*
• CPU	-	Execute*
• Communications Handler		Transmit
		Receive
		ReceiveJob

Process* and Execute* are starred to indicate that there is more than one version of the relevant method available. Only one of which is used in any single simulation run.

The full structure of the NodeObj as illustrated in Figure 4.6, shows other methods apart than those used to accomplish the four core tasks. These are used in initialisation, and housekeeping. They have no effect upon the simulation whilst it is in normal operation. A quick scan of the definition of a NodeObj seems to reveal a myriad of fields, but in fact only three of them are truly fields/attributes in the object oriented sense. Two of these, nodepower and JobQ.numberIn, are the metrics communicated between nodes in implementing the information policy of various load sharing algorithms. The other, responseT, is the main performance metric returned to GenesisObj to be used in compilation of overall system performance. Nodepower (REAL) and JobQ.numberIn (INTEGER), represent the processing power and current load of a node respectively. Whereas responseT is a statistical monitor object (SREAL) which stores the overall statistics on all jobs executed at the node.

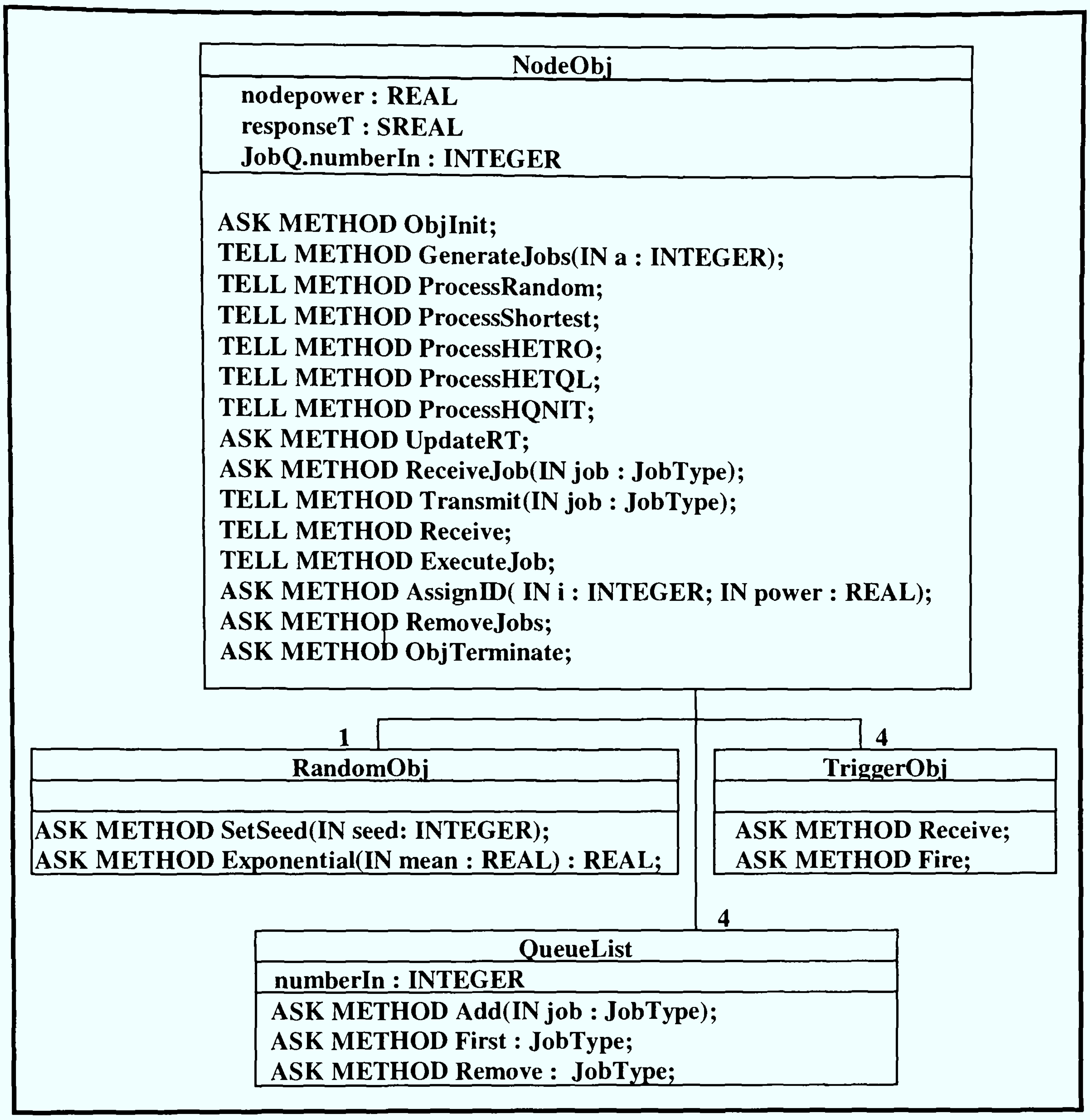
NodeObj
nodepower : REAL
responseT : SREAL
JobQ.numberIn : INTEGER
ASK METHOD ObjInit;
TELL METHOD GenerateJobs(IN a : INTEGER);
TELL METHOD ProcessRandom;
TELL METHOD ProcessShortest;
TELL METHOD ProcessHETRO;
TELL METHOD ProcessHETQL;
TELL METHOD ProcessHQNIT;
ASK METHOD UpdateRT;
ASK METHOD ReceiveJob(IN job : JobType);
TELL METHOD Transmit(IN job : JobType);
TELL METHOD Receive;
TELL METHOD ExecuteJob;
ASK METHOD AssignID(IN i : INTEGER; IN power : REAL);
ASK METHOD RemoveJobs;
ASK METHOD ObjTerminate;
1
RandomObj
ASK METHOD SetSeed(IN seed: INTEGER);
ASK METHOD Exponential(IN mean : REAL) : REAL;
4
TriggerObj
ASK METHOD Receive;
ASK METHOD Fire;
4
QueueList
numberIn : INTEGER
ASK METHOD Add(IN job : JobType);
ASK METHOD First : JobType;
ASK METHOD Remove : JobType;

Figure 4.6 Full Structure of the NodeObj

Each NodeObj instance has four queue structures and associated TriggerObj's. The queues are used to store jobs as they pass from one state to another between generation and final execution. Their specific use is as follows:

- lpQ - used to queue jobs that are eligible for possible transfer.
- rxQ - used as a buffer for jobs that have been transferred from another node.
- jobQ - used to queue jobs that have been allocated for execution at a node.
- txQ - only used by ProcessRandom, as a buffer for jobs that are to be transferred.

The associated TriggerObj of the same name and suffix sig has its Release method activated when a job is added to a queue.

To aid the understanding of the main methods used in the NodeObj and to complement the forthcoming description Figure 4.7 shows a schematic of their interaction.

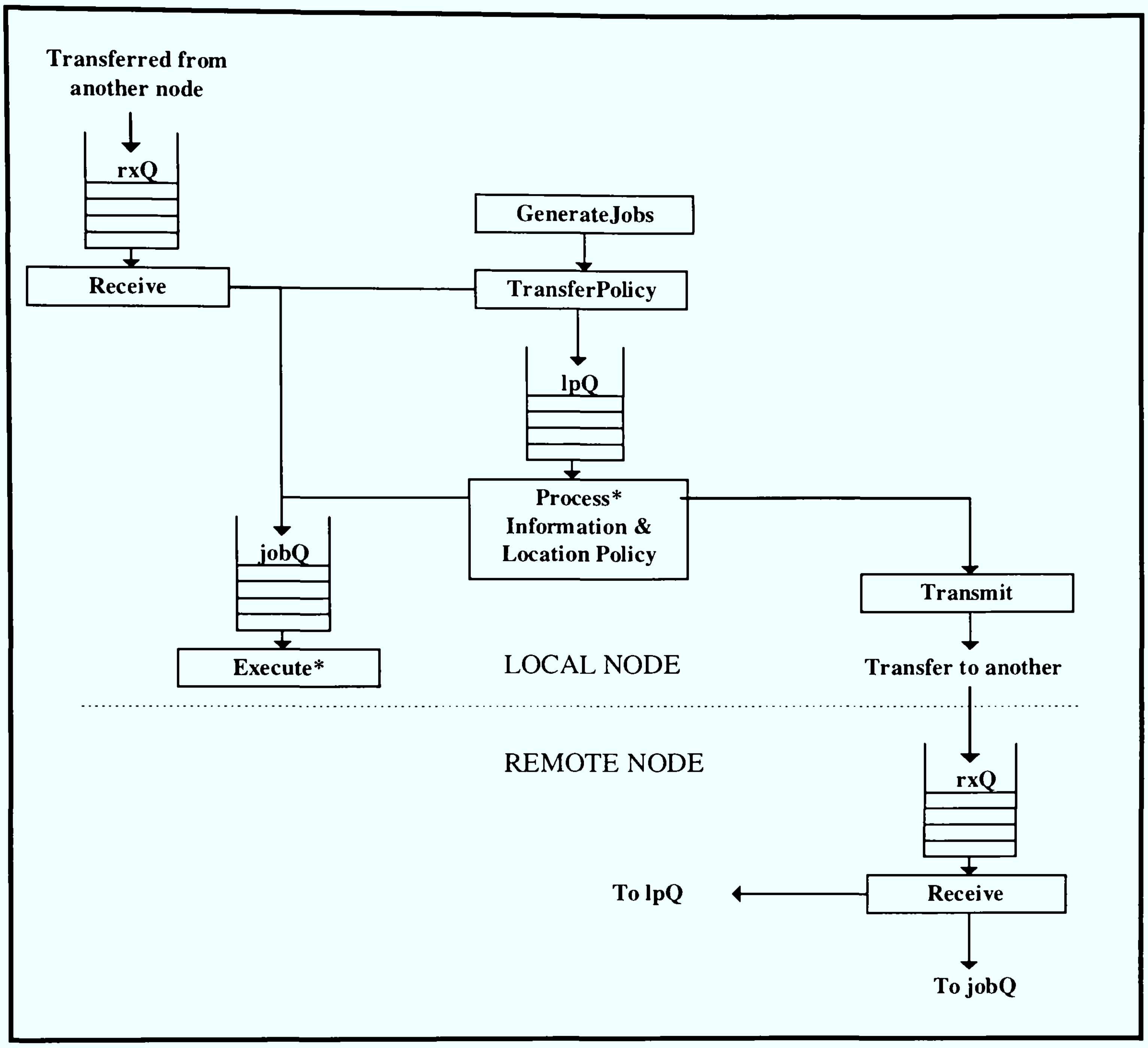

Figure 4.7 Schematic of Method Interaction

4.3.4 NodeObj Method : GenerateJobs

The main function of GenerateJobs is to provide a stream of JobTypes, representing the locally generated load. This method will continue for the length of the simulation. The mean interarrival time is calculated from the required utilisation and the power of the node. Actual interarrival times are assumed to be exponentially distributed. Job service times are also assumed to be exponentially distributed. The initial mean service time is the same for all nodes regardless of power. However actual service time for a job may change if it transferred to another node for execution.

As stated in chapter 3 this study will not investigate the possibility of using a transfer limit of greater than one. Thus any jobs transferred must be executed at their first destination node. With this in mind the transfer policy of the load sharing algorithm is only applied as jobs are generated in the system. This saves simulation overhead in two way, firstly transfer policy is performed in a minimum of instructions and secondly after any transfer no overhead is incurred in checking whether transfer policy should be applied again.

The load at a node is effectively the size of the ready to run queue (JobQ). This queue also contains any currently executing job, which will not be removed until it's execution has completed. A newly arrived job is considered eligible for transfer if accepting it for execution would cause the current load to exceed a threshold level. The value of the threshold will vary according to the algorithm in question. To curtail unnecessary overhead the number in the JobQ is compared directly with the set threshold and so the new job can be considered eligible for transfer if the threshold is merely equalled. If this is the case the new job is added to the queue of jobs for which the information and location policies of the load sharing algorithm will be performed. A software interrupt in the form of a TriggerObj (lpsig) Release method is used to signal this fact to the relevant Process* method.

Should the job be accepted for processing at its initial point of entry to the system it is added to the JobQ discussed earlier, but only after its true service time with relation to the power of the node has been calculated and substituted for the original servicetime. The Release method of another TriggerObj (sig) is used in alerting the Execute method of the node that a new job has arrived in the JobQ.

4.3.5 NodeObj Methods : Process*

Originally the process methods were designed to fulfil the full load sharing component of the model. But as was explained in the previous section the transfer policy has been moved for the sake of economy. However the remainder of load sharing activities are accomplished through these methods. There are five process methods as each is the equivalent of a different algorithm. Algorithm and process method are linked by the suffix to the process keyword, e.g., ProcessRandom implements the Random algorithm. The actual method used is selected at the outset of the simulation. Only one method is used for the whole of any run.

All the different process methods have the same basic structure. An endless loop, that is either executing load sharing policies or waiting for a TriggerObj (lpsig) to 'Release', indicating that a new job eligible for possible transfer has arrived at the node. Jobs eligible for transfer are taken from the lpQ on a FIFO basis and are processed sequentially. If the queue was dispensed with and the Process methods called directly from the GenerateJobs methods then concurrent execution of process methods could arise. The result of which would be that the full delay due to load sharing would not be accounted for.

The simplest of these methods is ProcessRandom. As no information policy is used in a Random algorithm the only property required is the ability to select a node at random. This is accomplished via a RandomObj (globalrandom) and provided the randomly picked destination node is not the same as the sender the job is sent to it. The transfer of the job starts with the transmit method described in the next section. The time taken in randomly selecting a destination node is considered negligible. For this reason no simulation time is passed in this method.

ProcessShortest involves many of the activities at the core of all the other Process methods. Firstly a sequence of randomly generated possible destinations is needed, the total number is dependent upon probe limit. To generate these a procedure called UniqueRandom is used. All possible destinations are unique and stored in an array. One is used in each repetition of a loop that carries out location policy. The maximum number of repetitions is defined by the probe limit. The gathering of system information imposes overhead on both nodes involved as well as the delay to the eligible job of RPC activity. The effect of these overheads is to delay the execution of any jobs currently executing on the respective systems. These delays are effected by interrupting the ExecuteJobs method and causing the 'ON INTERRUPT' statements to be executed. If a job is currently executing it is delayed by extending its servicetime.

Only the best results in the form of current lightest discovered load are stored (minload), together with the node involved (mindest). Where lightest load is a combination of load and nodepower as it is in all the heterogeneous algorithms this is calculated as the information is gathered and stored in 'minload'. The Shortest algorithm allows immediate transfer to any node that is discovered to be idle. So its Process method checks at the end of each information policy loop to see if an idle node has been probed and transfers the job if this is the case. Should idle transfer not be allowed as with HQNIT, all probing loops must be executed before a job could be transferred.

After probing has completed and assuming the job has not been transferred the minload value is compared to a metric level. This may take the form of a fixed threshold (Shortest, HETRO) or the length of local jobQ (HETQ, HQNIT). If minload is the lesser of the two values the job is transferred to the destination stored in mindest. This will involve the use of the communications methods. Otherwise it is added to the local jobQ for local execution, a TriggerObj (sig) is used to signal the event.

4.3.6 NodeObj Methods To Achieve Inter-node Communication

Three methods are used in the process of transferring jobs between nodes : Transmit, Receive and ReceiveJob. Transmit is only essential when simulating the Random algorithm.

The ProcessRandom passes no simulation time and so if no mechanism were used to queue jobs, they could be transmitted concurrently and the full cost of transmission would not be effected. The Transmit method takes jobs from the transmit queue and forwards them to the node specified in the jobs destination field. Delay to the nodes involved is achieved in the same manner as when probing, by interrupting the ExecuteJob method of the communicating nodes. Finally some time is passed in the method itself, the delay to the queue of jobs waiting for transmission. For other Process methods there is no possibility of jobs competing for transmission facilities as they are spaced far enough apart by the execution of their Location policies. This means it is safe to place the code contained in Transmit inside the Process method after dispensing with the procedures to manipulate txQ.

ReceiveJob is used as a form of buffer to process incoming jobs, as they could arrive from many different sources at the same time. ReceiveJobs puts them all in a queue for Receive to actually execute the delay to the transmitted job. This mechanism ensures that each job experiences the correct transmission delay.

4.3.7 NodeObj Methods : ExecuteJob*

The only function of the CPU in the simulation model is to execute the jobs found in the jobQ. This is handled by the ExecuteJob* methods. Two Executejob methods were implemented to cover both of the general job scheduling strategies, run to completion and pre-emptive scheduling [Tan87]. Run to completion will be referred to as sequential execution and is implemented by ExecuteJob. Pre-emptive scheduling is more commonly known as multi-programmed operation and is implemented by ExecuteJobMulti. The one prevalent in the workstation environment is multiprogramming, in which various schemes for scheduling the workload have been proposed [Bac86]. Even the most simple, round robin, involves a very high overhead when attempting to simulate it. Round robin scheduling is used in ExecuteJobMulti. The more sophisticated scheduling algorithms suggested by Bach and actually inplemented on Sun workstations are based upon priority schemes. These are not viable to implement via simulation due to their complexity.

The structure of the ExecuteJob method is a familiar one, an infinite loop processing the contents of a queue or waiting for a signal that another job has arrived in the queue. Processing a job merely involves executing a WAIT DURATION loop for the time specified by a job's servicetime. Should the method be interrupted whilst in the WAIT loop then the jobs unexpired servicetime is calculated and the loop started again. Once a job's servicetime has expired statistics are updated and any memory allocated to the job record is de-allocated. Contrasting with the operations involved in ExecuteMultiJob shows why the latter has such high associated overhead. Using a quantum of 100ms [Bac86] would involve a job of average servicetime looping through one hundred times, before any consideration of possible interrupts.

This high overhead begs the question of whether the simulation of multiprogrammed scheduling is necessary for the purposes of this investigation. Kleinrock [Kle76] shows that although multiprogrammed scheduling gives fairer treatment to individual jobs no advantage is gained in overall average response time. Table 4.2 shows a comparison of results from simulations using both ExecuteJob and ExecuteJobMulti methods. The systems investigated are homogeneous in nature, and the SHORTEST algorithm is used.

	Average Response Time	
Utilisation & Threshold	ExecuteJob	ExecuteJobMulti
70%, TH1	14.55	13.96
90%, TH2	23.81	23.45

Table 4.2 Run to Completion and Pre-emptive Scheduling Response Times.

The question of whether average queue size is affected by the scheduling method is addressed by Little's result, in which it is seen that queue size is related solely to average arrival rate and response time.

All simulations using the ExecuteJobMulti method took at least ten times as long as their sequentially scheduled counterparts. As no significant difference was observed between the two, the quicker version (ExecuteJob) was used in the simulations, for which results are presented in Chapter 6.

4.3.8 StopAllObj

The third and last object constructed and used in the simulation model is the StopAllObj. Not unsurprisingly this object is called to stop the active part of the model when simulation time has expired. This is achieved through the use of StopAllObj's only method, Finish. To allow the orderly disposal of the memory allocated for each NodeObj some administration must be performed before the StopSimulation command is issued. This involves forcing some of the methods in continuous loops to exit them, thereby guaranteeing that all the TriggerObj's used can be disposed of.

4.4 Validation and Verification of the Simulation Model

One of the most vital processes involved in the development of any model is to ensure that it is a significantly accurate representation of the system it represents. For only when this has been shown can the results provided by the model be held considered credible. Verification and validation are the means by which a satisfactory level of credibility can be established.

4.4.1 Verification

The verification of a model is the process of checking that the model has been built right [Ban96]. From the design stage a conceptual model of the system will have been developed. The design of this conceptual model and any assumption made must be reflected in the final implementation. The validity of any assumptions made is not questioned in the verification process, but left to validation.

Verification can also be thought of as debugging [Jai91]. This idea is particularly relevant to the simulation model described in this chapter as various software engineering debugging techniques were used in the verification process. The methods used and the subsequent results are detailed below. The combination of the results gained led to the conclusion that the model was suitably verified.

- A flow diagram was drawn, Figure 4.7. This showed each logical stage in the operation of the system model after initialisation. The methods of the NodeObj object were then constructed to fulfil the operations outlined in the diagram
- The code was at all times well commented enabling others who were not involved in its construction to be able to check its logical flow and ability to perform the functions desired. This method of verification was enhanced by the assistance of experts in the area of computer simulation and distributed systems who read through the code during model development. Their questions would often reveal any discrepancies between the conceptual model and that implemented.
- Simplified runs of the model were performed, allowing implementation details to be checked on a step by step basis. Print statements made it possible to see the changes that occurred to model variables with each occurrence of an event.
- A wide variety of input parameters were used to test the reasonableness of the model. These included, runtime, interarrival time, job servicetime, power of nodes and number of nodes. Small changes in input parameters had little effect on the end result, whereas large changes did have a noticeable effect.
- During and at the conclusion of each simulation run a number of statistics were gathered in addition to those of primary interest. These ancillary results were used to assert the reasonableness of the model by checking for consistency across a set of values, i.e. nodes of the same power.
- Each simulation run was executed a number of times with different seeds for any random number generators, to ensure that the results were independent of the seed used.

4.4.2 Validation

The validation of a model consists of comparing its behaviour to that of a real system. The aim of which is to ensure that the model if structured correctly and based upon valid assumptions should accurately represent the system it is modelling.

A three step approach to validation was developed by Naylor and Finger [Nay67]. This has been widely accepted as a suitable general technique, [Jai91, Ban96] and is used in validating the load sharing model. The three steps involved are:

1. Build a model with high face validity.
2. Ensure all assumptions made are reasonable.
3. Validate input output transformations.

The first step can also be referred to as utilising expert intuition. For the model should appear at face value to be reasonable to experts in the field in which it is to be used. The same experts should also look at the model output and check for reasonableness. During the development of the load sharing it was periodically examined by people knowledgeable in both the fields of distributed systems and communications. With the conclusion that the model appeared to be a accurate representation of the subject system. A further check on face validity is to use sensitivity analysis. Where the model should behave in the expected way if input values change. The impact of differing input parameters was judged to follow the accepted norm in the cases where previous experience could be called upon.

The assumptions made in the construction of a simulation model fall into two general categories. Structural assumptions are those concerning simplifications or abstractions of how the real system actually operates. An example in this study would be the assumption that the time to execute an RPC could be fixed for all nodes. The validation for these assumptions was contained in the arguments of Chapter 3 and design analysis earlier in this chapter. The second category of model assumptions are those about the data used, in both the constitution of the model and input parameters, i.e., number of nodes, initial loading, system utilisation and job servicetime. Validation of data assumptions is difficult in the load sharing case as working implementations are rare. So where possible earlier research in the field has been used in formulating parameters. This is combined with using a wide set of input values and model scenarios to negate the effect of any bias due to a lack of hard data.

The validation of input output transformations can be regarded as the truest test of a model. For on completion it would prove that the model could provide accurate predictions of the operation of the system it simulates. Ideally the conditions simulated will be readily encountered in the real world to provide results for comparison. If this is not to be the case historical data sets can be used in the validation process. Unfortunately these forms of direct validation are limited by a lack of load sharing implementations. However some alternatives are available and these have been used in the validation process.

Queuing theory can be used in determining response times for M/M/1 systems. A close correlation was observed between these results and those derived from the model with no load sharing implemented. As this was constant with a variety of input parameters it could be used to validate some of the core model functions, such as load generation and execution. Also available for comparison were the results in the literature. In many homogeneous systems the results derived by other researchers had been generally accepted as true. The model with a change of input parameters could duplicate this earlier work. By reaching the same conclusions as in the reported work the load sharing capabilities of the model were proved. To validate the heterogeneous aspects of the model was more difficult, prompting the implementation described in the next chapter. While the implementation itself is still a type of model the consistency of results, provides validation of both approaches.

4.4.3 Calibration

When dealing with the verification and validation of a simulation model, the subject would be incomplete if some attention were not paid to the process of arriving at the general simulation parameters. Calibration as this process is known [Ban96] will run in tandem with validation. Calibration involves refining a model's general simulation parameters, with the aim that the model's results will reflect the steady state performance of the system simulated. The parameters investigated in the calibration period were:

- Run length
- Initialisation period
- Number of repetitions

The first task in deciding upon the run length of a simulation is to determine the form of the output data. There are two possible types, discrete and continuous time data [Ban96]. The former occurs when output data comes in the form {Y1, Y2 ....Yn,} an example of which would be the response time of jobs. Whereas the latter's output data comes in the form $\{Y(t), 0 \leq t \leq T_E\}$ an example of which is average queue length for or at a resource. The period of a simulation run in which discrete time data is collected would normally be determined by a set number of intervals i.e., total number of jobs processed in a system. Continuous time data is best collected over a set period of time.

Although the primary objective of the simulation model is to determine the average job response time in systems the run lengths are determined by a fixed time period. This is because is not practical to limit simulations to a set number of jobs. Calculating when the finishing point occurred with many job generating sources would involve considerable extra overhead. Instead the run length is determined by a set time period. The time period is sufficiently long enough to ensure that the number of discrete events between simulations varies by only a very small proportion.

For the purposes of determining a sufficient run length sample runs were analysed using their ensemble averages. Ensemble averages are obtained by splitting each run into a set of equal periods known as batches, after a number of replications the mean of each batch is calculated, the result is the ensemble average for that batch. Each replication uses a different seed so that each batch and associate ensemble average will be independent. This negates the correlation between batches in the same run.

The ensemble averages were plotted against the upper and lower 95% confidence levels. This enabled various factors to be investigated. Firstly no substantial initialisation bias was observed. At batch intervals of 5000 seconds, even at low levels of system utilisation steady state performance was soon reached. Therefore an initialisation period of 5,000 seconds was considered adequate to bring the system to a steady state.

The number of repetitions used was five and this felt appropriate for all subsequent runs. The total length of the run was guided by the recommendation that a suitable number of batches was between 10 and 30 [Ban96]. The number selected was 12, giving a total simulation time of 60,000 seconds of which the first 5,000 were disregarded. At low levels of system utilisation this run length would allow tens of thousands of jobs to be processed in the standard system of 20 nodes.

5. Implementing the Load Sharing Scenario

5.1 Introduction

The implementation of a load sharing scenario was undertaken as a means of validating and verifying the simulation studies performed. This is needed both for the assumptions made in model construction and the results obtained. The design of the implementation follows the general structure of the model described in Chapter 4. However some deviation was unavoidable as a UNIX workstation is not as flexible as a simulation language. Where this has taken place will be highlighted in the forthcoming chapter.

The code used on all the workstations comprising the implemented system is identical. The only aspect where a case for variation exists is in the power rating of each machine, which varies according to individual processing power. However the duplication of code could lead to the introduction of errors and so machine type is determined at start-up and a hard coded value for power rating used according to the result. Originally processing power is determined by executing the same simple loop on each class of machine. After many thousands of iterations time is measured and the power rating set accordingly. This value is used in the information policy of the load sharing algorithms investigated as well as the generation of the offered load at each node. The mean interarrival rate is inversely proportional to the power rating. As in the simulation model this ensures that all machines have an equal original utilisation.

The system for which the implementation code is designed is a network of Sun workstations. All of the workstations use the SunOS 5.x operating system [Sun92], based on the System V Release 4 (SVR4) UNIX operating system. All the code used is written in the 'C' programming language [Ker84]. The code is non-obtrusive in operation and as such requires no rebuilding of the kernel or other operations requiring super-user permissions. A fully commented listing of all the code used is available in Appendix 2.

This chapter will give an overview of the operation of, and interaction between, the processes constituting the implementation. In particular attention will be focused on areas that forced deviation from the simulation design or are integral to the operation of the system.

5.2 Overview of Implementation Code

The implementation code is organised into seven distinct files. These take the form of header file (hetro.h) and six separate programs. The header file contains all constant declarations relating to the implementation, thereby allowing changes to be made quickly and in a consistent manner. There are also a number of function definitions contained in the header file.

All six of the programs listed below are used to generate a different process.

- generatejobs.c ➔ *generatejobs*
- processjobs.c ➔ *processjobs*
- executejob.c ➔ *executejob*
- serveprobe.c ➔ *serveprobe*
- remxclient.c ➔ *remxclient*
- remxserver.c ➔ *remxserver*

Where possible the process generated has the same name as its equivalent method in the simulation model object, NodeObj. All inter-node communication is carried out with the use of the Remote Procedure Call (RPC). This is the primary function of the latter three programs.

The load sharing scenario is started by one process *generatejobs,* an instance of which must be invoked on each of the workstations involved. This process will spawn *serveprobe* and *remxserver* immediately and *processjobs* after a brief initialisation sequence. The only purpose of *generatejobs* after its initial stages is to provide a stream of "jobs", representing the offered workload to a node, for *processjobs* to deal with..

The processes *serveprobe* and *remxserver* are RPC servers, used to handle incoming RPC's from prospective clients and will run continuously. Also running continuously are the processes *generatejobs* and *processjobs.*

The hub of all activity is the *processjobs* process and the only process it does not directly communicate with is *remxserver*. An illustration of the scope of its activities and the relationship between all six process types is provided in Figure 5.1. This diagram also shows the flow of information between processes situated both locally and remotely. Each node will possess a server and client side. Processes that can exist in concurrent instances are indicated by a multiple entity symbol.

As a fully commented listing of the implementation code is contained in Appendix 2 a line by line analysis of its composition will not be undertaken. Instead the operation and functionality of each program will be outlined. This is achieved by a textual summary followed by pseudo code version of the program concerned.

5.2.1 Generatejobs.c

The initial actions of this program, concerning the spawning of other processes, were covered in the previous section. However they are not the only tasks performed before the endless loop that deals solely with the generation of new work is entered. Between the creation of the two RPC servers and the *processjobs* process, two essential features are configured. A suitable seed must be derived and supplied to the random number generating function, followed by the creation and attaching of a shared memory segment. Shared memory segments need only be created once but must be individually attached by any process wishing to use them. After *processjobs* has been spawned, the program waits for a signal (SIGUSR2) that the new process is fully operating before proceeding to the actual generation of workload. This level of co-ordination is necessary to prevent the possibility of a signal being sent to a non-existent process and the subsequent termination of the sending process. When dealing with multiple processes the scheduling of execution is determined by the operating system and not the programmer.

The endless loop that generates workload, uses the same random number generator for both interarrival times and service times of jobs. Job interarrival times are based on the power of the node. Node power is determined from the node's machine name as supplied by the uname system call. The details of jobs are stored in records placed in the shared memory segment, where they can be retrieved by *processjobs*. Every time a new record is stored a signal is sent to *processjobs*, which is identified by its process identification number (pid), supplied to the parent (*generatejobs)* on creation of its child (*processjobs)*.

Two signal handlers are used in this process, to catch SIGUSR1 and SIGUSR2. Without signal handlers to catch signals and perform their dedicated action the default UNIX action will be carried out. In the case of most signals this is to terminate the receiving process. The first handler catches a signal (SIGUSR1) from *processjobs* to indicate that the load sharing scenario has reached the end of its run time. The actions taken on reception are to write job generation statistics to a file before terminating the *generatejobs* process. The second signal handler does not actually perform any new tasks. It's sole purpose is to catch the signal (SIGUSR2) from *processjobs.* This indicates that the process is successfully functioning and so it is safe for *generatejobs* to continue.

```
START
    Spawn and execute serveprobe - RPC server to answer incoming probes
    Spawn and execute remxserver - RPC server to execute incoming jobs
    Determine node type and assign node power
    Initialise random number generators, ts and ta parameters
    Generate unique seed
    Initialise random number generator
    Create, if not already in existence, and attach shared memory segment
    Spawn and execute processjobs - The process that initially handles all
    generated jobs
    LOOP
        generate interarrival time
        sleep for interarrival time
        assign job length and creation time
        place job record in shared memory segment
        send signal (SIGUSR1) to processjobs
    END LOOP
END

Signal (sigusr1) - catch SIGUSR1 from processjobs, write stats and exit
Signal (sigusr2) - catch SIGUSR2 from processjobs
```

Figure 5.2 GenerateJobs.c (Psuedo Code)

5.2.2 Processjobs.c

As with *generatejobs* the bulk of the activity carried out by *processjobs* takes place in an endless loop. However some initialisation procedures must be carried before the process reaches this stage. The majority are concerned with setting up a shared memory segment through which job details are obtained from *generatejobs*. Others include enabling an alarm to provide a means of periodic reporting and sending a signal (SIGUSR2) to the parent process.

```
START
    Initialise linked lists for job details
    Initialise and attach shared memory segment with same id as that used by
    generatejobs
    Determine node type and assign node power
    Set first report period
    Send signal to wake-up generatejobs
    LOOP
        IF shared memory segment is empty
            pause waiting for signal from generatejobs
        ELSE
            IF local load > threshold value
                initiaite HQNIT load sharing policies via lsalg()
            END IF
            IF suitable destination node is discovered
                spawn and execute remxclient
            ELSE
                spawn and execute executejob
                increment local load
            END IF
            store job details in link list
        ENF IF
    END LOOP
END
```

Signal (sigusr1)	Catch signal from generatejobs indicating a new job has been generated.
Signal (sigcld)	Catch death of child signal indicating a child process has terminated. The process will be from executejob (locally executed job) or remxclient (remotely executed job). Determine exit status of child process, current time and ID. Store these details in a link list
Signal (sigalrm)	Timer alarm, write report period stats to file and reset alarm for another period. If run time has expired send signal to generatejobs and write contents of link lists to permenant storage.
lsalg()	Randomly pick nodes for probing. Probe via RPC mechanism and implement HQNIT location policy on results returned. Repeat until probe limit is reached. Return result to main program indicating whether a suitable node has been discovered.

Figure 5.3 Processjobs.c (Pseudo Code)

The main while loop functions in the same manner as the loops in the simulation model object, NodeObj. If no jobs are present in the shared memory block the loop pauses, waiting for a signal to indicate a new arrival. Continuous checking of a memory location would incur substantial overhead and so is not practical. A form of software interrupt must be used. Unlike the simulation model all three load sharing policies are performed in *processjobs*. Although transfer policy could be integrated into generatejobs no advantage is gained in terms of efficiency. In fact circumventing processjobs in any way leads to problems in calculating final response times.

On the identification of an eligible job a function is called that performs both the information and location policy. In the case of the code presented the HQNIT algorithm is implemented. The load sharing function randomly picks nodes to which RPC's are made. The results returned are used to select a destination node in the same manner as in the simulation model. Control then returns to the main body of the program.

If a suitable node is selected a *remxclient* process is spawned. This process is used to start execution of the job on the destination node selected. Should no suitable node have been found or the job not have been deemed eligible for transfer, it is executed locally. This requires the spawning of an *executejob* process. In this case the local load must be incremented. Wherever the job is to be executed a record of its current state is added to a link list. This list will contain records of all jobs originating at the respective node in their pre-execution state.

Three signals are caught by this process. The first (SIGUSR1) used as an interrupt to signal a newly generated job. The second (SIGCLD) is sent by any terminating child process, in this case an instance of *executejobs* or *remxclient,* to its parent. Upon receiving this signal the exit status, which indicates the execution history of the terminating process, is determined. There are four possible exit modes:

- Successful local execution.
- Successful remote execution.
- Unsuccessful remote execution due to time-out.
- Unsuccessful remote execution as destination server unreachable

The exit status, completion time and id of the terminating process are stored in a link list of job state records post-execution.

The final signal handler declared deals with incoming signals (SIGALRM) indicating that a report period has expired. A brief summary of node status and history is printed to a file. Information of this type is essential for program development and validation. When the total running time has expired, a signal (SIGUSR1) is sent to *generatejobs* and the contents of both link lists written to file.

5.2.3 ExecuteJob.c

The only purpose of *executejob* is to provide a workload to the CPU of the machine on which it executes. This is accomplished by executing a simple loop of arithmetic operations. The number of repetitions of this loop is proportional to the original length of the job to be executed. A new instance of *executejobs* is created for each job allocated to the node for execution.

```
START
   Attach and initialise shared memory segment with same id as generatejobs
   Calculate number of loops in proportion to job size
   REPEAT
        perform simple arithmetic tasks
   UNTIL repetiitions completed
   Decrement local load
   Exit
END
```

Figure 5.4 Executejob.c (Pseudo Code)

The shared memory segment used for communicating job details between *generatejobs* and *processjobs* is also used by *executejob*, but for a different purpose. The record of local load is stored in this segment and must be accessed by *executejob* on completion of the loop sequence. At this point the local load is decremented by one and then the process terminates. Where the job originated is of no consequence and the exit status is always the same. The possibility of a job failing to execute fully is not considered as it is not considered a possibility in the simulation model.

Multiple instances of *executejob* may exist on the same node. Their sequence of execution is controlled by the scheduling policies of the Solaris 2 operating system [Sun90a]. These are multiprogrammed in nature.

5.2.4 ServeProbe.c

The philosophy of the RPC mechanism is based upon hiding the details of all network code from the programmer with the use of stub procedures [Ste90, Blo92]. This allows the actual server procedures to be written in the same manner as a local procedure call.

The RPC code that services an incoming probe must connect to the same shared memory segment used by the other procedures on the local node. In order to return the correct load value. As server procedures should never exit unless explicitly killed, the shared memory segment only need be attached on the first call. The power rating is determined by the nodes machine name and this too need only be performed on the initial call

A data structure is required to return the parameters as only a single pointer can be passed back from the routine. No arguments are required by the remote procedure for its operation but a dummy one is passed to satisfy the demands of the Sun RPC implementation.

```
START
   IF called for the first time
      Attach and initialise shared memory segment with same id as generatejobs
      Determine machine name and assign node power
   END IF
   Put nodepower and load into the data structure specified in RPC definition
   Return data structure to calling RPC
END
```

Figure 5.5 Serveprobe.c (Pseudo Code)

5.2.5 Remxclient.c

All of the actions performed by *remxclient* are concerned with starting an RPC to execute a job remotely. If *processjobs* itself tried to directly start the remote procedure it would pause to wait for the reply. Hardly ideal in a multiprogrammed environment especially when it is the hub of so much activity. By spawning a *remxclient* process to handle the remote execution it is free to continue and attend to other tasks. Multiple occurrences of *remxclient* must be possible as at a busy node many jobs could be executing remotely.

Another advantage of *processjobs* spawning a *remxclient* process is that the SIGCLD signal can be used to interrupt *processjobs* on the completion of a job. This allows the same signal handler to deal with all types of terminating jobs no matter where they were executed. The exit status returned by *remxclient* has three possible values as there are more possible outcomes with the complication of network communication.The three possibilities are successful execution of the job, unsuccessful execution and timeout. Although this feature may appear redundant with the comments presented in section 5.3.4, it has been retained to make the implementation scenario more flexible.

```
START
    Call remxserver procedure on remote machine
    Pass servicetime of job to remote server
    Exit with exitsatatus set according to result
END
```

Figure 5.6 Remxclient.c (Pseudo Code)

5.2.6 Remxserver.c

The second RPC server procedure *remxserver,* deals with calls from *remxclient.* The only parameter passed is the servicetime of the job to be executed. Which is then passed to the *executejob* process that is spawned. As the remote execution of a job will add to the load of the remote node selected its load count must be incremented. This is incorporated into the *remxserver* code. The actual incrementation takes place as soon as possible so that any load sharing decisions made in the near future are as accurate as possible. This can be before the job has started to be processed as no node is allowed to reject a job and in all the algorithms evaluated the transfer limit is one.

```
START
    IF called for first time
        Attach and initialise shared memoty segment with same id as generatejobs
    END IF
    Increment local load
    Spawn executejobs
    Wait for executejobs to finish
    Return control back to calling remxclient process
END
```

Figure 5.7 Remxserver.c (Pseudo Code)

Unlike the *serveprobe* process which is iterative in nature the *remxserver* must behave as a concurrent server. This is due to the far greater time *remxserver* takes to service requests. Without concurrency requests from calling procedures may be queued and so full multiprogrammed operation not achieved and more crucially the true load of a node will not be reflected in its load count. Enabling the server to execute in a concurrent fashion involved manipulation of the server stub code. The details are described in the later section on implementation specific RPC features.

5.3 Crucial Elements of the Implementation Code

5.3.1 Random Number Generation

In order for the results of the load sharing implementation to be of use in validating simulation results the offered load to each node must have exponentially distributed interarrival and service time distributions. An added complication in the code concerned is that it will run on many separate entities that will all need different streams of random numbers. In the simulation model this was not a problem as MODSIM offered a wide variety of statistical distributions for random number generation and as all nodes could be easily manipulated different generation patterns could be set at each.

The C language offers a selection of functions that return random numbers but none which can supply an exponentially distributed stream. The function used is drand48(), which returns a random double precision number between 0 and 1. A seed must be supplied to srand48(), which is used to provide an initialisation point for drand48().If a seed is not supplied all the random number generation will start from the default seed. As the numbers supplied are uniformly distributed between 0 and 1 the Inverse Transformation Method [Leu87] is used to produce an exponentially distributed stream for interarrival time and servicetime. The same random number generator is used for both.

$$\boxed{x = -\lambda * \ln [R]}$$

Figure 5.8 Inverse Transformation Method (x is a random variable of exponential distribution with mean λ and R is a random number between 0 and 1)

The problem of unique streams of numbers can only be solved by supplying unique seeds to srand(). Initially the current time in seconds was used but this proved unsatisfactory as it could give the same value for some nodes. One solution is to stagger the starting time for each node but this would lead to a longer initialisation period. So the time in seconds was combined with the ASCII value of the nodes network name, ensuring a unique seed on every occasion.

5.3.2 Inter Process Communication

In any system where two or more processes interact with each other some form of Inter Process Communication (IPC) must be established. This topic is the subject of lengthy chapters in any volume on UNIX based operating systems [Bac86, Tan87, Ste92]. Therefore only a brief note of the alternatives to the method used in the implementation will be made.

Three forms of IPC are commonly used between processes on the same host, collectively known as System V IPC [Bac86, Ste92].

1. Message queues - formatted message lists stored in the kernel.
2. Semaphores - used to share single resources between multiple processes.
3. Shared memory - a region of memory which multiple processes can access.

The method used in the implementation is shared memory as it is the fastest of the three and minimising overhead is of prime importance. A process creates a shared memory segment by using the shmget() system call, this will return a unique identifier. If the segment already exists the identifier can still be retrieved by supplying different permissions. Once this identifier is known the shared memory segment can be attached to the processes address space. Once a shared memory segment is attached it is always referenced by its starting address. Shared memory segments are inherited by children i.e. after fork(), but are not shared after an exec(). So although most of the processes used in the implementation are initially created by a fork() call from a process already attached to the shared memory segment this connection is lost as they are the direct result of an exec() system call. The fork() call merely creates an identical copy of a process, while the exec() replaces the copy with a new program from disk. Hence the initialisation and attachment of the shared memory segment forms the start of many of the processes used in the implementation, as they have lost any connection originally held.

The greatest use of the shared memory segment is made by *generatejobs* and *processjobs,* in communicating the details of the offered load. Problems in synchronising access to the shared memory segment are avoided by only allowing one process the ability to update the contents. A signal is sent on completion of this task and only then will details be read.

The other use of the shared memory segment is to store the local load state, for which the first two bytes are used. This data is accessed by four of the six process types, *processsjobs*, *serveprobe*, *remxserver* and *executejob*. Here it is impossible to synchronise access without the use of a control mechanism. The simplest method of control is record locking [Ste92]. A file is created, the first byte of which can be locked. The byte can then act as a quasi semaphore by restricting access to the shared memory segment to the process that locked it. An exclusive or shared lock can be used. The choice depends upon whether the intended operation was a read or write. The lock must be blocking in case the resource is in use.

This form of resource control was found to be very susceptible to the problem of deadlock [Tan87]. Further observation showed that the slight inaccuracy in load which resulted from uncontrolled access to the load value would only exist for a limited time and the effect on the full scenario would be negligible. The added complication of implementing a true semaphore scheme could not be justified and so access to the local load was not constrained in any way.

5.3.3 The Process Lifecycle

The operation of the load sharing scenario calls for various processes to be created, perform a function and then exit. For *executejob, remxclient and remxserver* this process will happen thousands of times. To enable the performance of each node to be calculated a reliable check must be kept on each job from its time of arrival in the system to eventual execution and the notification of the result to the originating node. This operation is performed by taking advantage of the manner a processes lifecycle is handled in the UNIX operating system.

The only way to create a new process is through the use of the fork() function. Which creates an identical copy of the calling process. The only difference is that the call returns 0 to the child and the pid of the child to the parent. This information is essential in keeping track of the progress of the job. After a fork() the parent process can wait for the child to terminate or continue its own execution. Whichever course of action is chosen the child will send a signal (SIGCLD) on its death. This must be caught by the parent otherwise the child will become a zombie process.

To invoke another program once a child has been spawned the exec() function is used. There are 6 different varieties of exec() [Ste92]. In the implementation execlp() is

Figure 5.9 shows the concepts mentioned above in the context of the load sharing scenario and in particular the collating of job response time. All processes created after initialisation of the implementation are a result of the action of *processjobs*. If a job is to be executed locally a new process is forked and the *executejob* program started. When the pid of the new process is returned to *processjobs* it is stored with all the job details in memory. On completion *executejobs* will terminate and the SIGCLD signal will be caught by *processjobs*. The signal handler that catches it will execute the wait() function to retrieve the pid of the completed job, which is stored with current time and exit status in another construct in memory. After the run has finished, the starting and finishing records for each job can be identified using the stored pid numbers and the response time calculated. Calculating response time dynamically was found to produce an unacceptable level of overhead, along with disasterous linked list problems if job execution time was short.

Should a job be selected for transfer then *remxclient* is executed after the forking of a new process. Several other processes are created but control will eventually return to *remxclient* and it will then terminate, causing a signal (SIGCLD) to be sent back to *processjobs* which is handled in the same way as if it had originated from *executejob* as in the case of a locally executed job.

One minor problem can occur with this scheme, due to the fact that the operating system used cannot support a signal queue of more than one. Should signals arrive from several terminating processes within a short duration of each other some could be lost and the relevant record not written to memory. Without both starting and finishing time it is impossible to calculate response time and the job is effectively lost. Tests showed that while this did occasionally happen less than 0.02% of signals were lost in this way, a small enough fraction of the total load to be considered negligible.

5.3.4 Implementation Specific RPC Features

During the development of the RPC's used in the implementation three areas of particular interest came to the fore. One concerning the actual construction of the RPC's used and the other two their susequent use.

The idea of a concurrent server was first mentioned in section 5.2.7 on the *remxserver* process. A normal RPC call will be iterative in nature, dealing with a request from one client in full before processing the next request. This situation is illustrated by

As *remxserver* may have to deal with requests taking some time to service, and occurring simultaneously from many clients, it must act in a concurrent / multi tasking manner. Otherwise not only will it not be acting in a multiprogrammed fashion but more importantly the true load at a node can not be ascertained, as many jobs may be queuing for service at a remote server. These jobs cannot be taken into account in any load sharing policy and so will lead to the sub optimal operation of any algorithm used.

Simple forking of the *remxserver* process when it is called is not a possible solution due to the nature of RPC's [Blo92]. RPC's have all networking details hidden from the user in blocks of code known as stubs [Ste90] The stub receives the client request as a network message, decodes the arguments from the network message and invokes the server function. Without considering the relevant network communication factors involved in an RPC no solution can be found. In the case of SUN RPC's this code is generated from a simple interface declaration using the rpcgen compiler [Sun90]. The use of rpcgen considerably simplifies the task of RPC programming over writing all the code by hand. However it is the server stub code that must be altered to achieve concurrency.

The server stub code is altered after the point at which the RPC environment has been established. Of particular note is that the parameters required for returning a reply to the client must have been collected. Then just as the service routine is about to be invoked an identical process is spawned which will invoke and execute the required service. When finished the child will exit() and return control to the calling client. Meanwhile after a successful fork() the parent will return to wait for more incoming calls, never executing any services itself and so never reaching the exit statement. This operation is illustrated in Figure 5.10b. The original serverstub has spawned three children each of which has invoked the server routine with the parameters passed by the respective client. These routines can now execute concurrently and the server is still available for anymore incoming requests.

The second area of interest is in the transport protocol to be used. Sun RPC allows UDP or TCP to be used, providing connectionless or connect oriented communication [Hal92]. In early versions of the implementation RPC's used UDP as the transport protocol. UDP is packet based and has a relatively short time-out which cannot be changed, hence if the operation involved is time consuming i.e. remote execution then the request will be resent. Once only execution is essential for load sharing activities to be effective and could not be guaranteed. Therefore the TCP protocol was used as only this protocol could ensure that the remote procedure was executed at most once. Although TCP is slightly more expensive in terms of overhead than UDP, it is used for probing as well for the sake of consistency.

The last concern also relates to the use of the TCP transport protocol. All communications protocols have a time-out facility to prevent communication channels hanging open on remote failure. The subject of interest in this context is the length of the time-out. The default is 25 seconds and this could easily be reached when the system was under heavy load. Various attempts were made to set the time-out based on a combination of average service time and loading. These were thwarted by the large degree of heterogeneity present in some of the systems investigated. Which led to some huge differences in expected execution times. The only solution is to tailor time-outs to individual RPC's. As this would mean setting constantly changing time-out levels it is not a practical proposition.

However as the system appeared very robust and no jobs were failing either due to network error or process failure at the executing node, it was decided to set the time-out to the length of the scenario run time. thereby ensuring that no jobs would be lost due to the working of the transport protocol.

6. Experimental Results

6.1 Introduction

The results presented in this chapter are from two different sources, simulation and implementation. All conclusions regarding the performance of the algorithms are derived from the simulation results. The implementation studies were performed to validate the simulation results and the assumptions made in constructing the simulation model.

After the main aim of reducing system wide response times, the three most important properties desirable in any load sharing algorithm are adaptability, scalability and stability. All these properties are investigated, but to differing degrees depending on the algorithm concerned. Investigations into an algorithm were halted once it proved to have serious flaws or to be surpassed by another.

	Power	Fraction of total power		Power	Fraction of total power	skew	CV
A1	0.350	0.21	B1	1.975	0.79	0.206	0.634
A2	0.417	0.25	B2	1.875	0.75	0.149	0.510
A3	0.500	0.30	B3	1.750	0.7	0.094	0.375
A4	0.667	0.40	B4	1.500	0.6	0.028	0.167
A5	0.830	0.50	B5	1.250	0.5	0.004	0.042
A6	1.167	0.70	B6	0.750	0.3	-0.004	0.042
A7	1.330	0.80	B7	0.500	0.2	-0.028	0.167
A8	1.500	0.90	B8	0.250	0.1	-0.094	0.375
A9	1.583	0.95	B9	0.125	0.05	-0.149	0.510
A10	1.650	0.99	B10	0.025	0.01	-0.206	0.634

Table 6.1 System Composition With Nodes Divided 12:8

Initially all algorithms were evaluated under the same operating conditions across the systems described in Table 6.1. After the elimination of unsuitable algorithms, the remaining ones were subjected to further and more rigorous testing. In order to establish the most practical algorithm over a wide range of conditions, through investigation of the ability of each to be adaptable, scalable and stable was undertaken.

6.2 Comparison of Algorithms

6.2.1 Simulation Parameters

The full set of algorithms described in Chapter 3 were applied to all of the heterogeneous systems outlined in Table 6.1. The simulation parameters used are shown in Table 6.2. Any deviation from these parameters is due to the nature of the algorithm concerned and attention is drawn to the fact in discussion of the results.

Parameter	**Setting**
Run Length	60,000 seconds
Initialisation Period	5,000 seconds
Repetitions	5
Threshold	Variable 1 : 3
Probe Limit	Variable 1 : 10
System Size	20 nodes
System Heterogeneity	Coefficient of Variance 0.042 : 0.634 Skewness -0.206 : 0.206
System Utilisation	low (50%), medium (70%), High (90%)
Average Job Size	10 seconds duration
Average Interarrival Time	Inversely proportional to power of node

Table 6.2 Simulation Parameters in Algorithm Comparison

6.2.2 Bounds on Performance

The no load sharing or M/M/1 scenario is considered as it provides a useful upper bound on performance. The M/M/1 performance at any degree of system heterogeneity will always be the same if the offered load at a node is proportional to the power of that node. For example when considering system A1/B1 from Table 6.1 and assuming

balancing line. This is because the powerful nodes are starting to work at almost full capacity and so the less powerful ones must take a greater overall share of workload.

6.2.3 Algorithms Proposed Primarily for use in Homogeneous Systems

The RANDOM algorithm was introduced in Chapter 2. Earlier work described in the literature and discussed there has shown it to be a simple but effective means of load sharing in the homogeneous environment. In a heterogeneous system its inherent simplicity could prove to be an advantage in terms of adaptability and scalability. For example the problem of how much system information to gather as specified by size of probe limit, can be dispensed with.

The RANDOM algorithm does have one variable parameter, which is the transfer threshold. Results are presented for the three different values used, 1,2 and 3. As a high transfer threshold effectively restricts load sharing 3 was considered a suitable maximum. Figures 6.3a - 6.3c show the algorithms performance over the range of heterogeneous systems described in Table 6.1. Predicted performance in a system with no load sharing capability is indicated by the broken line. The heterogeneity of the systems is primarily rated by degree of skewness.

When applied to a homogeneous system this algorithm shows consistent improvement over the no load sharing situation. Whereas in the heterogeneous environment improvement is limited to a small subset of the systems considered. As the rate of system utilisation increases the sphere of effectiveness rapidly decreases, to the point at which only when the underlying system is almost homogeneous can any benefit be derived. For all the other systems considered the algorithm introduces a high level of instability.

The algorithm is most effective when a high threshold is used, limiting the scope for performance improvement to nodes that are heavily loaded. This indicates the lack of sensitivity inherent in a random location policy. In negatively skewed systems the more powerful nodes are in the majority and the opposite is true in positively skewed systems. Performance in positively skewed systems is markedly better than their negatively skewed counterparts. This is due to the smaller difference in processing power of the nodes making up the positively skewed systems, which lessens the effect of flawed location decisions, a consequence of the random location decisions made.

faster on the idle node. To address this problem the HQNIT algorithm was devised. All location decisions made by this algorithm account for the relative power of the nodes involved, even those involving an idle node. To accomplish this a notional extra job is added to all probed loads to represent the transferred job. An extra job is also added to the current load at the source node for purposes of the location decision. This has the added advantage of making the comparison even more realistic as response time at each node is calculated including the cost of the extra job.

In systems where utilisation is low the advantages of using the HQNIT algorithm are clear. Even at very low degrees of heterogeneity an improvement over the other algorithms can be seen. The difference in response times increases rapidly as the systems considered become more heterogeneous. The simplest explanation for these performance traits is that lowly utilised systems will have many idle nodes at any one time. Of these many will be low powered nodes which could be probed by high powered ones with excess load. Therefore the probability of a poor transfer decision due to immediate transfer to an idle node is high. HQNIT by not allowing this operation will have a performance advantage. An advantage that will increase with system heterogeneity as the penalty for poor transfer decisions also rises.

At a medium level of utilisation the HQNIT algorithm still produces the best performance but the margin of improvement over HETQL has reduced. This change can be attributed to fewer nodes entering an idle state as utilisation has increased and so the number of bad location decisions with idle transfer will fall. HETQL shows longer response times at negative heterogeneity as the difference in power of nodes is greater, hence the penalties for bad location decisions are greater. The opposite applies for HQNIT and so its response times improve.

The performance comparison at high utilisation indicates that there may be other factors attributing to algorithm performance other than accuracy of location decision and transfer threshold. If these were the only factors the difference between HQNIT and HETQL could reasonably be expected to decrease but not to change in the dramatic fashion illustrated. The problem of idle node transfer will have decreased with the increased utilisation of the system, reducing the differences in performance characteristics of both algorithms. The relative performance of HQNIT deteriorates in all circumstances except those of extreme negative skew.

An extra cost is incurred when using the HQNIT algorithm as it must always probe up to its probe limit, in the results presented in Figures 6.12a - 6.12c this was 5. At high utilisation the total number of job transfers for both HQNIT and HETQL is the same at approximately 50 %. If all of these were transfers to idle nodes, discovered in the case of the HETQL algorithm with the first investigative probe, the extra overhead to the HQNIT algorithm would be 4 probes per transferred job (120 ms to each probing node and 40 ms to each probed node). This is a highly improbable scenario at such a high level of utilisation but even so the total extra overhead would only be an average of 80 milliseconds per job. Less than 1% of average execution time. Therefore it seems unlikely that this could be the sole cause of the observed changes in performance

An answer may lie in examining the final loading statistics when using the IDEAL algorithm, Figures 6.2a - 6.2c. As this algorithm has no overhead and assumes each node has perfect knowledge of the system state, it should distribute load in the optimum fashion. At low utilisation the higher power nodes in each system are assigned a far higher proportion of the overall load than their collective power and vice versa for the low power nodes.

Increasing the system utilisation to a medium level sees the proportion of work done by each group draw closer to the load balancing line. This is the notional point where system load is balanced proportionally across all nodes. In a highly utilised system the node groups draw still closer to the load balancing scenario. Whilst the IDEAL case is unobtainable in current distributed systems the trend of approaching load balancing as system utilisation increases may be applicable.

The final columns in Table 6.3 show the proportion of load executed at each node group for a sample set of systems. At low utilisation HQNIT distributes the majority of load to the high power nodes. A similar if slightly less prevalent pattern can be observed at medium utilisation. In contrast the HETRO and HETQL algorithms distribute load far more proportionally and in more heterogeneous systems even favour the lower powered nodes. As discussed previously the HQNIT algorithm performs best in both cases reinforcing the conclusions drawn from the IDEAL scenario. However at higher system loading HETQL with its more 'balanced' loading patterns shows an improvement in comparison to HQNIT. Even HETRO with its higher transfer threshold vastly improves its relative performance.

As the explanation for the improvement cannot lie in superior location decisions another factor must exist. The implication of these results is that although the HQNIT algorithm has in theory the best job distribution mechanism, as it uses all the information available to establish the optimum site for execution, there is still a place for some semi-random distribution of jobs, as accomplished by the idea of immediate idle transfer. Semi-random in the sense that idle nodes must still be identified, but then can be assigned jobs even if they do not appear to be the optimum execution site.

For an example of this concept consider the system with heterogeneity of -0.094 operating at high utilisation. Analysis of simulation results for HQNIT showed that no jobs were ever transferred to the low power nodes. In addition almost half of their original workload was transferred to the more powerful nodes. As a consequence of which led to these nodes being severely under-utilised even though all transfers were made on the basis of finding the shortest time to finish execution. At no individual point did it seem sensible to transfer a job to a low powered node although the overall response time would have benefited from it. however when the HETQL algorithm was applied with its ability to transfer to idle regardless of other factors the low power nodes were utilised at a higher rate and overall response time reduced.

6.3 Further Investigations Into The Behaviour of Algorithms

6.3.1 Adaptability, Scalability and Stability

For any load sharing algorithm to be judged acceptable for use in the distributed systems environment it must be flexible enough to cope with the many varieties of system possible. Three of the most important properties in which this flexibility should be apparent are adaptability, scalability and stability. To establish whether these properties were supported further investigations were carried out into the performance of both HETQL and HQNIT algorithms.

Adaptability is the property of an algorithm to cope with the changes in the structure and operational conditions of the system on which it is to operate. With regard to heterogeneous environments the foremost of these is the degree of heterogeneity of the system. A wide selection were investigated in the previous section but they were all constructed from the same ratio of different nodes, i.e. 12 : 8. The effect of changing the ratio to 18 : 2 was investigated in order to further extend the investigation into algorithm adaptability. Loading patterns can also vary from system to system, or even on the same system with time, an algorithm's adaptability should be able to encompass these changes. The 12:8 split system was used in this investigation but instead of using a proportional loading scenario two new loading schemes were implemented. Algorithm's performance under each was evaluated.

Scalability implies that an algorithms performance is independent of system size. Larger systems based on the 12:8 split system were simulated to assess scalability. Of concern was not only overall algorithm performance but also the affect of varying probe limit with system size.

Stability is a general property of an algorithm that should be exhibited at all times. In effect it is tested with any change in the system parameters. All of the investigations in this section were performed at all three levels of utilisation (low, medium and high).

6.3.2 18:2 Split Systems

Primarily for the assessment of algorithm adaptability another ratio of system node groups was investigated. The main idea behind the change was to discover whether the ratio of low to high power nodes affected relative algorithm performance, in addition to the changes in heterogeneity that would be associated with the changing system composition. The split selected was 18 : 2. A choice made to provide a sharp contrast to the constitution of the 12:8 system, while not increasing overall system size. Changing two variables would make any results difficult to relate to those previously collected All the parameters described in Table 6.2 are still valid. The systems used are described in Table 6.4.

Unfortunately a system make-up of this nature limits the scope of investigation into negatively skewed systems. However the results that have been gathered show enough to recognise important trends. All of the runs reported when using this system split are from algorithms using a probe limit of 5. Simulations were run using a probe limit that varied from 1 to 10 and as with 12 : 8 split systems the optimum was around 5. In some cases a higher probe limit did achieve lower response times but never more than 5% less than when using 5 probes. Where the improvement did occur it was not across the whole range of heterogeneity studied. The performance of each algorithm is contrasted in Figure 6.13.

The consistency of results, reflected in performance trends and actual response time figures extending over the full range of system heterogeneity indicate that both algorithms can adapt to the changing structures observed. This fact shows the adaptability of both algorithms evaluated. Comparison to the earlier performance on 12:8 systems extends the sphere of adaptability. Correlation of results over both system splits, enforces the usefulness of system skew as a metric of heterogeneity. Finally algorithm stability has again been demonstrated over different operating conditions as at no point does behaviour become erratic or worse than the no load sharing case..

6.3.3 Varying the Offered Load

In the main study all of the nodes experienced the same original utilisation. This was ensured by making the interarrival time of jobs inversely proportional to the power of the nodes. However there is a possibility that load will not be so fairly distributed. A user given the choice of two machines on which to execute a given workload could reasonably be expected to choose the most powerful assuming everything else is equal. To examine the algorithms performance under different loading conditions two scenarios were developed. Both of these assume low power nodes are more likely to be lightly utilised in comparison to overall system utilisation and high power nodes proportionally more highly utilised.

In the first set of simulations 50% of the low powered nodes receive no load at all and the remaining 50% a load proportional to their power. The shortfall in total system load is divided between the high power nodes. A similar principle is used in the second evaluation but in this case no load at all is offered to the low power nodes. A full set of results for each loading pattern are presented in Figures 6.14a and 6.14b. In all cases the probe limit used was 5.

When using either of the new loading strategies the relative performance of both algorithms differs in comparison with that from earlier simulations when proportional loading was used. At low and medium levels of utilisation any change is limited to systems of a low degree of heterogeneity. In these systems the relative inequality of loading between high and low power nodes is the greatest, and the difference in processing power the smallest. Therefore any algorithm that transfers immediately on finding an idle node will have a slight advantage. This reduction is due to the reduction in location policy cost, obtained through the saving in number of probes that need be made,

for HETQL. The percentage of attempts that are successful has increased for HETQL and decreased for HQNIT, even so the HQNIT algorithm still maintains its performance advantage in the majority of systems. This is due in part to the relative harshness of the cost of a bad location decision affecting HETQL response times. The other advantage gained by not transferring to idle nodes is that the number of jobs executed on high power nodes increases. In the most negatively skewed systems all work is carried out on these nodes, if no load is offered to the low power ones.

The increased loading of high power nodes does have significant effect upon HQNIT performance at high utilisation, for at this point it is advantageous to share the load proportionally. In some systems the load at any individual high power node never reaches the size at which it seems practical to transfer a job to a slow node. As opposed to HETQL which manages to share the load in all systems. In the scenario where low power nodes receive no offered load, they will contribute nothing to the processing performed in the system. Performance only improves at the point where the high power nodes constitute a large enough proportion of total power to render the rest insignificant, as is the case at extreme negative heterogeneity. This provides another example of how the introduction of semi-random job location can provide the best form of load sharing.

With regard to stability, the performance of both algorithms holds up well at lower utilisation. The cost of extra probes only having a marginal effect on HQNIT performance. At higher levels HQNIT comes a distant second in respect to HETQL, which maintains steady performance characteristics under all the loading patterns tested.

6.3.4 Larger 12:8 systems - Scalability

Two new sizes of system are used in evaluating the scalability of the algorithms, 40 and 80 nodes. Both systems have the same ratio of nodes groups as used in the original simulations, 12:8. Evaluating the algorithms on systems larger than 80 nodes imposes a heavy computational burden to no obvious advantage. Quadrupling the original size should provide an adequate test for scalability. The results shown in Figures 6.15a and 6.15b are those obtained when using a probe limit of five. To allow direct comparison with the performance of a 20 node system.

The performance of both algorithms in larger systems matches almost exactly that over 20 nodes. Only one difference is noticeable, that at high utilisation there is a universal improvement in performance, from 20 to 40 nodes, and between 40 and 80

node to always being selected. The result of this is an inefficient allocation of the system load, again pointing to the value of semi-random allocation of load to idle nodes.

However the main conclusion that can be drawn from the evaluation of both algorithms over larger systems is that they are indeed scalable. The quality of scalability extends to the probe limit used by each. In that a probe limit of 5 can be used with all the system sizes investigated and provide efficient load sharing.

6.3.5 The Effect of System Parameters on Load Sharing Performance

The comparison of the simulation results presented in this chapter has shown that system parameters, as well as the type of load sharing algorithm used, influence the final response times achieved. Therefore consideration of the relative effects of these parameters is needed in order to complete discussion of the simulation results. Four system parameters have the greatest potential effect on the observed performance: probing, transfer cost, queue length and execution time.

In all circumstances probing is assumed to have a fixed delay as little information is required to be retrieved by each probe However the number of probes as defined by the probe limit used can have a significant effect. Given an average service time of 10 seconds a high probe limit has limited effect on performance for all algorithms. However if average service time is reduced the relative cost of each probe will increase. This will lead to a reduction in the comparative superiority of the performance of the HQNIT algorithm, as this uses a more expensive location policy. Should the relative cost of probing become comparable to the average service times of the jobs executed in the system then load sharing itself would be redundant, as the RANDOM has been shown to be ineffective in heterogeneous systems.

In the systems considered in this study, transfer cost is equivalent to that of one probe. Therefore while the relative cost of probing is still acceptable transfer cost will not hinder system performance. Should data used in the execution of a job be cached locally transfer cost will have greater effect on performance. The higher the cost of moving this data the less effective load sharing will be. This should not effect the relative performance of HETQL or HQNIT as they both move the same proportion of jobs, HETRO at high system utilisation moves less and so may become relatively more effective.

The size of job queues at nodes is an influential factor in observed performance. Algorithms that can balance queue lengths in terms of execution time are the most efficient, hence the superiority of HETQL and HQNIT. If load sharing is to be thought of in any terms a balancing operation it is in the balancing of weighted queue lengths.

Lastly and in terms of this study the most important system parameter is relative execution time of the jobs in the system. When the degree of heterogeneity is high and there is a large difference between the execution times at each node an algorithm that avoids transferring from high to low power nodes has a great advantage, i.e. HQNIT. However as has been noted in the previous section HQNIT can suffer from not utilising low power nodes to their full extent. This is shown in the response time peaks at approximately -0.1 skewness. After this point the low power nodes constitute such a small proportion of system load that their redundancy has a negligible effect.

6.4 Implementation Results

6.4.1 Practical Limitations and Parameters Used

Implementing a load sharing algorithm on a scale equivalent to the simulation model was restricted by several practical limitations. The main one was that only facilities already available at the University could be used and these would have to be solely employed in the evaluation process to give any interpretable results. Fortunately there was a group of machines all residing on one network and intended for undergraduate use. They satisfied the dual requirements of being unused for long periods (at night) and resident on a network with little other traffic. Two types of Sun workstation were present on this network, 10 IPXs and 20 Sparc5s. This set-up made available a 12:8 heterogeneous system on which to run the load sharing implementation. The implementation design and all intended validation was based around this arrangement.

The processing power of each node type was determined by running a set workload, the forerunner of the executejob.c script, on both nodes. The ratio of IPX processing power to Sparc5 was found to be 1:3.45. To keep the relative node powers in the same format as those used in the simulation model the actual values used were adjusted to account for a notional machine of power 1, on which the loop in *executejob.c* would run 60 times in 10 seconds. The adjusted powers were 0.405 for an IPX and

1.395 for a Sparc5 giving a total processing power of 20. Skewness and coefficient of variance for the implementation system are -0.047 and 0.236 respectively.

The periods during which the load sharing implementation could be run were restricted to the time that the computing labs were closed. For six days of the week this meant an upper limit of 12 hours. The exception, Sundays, allowed longer runs to be carried out, but to be consistent and to perform the necessary analysis in a reasonable time the run length was restricted to 40,000 seconds. Although this period is shorter than the simulation runs and therefore not ideal the compromise is unavoidable.

System utilisation is varied across the same range as in the simulation model, low medium and high. The probe limit is set at 5 for all experiments on the implementation system.

6.4.2 Measurement Results For a Heterogeneous System

One of the intended aims of the load sharing implementation was to establish whether the extra network traffic resulting from load sharing activities had any effect on the performance of the underlying communications network. Should this be the case the subsequent increase in communication latency would have a detrimental effect on response time. This factor is not catered for in the simulation studies as it was considered too complex a problem to model.

The network of Sun workstations identified in the previous section was an ideal testbed for such a study. Unfortunately just as the load sharing implementation code had been finished and tested the layout of the network in question was altered. This was the result of a decision by the University authorities and not reversible. The changes involved the IPX machines being replaced by more Sparc5s. After which the lower powered workstations were sited on another nearby network from which access to the original was possible via a single router.

The delay imposed by the router was measured using the ICMP protocol (ping) and found to average 2 ms. While it could not be considered negligible the extra cost was relatively small in comparison to the total cost of an RPC. Therefore the new arrangement was still considered adequate for most investigations into the properties of the load sharing implementation. Although the IPXs were only separated by one router any effect due to traffic congestion would be considerably reduced, as the total traffic volume was diluted by spreading it across two networks. The problem of underlying network performance is addressed later with the use of a homogeneous system.

With the implementation spread across the two adjacent networks it is still possible to perform the other validation tasks intended, which are:

- Validate model design by demonstrating that load sharing process is a feasible objective.
- Validate that all jobs are successfully executed.
- Validate that the delays used to represent RPCs are reasonable.
- Validate that the overhead in performing load sharing processing is negligible.

Rather than attempt to reach conclusions using the simulation results already presented in this chapter, further results were obtained from the model using the exact values for node power that occur in the implementation system. The make-up of the implementation system is described in Table 6.5a. Simulation and implementation results are presented in Tables 6.5b and 6.5c. The 95% confidence intervals are shown alongside the respective response times.

The first conclusion that can be made from the results of the measurement study is that it confirms load sharing is a viable prospect. All the response times gained are significantly lower than their equivalent when load sharing was not enabled. In addition the load sharing mechanism implemented was shown to be robust. No transferred job failed to execute or return to the originating node. A count was made of all finishing jobs with an exit status of 3 indicating a failure to terminate successfully. In all the measurement runs undertaken it was zero. The successful completion of all jobs validates the simulation assumption than none will be lost.

Tables 6.5b and 6.5c provide two distinct areas in which to compare both sets of results and so to validate the simulation. These are workload distribution and overall response time. The workload statistics indicate the behaviour of nodes whilst load sharing. On the basis of the results presented it would seem fair to say that the simulation is a fair representation of the real behaviour of the HQNIT algorithm. As the proportion of total workload executed at origin, refused transfer therefore executed at origin and transferred is almost the same at all utilisations. The final allocation of workload is identical except at high utilisation where the difference is only 1%.

	Power	Fraction of total power		Power	Fraction of total power	Skew	CV
A	1.395	0.84	B	0.405	0.16	-0.047	0.236

Table 6.5a Implementation System Composition

Simulation						
Utilisation	Response Time		Executed at origin %	Refused Transfer %	Transferred %	Processed at %
Low	10.15 +/-0.06	A	39	9	36	89
		B	11	0	5	11
Medium	12.18 +/-0.10	A	20	21	43	90
		B	9	0	7	10
High	20.84 +/-0.58	A	4	30	50	88
		B	5	0	11	12

Table 6.5b Simulation Results

Implementation						
Utilisation	Response Time		Executed at origin %	Refused Transfer %	Transferred %	Processed at %
Low	10.28 +/-0.14	A	38	11	35	89
		B	11	0	5	11
Medium	12.79 +/-0.21	A	19	21	44	90
		B	9	0	7	10
High	22.05 +/-0.61	A	3	29	52	87
		B	5	0	11	13

Table 6.5c Implementation Results

The second stage in the validation process involves comparing the response times of the two experimental methods. At a low level of utilisation the results from both simulation and measurement agree with 95% confidence. At higher levels of utilisation this is not the case although agreement is close. The trend is for the implementation results to be higher and this may be due to overhead not considered in the simulation rather than any more serious flaw. The cost of negotiating the router would a add few milliseconds to RPC cost in communication between low and high power nodes. However a more weighty factor could be operational overhead at each node. This is the system time used by each process during the implementation period.

Of the six processes three are solely used for RPC communication, remxclient, remxserver and serverprobe. Their overhead is accounted for in the simulation as part of the communication delays. Of the remaining three executejob overhead is accounted for in the total job execution time, leaving generatejobs and processjobs. Generatejobs would not be necessary in a true system and so its overhead can be deducted from the implementation results. Processjobs is of particular interest as any overhead due to load sharing activity would be incurred by this process, as would the overhead of collecting statistics and sending probes.

The system time used by processes of interest on a Sparc5 workstation during a simulation run at high utilisation are shown in Table 6.6. The simulation run was performed on a homogeneous system so that the number of probes sent and received at each node were the same. The overheads calculated can be applied to any system. Serveprobe is included to give a guide to the amount of time spent answering probes, as this should be approximately the same as the time spent sending them, the proportion of time spent by processjobs on this activity can be assessed. Once deducted from the total time used by processjobs an estimate of the unaccounted overhead can be made. Generatejobs also incurs an overhead but it is small enough to be ignored. The final figure for unaccounted overhead is 13 ms per job.

These overheads will only apply to Sparc5 workstations. The IPX overheads were found to be larger at approximately 31ms per job. This overhead unaccounted for in the simulation model accounts for the discrepancy in the results between simulation and implementation.

Process	Total Time (sec)	Time per job (ms)
generatejobs	1	0.2
processjobs (load sharing)	117	32
serveprobe	70	19
processjobs (no load sharing)	18	5

Table 6.6 Load Sharing Overheads

Of the total overhead figure some will be implementation oriented and not due to load sharing activity. Running the implementation without allowing load sharing allowed a figure for these activities to be derived. A final estimate of the overhead due to load sharing activities on a Sparc5 is approximately 8 ms per job. The overhead due to load sharing is too large to be considered negligible but does not represent a significant cost when compared to the average job used in the simulation studies. Even on an IPX the overhead due to load sharing activities will be below 0.25% of total job time.

The implementation overhead helps to explain the difference between both sets of results. With this in mind the assumptions on RPC costs are assumed to be valid, with the provision that the underlying network is unaffected by the increased traffic.

6.4.3 Implementation Results From a Homogeneous System

While the intended aim of the implementation scheme was to validate the simulation results of load sharing over heterogeneous systems, load sharing over a homogeneous system was also examined. Although the results were used in the assessment of overhead, the primary use was to determine the effect of the extra traffic generated due to load sharing activity on the underlying communications network. Performance when load sharing with a probe limit of 3 was compared to that when using a probe limit of 10. The HQNIT algorithm was used as it generates the greatest volume of traffic and was already implemented in the *processjobs.c* code.

The network of Sparc5 workstations were used for this study. Although only 20 out of a possible 30 were used the other 10 were idle overnight so it is unlikely that significant other traffic would of been present on the segment. The *executejob.c* code adjusted so that a job of 10 seconds would actually run for that long, i.e. the work performed was increased by a factor of 1.395. Using the IPX machines was not considered as there were less than 20 available. Table 6.7 shows the response times using different probe limits with 95% confidence intervals.

Model & Algorithm	**Low_Util**	**Medium_Util**	**High_Util**
Simulation model PL=3	10.84 +/- 0.05	12.94 +/- 0.06	20.53 +/- 0.47
Implementation, PL=3	10.67 +/- 0.03	12.81 +/- 0.08	20.70 +/- 0.23
Simulation Model, PL=10	10.41 +/- 0.04	11.51 +/- 0.06	18.04 +/- 0.52
Implementation, PL=10	10.60 +/- 0.19	12.36 +/- 0.39	20.18 +/- 0.76

Table 6.7 Homogeneous System Simulation / Implementation Comparison

When using a probe limit of 3 the results both sets of results agree. The load sharing overhead incurred in the implementation scenario is partially compensated for by the reduced RPC delay. All communicating nodes are Sparc5s and so the delay of 30ms is rather generous. There is certainly no reason to suspect that any extra delay has been imposed due to traffic congestion. This is not the case when a higher probe limit is used. At high utilisation the overall response time for the implementation study is significantly higher than it simulated counterpart. The difference between the two sets of results decreases with system utilisation when there is a corresponding reduction in communication rate. At low system utilisation both simulation and implementation results agree again.

When system utilisation is high and a probe limit of 10 is used an average of 64 separate inter-node communications will occur every second. With the exponential nature of job arrival this rate will be exceeded at times. Therefore it is not unfeasible to surmise that the extra cost observed could be due to congestion on the underlying network.

The idea of network congestion resulting in greater probe costs can be considered with the use of a simple queued server model of the network. The network costs of each probe depend upon the volume of data to be transmitted, communications protocol used and speed of network. The RPC that forms the core of each probe consists of four separate parts, the registration with the server and subsequent confirmation to client followed by actual service call with reply. None of these operations requires the transfer of significant quantities of data but the TCP protocol used is expensive, as it provides a reliable method of communication, requiring the exchange of many packets of data.

Assuming the following:

Average length of data packet = 100B (800 bits)	Packets per TCP connection = 8
Packets per RPC = 8 x 4 = 32	Network Speed = 10 Mbits/s

RPC's per second = $N * ta * Tr * P$

(N = nodes, ta = job arrival rate, Tr = Fraction of jobs transferred, P= probelimit + 1)

At high system utilisation and probe limit of 3

$$\text{N/W utilisation}(\rho) = \frac{\textbf{Total\_ Traffic}}{\textbf{Bandwidth}} = \frac{800 * 32 * 20 * 0.09 * 4}{10^7} = 0.016$$

If Mean service Time = 1

$$\text{Total delay} = \frac{E(ts)}{1 - \rho} = 1.016$$

Queuing delay = 1.6% of servicetime

At high system utilisation and probe limit of 10

$$\text{N/W utilisation}(\rho) = \frac{\textbf{Total\_ Traffic}}{\textbf{Bandwidth}} = \frac{800 * 32 * 20 * 0.09 * 11}{10^7} = 0.046$$

If Mean service Time = 1

$$\text{Total delay} = \frac{E(ts)}{1 - \rho} = 1.048$$

Queuing delay = 4.8% of servicetime

Although the level of network utilisation is far higher when using 10 probes than 3 it will only account for an extra delay of 5% in comparison to the simulation assumptions. The effect of collisions on the network medium will also add an extra cost, but this cannot be reflected in a simple queuing model. With a small average packet size the detrimental effect on CSMA/CD network performance due to collisions can be considerable [Sta91]. Another factor, not related to network congestion, as to why the implementation results when using 10 probes are higher than those reported by the simulation, is the overhead in generating the random numbers used to decide probe destination. Random number generation forms the bulk of all processing overhead due to load sharing in the implementation and the effect upon average response time will increase with probe limit.

The bulk of simulations used to investigate the algorithms use a probe limit of five which even at high utilisation produce less traffic than the low utilisation 10 probe implementation. There was no evidence of traffic congestion affecting results in this particular run. It is therefore reasonable to suppose all the 5 probe runs simulated would be unaffected. With the spectre of traffic congestion removed the timings for RPC's can be considered validated.

7. Closing Remarks

7.1 Summary of Algorithms Investigated

During the course of Chapter 6 a total of five algorithms were investigated: RANDOM, SHORTEST, HETRO, HETQL and HQNIT. The first two had been suggested in earlier work on load sharing in homogeneous systems, with the remaining three original to this study. Once it was apparent that the performance of any of the five algorithms was easily surpassed by the others, the algorithm concerned was dropped and further investigation concentrated on the remainder.

A brief summary of each, including if applicable, why it was considered inappropriate for heterogeneous systems is given below as a prelude to the conclusions drawn from this study.

The RANDOM algorithm uses the most basic of load sharing policies and as such is the simplest in operation. When the load at any node breaches a pre-set threshold the newly arrived job is transferred to a node selected at random. The performance of RANDOM suffered greatly as system heterogeneity increased. Alternatively SHORTEST with the use of an information policy, based location decisions on the load at potential destinations in the system. The value of this facility was reflected in a vast improvement over RANDOM at all levels of heterogeneity, combined with a satisfactory degree of stability.

HETRO the first algorithm designed for heterogeneous systems operated in a similar fashion to SHORTEST but used the processing power of potential destinations to achieve a weighted load, used in the location policy. With the benefit of better system knowledge HETRO gave lower response times especially at higher degrees of heterogeneity. The reliance on simple fixed thresholds was removed with the introduction of HETQL, which based location decisions on a comparison of destination and local queue length, this gave a performance advantage in all the systems evaluated. The last algorithm suggested was HQNIT, designed to stop the possibility of inefficient transfer to idle nodes by basing all location decisions on a full comparison between source and destination circumstances. HQNIT was the most successful in the majority of conditions but not all, in some its performance was surpassed by HETQL.

7.2 Conclusions

The earlier chapters of this thesis have introduced the topic of load sharing in heterogeneous systems. This mechanism for reducing system response time is controlled through the use of an algorithm. Several, specifically designed for heterogeneous environments have been suggested and their performance compared to a selection of algorithms for homogeneous systems. Two methods have been used in the evaluation of these algorithms, simulation and measurement via implementation. The majority of conclusions on algorithm performance are drawn from the simulation model. Whilst the implementation study was intended primarily for validation purposes it too has provided ideas of interest.

In the first chapter three general aims were set as the goals for the work described in later pages. This conclusion draws together the findings of both evaluation models and shows how they satisfy the intended aims.

- **Load sharing algorithms for homogeneous systems:**

  Two algorithms commonly used as benchmarks in studies on homogeneous systems were investigated., RANDOM and SHORTEST. Evaluation showed that random location policies have no place in load sharing algorithms for heterogeneous systems. The advantages of simplicity and minimised overhead inherent in such policies are far outweighed by the catastrophic results of transferring work to nodes where execution will take far longer than at the original site. Only in systems bordering on homogeneity is any advantage over the no load sharing case (upper bound) observed.

More success was achieved when the SHORTEST algorithm was employed. Average response times well below the upper bound were achievable at all levels of system utilisation in all the differing systems used. However at high utilisation performance does degrade rapidly with degree of heterogeneity, prompting the design of algorithms which could explcitly take into account the heterogeneity in system nodes.

- **Evaluation of load sharing algorithms specifically designed for heterogeneous systems:**

  Three algorithms were suggested in Chapter 3, HETRO, HETQL and HQNIT. HETRO is based on the design of SHORTEST but uses the power of the nodes probed in its location decisions. Simulation proved HETRO to be an improvement upon SHORTEST in all the circumstances tested. It copes with differing degrees of heterogeneity through a strategy of weighting the load at each node by relative processing power. However the need for some form of adaptive threshold was still a pitfall. Without that, prior knowledge of system utilisation was required for the algorithm to operate at its optimum.

  The HETQL algorithm solved the changing threshold problem by basing location decisions upon local load and weighted remote load. The increase in accuracy of location decisions when using a local load based policy enabled an optimum transfer threshold of 1 to be used at all times. Whilst little difference was observed between the performance of HETRO and HETQL at low and medium utilisation at high levels the latter showed significant improvement across all degrees of heterogeneity. HETQL allows immediate transfer to any idle node discovered whilst probing, a potential weakness in heterogeneous systems. For this reason a refined version was investigated. The HQNIT algorithm does not allow transfer without considering the processing power of any potential destination node.

  HQNIT uses a location policy that accounts for all the system information available, in terms of load and power. The eligible job is sent for execution to the node on which it will complete first. While HQNIT outperforms HETQL at low and medium system utilisation it does not at higher levels. The exception occurs at extreme negative degrees of heterogeneity. Discounting the cost of the more extensive HQNIT information policy another reason is needed to explain why an algorithm that makes less informed decisions provides better performance.

The answer lay in examining the load distribution in the IDEAL scenario, the lower bound for algorithm performance. In all circumstances the high power nodes took were allocated proportionally greater percentage of the total workload, but as utilisation increased this proportion decreased. Examination of the workload allocation for each of the heterogeneous algorithms revealed that HQNIT favoured the high power nodes at all times. Whereas HETQL and HETRO favoured low power nodes at low and medium utilisation with a more balanced approach at high utilisations. HETRO with its unsophisticated location policy cannot outperform HQNIT but HETQL does, as its loading at high utilisation is closer to the optimum. HQNIT does not take advantage of the latent processing power available in the low power nodes and performance suffers accordingly. As HQNIT makes the best location decisions possible with the data available there appears to be an advantage in certain circumstances to invoking a form of random allocation.

The definitions of load balancing and load sharing common in homogeneous systems are not applicable in heterogeneous systems. Investigation of an IDEAL scenario over a range of systems, shows that the optimum solution is not achieved by equalising load amongst the nodes. The workload of the more powerful nodes should be far in excess of their proportion of processing power. When considering heterogeneous systems it is not enough to base a load sharing strategy around ensuring that no node is idle. However at high utilisation the use of immediate idle transfer can offer advantages at some degrees of heterogeneity.

The optimum amount of system information required for all the algorithms studied is approximately the same at 5 probes. Although the HETQL algorithm can benefit slightly by using a higher probe limit the improvement is not universal. HQNIT performance starts to degrade almost immediately at higher probe limits. System size appears to have little bearing on the optimum probe limit. While HETQL can reduce response times with higher probe limits the increase in information required is not relative to the size of the system. The benefits that are gained must be weighed against the difficulty of implementing an adaptive probe limit.

Both the HETQL and HQNIT algorithms have been shown to be adaptable, scalable and stable under a wide variety of changing conditions. Not all possible circumstances have been considered but varying load distribution, system size and degree of heterogeneity give a reasonable basis for this statement.

The main factors for differentiation between the two algorithms are system utilisation and degree of heterogeneity. HQNIT is superior at low to medium levels of system utilisation in the vast majority of systems investigated, especially those with a high degree of heterogeneity. At high levels of utilisation HETQL enables lower response times the exception being systems with a high negative skew.

- **Implement a load sharing scheme on a distributed system in order to validate the simulation model and examine questions impractical to simulate:**

A load sharing implementation was constructed and its performance measured over a distributed system of 20 nodes. The HQNIT algorithm was found to operate satisfactorily over the system available. There is no reason to suspect that the other algorithms simulated would not also be viable.

Each node in the implementation used a multi-threaded operation to generate, process and finally execute offered workload. The mechanism used was robust enough to guarantee no jobs failed to execute or were lost due to inter-node communication. Remote Procedure Calls were used to perform said communication. The delays used in the simulation model were shown to be reasonably accurate.

The measurement results were in agreement with those derived from study using the simulation model. Overhead incurred by the implementation in generation of workload and processing performance statistics was found to be negligible in terms of overall response time. The overhead of the load sharing process was assumed to be negligible in the simulation model and measurement found it to constitute less than 0.1% of total response time in the cases observed.

Implementing a load sharing scenario enabled the result of the extra traffic generated on the performance of the underlying communication network to be observed. The results indicate that network performance is unaffected in all cases when using a probe limit of 5 on a system of 20 nodes. However network performance can be detrimentally effected if probe limit is much higher. The same result can be expected if system size increases and all the nodes are based on the same LAN. Therefore in these circumstances it would be prudent to consider load sharing performance as indicated by the simulation model as an optimum value.

7.3 Further Work

During the course of this work several avenues for further investigation have suggested themselves, but have been left unexplored due to the limits of time and available resources. Possible new areas have arisen in both the simulation and implementation of load sharing algorithms.

The large number of variable parameters that have been noted when applying the simulation model could all warrant further study. However the two that would appear to be of most interest are system composition and average job size. All the systems investigated to date have been comprised of two different types of node. Using a far greater mix of node types would provide a further test of the algorithms proposed for heterogeneous systems. This would be possible without changing the simulation model in any way, with the exception of some variable parameters.

The implementation study indicated that the volume of load sharing traffic could become a problem in a larger, single segment based system. Therefore it may be beneficial to investigate the performance of an algorithm that transfers immediately on finding a node that is more lightly loaded. This is the same principle as that used in THRESHOLD [Eag86a, Zho87]. The idea was not pursued in this study as versions of HQNIT or HETQL using this technique could produce a performance improvement over the originals.However if the problem of network congestion does cause a significant problem this would not be the case. Implicit in this idea is that a means of monitoring network congestion could be built into the simulation model, which may prove to be a challenge.

Having validated the delays used in the simulation model to represent the overhead involved in the load sharing process, the model can now be used to determine the effect of reducing average job size, in order to find the minimum job size at which load sharing is still cost effective.

The increase in commercially available performance measurement software [HP96a, BGS97] prompts possible changes in the programs written to control load sharing in the workstation environment. The means of calculating load, at present somewhat artificial, could be replaced with the true CPU queue length. Other metrics available via performance software would provide a clear picture of the state of each node involved in any load sharing activity. Operating system metrics and network performance data could be compared to investigate fully the effects of load sharing traffic on network performance. Measurement of response times is also now available with the Application Response Measurement initiative [HP96b].

Enterprise management products [CA96] have begun to consider the load sharing problem. They claim success in distributing workload to the systems with most resource available. If possible (guarded technology may be a problem) it would be interesting to contrast the commercial approach and the one taken in this study.

Bibliography

[Ald92] Aldy.N, Nagi.M, Selim.S, "Performance Evaluation of Job Scheduling in Heterogeneous Distributed Systems", *Proc. European 1992.*

[ARM96] *Application Response Measurement API Guide*, Hewlett-Packard Company, 1996.

[Art89] Y.Artsy, R.Finkel, "Designing a Process Migration Facility - The Charlotte Experience", *Computer,* Sept. 1989, pp 47-56.

[Bac86] Bach.M.J, "The Design of the UNIX Operating System", *Prentice Hall,* 1986.

[Bak92] Baker.D, Haddledon.R, Wika.K, "A Distributed Scheduling Simulation" :*Proc. 1st International Symposium on High Performance Distributed Computing*, 1992.

[Ban89] S.Banawan, J.Zahorjan, "Load sharing in Hierarchical Distributed Systems", *Proc. of the 1989 Winter Simulation Conference,* pp 963-970.

[Bau89] K.Baumgartner, B.Wah, "GAMMON: A Load balancing Strategy for Local Computer Systems with Multi-access Networks", *IEEE Transactions on Computers,* Vol 38, No 8, 1989, pp 1088-1109.

[Ben94] K.Benmohammed-Mahieddine, P.Dew, "A Periodic Symmetrically-Initiated Load Balancing Algorithm for Distributed Systems", *Operating Systems Review,* 28(1), 1994, pp 66-77

[Ben95] K.Benmohammed-Mahieddine, P.Dew, "A Testbed for the study of Load balancing Algorithms on Distributed Systems", *Advances in Modelling & Analysis,* Vol. 24, No 1, 1995, pp 19-30.

[Ber93] G.Bernard, D.Steve, M.Simatic, "A Survey of Load Sharing in Networks of Workstations", *Distributed. System. Engineering*, Vol. 1, 1993 , pp 75-86.

[BGS97] *BEST/1 Performance Assurance for UNIX*, BGS Systems, 1997.

[Bis95] M.Bishop, M.Valence, L.Wisniewski, "Process Migration for Heterogeneous Distributed Systems", *Dartmouth College Technical Report Series,* PCS-TR95-264, 1995.

[Boo91] Booch.G, "*Object Oriented design with Applications*", The Benjamin Cummings Publishing Corp, 1991.

[Blo92] Bloomer.J, *Power Programming With RPC,* O'Reilly & Associates Inc, 1992.

[CA96] *CA-UNICENTER TNG Product Description*, Computer Associates, 1996.

[CACa93] *MODSIM II User's Manual*, CACI Products Company, 1993.

[CACb93] *MODSIM II Tutorial*, CACI Products Company, 1993.

[CACc93] *MODSIM II Reference Manual*, CACI Products Company, 1993

[Cas87] T.Casavant, J.Kuhl, "Analysis of Three Dynamic Distributed Load-Balancing Strategies With Varying Global Information Requirements", *Proc. 7th International Conference on Distributed Computing Systems*, 1987, pp 185-191.

[Cas88] T.Casavant, J.Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems", *IEEE Transactions on Software Engineering*, Vol. 14, No 2, 1988, pp 141-154.

[Cho79] Y.Chow, W.Kohler, "Model for Dynamic Load balancing in a Heterogeneous Multiple Processor System", *IEEE Transactions on Computing,* Vol. C-28, No 5, 1979, pp 354-361.

[Cou94] G.Coulouris, J.Dollimore, T.Kindberg, *Distributed Systems - Concepts and Design,* Addison Wesley 1994.

[Dan95] S.Dandamudi, "The Effect of Scheduling Discipline on Sender-Initiated and Receiver-Initiated Adaptive Load Sharing in Homogeneous Distributed Systems", *Ottawa University Technical Report Series,* No. 95-25 1995.

[Eag86a] D.L.Eager, E.D.Lazowska, J.Zahorajan, "Adaptive Load Sharing in Homogeneous Distributed Systems", *IEEE Transactions on Software Engineering*, Vol. SE-12, No 5, 1986, pp 662-675.

[Eag86b] D.L.Eager, E.D.Lazowska, J.Zahorajan, "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing", *Performance Evaluation*, Vol. 6, 1986, pp 662-675.

[Eag88] D.L.Eager, E.D.Lazowska, J.Zahorajan, "The Limited Performance Benefits of Migrating Active Processes for Load Sharing", *Proc. ACM Sigmetrics*, 1988, pp 63-72.

[Fer87] D.Ferrari, S.Zhou, "An Empirical Investigation of Load Indices for Load Balancing Applications", *Proc. Performance 87 12th IFIP International Conference on Computer performance*, 1987, pp 515-528.

[Gra92] P.Gray, *"Distributed Systems - A Business Strategy for the 1990s"*, McGRAW-HILL Book Company Europe, 1992.

[Gra94] I.Graham, *"Migrating to Object Technology"*, Addison-Wesley, 1994.

[Hou94] C.Hou, K.Shin, "Load Sharing with Consideration of Future task arrivals in Heterogeneous Distributed Real-Time Systems", *IEEE Transactions on Computers*, Vol. 43, No 9, 1994, pp 1076-1090.

[HP96a] *MeasureWare Agent : Users Manual*, Hewlett-Packard Company, 1996.

[HP96b] *Application Response Measurement API Guide*, Hewlett-Packard Company, 1996.

[IEEE90] *"Information Technology-Portable Operating System Interface (POSIX)"*, IEEE 1003, 1990.

[Ker84] B.Kernighan, D.Ritchie,"*The C Programming Language*", Prentice-Hall, 1984.

[Kan92] K.Kant, "*An Introduction to Computer System Performance Evaluation*", McGRAW HILL International Editions, 1992.

[Kim92a] C.Kim, H.Kameda, "An Algorithm for Optimal Static Load Balancing in Distributed Computer Systems", *IEEE Transactions on Computing*, Vol. 41, No 3, 1992, 381-388.

[Kim92b] J.Kim, J.Liu, Y.Hao, "An All Sharing Load Balancing Protocol in Distributed Systems on the CSMA/CD Local Area Network", *Proc. Distributed Computing Symposium*, 1992, pp 82-89.

[Kle76] L.Kleinrock, "*Queuing Systems: Volume 2, Computer Applications*", John Wiley & Sons, 1976.

[Kle85] L.Kleinrock, "Distributed Systems", *Communications of the ACM,* Vol. 28, No 11, 1985, pp 1200-1213.

[Kre92] O.Kremien and J.Kramer, "Methodical Analysis of Adaptive Load Sharing Algorithms", *IEEE Transactions on Parallel and Distributed Systems,* Vol. 3, no 6, 1993, pp 747-760.

[Kru87] P.Krueger, M.Livny, "The Diverse Objectives of Distributed Scheduling Policies", *Proc. 7th International Conference on Distributed Computing Systems*, 1987, pp 242-249.

[Kru88] P.Krueger, M.Livny, "A Comparison of Pre-emptive and Non-preemptive Load Distributing", *Proc. of the 8th International Conference on distributed Computing Systems,* 1988, pp 123-130.

[Kru94] P.Krueger, N.Shivaratri, "Adaptive Location Policies for Global Scheduling", *IEEE Transactions on Software Engineering*, Vol 20, No 6, 1994, pp 432-444.

[Kun91] T.Kunz, "The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme", *IEEE Transactions on Software Engineering*, Vol. 17, No 7, 1991, pp 725-730.

[Leu87] C.Leung, "*Quantitive Analysis of Computer Systems*", Prentice Hall, 1987.

[Liv82] M.Livny, M.Melman, "Load Balancing in Homogeneous Broadcast Systems", *Proc. ACM Computer Network Performance Symposium*, April 1982, pp 47-55.

[Mah93] A.Mahamuni, T.Gonsalves. B.Ramamurthi, "Efficient Load Information Management for Load Sharing in Distributed Systems', *Computer Networks Architecture and Applications,* C-13, 1993, pp 43-54.

[Mir89a] R.Mirchandaney, D.Towsley and J.Stankovic, "Analysis of the Effects of Delays on Load Sharing ", *IEEE Transactions on Computers,* Vol. 38, No 11, 1989, pp 1513-1525.

[Mir89b] R.Mirchandaney, D.Towsley and J.Stankovic, "Adaptive Load Sharing in Heterogeneous Systems", *IEEE Transactions on Computers,* Vol. 38, No 11, 1989, pp 1513-1525.

[Muk91] M.Mukta, M.Livny, "The Available Capacity of a Privately Owned Workstation Environment", *Performance Evaluation, Vol. 12, 1991*, pp 269-284.

[Nay67] T.Naylor, J.Finger, "*Verification of Computer Simulation Models*", Management Science, Vol. 2, pp B92-B101.

[Ni81] L.Ni, K.Abani, "Non-preemptive load balancing in a Class of Local area Networks", *Proc. IEEE Computer Networking Symposium,* 1981, pp 113-118.

[Phi90] I.Phip, "Dynamic Load balancing in Distributed Systems", *Proc. Southeastcon 1990*, pp 304-307.

[Rom91] C.Rommel, "The Probability of Load balancing success in a Homogeneous Network", *IEEE Transactions on Software Engineering*, Vol. 17(9), 1991, pp 922-933.

[Rum91] J.Rumbaugh, M.Blatha, W.Premerlani, F.Eddy, W.Lorensen, *"Object-Oriented Modeling and Design"*, Prentice Hall International, 1991.

[Sar95] A.Sarraf, J.Senior, A.Wiseman, 'New Technique to Assess the Asymmetry of the Traffic Load Offered to LAN's", *Proc. Second Communication Networks Symposium*, 1995, pp 77-80.

[Shi92] N.Shivaratri, P.Krueger, M.Singhal, "Load Distributing for Locally Distributed Systems", *Computer,* Dec. 1992, pp 33-44.

[Shi94] S.Shi, D.Lin, C.Wang, "Dynamic Load Sharing Services with OSF DCE", *First International Workshop on Services in Distributed and Networked Environments,* 1994, pp 178-186.

[SPE96] System Performance Evaluation Co-operative, *http://www.specbench.org,* 1996.

[Sri92] P.Srimani and R.Reddy, "Load Sharing in Soft Real-time Distributed Systems", *International Journal of Systems Science*, Vol. 23(7), 1992, 1115-1130.

[Sta91] W.Stallings, "*Data and Computer Communications*", Maxwell Macmillan International Editions, 1991.

[Sta84] J.Stankovic, "Simulations of Three Adaptive, Decentralised Controlled, Job scheduling Algorithms", *Computer Networks,* Vol. 8, 1984, pp 199-217.

[Sta85] J.Stankovic, " Stability and Distributed Scheduling Algorithms", *IEEE Transactions on Software Engineering,* Vol. SE-11, No 10, 1985, pp 1141-1152.

[Sun90] Sun Education, "Priorities and Scheduling", *Revision D.2*, 1990, pp 536 - 553.

[Tan85] A.Tanenbaum, R.Van Renesse, "Distributed Operating Systems", *ACM Computing Surveys*, Vol. 17, No 4, 1985, pp 421-470.

[Tan85a] A.Tantawi, D.Towsley, "Optimal Static Load Balancing in Distributed Computer System", Journal of the ACM, Vol. 32, No. 2, 1985, pp 445-465.

[Tan87] A.Tanenbaum, "*Operating Systems - Design and Implementation",* Prentice Hall International, 1987.

[Tan95] T.Tanenbaum, M.Litzkow, "The Condor Distributed Processing System", *Dr. Dobb's Journal,* Feb. 1995, pp 40-48.

[The89] M.Theimer, K.Lantz, "Finding Idle Machines in a Workstation-Based Distributed System", *IEEE Transactions on Software Engineering*, Vol. SE-15, No 11, 1989, pp 1444-1457.

[Wan85] Y.Wang, R.Morris, "Load Sharing in Distributed Systems", *IEEE Transactions on Computers*, Vol. 34(3), 1985, pp 204-217.

[Wan94] J.Wang, L.Tee, Y.Huang, "Load Balancing Policies in Heterogeneous Distributed Systems", *Proc. 26th Symposium on System Theory*, 1994, pp 473-477.

[Wil95] C.Wills, D.Finkel, 'Scalable Approaches to Load Sharing in the Presence of Multicasting', Computer Communications, Vol. 18 No. 9, 1995, pp 619-630.

[Yum81] T.Yum, "The Design and Analysis of a Semidynamic Deterministic Routing Rule", *IEEE Transactions on Communications*, COM-29, No 4, 1981, pp 498-504.

[Zho87] S.Zhou, D.Ferrari, "A Measurement Study of Load balancing Performance", *Proc. 7th International Conference on Distributed Computing Systems*, 1987, pp 490-497.

[Zho88] S.Zhou, "A Trace Driven Simulation Study of Dynamic Load Balancing", *IEEE Transactions on Software Engineering*, Vol. 14(9), 1988, pp 1327-1341.

[Zho93] S.Zhou, X.Zheng, J.Wang, P.Delisle, 'Utopia : a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems', *Software Practice and Experience, 1993*, Vol. 23(12), pp 1305-1336.

Appendix 1. Simulation Code

A1.1 Definition Module

```
DEFINITION MODULE Hetrodelaylib;              { Module in which all model definitions are made}

FROM RandMod IMPORT RandomObj, FetchSeed;
FROM ListMod IMPORT QueueList;
FROM SimMod IMPORT SimTime, StartSimulation, StopSimulation, TriggerObj;
FROM StatMod IMPORT RStatObj, IStatObj, SREAL;

CONST


{--------------------------------------------------------------------------------------------------}

TYPE
        JobType = RECORD                      {structure used to represent a job}
        origin : INTEGER;
        arrivaltime : REAL;
        servicetime : REAL;
        transfertag : INTEGER;
        destination : INTEGER;
        END RECORD;

        Hetrorecord = RECORD
        numberofnodes : INTEGER;
        powerofnodes : REAL;
        END RECORD;

        HetroArray = ARRAY INTEGER OF Hetrorecord;

        ArrayType = ARRAY INTEGER OF INTEGER;

{--------------------------------------------------------------------------------------------------}
{genesis object used to initialise, start and collect final statistics from the simulation model}

GenesisObj = OBJECT

overallRT, overallBAT : SREAL;
TELL METHOD InitialiseNodes(IN defarray : HetroArray; IN seed : INTEGER; IN batchtime : REAL);
ASK METHOD ObjTerminate;
ASK METHOD PerfStats() : REAL;
ASK METHOD Batchresults() : REAL;

END OBJECT;
{--------------------------------------------------------------------------------------------------}
{ node object used to perform all the actions required from a node}

NodeObj = OBJECT;

jobQ, txQ, rxQ,lpQ : QueueList;               {queue of job types}
nodeID, probecount, successcount : INTEGER;
sig, tXsig, rXsig, lpsig, batchsig : TriggerObj;
randomnode1 : RandomObj;
nodeRT, responseT, jobLength, taTime, jobLengthRec, jobLengthRef, nodeBRT, batchRT : SREAL;
nodepower, lastTa : REAL;
currentjob : JobType;

ASK METHOD ObjInit;
TELL METHOD GenerateJobs(IN a : INTEGER);
TELL METHOD ProcessRandom;
TELL METHOD ProcessSHORTEST;
TELL METHOD ProcessHETRO;
TELL METHOD ProcessHETQL;
TELL METHOD ProcessHQNIT;
ASK METHOD UpdateRT( IN job : JobType);
ASK METHOD ReceiveJob(IN job : JobType);
TELL METHOD ExecuteJob;
TELL METHOD Transmit;
TELL METHOD Receive;
ASK METHOD AssignID( IN i : INTEGER; IN power : REAL);
ASK METHOD RemoveJobs;
ASK METHOD ObjTerminate;

END OBJECT;
{--------------------------------------------------------------------------------------------------------------------------------}
{object to cease simulation}
StopAllObj = OBJECT

TELL METHOD Finish;

END OBJECT;
{--------------------------------------------------------------------------------------------------------------------------------}

{procedure used by load sharing algorithm methods to select nodes for probing}
PROCEDURE UniqueRandom(IN Probelimit : INTEGER; IN nodeID : INTEGER;INOUT numarr : ArrayType);

{procedure used during initialisation to select load sharing algorithm to use}
PROCEDURE PickAlgorithm(IN ID : INTEGER);

{--------------------------------------------------------------------------------------------------------------------------------}
{ Global variables }
VAR
            AvInterArrivalTime, AvServiceTime, TDelay, probingDelay,
            probedDelay, minSize, batchtime : REAL;
            test : BOOLEAN;
            globalrandom, random1 : RandomObj;
            nodearray : ARRAY INTEGER OF NodeObj;
            ProbeLimit, NofN, NofDN, Threshold, TransferLimit,
            Algorithm : INTEGER;


END MODULE.
```

A1.2. Implementation Module

```
IMPLEMENTATION MODULE Hetrodelaylib;

FROM RandMod IMPORT RandomObj, FetchSeed;
FROM ListMod IMPORT QueueList;
FROM SimMod IMPORT SimTime, StartSimulation, StopSimulation, TriggerObj, Interrupt;
FROM StatMod IMPORT RStatObj, SREAL, SINTEGER;

{--------------------------------------------------------------------------------------------------------}

{Creates new random number generator with seed passed down. Creates array of nodes the size of the desired system
and then creates the actual nodes themselves. A node needs an ID and to be given a power rating. The random number
generator is used to derive a seed for each node which is used in its Generatejob method. Other methods to run constantly
are started as well. The Process procedure selects and starts the desired algorithm type. The loop at the end of this
method is used for collecting the batch results needed in determining initialisation period and run length.}


OBJECT GenesisObj;

{method that performas initialisation of all nodes in the system}
TELL METHOD InitialiseNodes(IN defarray : HetroArray; IN seed : INTEGER; IN batchtime : REAL);
VAR
            node : NodeObj;
            i, genSeed, j, offset,ID, loopct : INTEGER;
            power, BRT : REAL;
BEGIN
            NEW (random1);
            NEW (globalrandom);
            ASK random1 TO SetSeed(seed);
            offset := 0;
            loopct := 1;
            NEW(nodearray, 1..NofN);
```

```
        FOR i:=1 TO NofDN {for each group of different powered nodes}
                FOR j:=1 TO defarray[i].numberofnodes              {for each node in a group}
                        power := defarray[i].powerofnodes;         {initialise node values}
                        ID := j+offset;
                        NEW(node);
                        nodearray[ID] := node;
                        ASK nodearray[ID] TO AssignID(ID,power);
                        genSeed := ASK random1 UniformInt(1, 10000);
                        TELL nodearray[ID] TO GenerateJobs(genSeed); {start methods to run for duration}
                        TELL nodearray[ID] TO Transmit;
                        TELL nodearray[ID] TO Receive;
                        TELL nodearray[ID] TO ExecuteJob;
                        PickAlgorithm(ID);
                        INC(j);
                END FOR;
                offset := defarray[i].numberofnodes + offset;
                INC(i);
        END FOR;

        LOOP
                WAIT DURATION batchtime   {loop used in compiling batch times}
                BRT := ASK SELF TO Batchresults;
                OUTPUT("Batch ",loopct," RT ", BRT);
                INC(loopct);
                END WAIT;
        END LOOP;
END METHOD; {end of initialisation method}
{------------------------------------------------------------------------------------------------------}

{This Method is needed to collect batch statistics}
ASK METHOD Batchresults() : REAL;

VAR
        I : INTEGER;
        totalCount : INTEGER;

BEGIN
        ASK(GETMONITOR(overallBAT,RStatObj))Reset();
        totalCount := 0;
{The number of jobs executed in the system during this batch are are totalled up}
        FOR I := 1 TO NofN
                totalCount := ASK(GETMONITOR(nodearray[I].batchRT,RStatObj))Count + totalCount;
        END FOR;
{ Each nodes contribution to the average response time is calculated and added to the total, after which the statistical object
is reset for the next batch}

        FOR I := 1 TO NofN
                overallBAT :=
FLOAT(ASK(GETMONITOR(nodearray[I].batchRT,RStatObj))Count)/FLOAT(totalCount)*
ASK(GETMONITOR(nodearray[I].batchRT,RStatObj))Mean();
ASK(GETMONITOR(nodearray[I].batchRT, RStatObj))Reset();
        END FOR;
{The average response time for the batch is returned to the calling object}
RETURN(ASK(GETMONITOR(overallBAT,RStatObj))Sum);

END METHOD; {end of method}
{------------------------------------------------------------------------------------------------------}

{The method which collates statistics on the total simulation run time}
ASK METHOD PerfStats() : REAL;
CONST
        format="*.** ***.** ***** ***.** ****** ***.** ****** ***.** ****** ***.** ******";

VAR
        I : INTEGER;
        overallPR: SREAL;
        TotalCount : INTEGER;
        ProbeRes : REAL;
BEGIN
        TotalCount := 0;
{The number of jobs executed in the system during the total run time are are totalled up}
        FOR I := 1 TO NofN
                TotalCount := ASK(GETMONITOR(nodearray[I].responseT,RStatObj))Count + TotalCount;
        END FOR;
```

```
{The contribution of each node to the  metrics collected is calculated and printed out}
            FOR I := 1 TO NofN
                        overallRT                                                                              :=
FLOAT(ASK(GETMONITOR(nodearray[I].responseT,RStatObj))Count)/FLOAT(TotalCount)*
ASK(GETMONITOR(nodearray[I].responseT,RStatObj))Mean();
{Lists average response times and total number of jobs executed at node}

PRINT(nodearray[I].nodepower,ASK(GETMONITOR(nodearray[I].responseT,RStatObj))Mean()
,ASK(GETMONITOR(nodearray[I].responseT,RStatObj))Count,
{Lists average response times and total number of jobs originating at node}

ASK(GETMONITOR(nodearray[I].nodeRT,RStatObj))Mean() ,ASK(GETMONITOR(nodearray[I].nodeRT,RStatObj))Count,
{ Lists number of and average lengths of jobs executed at origin}

                                                ASK(GETMONITOR(nodearray[I].jobLength,RStatObj))Mean()
,ASK(GETMONITOR(nodearray[I].jobLength,RStatObj))Count,
{ Lists number of and average lengths of jobs executed at origin but refused transfer}

                                             ASK(GETMONITOR(nodearray[I].jobLengthRef,RStatObj))Mean()
,ASK(GETMONITOR(nodearray[I].jobLengthRef,RStatObj))Count,
{Lists number of and average lengths of jobs that have been transferred to other nodes}

ASK(GETMONITOR(nodearray[I].jobLengthRec,RStatObj))Mean()
,ASK(GETMONITOR(nodearray[I].jobLengthRec,RStatObj))Count
{Lists number of and average lengths of jobs that have been received from other nodes}

) WITH format;
            END FOR;

{return overall average response time to calling object genesisObj}
RETURN ASK(GETMONITOR(overallRT,RStatObj)) Sum;
 ASK(GETMONITOR(overallRT,RStatObj))Reset();

END METHOD;

{------------------------------------------------------------------------------------------------------}
{method to free all memory associated with the genesisObj after run-time has expired}
ASK METHOD ObjTerminate;
VAR
            i : INTEGER;
BEGIN
            DISPOSE(random1);
            DISPOSE(globalrandom);
            FOR i:= 1 TO NofN
                        DISPOSE(nodearray[i]);
            END FOR;
            DISPOSE(nodearray);
END METHOD;

END OBJECT; {end of genesisObj}

{------------------------------------------------------------------------------------------------------}

{Before nodeObj starts  objects it uses as  triggers and queues are initialised}

OBJECT  NodeObj;

ASK METHOD ObjInit;
BEGIN
            NEW(sig);               {triggers}
            NEW(rXsig);
            NEW(tXsig);
            NEW(lpsig);
            NEW(jobQ);              {queues}
            NEW(rxQ);
            NEW(txQ);
            NEW(lpQ);
            NEW(randomnode1);
END METHOD;
{------------------------------------------------------------------------------------------------------}
```

```
{ Jobs are generated at each node and then the threshold at each node is checked if the threshold is not exceeded by the
arrival of a new job the job is added to the local queue for execution, otherwise it is placed in the queue of the Process
method selected in Initialisenodes. In effect the transfer policy is carried out here}

TELL METHOD GenerateJobs( IN a : INTEGER);

VAR
        newjob : JobType;
        ta, ts, hetroInterarrivalTime, tempTS : REAL;

BEGIN

        ASK randomnode1 TO SetSeed(a);
        hetroInterarrivalTime := AvInterArrivalTime / nodepower;
{Interrarrival time is in direct proportion to nodepower, this ensures that the original utilisation at each node is the same.}
        LOOP
{Exponentially distributed interarrival times are equivalent to a  poission arrival rate}
                ta := ASK randomnode1 Exponential(hetroInterarrivalTime);
                WAIT DURATION ta END WAIT;
{A new job is created and its arrival time and service time are stored in the record structure}
                NEW(newjob);
                newjob.origin := nodeID;
                newjob.arrivaltime := SimTime();
                newjob.servicetime := ASK randomnode1 Exponential(AvServiceTime);
{The transfer policy, based around a simple pre- determined threshold. If the job is considered eligble for transfer it is added
to the Process queue and a signal released to indicate this fact}
                IF(jobQ.numberIn >= Threshold)
                        ASK lpQ TO Add(newjob);
                        ASK lpsig TO Release;
                ELSE
{ get original job length for statistical purposes and then calculate actual servicetime on executing machine. The job is then
added to the queue for execution and a signal sent to indicate this fact}
                        jobLength := newjob.servicetime;
                        newjob.servicetime := newjob.servicetime/nodepower;
                        ASK jobQ TO Add(newjob);
                        ASK sig TO Release;
                END IF;
        END LOOP;
END METHOD;
{------------------------------------------------------------------------------------------------------------------}

{Random algorithm or blind location, without the use of any system state information the eligble job is sent to a randomly
picked node for execution.}

TELL METHOD ProcessRandom;
VAR
        job : JobType;

BEGIN
LOOP
{If there are jobs waiting to be processed, pick any node at random and send the job to that node for processing. Otherwise
wait for the signal that jobs are waiting to be processed. A queue (txQ) is used to buffer jobs and prevent the possibility of
concurrent transmission The means of picking a ranom node is unsophisticated in design as normally it will be successful
on the first attempt}

        IF lpQ.numberIn > 0
                job := ASK lpQ TO Remove();
                REPEAT
                        job.destination := ASK globalrandom UniformInt(1, NofN);
                UNTIL job.destination <> nodeID;
                INC(job.transfertag);
                ASK txQ TO Add(job);
                ASK tXsig TO Release;
        ELSE
                WAIT FOR lpsig TO Fire;
                END WAIT;
        END IF;
END LOOP;
END METHOD;
{------------------------------------------------------------------------------------------------------------------}

{ A location and information policy developed for homogeneous systems}

TELL METHOD ProcessSHORTEST;
```

```
VAR
          numarr : ArrayType;
          pl, destination, minload, mindest : INTEGER;
          job : JobType;
BEGIN
          NEW(numarr, 0..ProbeLimit);   {create array for random numbers}
          LOOP
          IF lpQ.numberIn > 0
                    job := ASK lpQ TO Remove();
                    pl := ProbeLimit;
                    UniqueRandom(ProbeLimit,nodeID,numarr);          {get random numbers}
                    minload := Threshold;
                    WHILE pl > 0 {until the probe limit has expired}
                              destination := numarr[pl];
{probing effects both local and remote node as well as the current job}
                              Interrupt(nodearray[destination], "ExecuteJob");
                              Interrupt(nodearray[nodeID],"ExecuteJob");
                              WAIT DURATION probingDelay END WAIT;
{if the remote nodes load is less than the threshold it becomes a possible destination for the current job}
                              IF minload > ASK nodearray[destination] jobQ.numberIn;
                                        minload := ASK nodearray[destination] jobQ.numberIn;
                                        mindest := destination;
                              END IF;
{if the remote node is idle the current job is immediately transferred to it}
                              IF minload = 0
                                        INC(job.transfertag);
                                        job.destination := destination;
                                        Interrupt(nodearray[destination], "ExecuteJob");
                                        Interrupt(nodearray[nodeID],"ExecuteJob");
                                        {transmit job to selected node}
                                        ASK nodearray[job.destination] TO ReceiveJob(job);
                                        WAIT DURATION (TDelay + 0.001) END WAIT;
                                        EXIT;        {exit construct as job processing finished}
                              END IF;
                              DEC(pl);
                    END WHILE;
{after probe limit has expired if no suitable node has been found add job to local processing queue}
                    IF minload >= Threshold
                              jobLengthRef := job.servicetime;
                              job.servicetime := job.servicetime/nodepower;
                              ASK jobQ TO Add(job);
                              ASK sig TO Release;
{otherwise send  to least busy node found}
                    ELSIF minload > 0
                              INC(job.transfertag);
                              job.destination := mindest;
                              Interrupt(nodearray[destination], "ExecuteJob");
                              Interrupt(nodearray[nodeID],"ExecuteJob");
                              ASK nodearray[job.destination] TO ReceiveJob(job);
{Transmit job to selected node}
                              WAIT DURATION (TDelay + 0.001) END WAIT;
                    END IF;
          ELSE
                    WAIT FOR lpsig TO Fire;
                    END WAIT;
          END IF;
          END LOOP;

END METHOD;

{--------------------------------------------------------------------------------------------------------}
TELL METHOD ProcessHETRO;{hetro ALGORITHM 4}
{This version works the same way as Shortest but instead of raw ready to run queue length a value weighted by the
respective powers of the nodes concerned is used}

VAR
          numarr : ArrayType;
          pl, destination, mindest, sent : INTEGER;
          minload, load : REAL;
          job : JobType;
BEGIN
          NEW(numarr, 0..ProbeLimit);
          LOOP
          IF lpQ.numberIn > 0
```

```
                        job := ASK lpQ TO Remove();
                        sent := 0;
                        pl := ProbeLimit;
                        UniqueRandom(ProbeLimit,nodeID,numarr);
                        minload := FLOAT(Threshold);
                        WHILE pl > 0
                                destination := numarr[pl];
                                Interrupt(nodearray[destination], "ExecuteJob");
                                Interrupt(nodearray[nodeID], "ExecuteJob");
                                WAIT DURATION probingDelay END WAIT;
                                load        :=        FLOAT(ASK        nodearray[destination]        jobQ.numberIn
)*(nodepower/nodearray[destination].nodepower);
                                IF minload > load;
                                        minload := load;
                                        mindest := destination;
                                END IF;
                                IF minload = 0.0
                                        INC(job.transfertag);
                                        job.destination := mindest;
                                        Interrupt(nodearray[job.destination], "ExecuteJob");
                                        Interrupt(nodearray[nodeID], "ExecuteJob");
                                        ASK nodearray[job.destination] TO ReceiveJob(job);
                                        WAIT DURATION (TDelay + 0.001) END WAIT;
                                        EXIT;
                                END IF;
                                DEC(pl);
                        END WHILE;
                        IF minload >= FLOAT(Threshold)
                                jobLengthRef := job.servicetime;
                                job.servicetime := job.servicetime/nodepower;
                                ASK jobQ TO Add(job);
                                ASK sig TO Release;
                        ELSIF minload > 0.0
                                job.destination := mindest;
                                Interrupt(nodearray[mindest], "ExecuteJob");
                                Interrupt(nodearray[nodeID], "ExecuteJob");
                                {INC(job.transfertag);}
                                ASK nodearray[job.destination] TO ReceiveJob(job);
                                WAIT DURATION (TDelay + 0.001) END WAIT;
                        END IF;
                        ELSE
                                WAIT FOR lpsig TO Fire;
                        END WAIT;
                        END IF;
                END LOOP;
END METHOD;


{-----------------------------------------------------------------------------------------------------------------------}
TELL METHOD ProcessHETQL;
{ Similar in operation to the HETRO method the difference lying in the the use of local load queue length instead of a
threshold value in the location policy. }

VAR
        numarr : ArrayType;
        pl, destination, mindest, sent : INTEGER;
        minload , load : REAL;
        job : JobType;
BEGIN
        NEW(numarr, 0..ProbeLimit);
        LOOP
        IF lpQ.numberIn > 0
                job := ASK lpQ TO Remove();
                sent := 0;
                pl := ProbeLimit;
                UniqueRandom(ProbeLimit,nodeID,numarr);
                minload := FLOAT(jobQ.numberIn);
                WHILE pl > 0
                        destination := numarr[pl];
                        Interrupt(nodearray[destination], "ExecuteJob");
                        Interrupt(nodearray[nodeID], "ExecuteJob");
                        WAIT DURATION probingDelay END WAIT;
                        load        :=        FLOAT(ASK        nodearray[destination]
jobQ.numberIn)*(nodepower/nodearray[destination].nodepower);
```

```
                    IF load = 0.0
                            sent := 1;
                            INC(job.transfertag);
                            job.destination := destination;
                            Interrupt(nodearray[destination], "ExecuteJob");
                            Interrupt(nodearray[nodeID], "ExecuteJob");
                            ASK nodearray[job.destination] TO ReceiveJob(job);
                            WAIT DURATION (TDelay + 0.001) END WAIT;
                            EXIT;
                    ELSIF minload > load
                            minload := load;
                            mindest := destination;
                            sent:=2;
                    END IF;
                    DEC(pl);
            END WHILE;
            IF sent = 2
                    INC(job.transfertag);
                    job.destination := mindest;
                    Interrupt(nodearray[mindest], "ExecuteJob");
                    Interrupt(nodearray[nodeID], "ExecuteJob");
            ASK nodearray[job.destination] TO ReceiveJob(job);
            WAIT DURATION (TDelay + 0.001) END WAIT;
            ELSIF sent = 0
                    jobLengthRef := job.servicetime;
                    job.servicetime := job.servicetime/nodepower;
                    ASK jobQ TO Add(job);
                    ASK sig TO Release;
            END IF;
    ELSE
            WAIT FOR lpsig TO Fire;
            END WAIT;
            END IF;
    END LOOP;
END METHOD;
{----------------------------------------------------------------------------------------------------------------------}
```

{The queue length at the local node is used rather than a fixed threshold in the location decision,a form of bias is implemented but only in the sense that the execution times at each node are compared transfer occuring if a remote node has a shorter predicted execution time.Immediate transfer to an idle node is not possible, the full probe limit is used and only then is the location decision made }

```
TELL METHOD ProcessHQNIT;
VAR
        numarr : ArrayType;
        pl, destination, mindest, sent : INTEGER;
        minload , load : REAL;
        job,tempjob : JobType;
BEGIN
        LOOP
                IF lpQ.numberIn > 0
                        job := ASK lpQ TO Remove();
                        sent := 0;
                        NEW(numarr, 0..ProbeLimit);
                        pl := ProbeLimit;
                        UniqueRandom(ProbeLimit,nodeID,numarr);
                        {local load incremented by 1 to account for eligible job}
                        minload := FLOAT(jobQ.numberIn + 1);
                        WHILE pl > 0
                                destination := numarr[pl];
                                Interrupt(nodearray[destination], "ExecuteJob");
                                Interrupt(nodearray[nodeID], "ExecuteJob");
                                WAIT DURATION probingDelay END WAIT;
                                {remote load calculated with the eligible job accounted for}
                                load    :=    FLOAT(ASK    nodearray[destination]    jobQ.numberIn    +    1
)*(nodepower/nodearray[destination].nodepower);
                                {Update best possible destination if suitable node found}
                                IF minload > load
                                        minload := load;
                                        mindest := destination;
                                        sent:=1;
                                END IF;
                                DEC(pl);
                        END WHILE;
{if a suitable node has been discovered (sent =1) the the eligible job is dispatched to it}
```

```
                    IF sent = 1
                              INC(job.transfertag);
                              job.destination := mindest;
                              Interrupt(nodearray[mindest], "ExecuteJob");
                              Interrupt(nodearray[nodeID], "ExecuteJob");
                              ASK nodearray[job.destination] TO ReceiveJob(job);
                              WAIT DURATION 0.031 END WAIT; {Transmit Delay}
                    ELSIF sent = 0
                              jobLengthRef := job.servicetime;
                              job.servicetime := job.servicetime/nodepower;
                              ASK jobQ TO Add(job);
                              ASK sig TO Release;
                    END IF;
                    DISPOSE(numarr);
          ELSE
          WAIT FOR lpsig TO Fire
          END WAIT;
          END IF;
     END LOOP;
END METHOD;

{----------------------------------------------------------------------------------------------------------------}
{This method called by transmit, adds a job to a nodes recieve queue and releases a trigger to tell the node to examine its
recieve queue if it is not currently doing so. This  method is needed to buffer jobs they are put in an orderly queue by
receive. }

ASK METHOD ReceiveJob ( IN job : JobType);

BEGIN
{This is where the original length is collected and new servicetime calculated, for all transferred jobs}
          jobLengthRec := job.servicetime;
          job.servicetime := job.servicetime/nodepower;
          ASK rxQ TO Add(job);
          ASK rXsig TO Release;
END METHOD;

{----------------------------------------------------------------------------------------------------------------}
{ This method is constantly running and processes the contents of a nodes transmit queue, or waits for a trigger to signal
that a job has entered the transmit queue. On interrupt the WAIT is exited enabling the tXsig trigger to be DISPOSED of}

TELL METHOD Transmit;

VAR
          job : JobType;
BEGIN
          LOOP
                    IF txQ.numberIn > 0
                              job := ASK txQ TO Remove();
                              ASK nodearray[job.destination] TO ReceiveJob(job);
                              WAIT DURATION TDelay
                              ON INTERRUPT
                                        EXIT;
                              END WAIT;
                    ELSE
                              WAIT FOR tXsig TO Fire
                              job := ASK txQ TO Remove();
                              ASK nodearray[job.destination] TO    ReceiveJob(job);
                              WAIT DURATION TDelay END WAIT;
                              ON INTERRUPT
                                        EXIT;
                              END WAIT;
                    END IF;
          END LOOP;
END METHOD;

{----------------------------------------------------------------------------------------------------------------}
{ This method is constantly running and processes the contents of a nodes recieve queue, or waits for a trigger to signal
that a job has entered the recieve queue. When a job is recieved it is passed to the ExecuteJob method of that node to be
executed, or transferred if threshold is exceeded and transfertag limit is not. On interrupt the WAIT is exited enabling the
rXsig trigger to be DISPOSED of. In this version the idea of a transfer tag is not implemented, jobs must be implemented on
the node they are transferred to.}

TELL METHOD Receive;
VAR
```

```
        job : JobType;
BEGIN
        LOOP
                IF rxQ.numberIn > 0
                        job := ASK rxQ First();
                        WAIT DURATION TDelay END WAIT;
                        ASK jobQ TO Add(job);
                        ASK sig TO Release;
                        job := ASK rxQ TO Remove();
                ELSE
                        WAIT FOR rXsig TO Fire
                        job := ASK rxQ First();
                        WAIT DURATION TDelay END WAIT;
                        ASK jobQ TO Add(job);
                        ASK sig TO Release;
                        job := ASK rxQ TO Remove();
                        ON INTERRUPT
                                EXIT;
                        END WAIT;
                END IF;
        END LOOP;
END METHOD;

{----------------------------------------------------------------------------------------------------------------}
TELL METHOD ExecuteJob;
{ This method runs continuously simulating the execution of jobs as they reach the node. This is the FCFS version where
jobs are executed sequentially. Theeffect of having to deal with RPC activity is implemented by adding the delayto the
unexpired job servicetime}

VAR
        job : JobType;
        intChk : INTEGER;
        stopTime, startTime : REAL;
BEGIN
        LOOP
        IF jobQ.numberIn > 0
                job := ASK jobQ First;
{continue until job service time is fully expired}
                REPEAT
                        startTime := SimTime();
                        WAIT DURATION job.servicetime
                                intChk := 0;
{continue until job service time is fully expired}
                        ON INTERRUPT
                                stopTime := SimTime();
{ recalculate unexpired servicetime}
                                job.servicetime := job.servicetime-stopTime+startTime+probedDelay;
                                intChk :=1;
                                END WAIT;
                UNTIL intChk = 0; { if chk = 0 job has completed}
{update statistical counters}
                responseT := SimTime() - job.arrivaltime;
                batchRT := SimTime() - job.arrivaltime;
                ASK nodearray[job.origin] TO UpdateRT(job);
                job := ASK jobQ TO Remove();
                        DISPOSE(job);
        ELSE
                WAIT FOR sig TO Fire

                ON INTERRUPT

                END WAIT;
        END IF;
        END LOOP;
END METHOD;
{----------------------------------------------------------------------------------------------------------------}
{ COMMENTED OUT TELL METHOD ExecuteJob; {ProcessMultiJob}
MULTIPROGRAMMING VERSION    This method runs continuously simulating the execution of jobs as they reach the
node, multiprogramming version

VAR
        job : JobType;
        intChk : INTEGER;
        stopTime, startTime, quantum, origQuantum : REAL;
BEGIN
        LOOP
        IF jobQ.numberIn > 0
                        currentjob := ASK jobQ First;
                                REPEAT

                        quantum := 0.10;
                        origQuantum := quantum;
                                IF currentjob.servicetime < quantum

                                quantum :=currentjob.servicetime;
                                origQuantum := quantum;
                                END IF;
                        REPEAT
                                startTime := SimTime();
                                        WAIT DURATION quantum
                                        intChk := 0;
                                ON INTERRUPT
                                        stopTime := SimTime();
                                        quantum := quantum-stopTime+startTime+probedDelay;
                                        intChk :=1;
                                END WAIT;
                        UNTIL intChk = 0;
                        currentjob.servicetime := currentjob.servicetime - origQuantum;
                        IF currentjob.servicetime = 0.0
                                responseT := SimTime() - currentjob.arrivaltime;
                                job := currentjob;
                                IF (ASK jobQ Last) <> currentjob;
                                        currentjob := ASK jobQ Next(currentjob);
                                        ASK jobQ TO RemoveThis(job);
                                        DISPOSE(job);
                                ELSIF jobQ.numberIn > 1
                                        currentjob := ASK jobQ First;
                                        ASK jobQ TO RemoveThis(job);
                                        DISPOSE(job);
                                ELSE
                                        ASK jobQ TO RemoveThis(job);
                                        DISPOSE(job);
                                END IF;
                        ELSE
                                IF (ASK jobQ Last) <> currentjob;
                                        currentjob := ASK jobQ Next(currentjob);
                                ELSIF jobQ.numberIn > 1
                                        currentjob := ASK jobQ First;
                                END IF;
                        END IF;
                UNTIL jobQ.numberIn = 0;
        ELSE
                WAIT FOR sig TO Fire
                ON INTERRUPT
                END WAIT;
        END IF;
        END LOOP;
END METHOD;
}
{------------------------------------------------------------------------------------------------------------}
{stats on jobs executed at a node are collected as well as those originating at a node}
ASK METHOD UpdateRT(IN job : JobType);

BEGIN
        nodeRT := SimTime() - job.arrivaltime;
        nodeBRT := SimTime() - job.arrivaltime;
END METHOD;

{------------------------------------------------------------------------------------------------------------}

ASK METHOD AssignID(IN i : INTEGER; IN power : REAL);
{ This method is used to initialise a node with its ID number and power}
BEGIN
        nodeID := i;
        nodepower := power;
END METHOD;

{------------------------------------------------------------------------------------------------------------}
```

```
{ At the end of each simulation the first job in the transmit queue must be removed, but as it will be in another nodes recieve
queue must not be  DISPOSED of, all other jobs in the queue can be DISPOSED of}

ASK METHOD RemoveJobs;

VAR       temp : JobType;
BEGIN
          IF txQ.numberIn > 0
                    temp := ASK txQ TO Remove();
          END IF;
          WHILE txQ.numberIn > 0
                    temp := ASK txQ TO Remove();
                    DISPOSE(temp);
          END WHILE;
END METHOD;

{-------------------------------------------------------------------------------------------------------------------}
{ This method DISPOSES of any items using up memory at the end of each simulation run.}
ASK METHOD ObjTerminate;

VAR
          i: INTEGER;
          temp : JobType;
BEGIN
{Jobs assigned for local processing are removed from the job.Q}
          DISPOSE(randomnode1);
          WHILE jobQ.numberIn > 0
                    temp := ASK jobQ TO Remove();
                    DISPOSE(temp);
          END WHILE;
          DISPOSE(jobQ);
{The first node in the system prompts a system wide removal of jobs from tx.Q's}
          IF nodeID = 1
                    FOR i:=1 TO NofN
                     ASK nodearray[i] TO RemoveJobs;
                    END FOR;
          END IF;
          DISPOSE(txQ);
{With the transmit queues empty any jobs in the rx.Q's can be removed}
          WHILE rxQ.numberIn > 0
                    temp := ASK rxQ TO Remove();
                    {OUTPUT("rxQ",nodeID);}
                    DISPOSE(temp);
          END WHILE;
          DISPOSE(rxQ);

          {DISPOSE(sig);}
          DISPOSE(rXsig);
          DISPOSE(tXsig);
END METHOD;

END OBJECT;
{-------------------------------------------------------------------------------------------------------------------}

{ This object has one method that stops the simulation although first it must interrupt certain methods in each node object to
allow the DISPOSAL of the various triggers used}

OBJECT StopAllObj;

          TELL METHOD Finish;
          VAR
                    i : INTEGER;
          BEGIN
                    FOR i := 1 TO NofN
                              Interrupt(nodearray[i],"ExecuteJob");
                              Interrupt(nodearray[i],"Receive");
                              Interrupt(nodearray[i],"Transmit");
                    END FOR;
                    StopSimulation;
          END METHOD;
END OBJECT;
{-------------------------------------------------------------------------------------------------------------------}

{ A procedure to generate a set of unique nodes to probe}
```

```
{PROCEDURE UniqueRandom(IN Probelimit : INTEGER; IN nodeID : INTEGER;
                              INOUT numarr : ArrayType);

VAR
        pl, i, temp, j  : INTEGER;
        test : BOOLEAN;

BEGIN
                pl := Probelimit;
                numarr[0] := nodeID;
                i := 1;
                WHILE i <= Probelimit
                        test := TRUE;
                        REPEAT
                         temp := ASK globalrandom UniformInt(1, NofN);
                        UNTIL temp <> nodeID;
                        FOR j := 1 TO i
                                IF temp = numarr[j-1]
                                        test := FALSE;
                                        EXIT;
                                END IF;
                        END FOR;
                        IF test = TRUE;
                                numarr[i] := temp;
                                INC(i);
                        END IF;
                END WHILE;
END PROCEDURE;}
{---------------------------------------------------------------------------------------------------------------}

{ A procedure to generate a set of unique nodes to probe}

PROCEDURE UniqueRandom(IN Probelimit : INTEGER; IN nodeID : INTEGER;
                              INOUT numarr : ArrayType);

VAR
        pl, i, temp, j  : INTEGER;
        test : BOOLEAN;
        choiceArray : ArrayType;

BEGIN
        NEW(choiceArray, 1..NofN);
        FOR i:=1 TO NofN   {initialise array to contain a set of integers}
                choiceArray[i] := i;
        END FOR;
{ensure it is impossible to pick the source node as a destination}
        choiceArray[nodeID] := 1;

        FOR i:=1 TO ProbeLimit
                temp := ASK globalrandom UniformInt( i+1, NofN ); {pick random number}
                numarr[i] := choiceArray[temp]; {put selected nodeID into array}
                choiceArray[temp] := choiceArray[i+1]; {remove selected nodeID from choice}
        END FOR;
        DISPOSE(choiceArray);

END PROCEDURE;

{---------------------------------------------------------------------------------------------------------------}
{select algorithm to use for length of run}
PROCEDURE PickAlgorithm(IN ID : INTEGER);
BEGIN
        CASE Algorithm
        WHEN 1:
                TELL nodearray[ID] TO ProcessRandom;
        WHEN 2:
                TELL nodearray[ID] TO ProcessSHORTEST;
        WHEN 3:
                TELL nodearray[ID] TO ProcessHETRO;
        WHEN 4:
                TELL nodearray[ID] TO ProcessHETQL;
        WHEN 5:
                TELL nodearray[ID] TO ProcessHQNIT;
        OTHERWISE
                OUTPUT("illegal algorithm");
                StopSimulation;
        END CASE;
END PROCEDURE;


END MODULE.
```

A1.3. Main Module

```
MAIN MODULE loadshare;

FROM Hetrodelaylib IMPORT AvServiceTime,NofN,NofDN,batchtime,
                        Threshold,TransferLimit,TDelay,
                        probedDelay,probingDelay, minSize,
                        Algorithm,ProbeLimit,AvInterArrivalTime,
                        JobType, Hetrorecord,ArrayType, HetroArray,
                        GenesisObj,NodeObj,StopAllObj;
FROM RandMod IMPORT RandomObj, FetchSeed;
FROM ListMod IMPORT StatQueueList;
FROM SimMod IMPORT SimTime, StartSimulation, StopSimulation, TriggerObj;
FROM StatMod IMPORT RStatObj, IStatObj, SREAL;


CONST

TYPE


VAR
        i,U,PLmin,PLmax,Umin, Umax, Ustep, totalnodes,seed,reps : INTEGER;
        AvResponseTime, runtime : REAL;
        genesis : GenesisObj;
        stopit : StopAllObj;
        rec : Hetrorecord;
        diffnodes : HetroArray;

BEGIN
        TransferLimit := 1;
        TDelay := 0.030;                {30 ms delay due to rpc }
        probingDelay := 0.030;                  {30 ms delay due to rpc }
        probedDelay := 0.010;                   {10 ms delay in answering rpc }

        {user input of run parameters }
        INPUT(runtime);     {runtime}
        INPUT(batchtime);
        INPUT(AvServiceTime);
        INPUT(Threshold);
        INPUT(Algorithm);
        INPUT(NofN);
        INPUT(NofDN);



        NEW(diffnodes, 1..NofDN);       {initialise node array}
        FOR i:=1 TO NofDN               {for each different group of nodes}
                NEW(rec);
                diffnodes[i] := rec;
                INPUT(diffnodes[i].numberofnodes);      {user input expected number of nodes in group}
                totalnodes := totalnodes + diffnodes[i].numberofnodes;
                IF (totalnodes > NofN);
                        OUTPUT("TOO MANY NODES < START AGAIN");
                        HALT;
                END IF;
                INPUT(diffnodes[i].powerofnodes);       {user input expected power of nodes in group}

                INC(i);
        END FOR;
        {user input more run time parameters
        INPUT(PLmin);                           }
        INPUT(PLmax);
        INPUT(reps);
        INPUT(Umin);
        INPUT(Umax);
        INPUT(Ustep);

        FOR U := Umin TO Umax BY Ustep;
        FOR ProbeLimit := PLmin TO PLmax;
        FOR seed := 1 TO reps;
                AvInterArrivalTime := 1000.0/FLOAT(U);
                NEW(genesis);
                NEW(stopit);
                TELL genesis TO InitialiseNodes(diffnodes, seed, batchtime);
                TELL stopit TO Finish IN runtime;
                StartSimulation;
                AvResponseTime := ASK genesis TO PerfStats();
                OUTPUT("Overall RT at PL",ProbeLimit," Utilisation ",U,"% = ",AvResponseTime);
                DISPOSE(genesis);
                DISPOSE(stopit);
        END FOR;
        END FOR;
        END FOR;

        OUTPUT();OUTPUT();
END MODULE.
```

Appendix 2. Implementation Code

A2.1 Generatejobs.c

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <math.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stddef.h>
#include <signal.h>
#include "hetro.h"
#include <sys/utsname.h>



double drand48();
void srand48();
double log();
int shmid, jobsgen = 0;
char *shmaddr;
pid_t pid, pid_sp, pid_rs;
struct utsname name;

static void sigusr1(); /* signal functions */
static void sig_usr2();

void main()

{
record *data_ptr, *st_seg, *end_seg;
double ta, seed;
long SRseed;
int node_id,temp;
float mean_ta,time,node_power;
struct timespec tv;
FILE *fptr;

        if((pid_sp = vfork())<0)        /* create RPC server that services probes */
                exit(1);
        if(pid_sp == 0)
                execlp("./serveprobe","serveprobe",(char *) 0);
        if((pid_rs = vfork())<0)        /* create RPC server that services remote execution */
                exit(1);
        if(pid_rs == 0)
                execlp("./remxserver","remxserver",(char *) 0);


        uname(&name);                   /* retreive node description */

        signal(SIGUSR1, sigusr1);       /* catch death of processjobs */
        signal(SIGUSR2, sig_usr2);      /* catch synchronisation signal */

        get_time(&seed);                                        /* use seed based upon clocktime */
        SRseed = (long)seed / *(name.nodename);
        SRseed = SRseed / *((name.nodename)+1);                 /* ensure seed is unique */
        SRseed = SRseed * *((name.nodename)+2);

        /*
        fptr = fopen("seedNo","a");
        fprintf(fptr,"%s seed = %d\n",name.nodename,SRseed);
        fclose(fptr);
        */

    if(strcmp(name.machine,"sun4c")==0)                 /* Assign node_power */
        node_power=0.395;

    if(strcmp(name.machine,"sun4m")==0)
        node_power=1.405;

/*set mean interarrival time to be proportional to node power */
        mean_ta = (MEAN_TS / UTIL) / node_power;

        srand48((int)SRseed);               /* initialisation entry point for random number generator */

/*attach shm segment using default values for shmaddr and shmflg to allow compiler to decide location*/
        if ((shmid = shmget (SEG_KEY, SEG_SIZE, SEG_EXCL)) == -1)
                shmid = shmget(SEG_KEY,SEG_SIZE,SEG_PERM);
        shmaddr = shmat(shmid,0,0);

/* struture pointer assigned to start of shm segment */
        data_ptr =(record *)  shmaddr;

        data_ptr = data_ptr + 1;
        data_ptr->ts = 0;           /* make sure location empty */
        st_seg = data_ptr;          /* fix start of segment */
        end_seg = st_seg +300;      /* fix end of segment */

        if((pid = vfork())<0)       /* create processjobs process */
                exit(1);
        if(pid == 0)
                execlp("./processjobs","processjobs",(char *) 0);
        pause();            /* wait for signal that processjobs has been sucessfully created */

        while(1)    /* endless loop to generate jobs */
        {
                if (data_ptr >= end_seg)            /* if end of segment */
                        data_ptr = st_seg;          /* go back to start */

                ta = -mean_ta * log(drand48()); /* calculate exponentially distributed ta */
                tv.tv_sec = (long) ta;

                tv.tv_nsec = (long) ((ta - tv.tv_sec) * 1000000000 ); /* convert to nanoseconds*/

                nanosleep(&tv, NULL );          /*sleep for ta*/

                /* calculate exponentially distributed ts and store in shm */
                data_ptr->ts = -MEAN_TS * log(drand48());

                get_time(&(data_ptr->starttime));           /* store job starttime*/


                data_ptr++;                                 /* increment pointer to next location */
                data_ptr->ts = 0.0;                         /*make sure location empty */

                kill(pid,SIGUSR1);                          /* send signal to proccessjobs */
                jobsgen++;

        }

}

static void sigusr1(signo)      /* signal handler to catch end of run signal from processjobs */
int signo;
{
FILE *fptr;
int statloc;

if(signo == SIGUSR1)            /* check signal type */
        {
        fptr=fopen("genjobsresults","a");                   /* prints jobs generated stats to file */
        fprintf(fptr,"Hostname = %s\n",name.nodename);
        fprintf(fptr,"Jobs generated = %d\n",jobsgen);
        fclose(fptr);
```

```
            _exit(0);                                          /* stops process */
            }
}

/* signal handler to catch synchronisation signal from processjobs */
static void sig_usr2( int signo)
{
if(signo == SIGUSR2)
            return;
}
```

A2.2 Processjobs.c

```
#include <sys/types.h>                                /* PROCESSJOBS.C*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <math.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <signal.h>
#include <sys/wait.h>
#include "hetro.h"
#include <rpc/rpc.h>
#include "Probe.h"
#include <sys/utsname.h>


int shmid, remex, localex, transct, lsgct;
char *shmaddr;
record *data_ptr;
int *local_load, jobfinished, jobcount, NumSUnreachable, NumTimeout,fd, *remjob_rec;
member **list_ptr, *list;
float overallrt, overallts, minload,SUnreachts, Timeoutts,node_power;
struct utsname name;
comp_memb *fin_list_ptr;

static void sig_usr1();                        /* signal handlers */
static void sigchld();
static void sigalrm();


void main()

{
record *st_seg, *end_seg, job_details;
pid_t pid;
char str_ptr[20],*destination_node,**d_node_ptr, pathname[30];
int transfer,temp_local_load;
FILE *fptr;
long lock_size = 0;
pid_t ppid;

d_node_ptr = &destination_node;
list  = NULL;                        /* pointer to start of list */
list_ptr = &list;           /* list_ptr points to address of list, to allow manipulation of address' */
fin_list_ptr = NULL;

            signal(SIGCHLD, sigchld);                  /* catch death of child signals */
            signal(SIGUSR1, sig_usr1);                 /* catch signals to the process */
            signal(SIGALRM, sigalrm);                  /* catch alarm signals */

/* try to create shm segment, if it is already in existance get shmid */
            if ((shmid = shmget (SEG_KEY, SEG_SIZE, SEG_EXCL)) == -1)
                        shmid = shmget(SEG_KEY,SEG_SIZE,SEG_PERM);

/* attach shm using default values for shmaddr and shmflg to allow compiler to decide location*/
            shmaddr = shmat(shmid,0,0);

            data_ptr = (record *) shmaddr;  /* make sure location empty */
            local_load = (int *) shmaddr;              /* get area for storage of local load*/
```

```
        *local_load = 0;

        remjob_rec = local_load +1;
        *remjob_rec = 0;

        data_ptr = data_ptr + 1;
        st_seg = data_ptr;              /* fix start of segment, use first location for local load */
        end_seg = st_seg +300;          /* fix end of sement */

        alarm(REPORT_TIME);             /* set alarm for next report period */

        uname(&name);                                               /* get host details*/
    if(strcmp(name.machine,"sun4c")==0)                             /* assign relevant node_power */
        node_power=0.395;
    if(strcmp(name.machine,"sun4m")==0)
        node_power=1.405;


        ppid = getppid();
        kill(ppid,SIGUSR2);

        while(1)            /* endless loop getting job details from shm */
        {
                if(data_ptr >= end_seg)         /* if end of segment */
                        data_ptr = st_seg;      /* go back to start */
                if(data_ptr->ts == 0.0)         /* if record null no new jobs have been created*/
                        {
                        pause();                /* pause until signal */
                        }
                else
                        {
                        sprintf(str_ptr,"%f", data_ptr->ts);        /* get job duration */
                        minload = 1 +(float) *local_load;           /* update minload */

                        transfer = 0;

                        if(*local_load > 0 )                        /* Invoke load sharing */
                                {
                                lsgct++;
                                transfer = lsalg(d_node_ptr);
                                }
                        if( transfer == 1 )                         /* execute job remotely */
                                {
                                transct++;
                                data_ptr->ex_loc = 1;
                                if((pid = vfork())< 0)          /* if fork fails*/
                                        {
                                        fptr=fopen("errorfile","a");
                                        fprintf(fptr,"%s exit1\n",name.nodename);
                                        fclose(fptr);
                                        _exit(1);
                                        }
                                if(pid == 0)
                                        {
                                        execlp("./remxclient","remxclient", *d_node_ptr,str_ptr,(char *) 0);
                                                /* initiate remote execution for job */
                                        }
                                }
                        else            /* execute job locally */
                                {
                                (*local_load)++;        /*increment local load */
                                data_ptr->ex_loc = 0;
                                if((pid = vfork())< 0)          /* if fork fails*/
                                        {
                                        fptr=fopen("errorfile","a");
                                        fprintf(fptr,"%s exit2\n",name.nodename);
                                        fclose(fptr);
                                        _exit(1);
                                        }
                                if(pid == 0)
                                        {
                                        /* spawn process to execute for job length */
                                        execlp("./executejob","executejob",str_ptr,(char *) 0);
                                        }
                                        }
                                jobcount++;
/* add job details to linked list for later processing */
                                add_to_list(list_ptr,pid,data_ptr);
                                data_ptr->ts = 0.0;                     /* set record to null */
                                data_ptr++;                                     /* go to next record */
                                }
                }
}

/* signal handler to catch SIGUSR1 from generatejobs, does not perform any other function */
static void sig_usr1(signo)
int signo;
{

if(signo == SIGUSR1)

                printf("");

}

/* signal handler to catch death of child signals from terminating executejob processes */
static void sigchld(signo)
int signo;
{

comp_rec c_rec,*c_rec_ptr;
int statloc;
double time;
FILE *fptr;

c_rec_ptr = &c_rec;

if(signo == SIGCLD)

                {
                /* get pid of terminated process and store in termination record*/
                c_rec_ptr->pid = wait(&statloc);

                /* determine exit status of terminated process and store in termination record */
                c_rec_ptr->estatus = (int)(WEXITSTATUS(statloc));

                jobfinished++;          /* increment job finished count */

                if(*local_load < 0)     /* should load fall below 0 record fact in error file */
                        {
                        fptr=fopen("errorfile","a");
                        fprintf(fptr,"%s load corrupted %d\n",name.nodename,*local_load);
                        fclose(fptr);
                        }
                /* get current time and store in termination record */
                get_time(&time);
                c_rec_ptr->stoptime = time;
                /* put termination record in linked list */
                add_to_fin_list(&fin_list_ptr,c_rec_ptr);
                }
}

/* signal handler to run set routines at alarm periods */
static void sigalrm(signo)
int signo;

{
double tempRT=0 , tempTS=0, totalRT=0, totalTS=0, repRT=0, repTS=0;
double endtime;
char pathname[30],pathnamein[30],pathnameout[30];
unsigned int repJF, totalJF=0, repjobs;
static int totaltime = 0, rp =0;
FILE *fptr,*resptrin, *resptrout;
record *rec, r;
comp_rec *c_rec, cr;
member *lp;
comp_memb *flp;
pid_t ppid;
```

```
c_rec = &cr;
rec = &r;

if(signo == SIGALRM)
        {
        /* initialise pathnames for results */
        strcpy(pathname,"results/");
        strcpy(pathnamein,"results/");
        strcpy(pathnameout,"results/");
        strcat(pathname,name.nodename);
        strcat(pathnamein,name.nodename);
        strcat(pathnameout,name.nodename);
        strcat(pathnamein,"in");
        strcat(pathnameout,"out");

        fptr=fopen(pathname,"a");                           /* open results file for node */
        fprintf(fptr, "%s",name.nodename);                  /* nodename */
        fprintf(fptr," Rep%d ", rp);                        /* repetition number */
        fprintf(fptr,"Jrec = %d ",jobcount);                /* jobs received */
        fprintf(fptr,"Jfin = %d ",jobfinished);             /* jobs finished */
        fprintf(fptr,"curr load = %d",*local_load);        /* current load */
        fprintf(fptr, " rem jobs rec =%d",*remjob_rec);  /* jobs transferred from other nodes */
        fprintf(fptr, "lsg = %d tct = %d\n",lsgct, transct);  /*  times load sharing invoked */
        fclose(fptr);
        totaltime+= REPORT_TIME;                            /* increment time passed */
        rp++;                                                       /* increment report period */
        if(totaltime == RUN_TIME)                           /* if run time expired */
                {

                get_time(&endtime);
                ppid = getppid();               /* get id of generatejobs */
                kill(ppid,SIGUSR1);             /* send signal to generatejobs */

                flp = fin_list_ptr;     /* initialise pointer to start of finished jobs linked list */
                lp = *list_ptr;                 /* initialise pointer to start of created jobs linked list */
                resptrin=fopen(pathnamein,"a");       /* open file to write input jobs records to */
                while(lp != NULL)                     /* while linked list not empty */
                        {
                        delete_from_list2(&lp,rec);     /* copy record from linked list */
                                                                /* write record to file */
                        fprintf(resptrin," %lf\n %d\n %lf\n ", rec->ts, rec->pid, rec->starttime);
                        }
                fclose(resptrin);

                resptrout=fopen(pathnameout,"a"); /* open file to write finished job records to */
                while(flp != NULL)                    /* while linked list not empty */
                        {
                        delete_from_fin_list(&flp,c_rec); /* copy record from linked list */
                                                                /* write record to file */
                        fprintf(resptrout," %lf\n %d\n %d\n ", c_rec->stoptime, c_rec->pid, c_rec->estatus);
                        }
                fclose(resptrout);
                }
        else            /* if run time not reached */
                alarm(REPORT_TIME);             /* reset alarm to end of next report period */

        }
}
```

/ lsalg is the function that actually carries out load sharing. If an appropriate node is discovered then destination_node is changed to point to it. Otherwise the value will remain as NULL. This function sends out probes in the form of RPC's to randomly selected nodes. The results of these probes, load and power are used to generate a weighted load, which is then compared to the local load or lowest weighted load so far discovered. Should an RPC fail for any reason it is ignored and the next one is started.*/*

```
int lsalg(node_ptr)
char **node_ptr;

{
float weighted_load;
char *hostnames[PROBE_LIMIT];
int ct,dummy,transfer;
CLIENT *ClientHandle;
Data *results;
struct timeval tv;
char *ptc;

ptc = (char*) &tv;                              /* pointer to time structure */

transfer = 0;
random_nodes(hostnames);                        /* get randomly picked nodes*/
for(ct=0;ct<PROBE_LIMIT;ct++)                   /* for set number of probes */
        {
        /* create client handle, contacts remote portmapper and gets tcp port for server */
        ClientHandle = clnt_create(hostnames[ct], PROBEPROG, PROBEVERS, "tcp");

        if(ClientHandle != NULL)                /* if remote portmapper contacted successfully*/
                {
                tv.tv_sec = PROBE_TIMEOUT;
                tv.tv_usec = 0;
                clnt_control(ClientHandle, CLSET_TIMEOUT, ptc); /* set probe timeout */
                results = getinfo_1(&dummy,ClientHandle);        /* initiate remote procedure call*/
                if(results != NULL)             /* if remote procedure seccessfully completed */
                        {
                        if(results->load < 0)                   /* just in case load is negative */
                                results->load = 0;
                /* calculate weighted load */
                        weighted_load = (node_power/results->power)*(results->load + 1);
                        if(minload > weighted_load)     /* if currently probed node is least loaded */
                                {
                                minload = weighted_load;        /* new minimum */
                                *node_ptr = hostnames[ct];      /* new destination */
                                transfer = 1;                           /* transfer on */
                                }
                        }
                clnt_destroy(ClientHandle);                     /* remove client handle */
                }
        }
return(transfer);                                               /* return transfer decision */

}
```

A2.3. Executejob.c

```
#include <stdio.h>
#include <stdlib.h>
#include "hetro.h"
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/utsname.h>

double atof();

main(argc, argv)
int argc;
char *argv[];

{
record *data_ptr;
double ts;
int shmid,*local_load,loopno,fd;
long lock_size = 0;
char *shmaddr;
int         a,b,c,ct,count;
struct utsname name;

/* attempt to create shared memory segment  if it is already in existance get the segment id */
if((shmid = shmget (SEG_KEY, SEG_SIZE, SEG_EXCL)) == -1)
        shmid = shmget(SEG_KEY,SEG_SIZE,SEG_PERM);

/* attach the shared memory segment */
shmaddr = shmat(shmid,0,0);
/* initialise local load */
local_load = (int*) shmaddr;

/* get service time of job from arguement passed to process */
ts = atof(argv[1]);
/* convert service time to correct length , 60 loops = 1 second on machine of rating 1*/
ts = ts * 60;
/* loopno must be an integer */
loopno = (int) ts;

/* work loops to use system time */
for(count=0;count<loopno;count++)
        {

        for(ct=0;ct<10000;ct++)
                {
                a++;
                b++;
                c++;
                a++;
                b++;
                c++;
                a++;
                b++;
                c++;
                a++;
                b++;
                c++;
                a++;
                b++;
                c++;
                a++;
                b++;
                c++;
                }
        }

 /* decrement local load */
 (*local_load)--;

 /* exit with status 0 */
 _exit(0);

 printf("exit failed");
 }
```

A2.4. Serveprobe.c

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "Probe.h"
#include "hetro.h"
#include <sys/utsname.h>

Data *getinfo_1(dummy)
int *dummy;

{
static int test = 0;
static Data results;
static int *load_ptr,fd;
int shmid;
char *shmaddr;
struct utsname name;
static float node_power;

if(test!=1)  /* on the first call to the procedure */
        {
        /* try to create shm segment, if it is already in existance get shmid */
        if((shmid = shmget (SEG_KEY, SEG_SIZE, SEG_EXCL)) == -1)
                shmid = shmget(SEG_KEY, SEG_SIZE, SEG_PERM);
        shmaddr = shmat(shmid,0,0);  /* attach the shared memory segment */
```

```
            load_ptr = (int *)shmaddr;              /* initialise local load */

            test = 1;
            uname(&name);                                       /* get host details */
            if(strcmp(name.machine,"sun4c")==0)     /* assign relevant node power */
                        node_power=0.395;
            if(strcmp(name.machine,"sun4m")==0)
                        node_power=1.405;
            }

results.load  = *load_ptr;                                  /* put load in results structure */
results.power = node_power;                         /* put power in results structure */

return &results;                                            /* return results structure */

}
```

A2.5. Remxclient.c

```
#include <rpc/rpc.h>
#include <stdio.h>
#include <stdlib.h>
#include "remexec.h"
#include "hetro.h"

main(argc,argv)

int argc;
char *argv[];
{


CLIENT *ClientHandle;
char *nodename = argv[1], *shmaddr, *ptc;
float ts;
int *result, shmid, timeout, load;
struct timeval tv;
static int test, *local_load;

ptc = (char*) &tv;

/* convert servicetime string to float */
ts = (float)atof(argv[2]);

/* create client handle, contacts remote portmapper and gets tcp port for server */
ClientHandle = clnt_create(nodename, REMEXECPROG, REMEXECVERS, "tcp");
/* if remote portmapper contacted successfully*/
if(ClientHandle != NULL)
            {
            tv.tv_sec = RUN_TIME;                   /* timeout set to run time */
            tv.tv_usec = 0;
            clnt_control(ClientHandle, CLSET_TIMEOUT, ptc); /* set execution timeout */
            result = remproc_1(&ts,ClientHandle);    /* initiate remote procedure call*/
            if(result!=NULL)                                    /* if RPC successful */
                        {
                        /* remove client handle */
                        clnt_destroy(ClientHandle);
                        _exit(1);               /* exit with status for successful remote  job execution */

                        }
            _exit(2);                   /*exit with status indicating  timeout */
            }
 _exit(3);                              /* exit with status indicating server unreachable */
}
```

A2.6. Remxserver.c

```
#include <rpc/rpc.h>
#include <stdio.h>
#include "remexec.h"
```

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <signal.h>
#include <sys/wait.h>
#include "hetro.h"
#include <sys/utsname.h>

int *remproc_1(ts)
float *ts;

{
int *statloc;
int result = 1;
static int *local_load, *remjob_rec, fd;
char str_ptr[20];
static int test;
int shmid;
char *shmaddr, pathname[30];
pid_t pid;
struct utsname name;

if(test!=1) /* on first call of procedure */
        {
        /* try to create shm segment, if it is already in existance get shmid */
        if((shmid = shmget (SEG_KEY, SEG_SIZE, SEG_EXCL)) == -1)
                shmid = shmget(SEG_KEY, SEG_SIZE, SEG_PERM);
        /* attach the shared memory segment */
        shmaddr = shmat(shmid,0,0);
        /* initialise local load */
        local_load = (int *)shmaddr;
        /* initialise remote jobs received count */
        remjob_rec = (int *)shmaddr;
        remjob_rec = remjob_rec +1;

        test = 1;
        uname(&name); /* get host details */
        }

sprintf(str_ptr,"%f", *ts);        /* convert servicetime to string format */

 (*local_load)++;                   /* increment local load */
(*remjob_rec)++;                    /* increment remote jobs received count */

if((pid = vfork())< 0)  /* fork new process */
        {printf("fork error, pid = %d\n", pid);
        exit(1);}
if(pid == 0)
        {
        execlp("executejob","executejob",str_ptr,(char *) 0);
         /* spawn process to execute for ts */
        }
pid = wait(statloc);    /* wait for child process to terminate */

return(&result);        /* return to calling process */
}
```

Appendix 3. Conference Papers Based on This Work

Gold Coast, Australia, 1996

IASTAD International Conference on Modelling, Simulation and Optimization

242-031.pdf

6-9 May 1996.

Manchester, England, 1995

Second Communication Networks Symposium,

Pages 265-9,

10-11 July 1995.

Performance Evaluation of Load Sharing Algorithms in Heterogeneous Systems

R.Leslie & S.Mckenzie
University of Greenwich
Wellington St, London SE18 6PF
Fax: 0181-331-8665
Tel: 0181-331-8669
email: r.leslie@gre.ac.uk, s.mckenzie@gre.ac.uk

Abstract.
This paper considers load sharing in heterogeneous systems where the heterogeneity is exhibited in the processing power of the constituent nodes. An algorithm is proposed that considers both the relative processing power of the nodes and their current load in its location policy. Versions of this algorithm using both threshold and local load based location policies are assessed. The information policy used is based on probing and the use of different probe limits is investigated. The algorithms are evaluated by simulation, on a model of 20 diskless workstations with differing processing powers. Various systems were modelled, all of 20 nodes and identical overall capacity but varying in their degree of heterogeneity. The performance of all versions of the algorithm are compared against an existing algorithm as well as theoretical upper and lower bounds. Results show the importance of considering relative processing power in algorithms for heterogeneous systems. Unlike homogeneous systems threshold based location policies are not as efficient as those using a variable comparison factor.

Keywords:
Load Sharing, Distributed Systems, Simulation, Performance.

1. Introduction.

In a distributed system there is a high probability that at any point in time some of its nodes will be highly utilised whilst others will be idle or lightly loaded. By using the ability of distributed systems to execute jobs at other than their originating node, work can be transferred from one node to another in order to achieve an improvement in overall system performance. This approach to system performance enhancement is referred to as load sharing or load balancing [1-5].

Load balancing has been used to refer to algorithms that attempt to equalise workload amongst the nodes, whilst load sharing algorithms attempt to ensure no node is idle. In this paper the term load sharing will be used in a broader sense, namely attempts to improve system performance by redistributing some of the workload. It will be shown that when considering heterogeneous systems, transferring to an idle node is not always the optimum solution nor is attempting to equalise the load at each node.
Earlier work [6] has shown that considering the relative power of the nodes in a heterogeneous system results in performance

improvement. In this paper a number of different systems, varying in degree of heterogeneity, are investigated with the use of a simulation model. Performance is measured by the average response time of the system which is accepted as the most important, but not the only, measure of performance[4]. In real time systems the best measure of performance is percentage of jobs lost [7].

The use of thresholds is a common feature in many load sharing algorithms. A new metric based upon local load is introduced. This is shown to offer advantages both in performance and scalability.

The algorithms proposed are based upon the use of the Remote Procedure Call (RPC) as a means of communication and resource sharing between nodes. The cost of the RPC's i.e. in delay experienced at both nodes involved, is included in the simulations. The delay experienced in gaining perfect knowledge of the system is prohibitive. So the optimum number of RPC's or probes that should be used is investigated, at various utilisations. It is shown that load sharing decisions can be based on limited system state information over a range of degrees of heterogeneity.

2. Load Sharing Algorithms.

Load sharing algorithms can be static or dynamic [1,8]. The algorithms investigated in this paper are dynamic and distributed.

It is assumed that the nodes comprising the systems investigated are multiprogrammed machines. The cost of process migration once a job has started execution can be excessive and is an operation that is difficult to implement on many current systems. So the algorithms studied are sender-initiated [9], where any load-sharing is implemented on the initial arrival of a job to the system.

By convention load sharing algorithms are described by dividing them into separate policies, as first introduced in [2]. Three policies are normally used: Transfer, Information and Location.

The Transfer policy controls which jobs should be made eligible for transfer. The most widely used means of establishing when a job should be eligible for transfer is to use a threshold based upon queue length. Should the arrival of a new job cause the queue for processing at the node to exceed the set threshold then that job is eligible for transfer. A limit to the number of times a job can be transferred is needed to prevent the problem of thrashing [10]. In this study a transfer limit of one is used. The Information policy describes the means by which system state information is disseminated amongst the nodes. In this study the Information policy is based on probing. Each node will send up to a set number(probe limit) of probes to other randomly picked nodes asking for their current loading. Probing is only carried out on the arrival of an eligible job, ensuring that the information retrieved is as current as possible. The information gathered is used in the Location policy, where the destination node, if any, is picked. Homogeneous systems can base all decisions upon loading information. Heterogeneous systems must take into account the processing power of the nodes. Thus information on a nodes processing power must be collected along with its loading. Many algorithms popular in the literature use a threshold based Location policy. A job is transferred if the remote node has a load less than a set threshold, normally the same threshold level as used in the Transfer policy. In this work we investigate a policy based on actual queue lengths.

Four algorithms are evaluated: SHORTEST, HETRO, HETQL, HETQLNIT. They are compared to theoretical upper and lower bounds of load sharing performance.

The **SHORTEST** algorithm first suggested in [2], uses a threshold based Transfer and Location policy. The threshold is set at 1 for system utilisations up to 70% and 2 for higher. The Information policy gathers the queue length at randomly picked remote nodes, considered a satisfactory measure of loading[12]. Should an idle node be discovered whilst probing, the eligible job is immediately transferred to that node. Otherwise nodes are probed up to the probe limit and the job transferred to the node with lowest loading if less than the threshold. If no suitable node is identified the job is processed locally.

HETRO uses a weighted load in its Location policy, this entails the Information policy gathering details of a remote nodes load and processing power. The weighted load is calculated as:

$$\boxed{\textit{weighted\_ load} = \frac{\textit{local\_ power}}{\textit{remote\_ power}} * \textit{remote\_ load}}$$

The transfer and location policies are both threshold based and transfer to any node found idle is immediate. As with SHORTEST the threshold varies with system utilisation.

HETQL differs from HETRO in that it does not use a threshold in either Transfer or Location policy. All jobs are considered eligible for transfer if the local node is busy, i.e. has a load of one, no matter what the system utilisation. The transfer decision in the Location policy is based on a comparison of weighted remote load and the current loading of the local node (queue of jobs at local CPU). Transfer occurs if the lowest weighted remote load is less than the local load or if an idle node is found.

HETQLNIT uses the same Transfer policy and queue length based Location policy as HETQL. The difference is that the newly arrived job is included in calculating both local and remote loads. Thus transfer to an idle node is not immediate or automatic. The power of the idle node will also be a determining factor. It may be better to bypass a slow idle node and send the job to a faster one even if it is not idle. The weighted load is calculated as:

$$\boxed{\textit{weighted\_ load} = \frac{\textit{local\_ power}}{\textit{remote\_ power}} * (\textit{remote\_ load} + 1)}$$

Transfer occurs if:

$$\boxed{\textit{local\_ load} + 1 > \textit{weighted\_ load}}$$

In order to give an lower bound to performance an **IDEAL** case scenario is used. This is based on simulation of an idealised load sharing scheme where complete knowledge of queue length and job sizes at all node is assumed available and each job is sent to the node where it will be completed in the least possible time. Once a job has been sent to a node it cannot be migrated. Transfer and information costs are assumed to be zero. This is the same principle as M/M/K [2] and NoCOST[13] or LB2[14], used as lower bounds in homogeneous and heterogeneous systems respectively. The upper bound is the M/M/1 case, with no load sharing.

3. System Model

The systems modelled are comprised of nodes that differ in processing power. The mean interarrival rate at each node is inversely proportional to power, ensuring that each node has the same original utilisation, as suggested in [14]. Interarrival times are exponentially distributed. Job sizes are exponentially distributed with a mean of 10 seconds for nodes of power 1.

The 10 systems studied all have 20 nodes and total power of 20 as shown in TABLE 1. They differ in the way the total power is partitioned between the majority nodes(A) and the minority nodes(B). Nodes in each category have the same processing power this is expressed in arbitrary units as only relative power is important. As the systems exhibit different levels of heterogeneity the question arises as to what metric to use in characterising it. Earlier work[15] used the ratio of processing power of the two classes of node. However, this metric is too restrictive as it can only be used for systems of 2 classes of node. In this work the skewness of distribution of power is used. This (together with the variance) proves a better measure of heterogeneity[16].

	TYPE - A			TYPE -B		
System Skew		Power	Fraction of total power		Power	Fraction of total power
-0.206	A1	1.650	0.99	B1	0.025	0.01
-0.149	A2	1.583	0.95	B2	0.125	0.05
-0.094	A3	1.500	0.90	B3	0.250	0.10
-0.028	A4	1.333	0.80	B4	0.500	0.20
-0.009	A5	1.167	0.70	B5	0.750	0.30
0.009	A6	0.833	0.50	B6	1.250	0.50
0.028	A7	0.667	0.40	B7	1.500	0.60
0.094	A8	0.500	0.30	B8	1.750	0.70
0.149	A9	0.410	0.25	B9	1.875	0.75
0.206	A10	0.350	0.21	B10	1.975	0.79

Table 1.

A probe limit of 3 is used for all algorithms[2,15] in the algorithm comparisons. System utilisation's of 50%, 70% and 90% were considered. Lower utilisation's offer little possible performance improvement to load sharing algorithms.

Communication delays are based upon measurements taken over LAN's at the University of Greenwich[17]. The delay in transferring a job is calculated assuming the existence of a common file server used by all nodes but on which no job can execute. Thus the cost of transferring a job is just the cost of a simple RPC to the destination node.

Experiments described in [17] have shown that the response times of RPC's do not vary significantly with the power or loading of the communicating nodes. Therefore the same delays are used in the simulations for all nodes. The cost of invoking a load sharing algorithm is considered negligible compared to the communication costs. The probing cost is 30ms to the probing node and 10ms to the probed node. The transfer cost is 30ms per job.

All simulations were implemented using MODSIM II (CACI). This is an object-oriented language designed for discrete-event simulation.

4. Results
Workload allocation for the ideal algorithm is shown in Figs 1-3. For clarity only a selection of results are illustrated. The "load balancing line" indicates where the points should lie if work were allocated proportionally to power.

At low utilisations, the vast majority of work is carried out on the high power nodes. Analysis of results shows weaker nodes are only used to execute jobs with short service times. As system utilisation increases results begin to group closer to the load balancing line. The powerful nodes are starting to work at almost full capacity and so the less powerful ones must take a greater overall share of workload.

Figs 4-6 contrast the relative performance of all the algorithms described in section 3. The upper bound or M/M/1 case is not shown on the graphs as the difference in scale would make the other curves unintelligible. All results shown are well below the upper bound.

At all levels of system loading and system heterogeneity the SHORTEST algorithm is outperformed by HETRO. At 50% utilisation the difference in response time is minimal at low heterogeneity. As heterogeneity increases it can be seen that HETQLNIT performs substantially better than the rest. This pattern is echoed in the results for 70% utilisation, with a smaller margin in improvement for the HETQLNIT. With a high utilisation of 90% the HETQLNIT algorithm ceases to be the optimum except for systems with a high negative skew. In other cases it is outperformed by the HETQL algorithm.

HETQL and HETQLNIT can be considered more scaleable than SHORTEST or HETRO as no change to algorithm parameters had to be implemented with changes in system utilisation. HETRO and SHORTEST use different threshold values at higher loads. An interesting point to note is that the general behaviour of HETQLNIT follows the same pattern as the IDEAL case.

Lastly in Figs 7-9 the effect of varying the probe limit, using HETQLNIT is presented. Performance increases with rising probe limit until 15-20% of nodes are probed. The benefits from using higher probe limits are marginal and are not obtained at all system utilisations. Only at 90% loading does the degree of heterogeneity have an effect. This is possibly due to swamping[2], where any lightly loaded node can become overloaded.

References.

[1] G.Bernard, D.Steve, M.Simatic, "A Survey of Load Sharing in Networks of Workstations", *Distrib. Syst. Engineering*, no 1, 1993 , pp 75-86.

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

[3] O.Kremien and J.Kramer, "Methodical Analysis of Adaptive Load Sharing Algorithms", *IEEE Transactions on Parallel and Distributed Systems*, Vol 3, no 6, 1993, pp 747-760.

[4] P.Kruger and M.Livny, "The diverse Objectives of Distributed Scheduling Policies", *Proc. 7th Int. Conf. on Distrib. Computing Syst*, 1987, pp 242-249.

[5] S.Zhou and D.Ferrari, "A Measurement Study of Load balancing Performance", *Proc. 7th Int. Conf. on Distrib. Computing Syst*, 1987, pp 490-497.

[6] R.Leslie and S.McKenzie, "Load Sharing in Distributed Systems", *Proc. 2nd Communication Networks Symposium*, 1995, pp 265-268

[7] P.Srimani and R.Reddy, "Load Sharing in Soft Real-time Distributed Systems", *Int. J. Systems Sci.*, Vol 23(7), 1992, 1115-1130.

[8] Y.Wang and R.Morris, "Load Sharing in Distributed Systems", *IEEE Transactions on Computers*, Vol 34(3), 1985, pp 285-217.

[9] P.Kruger and M.Livny, "A Comparison of Preemptive and Non-Preemptive Load Distributing", *Proc. 8th Int. Conf. on Distrib. Computing Syst*, 1988, pp 123-130.

[10] C.Rommel, "The Probability of Load balancing success in a Homogeneous Network", *IEEE Trans. on Software Eng*, Vol 17(9), 1991, pp 922-933.

[11] S.Zhou, X.Zheng, J.Wang and P.Delisle, 'Utopia : a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems', *Software Practice and Experience, 1993*, Vol 23(12), pp 1305-1336.

[12] D.Ferrari and S.Zhou, "An Empirical Investigation of Load Indices for Load Balancing Applications", *Proc. Performance 87 12th IFIP Int. Proc. on Computer performance*, 1987, pp 515-528.

[13] S.Zhou, "A Trace Driven Simulation Study of Dynamic Load Balancing", *IEEE Trans on Software Engineering*, Vol 14(9), 1988, pp 1327-1341.

[14] R.Mirchandaney, D.Towsley and J.Stankovic, "Adaptive Load Sharing in Heterogeneous Systems", *Proc. 9th Int. Conf. Distrib. Computing Syst.*, 1989, pp 298-306.

[15] A.Mahamuni, T.Gonsalves. B.Ramamurthi, "Efficient Load Information Management for Load Sharing in Distributed Systems', *Computer Networks Architecture and Applications*, C-13, 1993, pp 43-54.

[16] A.Sarraf, J.Senior, A.Wiseman, 'New Technique to Assess the Asymmetry of the Traffic Load Offered to LAN's", *Proc. Second Communication Networks Symposium*, 1995, pp 77-80.

[17] Leslie.R, "An Investigation Into The Costs Involved In Probing Heterogeneous Nodes Using Remote Procedure Calls": *Technical Report 1r02*, University of Greenwich, January 95.

Load Sharing In Distributed Systems

R.Leslie, S.McKenzie
Department of Computing and Information Systems
University of Greenwich

Abstract - This paper examines the performance of load-sharing algorithms when used on both homogeneous and heterogeneous systems. Algorithms described in the literature have been investigated and new algorithms are suggested that are tailored towards heterogeneous systems. It is shown with the use of simulation results that the suggested algorithms can provide better performance when used with heterogeneous systems and comparative results when used with homogeneous systems. The viability is shown of an algorithm that ignores idle nodes in preference to more heavily loaded nodes of a more powerful nature.

1. Introduction

When considering a system of computers, at any time there is a high probability that some will have a heavy load whilst others are idle or lightly loaded. Distributing jobs from the heavily to lightly loaded nodes can decrease the average response time of jobs in the system.

The distribution of jobs is achieved with the use of an algorithm. Static algorithms are based upon pre-determined load figures and so are limited in their use. Centralised algorithms are vulnerable to bottle-necks and failure of the machine on which the algorithm is running. Both static and centralised algorithms are discussed in [1,6]. The algorithms considered in this paper are dynamic and distributed.

The performance of load-sharing algorithms can be gauged against two measures. The worst case scenario is that where no load-sharing takes place at all. This is referred to as the M/M/1 case, each node has a single queue served by a single server. The second measure is the other end of the spectrum, where perfect load-sharing occurs, M/M/k. The system is represented by a single queue with multiple servers, no overhead is associated with the load-sharing process.

The potential gains achievable by load-sharing algorithms have been shown in previous papers[2,3,7]. However the system on which the suggested algorithms have been tested is invariably constructed of homogeneous machines. With the widespread use of heterogeneous systems, it was considered prudent to test the algorithms previously suggested on a variety of these systems. Heterogeneity in the systems considered was limited to differences in computational power.

The testing and development of load-sharing algorithms has been carried out through the use of the Modsim simulation language. The simulation model used has been validated against results published in previous papers and those gained through queuing theory analysis.

2. Dynamic Distributed Algorithms

Dynamic distributed algorithms can be sub-divided into three constituent parts known as policies. The terms *transfer* policy and *location* policy were first introduced in [2], transfer policy has been generally accepted as the policy determining whether a job should be executed locally or made available to be transferred to another node for execution. The most widely used transfer policy is the threshold policy based upon queue length, as a new job arrives at a node, the queue of jobs at that node is examined, if it is above a set threshold value the job is eligible for transfer. Eligibility for transfer does not imply that the job must be transferred.

Early definitions of location policy, the policy which decides where a job eligible for transfer should be transferred to, included the means of acquiring the information on which to base the decision. Later work [3,7] splits this definition into location and *information* policy, the latter concerning the acquisition of information upon which to base decisions. Using the three terms allows a clearer description of any algorithm and they are all used in this paper.

The simplest dynamic distributed algorithm is one which uses a random location policy, known as the RANDOM algorithm. First suggested in [2] and referred to in [3,7]. This algorithm uses a transfer policy based upon queue length thresholds. Once it has been decided that a job is suitable for transfer no information is gathered on which to base the transfer decision the job is randomly transferred to any node in the system. A limit (normally 1) must be put on the number of possible transfers or the problem of thrashing may arise. Where jobs are constantly transferred and never executed. When used in homogeneous systems the performance offered by this algorithm is always an improvement on the no load-sharing case. This is shown not to be the case in heterogeneous systems.

The SHORTEST algorithm [2] is more sophisticated as it bases the location decision on the information gathered upon the system state. When a job becomes eligible for transfer other nodes are probed as to their current load level. The nodes to be probed are picked at random, a limit is put on the number to be probed. The first idle node probed (load of zero) is the one to which the job is transferred. If an idle node is not

2. **Eager.D.L, Lazowska.E.D, Zahorajan.J,** *"Adaptive Load Sharing in Homogeneous Distributed Systems"* : IEEE Trans on Software engineering, Vol SE-12, No 5, May 86

3. **Kremien.O, Kramer.J,** *"Methodical Analysis of Adaptive Load-Sharing Algorithms"* : Trans on Parallel and Distributed Systems, Vol 3, No 6, Nov 92

4. **Leslie.R,** *"An Investigation of Simple Load-Sharing Algorithms Using Simulation"*: Technical Report, University of Greenwich, October 94

5. **Leslie.R,** *"An Investigation Into The Costs Involved In Probing Heterogeneous Nodes Using Remote Procedure Calls"*: Technical Report, University of Greenwich, January 95

6. **Wang.Y & Morris.R.J.T,** *"Load Sharing in Distributed Systems"* :IEEE Trans on Computers, Vol C-34, No 3, March 85

7. **Zhou.S,** *"A Trace Driven Simulation Study of Dynamic Load Balancing"* : IEEE Trans on Software Engineering, Vol 14, No 9, Sep 88