Transparent Process Migration Protocol For Distributed Applications in Heterogeneous Unix Networks

A Dissertation
Submitted in partial fulfillment of the requirement for the award of the degree of

MASTER OF ENGINEERING
(COMPUTER TECHNOLOGY & APPLICATIONS)

BY

Milan Saxena

Under the guidance of
Dr. Goldie Gabrani

Department of Computer Engineering
Delhi College of Engineering, New Delhi-110042
(University of Delhi)

February-2005

CERTIFICATE

This is to certify that the Dissertation entitled “submitted by MILAN SAXENA, in the partial fulfillment for the award of the degree of MASTER OF ENGINEERING” in “Computer Technology and Applications” of Computer Engineering Department of Delhi College of Engineering, Delhi , is a bonafide record of the work he has carried under my guidance and supervision.

Dr. Goldie Gabrani
(Asst. Professor)

Department of Computer Engineering

Delhi College of Engineering,

New Delhi-110042(University of Delhi)

CERTIFICATE

This is to certify that the Dissertation entitled “submitted by MILAN SAXENA, in the partial fulfillment for the award of the degree of MASTER OF ENGINEERING” in “Computer Technology and Applications” of Computer Engineering Department of Delhi College of Engineering, Delhi , is a bonafide record of the work he has carried.

Prof. D. Roy. Choudhary
(Professor)

Department of Computer Engineering

Delhi College of Engineering,

New Delhi-110042(University of Delhi)

Acknowledgement

It is a great pleasure to have the opportunity to extent my heart felt thanks to every body who helped me through the successful completion of this project.

I would like to express my sincere and profound gratitude to my supervisor **Dr. Goldie Gabrani** for her invaluable guidance, continuous encouragement, helpful discussions and wholehearted support in every stage of this Thesis. It is my privilege and honor to have worked under her supervision. It is indeed difficult to put her contribution in few words.

I would also like to thank my parents and my sisters for their encouragement and support and dedication, which has helped me in great sense to complete this course.

I am also thankful to my teachers at DCE viz. Prof D. R. Choudhary, Prof A. Dey, Dr. S.K. Saxena and Ms. Rajni Jindal for their support and encouragement.

I am thankful to my friend and classmate Mr. Rajeev Srivastva, Lecturer NSIT for his unconditional support and motivations during the last final week of thesis submission.

Last but not least I would like to thank my colleagues Arvind Tiwari, Abhinav Gupta, Anjali Sharma, Indu Arora, Naveen Dankal, SPM and DM, C-DOT for their all kind of support in completing this course.

(MILAN SAXENA)

M.E. (Computer Technology and Applications)

College Roll No. 17/CTA/2002

Delhi University Roll No. 4002

ABSTRACT

In this thesis, analysis and design of a protocol for the transparent process scheduling and process migration in heterogeneous unix networks has been proposed and implemented. Further, an algorithm to measure the load of any machine in distributed heterogeneous Unix network has been devised and it is analyzed for the performance and efficiency. Because a distributed system can be used effectively by its end users only if its software presents a single system image of this physically distributed system to the users, So all the resources of any node should be easily and transparently accessible from any other. A common kind of machine clusters typically used is a set of heterogeneous Unix workstations. While solutions are available for transparent sharing of resources such as files and printers in such a scenario, an important resource that is typically not shared is the CPU. Hence, even though there is an idle machine is the system, a heavily loaded machine cannot share the idle CPU transparently which signifies wastage of CPU cycles. Process migration can reduce the loss of such CPU cycles by migrating processes from a highly loaded machine to the idle (or lightly loaded) one. Remote execution or non-preemptive migration is a form of process migration where the processes are placed in remote nodes during creation. In a heterogeneous environment only non-preemptive process migration can be performed transparently.

We have proposed a transparent process scheduling and process migration protocol and a set of conventions to achieve load sharing for a cluster of heterogeneous machines each running some flavors of Unix. In the protocol, the single system image is also preserved by creating unique cluster wise process ids, keeping a traditional filename space, uniform user credentials, etc. The protocol itself do not provide all the features of process migration, but it rely on the underlying Unix network. For example, the protocol assumes that the machines already have a more or less uniform file system view (through NFS) and are uniformly administered for user credentials. These are declared as conventions that all machines in the cluster must ensure. The protocol also does not dictate the policy decisions and can work with any load sharing policy. In the current version of the protocol, there are a few limitations. For example, the current version of the protocol does not have support for migration of process from one architecture of Unix to another as, the migration time in this case is significantly high as compared to time required for the completion of process.

Table of Contents

List of Figures

List of Tables

CHAPTER 1 Introduction

The availability of low cost powerful microprocessors and high speed computer networks has radically changed the way computing is done today. The big, monolithic mainframes of the earlier days have been replaced by clusters of small but powerful computers connected by high speed networks. A distributed system is defined as a collection of autonomous computers interconnected by a communication network. The computers (or nodes) of a distributed system communicate only through message passing because there is no physically shared memory.

Distributed systems have many advantages to offer over sequential machines. Distributed and parallel computations can take advantage of the multiple processors available in distributed systems for achieving high speed gains. A distributed system also provides higher reliability as the failure of node does not bring down the entire system; making it highly available to its users. Further, the system is scalable as extending the system for more power would require addition of fewer components. For a sequential machine such extension would mean replacement of old components by new more powerful ones.

All these potential benefits of distributed system cannot be fully utilized by the users until all the resources are shared transparently by all the nodes in the system. Transparent sharing means that the users should be able to access all the resources physically available in any machine in the system without worrying about their physical locations. Transparency is important because users want to view the system as a traditional monolithic computer system, since they are more accustomed to the single system look of traditional timesharing systems where the location of a resource is irrelevant. In addition to the look of the system, if applications are written for accessing resources from known locations, they may fail in case some of the accessed resources migrate to other nodes. Transparency also prevents this by hiding the location information of the applications.

A number of such distributed system have been developed in the universities and research laboratories in the recent years. Some of the prominent examples include Amoeba [1],V[2,3], Charlotte [4], etc. most such systems work by running a copy of the same oper-

ating system on all the participating computers and these copies cooperate to provide a single system image of the system to its users.

The popularity of such systems, however, has not really grown. This is because of the fact that there is a large existing user and software base for Unix [5,6,7,8]. Several distributed systems therefore try to emulate Unix so that existing applications can be reused with little or no modification and so that the users get a familiar working environment.

However, there is another problem that must be solved by today's distributed system software. This problem arises because the distributed systems of today typically comprise of hardware and operating system software from a variety of vendors. Achieving the single system image in the face of this heterogeneity is a major challenge. Systems such as Amoeba etc., do not address the problem of operating system heterogeneity as they assume that all participating machines on the network run the same operating system. A few solutions are also available today that address this problem partially. Sun-NFS [9] and AFS(10) are examples of distributed file systems that are commonly used now to achieve a unified view of the file system on a network of heterogeneous workstations. Various other resources such as printers etc., are now routinely shared on Unix systems.

1.1 Transparent CPU Sharing

An important resource that is typically not shared transparently on Unix systems is the CPU. Several studies have shown that there is a wide disparity in the load of various machines in a distributed system at any given time of the day. While there are some machines that are heavily loaded, others are completely idle. What is desirable is that these machines should share the total processing load requirement of all the users so that the load distribution is more uniform and ultimately the users see an improvement in system performance. While there are user level commands available in Unix systems (such as rsh) that allow users to execute their jobs on any machine of their choice, such mechanisms are clearly not transparent. In an ideal setting, a user would just fire a job and the system would automatically select the best location (i.e., the least loaded machine) to execute the job. This is called transparent remote execution or load sharing. A stricter form of

such load sharing is called load balancing, wherein the system strives to balance the load on all machines at all times. While load sharing would just require the system to select the best machine for executing a newly submitted job and transparently transfer the job to that machine, load balancing would typically call for migrating a job to another machine, possibly during its execution.

1.2 Background

In this section we summarize a few basic terms and ideas related to load related to load sharing.

1.2.1 Preemptive and Non-preemptive Migration

The act of transferring a process from a node to another in a distributed system is called process migration. Migration of an already executing processes is called preemptive process migration. This requires the executing process to be stopped during execution (hence preemptive) and the state of the process is transferred to the target node where its execution is resumed. Initiating a process at a node different from the creator of the process is called remote execution. It is also called non-preemptive migration because it does not involve preemption.

It can be easily seen that non-preemptive process migration can be more easily implemented than preemptive migration. This is because the latter requires the system to checkpoint the state of a process already in execution and then transfer this state to the target machine. Clearly this is a very hard problem if the two machines are architecturally different, for in that case the checkpointing would have to be done at the source program level and not the executable code level. Tui (11) is an example of a system that allows executing processes to migrate to architecturally different machines. However, in this case, the process of migration is not transparent to the application program and the application must cooperate with the migration software in order to migrate successfully. Further, the migration process is also costlier in terms of time since the entire state (which might be quite large) has to be transferred to the destination machine. Studies have shown that this additional overhead severally restricts the performance benefit that can be obtained by using

preemptive task transfer as compared to non-preemptive process migration (12). Non-preemptive migration does not incur this additional cost since only newly submitted jobs are transferred to other machines and therefore there is no address space image to transfer. Further, heterogeneity is much more easily accommodated.

1.2.2 Load Balancing and Load Sharing

As and when user's job will arrive, they will be allocated and scheduled on various idle-Hosts present in the network. This involves the registration and de-registration of Idle-Hosts and proper scheduling algorithm to be implemented to get the fairness, liveliness and lack of starvation of job allocation to various hosts. This concept is called as Load Balancing, as before the execution of any process, the system may decide the host where the process can be scheduled fairly.

During the execution of various jobs it may happen that one particular host may be highly loaded with number of processes executing on it. Therefore a proper migration mechanism should be there to migrate the heavy task/process to a less loaded suitable idleHost, this concept is known as load sharing as on the fly the load of the complete distributed system be shared among the hosts. Load Sharing is an area of research where the process migration is being implemented with the help of concepts of shared memory or dynamic process creation and process migration from one machine to another machine.

1.2.3 Policy and Mechanism Issues

There are two orthogonal issues related to load sharing- policy and mechanism. The first one relates to the policies for migration. For example, when should a machine attempt to transfer a process to another machine, which process should be migrated and to which machine? The second issue relates to the actual methods of transferring processes and ensuring that a migrated job will get roughly the same environment as it would have on the machine where it actually originated.

1.2.4 Heterogeneity

In a distributed system heterogeneity can exist in several forms. Nodes, for instance, can have different architectures (architectural heterogeneity), can run different operating system (operating system heterogeneity) or have different volumes of resources available (configuration heterogeneity). Heterogeneity has a major impact on load sharing. It can easily be seen that, in general, transparent preemptive process migration is not possible between heterogeneity nodes.

1.2.5 Failure and Fault tolerance

Failures are an unavoidable part of a distributed system. In a distributed system, both nodes and the network links connecting them can fail. Based upon the behavior of the failed nodes, node failure can be further divided as a fail-stop or a Byzantine failure. A fail-stop failure means that the failed node does not do anything and simply ceases to operate, whereas Byzantine failure implies that the failed node behaves in a completely arbitrary and unpredictable manner. Byzantine failures are complex to handle and hence most of the distributed systems assume that node failures are fail-stop in nature.

Network partitioning is a type of network failure where the set of nodes in the system is divided into two or more partitions; a node in a partition can communicate with only nodes in that partition but not with any node in other partitions.

Apart from the benefits of load sharing, process migration can also be used for fault tolerance. For example, long running processes may be moved to a different node when a node is about to shutdown. In several cases, such prior notifications are sent to the processes by the operating system. Hence, if a process wants to continue execution, it can migrate to a new node. In systems where check pointing is employed, even a process terminated during a node failure may be restarted on another node from the check pointed state. Such fault tolerance measures can only be employed in a system that supports preemptive process migration. In systems supporting non-preemptive process migration care should be taken so that the failure of a component of the system (node, link, etc.) does not affect the behav-

ior or other components and when a process fails, all its effects should be removed from the system as if the process has existed.

1.2.6 Scalability

Performance of the components of a distributed system may vary depending upon the number of nodes in the system. A component is said to be scalable if its performance does not degrade with increasing number of nodes. While designing a process migration system care should be taken so that centralized components are not used. The centralized components become bottleneck as the number of nodes in the system increases. These are also the source of single point of failures, meaning the working of the system depends upon the working of these components. When any of the centralized component does not function, the entire system stops working. Moreover, broadcast messages for communication between nodes should not be used. Apart from the fact that these are not supported in all the networks, this also means increased network traffic. Moreover, if a broadcast message is targeted for a subset of nodes in the cluster, this also imposes processing power wastage for the rest of the nodes in the system. Hence, for scalability both the centralized components and broadcast message should be avoided in a process migration system.

1.3 Related Work

Several systems have been implemented in the past for transparent process migration. In this sections we shall look at some of these systems in brief. A comparative study of these systems is presented in appendix A.

Process migration can be implemented entirely at the user level, or by making modifications to the kernel itself. Implementations of both kinds have been reported in the literature (13).

1.3.1 User Level Implementations

Purely user level implementations usually work by intercepting system calls through a modified system call library. The modified system call library provides functions for three

kinds of activities. Firstly, the modified system library replaces system call functions with versions that examine the system call arguments and return values and thus attempt to memorize the external state of the process, like files opened, in their private variables. Secondly, it provides code for checkpointing the process state before migration. For Unix processes the state can be divided as internal state containing text. Data, stack segments, register context etc. and external state containing open files, etc. for saving the data and stack segments, the check pointing function typically determines the segment addresses and sizes and writes these segments in a file. Finally, the modified library provides restart routines that can be used to resume the execution of the checkpointed process form the saved state.

The modification of the system library implies that application programs need to be re-linked with this new library in order to be eligible for migration. On systems that support dynamic library linking, this may not be necessary. In Utopia (14) a slightly different approach is used where only a few programs, like the command shell, need to be changed to use a high level library interface for migrating the processes that they spawn. But the other applications need not be even re-linked.

Along with the modification system call library, a few daemons are also typically used in user level implementations. These are used for implementing the policy activities like detection of idle resource, load information exchange etc. some systems also use them as servers for providing services to remote processes like file or device access.

The advantage of user level implementations is the ease of portability to different flavors of Unix. However these systems are typically slower than ones in which migration is done at the kernel level. Moreover, user level implementations also have certain restrictions because of the non-accessibility of kernel data structure. For example, a Unix process can delete an open file. If such a process migrates to a remote machine, the source system should inform the remote system about this open file. But this cannot be done because the file does not have a file name and hence does not exist in the file system. User level implementations like Remote Unix, Condor, Utopia etc. all have similar restrictions. For example, Remote Unix and Condor do not allow a remotely executing process to create new child processes and Utopia does not support process groups across machines.

Remote Unix

Process migration under Unix started with University of Wisconsins Remote Unix (15). Remote Unix uses idle workstations to execute computation- bound batch jobs in back ground. Each machine has a queue of jobs which are submitted for execution in idle machines through the ru program. Since these jobs are executed in background, the standard streams, i.e. stdin, stdout, stderr of the jobs are redirected to files. The jobs in the queue execute in the idle machines and when finished the user submitting the job is informed through mail. While running at a remote machine, al file related system calls of a job are forwarded to the node where it was submitted The forwarded calls are executed by shadow process running in the home machine. Remote Unix provides an automatic checkpoiniting and execution control mechanism so that the job can migrate from one machine to another. When the owner of a workstation wants to regain control of his/her workstation, the jobs executing on that machine are automatically checkpointed to files in their respective home machines. When some other machine becomes idle, execution can resume in that idle machine. The system takes care of re-opening all the files in the appropriate modes, assigning the original file descriptor numbers to them, and positioning the file offsets to appropriate places.

Being a user level implementation, Remote Unix is portable to any Unix kernel. But it also has several limitations. It is mainly for computation intensive batch processing and does not support remote execution of interactive processes. The migrated process cannot open pipes, cannot send/receive signals, cannot make fork, exec, etc. system calls. Interprocess communication is also not supported. Programs that depend on knowing their own process id will not work correctly because the identification of an Remote Unix process changes after each checkpoint. Moreover, heterogeneity is not supported.

Condor

Condor [16,17] system is a descendant of Remote Unix. Condor also follows the same principle of utilizing idle machine cycles and has the same limitations as that of Remote Unix. Condor differs from Remote Unix in only its attempts to allocate jobs to the idle machines fairly and in some of the terminologies. Condor runs two daemons, viz. Started

and schedd, in every machine. The schedd daemon keeps track of all the jobs that have been submitted. Started daemon monitors information about the machine that is used to decide if it is available to run a Condor job, such as time since last keyboard or mouse activity, and the load on the CPU. Started also checkpoints and removes a job from the machine when the user comes back and starts working on the machine. The mechanism for fair allocation of these idle machines to users who have queued jobs is handled by a centralized machine manager on the basis of priority. The priority is calculated according to an algorithm that periodically increases the priority of those users who have been waiting for resources, and reduces the priority of these users who have received resources in the recent past. The purpose of the algorithm is to allow heavy users to consumer very large amounts of CPU cycles, and at the same time protect the response time for less frequent users. Like Remote Unix, the file system related system calls are forwarded to a shadow process in home machine, a mechanism called the remote system call. In Condor, a process is run on a machine with the architecture and operating system it was compiled for. Beyond that, jobs might have requirements as to how much memory they need, or special hardware. Owner of a job can specify the requirements and preferences of the job when it is submitted. Condor is highly portable on different platforms and also provides limited fault tolerance.

Although it works extremely well for its intended goals, Condor is slow and does not support IPC, signaling, etc. the use of centralized machine manager makes Condor non-scalable. In Condor, the processes are heavily dependent on the home node. The home node performs all the file system related system calls even though the process is executing remotely. This gives rise to residual dependencies. Residual dependencies create two problems. First, the overhead involved in carrying out operations on behalf of remote process affects the performance of the home machine. Secondly, the execution of the remote processes becomes vulnerable to the failure of the home machine.

Global Layer Unix

Global Layer Unix, called GLUnix [18] short, is another user level implementation of process migration system which provides the illusion of a single system image. The distinctive feature of GLUnix from three user level implementations are its globally unique

process id and signaling mechanism. Under GLUnix a process id is unique in a cluster of connected workstations and a process can send signal to any other process in the cluster.

Utopia

Utopia [14] is a load sharing system for large clusters of nodes. Transparency in file access is provided through a shared file name space. Utopia supports only remote execution i.e., non-preemptive process transfer. Load sharing policy, RESs (Remote Execution Servers) which implement the migration mechanism, a Load Sharing Library (LSLIB) and the Load Sharing Applications.

Every node runs a LIM and a RES. Load sharing applications are special applications written using LSLIB. LSLIB provides primitives to the load sharing applications for communicating with LIM and RESs. LIM is the policy component which interacts with other LIMs in the system for collecting load information which is used to help load sharing applications for selecting nodes for task initiation. A load sharing application can initiate a task at a remote node by contacting RES in that node. RES also acts as a middleman for transferring signals and input/output date between the remote task and the load sharing application. Scalability is taken care of by structuring the nodes in a logical hierarchy so that the load information exchange and task placement decisions do not cause a large overhead in the system. This is done by grouping the nodes into clusters and then arrange the clusters in a tree like hierarchy. Within a cluster, the nodes exchange detailed load information but a cluster only sends a condensed summary of the load situation in the cluster to other clusters. Thus load information travels along the edges of the tree. The final decision regarding the placement of a job is done in several stages. In the first stage only the top level cluster to which the job will be assigned is decided. The decision is then refined to finally decide which node in a cluster will execute the job.

There are two kinds of applications in Utopia. A few applications like shell etc., use the high level library (LSLIB) functions for load sharing. Load sharing shell, parallel make, etc. are examples of such applications. These programs are called direct applications which require complete rewriting because the LSLIB functions names are different from Unix system calls. But the indirect applications which use the facilities provided by the

direct applications can get the load sharing facility even without re-linking. For example, a load sharing shell can remotely execute a process to another machine. For this, the migrating program binary need not be changed. But, if such an migrated process (created from an unmodified program) creates a child process which calls exec, this child process cannot be migrated. This surely means loss of migration opportunity. Utopia does not support distributed process groups when the members execute in different machines.

1.3.2 Kernel Level Implementations

In contrast to the typical requirement of re-linking of existing programs in user level implementations, kernel implementations can allow binary compatibility to the applications. Hence, existing programs will run without modifications, or even re-linking. It also preserves the seem antics of Unix calls betters since all of the process state that is stored in the kernel date structures is accessible. Kernel implementations are also usually more efficient. However, a serious drawback is reduced portability, since it is much to write user level programs than changing the kernel. Each different version of Unix that is to be supported needs to be changed afresh. Typical kernel implementations such as Solaris MC, and OSF/1 AD TNC assume that all the participating machines run the same operating system and hence do not support heterogeneity in operating system.

Although the focus of this section is on Unix process migration, we also describe two non-Unix process migration, we also describe two non-Unix process systems here, namely V-system and Charlotte, which define interesting inter process communication model suitable for migration. Both of these run identical micro-kernels in every node of the system that provide transparent inter process communication across nodes.

V-system

The V distributed system [2,3,19] consists of a number of diskless workstations and a set of server machines all running the V kernel. This collection of machines is called a V domain. The kernels contain three major components, namely, the kernel server, device server and IPC. Kernel servers provide process and memory management services to processes, while device servers provide device access through a uniform I/O object abstrac-

tion, abbreviated as UIO. The processes can transparently access any server in the V domain through IPC primitives. Tuned to facilitate client/server kind of communication. For example, it has two primitives for sending messages. Servers send back requested data to a client with non-blocking reply primitive. However the send primitive is block until the reply comes back from the server. Process ids are used in the IPC primitives to indicate the communication peer. V supports preemptive process migration for using idle machine cycles. The connection less nature of the IPC mechanism makes migration easy even for processes that are communicating with other processes. If a process migrates while servicing requests from other (client) processes, the client processes are informed that the request cannot be satisfied since the process is migrating.

Apart from being a system for homogeneous environment, V is non scalable, because it uses broadcast message to ensure network wide unique process ids. V uses idle machine detection and migrates process only to idle machines.

Charlotte

Charlotte [4] provides a connection oriented IPC facility through the link abstraction. Links are bidirectional communication channels whose end points can be moved across the processes even crossing machine boundaries. Like V, a Charlottee process interacts with other processes and kernel using network transparent link facility. When a process migrates to another node, all the relevant kernel state, address space and link information are sent to the destination machine. The link endpoints of the migrating process are moved to the target machine using the Charlotte IPC facility. The process identification changes when a process migrates. The links are very difficult to program. In fact a high level language, called Lynx is created for this purpose. Charlotte uses a central file server that makes it vulnerable to failures (single point failure) and non-scalable.

Sprite

Sprite [20,21] is a Unix like operating system that uses transparent process migration to use idle machine cycles in a homogeneous cluster of workstations. Transparency in file accesses in provided through a global filename space. The global file system also helps in

process migration and limited inter process communication (single reader single writer named pipes only). Sprites notion of transparency is relative to the home machine of the processes. Home machine of a process is defined as the machine where the process would have executed if it had not migrated. A home node can initiate a process to an idle machine. When executing in a remote node, a process can make systems calls. System calls are categorized as location dependent and location independent system calls. Location independent calls are executed in the machine where the process is currently executing. But location dependent calls are always executed in the home machine. Process management system calls like fork, exec, wait, etc. and device related calls are examples of location dependent calls. The children processes inherit the home node of the parent and also the ps command in a node shows all processes whose home machine is the node itself. Hence a user gets the illusion that all the process are executing in his/her own workstation.

When a user of a idle machine comes back, all the remote processes are evicted back to its home node by the load-average daemon running in that machine. Preemptive process migration is used for eviction. Every machine runs a load-average daemon which apart from evicting processes also detects if the machine has become idle and informs this to a central migration server. Machines can ask the central migration server for an idle machine.

Sprite does well to utilize the idle CPU cycles, but also has a few problems. As in the case of Condor, Sprite processes are also heavily dependent on the home node. In fact part of the process state is kept in the home node and the home node also performs several operations when the process is executing remotely. This gives rise to residual dependencies. Sprite allows limited devices of any node except that of home machine. This restricts resource sharing. The use of a centralized migration server makes Sprite vulnerable to failures and non-scalable. Sprite does not support heterogeneity.

MOSIX

MOSIX [22,23] is an enhancement of BSD /OS [24] for adaptive resource sharing and is designed to run on a homogeneous cluster of Pentium based workstations, connected by

standard LANs or fast interconnection networks. Users interact with the system through their home nodes. Users create processes in three home node, and whenever the workload becomes higher than a threshold, some of the processes are transparently migrated to other less loaded nodes. A process is migrated either due to higher CPU load or higher memory contention leading to thrashing. A part from distributing the CPU load of the system, MOSIX also tries to reduce main memory thrashing and swapping out of pages by distributing the memory requirements among the nodes of the cluster. This is done using a memory ushering algorithm.

MOSIX uses decentralized control for both load sharing policy and mechanism and provides autonomy to every node for making process migration decisions independently. This enables it to be scalable to a large number of nodes. For efficient kernel communication a modified version of TCP.IP protocol is used, which reduces the overhead of initial connection setup.

Like Sprite, MOSIX also does not provide full transparency in accessing all the resources in the system. The transparency is defined relative to the home node. Also heterogeneity is not supported in the system.

LOCUS

LOCUS [25,26]is another Unix compatible kernel level implementation supporting load sharing. It has a global file system which allows transparent replication of files for improved file read performance. Unix file offset sharing is implemented through a token based protocol. LOCUS allows processes to migrate both during fork and exec system calls. But migration during fork is restricted between machines of same architecture only. LOCUS has support for heterogeneity and single reader and single writer pipes. Pipes are implemented in the shared file system. It also supports transparent device access for a limited class of devices.

LOCUS supports very limited inter process communication apart from signals. Also the token based protocol used for file access synchronization makes it inefficient.

OSF/ 1 AD TNC

OSF/1 AD TNC [27] is a multicomputer (NORMA) system from Open Software Foundation and Locus Computing Corporation based on OSF/ 1 MK single server on Mach 3.0 [28] micro-kernel. OSF/ 1 AD TNC has large number of nodes categorized into three types: nodes used for input/output and connectivity (1/O nodes or file server nodes), nodes dedicated for parallel applications (compute nodes); nods for interactive use (service nodes). The system enables the user to take full advantage of multicomputer hardware while supporting full OSF/1 (and thus Unix) semantics. It presents a single system image by means of a single file name space and access to all the system resources and OSF/1 facilities (file descriptor, socket, pipes, process management and even shared memory). Mach 3.0 transparent network services and memory object facility is utilized extensively for this purpose. Scalability is an important issue in a multicomputer with large number of nodes. Thus the system distributes control of the file system, socket protocol stack and process management subsystems. For accessing global resources like sockets or processes, the system uses virtual structures. Virtual structure points to a table of functions which are used to perform global operations on the object identified by the virtual structure. These global functions send messages to the appropriate nodes and call local functions of that node. The system supports process migration and automatic load balancing.

Although OSF/1 AD TNC has been able to keep the single system image and Unix semantics intact, doing so it has put certain restrictions and performance drawbacks. Mach 3.0 networking facility and memory object facility across the network has its own performance penalty. Also, operating system heterogeneity is not supported.

Solaris MC

The Solaris MC [29,30] operating system is a prototype distributed operating system that provides a single system image for a homogeneous cluster and provides high availability so that the cluster can node failures. Solaris MC is built as a set of extensions to the base Solaris UNIX system and provides the same API as the Solaris OS, running unmodified applications. Solaris MC supports remote execution only, where a process can change node only during exec system call. The system preserves Unix semantics for files through

a global file system called Proxy File System (PXFS). The file name space is the same for each node in the network. Process Management is done globally using CORBA. Each node has a node manger object which managers several virtual process objects. Every virtual process object represents a local process. a node can perform operations like sending signals, etc. by receiving a reference to the virtual process object of the target process from the process id of the process and by invoking a method on that object. The I/O subsystem makes it possible to access any I/O device from any node in the multi-computer without regard to the physical attachment of device to nodes. In Solaris MC, a node can transparently access every network interface available physically attached to any node in the cluster. This is done by three components of the networking subsystem: (a) demultiplexing the incoming packets to an interface to the appropriate node, (b) multiplexing the outgoing packets from various nodes onto a network device and (c) global management of network name space. Network services are accessed through a service access point of SAP (For TCP/IP, the SAPs are ports). Solaris MC keeps a database that maps SAPs to nodes. The database is maintained by the SAP Server which ensures that also enables a system wide process tracing mechanism through its distributed /proc file system implementation.

Solaris MC does not support heterogeneity in the system. It also is not able to provide POSIX controlling terminal and session semantics. It provides a weaker semantics for exec. Should the process machine after exec the file sharing semantics is not preserved.

1.4 The Scope of Our Work

This work is aimed at providing transparent resource sharing in a heterogeneous Unix network. The nodes can be of different architecture and may run different flavours of Unix. We have chosen Unix as the base operating system because of its popularity, its open nature and the large number of existing Unix applications.

Our basic approach to handle CPU sharing and heterogeneity in operating system is to define a standard protocol for transparent remote execution that all participating machines must implement. The protocol should not be biased by any particular flavour of Unix, and should not be defined in terms of abstractions that are peculiar to any Unix implementa-

tion. As a example of a system that does not follow this principle, OSF/1 AD TNC uses Mach ports for communication among machines. This makes the protocol unsuitable for Unix implementations that are not Mach based. The protocol should be able to tolerate both node and network failures. Thus is case of a partial failure, the rest of the system should be able to work in the usual manner. Finally, the protocol should not dictate the policies for load sharing. The policy decisions are an orthogonal matter and should be treated separately.

The standard protocol by itself is not sufficient to handle heterogeneity. We also define a set of conventions that each participating machine must adhere to. For example, each machine should assign new process ids in a certain manner. The advantage of separating the protocol and conventions from their implementation is that different systems can implement them differently.

1.5 Organization of the Thesis

The thesis is organized as follows:

Chapter 2: Performs a discussion over various issues in unix process migration.

Chapter 3: Explains the complete architecture of the proposed protocol for the process scheduling and process migration mechanism. This part also introduces the reader to some of the concepts which are to be used in the system.

Chapter 4: Explains the concepts of load measurement and discusses the parameters to be taken for the load measurement of the system, which mark it as being idleHost.This chapter will elaborate the load measurement parameters, their significances, the relation between the system parameter and system load. Finally a algorithm is discussed to measure the load of any distributed machine in heterogeneous unix network.

Chapter 5: Explains the design and implementation.This chapter describes the structure of the different messages flowing in the Process Migration system. These messages are used by the different modules in the system to exchange information. Some of the messages have identical structures. Members which occur in more than one message but have the

same meaning have been explained only once i.e the first time when they appear in the chapter.

Chapter 7: This chapter describes the system modeling for the proposed protocol. It designs various object models using OMTs. Finally the complete view of the system has been shown here.

Chapter 8: Discusses the proposed future work and the conclusion over the various aspects of the suggested protocol

Bibliography

Appendices

CHAPTER 2 Issues in Unix Process Migration

Providing transparent migration facility for Unix in heterogeneous environment is much more harder than the specially designed distributed systems like Amoeba, V, Charlotte, etc. This is because these systems were designed keeping the goals of distribution in mind. Unix, on the other hand, was designed as a centralized operating system much before the ideas of distributed systems came into widespread use. It is much harder therefore to adapt it to a distributed setting.

Several issues, peculiar to Unix process migration, need to be addressed for providing single system illusion to the users of a Unix cluster. In this chapter we briefly discus some of these issues.

2.1 Process State

The load sharing policy selects a local process for migration and a node where it has to be migrated. Once these are decided, the execution of the process is stopped and its state is extracted and transferred over the network to the target machine where it is restored and execution of the process is resumed. We now look at what comprises the state of a Unix process.

Each Unix process has a well defined context, comprising all the information needed to describe the process. this context has several components.

Users address space This is usually divided into several segments the program text, data, user stack, shared memory regions, and so on.

Control information The kernel typically uses two main data structure to store control information about a process t the u are and the proc structure.1 Each process also has its own kernel stack and address translation map (pages tables).

Credentials The credentials of the process include the user and group ids, controlling terminals etc.

Environment variables These are the strings of the form variable = value which are inherited from the parent. Most Unix systems store strings at the bottom of the user stack.

Hardware context This includes the contents of the general purpose registers and a set of special system registers. When a context switch occurs, these registers are saved in a special part of the u area (called the process control block, or PCB) of the current process.

Unix processes interact with the outside world through several abstractions, such as files, devices, pipes, sockets etc. The state of these abstractions collectively represent the external state of a process while the process context is called the internal state of the process. A process uses unsigned integers called file descriptors to access thee objects. The table of open file descriptions (user file descriptor table) for a process is typically stored in the u area. The external and the internal state together constitute the state of a process that should be sent to the destination machine.

2.2 Preemptive vs. Non-preemptive Migration

As mentioned earlier, preemptive migration is both harder to implement and more expensive than non-preemptive migration. Preemptive migration has a much larger overhead since the entire address space and the hardware context of the process needs to be checkpointed and transferred over the network. Also in presence of heterogeneity, for example when machines participating in migration have different architectures, the saved state needs to be mapped to equivalent state of the target node. Although this mapping can be done at the source level [11], no general solution exists for such mapping. Moreover doing it at the source level would mean loss of transparency because the application has to assist the migration of processes. For these reasons we have decided OT focus only on non-preemptive migration.

In Unix, new processes are created by using the fork system call. The fork call creates a new child process of the calling process. The child process has an (almost) identical address space and hardware content as the parent process. thus if a newly created process is migrated, the same difficulties as in the case of preemptive migration would be encountered. Typically, the child process soon makes an exec system call. This call is used to exe-

cute a new program in the context of an executing process. the old address space of the process is discarded and is replaced by a new one that is appropriate for the new program to be executed. Hence, for load sharing Unix systems supporting only non-preemptive process transfer, the exec call should be considered as the time when a process is considered for migration. The advantage is that the execution image of the process need not be saved and transferred to the destination. Instead only the name of the executable file to overlay the process memory image, open file descriptors, signal related information, etc. are sufficient. Systems such as Solaris MC take this approach and consider a process for migration when it makes the exec call.

2.3 Semantics Changes

Unix process management semantics are not easily amenable to extending over a distributed environment. The reason behind this is that Unix was primarily designed to run on uniprocessor machines and design decisions were made with this assumption in mind. As a result, certain process management features of Unix are difficult to extend over a network without a very large performance penalty. For example, in Unix a child process shares the file offset that it inherits from its parent process. But when the child process executes in a different node from the parent node, such as sharing is difficult to achieve without reduced efficiency of the file operations.

A distributed environment also introduces certain problems which are not present in the case of uniprocessors. For example, the failure model of distributed system is much more complex than for the uniprocessor case. There is no concept of partial failure in the case of uniprocessors because node failure essentially means a complete system failure. However in a distributed system a node failure does not bring down the entire system, because the rest of the system can work in the usual manner despite the failure. For this reason also certain changes need to be made to Unix system call semantics. For example, in a distributed Unix environment, the parent and child processes can execute in different nodes. When the parent process calls wait, it waits until the child process dies and an intimation comes to the node executing the parent process from that executing the child process. But when the node executing the child fails, such an intimation is not sent to the node running

the parent process. When the node executing the parent process detects the failure, it assumes that the child has died. Such a situation never arises for uniprocessor environment because both of them execute in the same node and if the node fails, both the parent and the child process fail.

This discussion suggests that Unix semantics require a few modifications to adapt to a distributed environment for efficiency and fault tolerance. However, most of the existing programs should be able to run unchanged even after these modifications. Hence, while modifying the semantics, care should be taken so that the deviation from the standard semantics is kept minimal and that the modification is made to features very rarely used so that majority of the existing applications can run unmodified. For example, as mentioned earlier, file offset sharing is inefficient to support in distributed setting. Howsoever, the efficiency of the file operations improves if the semantics of fork are changed slightly, i.e., the parent and child do not share inherited open file offsets when they execute on different machines. Moreover, the file offset sharing is so rarely used that majority of the applications work correctly without offset sharing.

2.4 Filename Space

If transparent load sharing is to be achieved in a network of Unix workstations, all the machines need to have the same logical view of the file system. Typical implementations assume the existence of some underlying distributing file system that makes this possible.

Condor, however, uses a different approach. It does not require a uniform file name space on all machines. Instead, all file system related (and several other) system calls are forwarded transparently to the home machine (i.e. where the process originated by a remote system call mechanism. This option is clearly not very attractive since it is inefficient, and makes the migrated process heavily dependent on the originating machine (residual dependency). Thus if a machine goes down, then even the processes that originated on this machine but had earlier migrated to other machines cannot continue execution.

2.5 User Credentials

In Unix, each user has a unique user id. In addition, each user can belong to several groups or users, and each group also has a unique group id. The user and groups ides together form the credentials of a user and are used for all authentication purposes.

In the scenario of process migration, it is quite clear that for every user these ids should be same in all the participating machines. This can be easily achieved if the machines in the system are uniformly administered. For example, Yellow Pages (YP) or the Network Information Service (NIS) (31) can be used to maintain same user and groups ids in all the nodes I the system. The Condor approach of remote system calls in an alternative to this approach.

2.6 Heterogeneity and Executable Filename

Nodes of a heterogeneous distributed system can have different architectures and may be running different flavors of Unix. This means that for the same program, the executable files for different machines will, in general, be different. An argument of the exec call specifies the executable file name. If the process is migrated when it makes the exec call, the call is completed on the destination machine. Thus the filename specified as the argument of the exec call should refer to binary of the program for the destination machine. Since the caller of exec does not know where the process will be migrated, we require that same filename corresponds to the appropriate binary on each machine. Among the system we described, only LOCUS implements such a mechanism using the so called hidden directories [25].

2.7 Process Id and Process Group Id

In Unix, each process has a unique process id. Additionally, processes may be grouped into process groups and each such groups has a unique process group id. Now if processes can migrate, it is desirable that processes and process groups have network wide unique ids. Preserving uniqueness of Unix process id is fairly simple in a single machine environments as this merely requires the operating system to check its internal tables and

find an unused id. Extending this to a network environment adds certain problems as these tables are now distributed among different nodes and searching through all the tables is clearly infeasible.

In some user level implementations, the process id changes when a process migrates. This causes loss of transparency, since other processes may have the old id for migrated process and such a process would not be able to communicate (through signals, for example) with the migrated process. To avoid this situation some implementations create network-wide unique virtual process ids and virtual process group ides which are translated through a global layer to the local process ids and process group ids (18) (27). Solaris MC creates unique process id by partitioning the process id into two parts- node id and local process id, where node id is the node which created the process.

Ideally, the process and process groups ides should be network-wide unique and should not change, once assigned. Further it should be easy to find anew unused id for a newly created process. A new process id should not clash with an existing process group id. Also, give a process id, it should be easy to locate a process in the network. This is needed again to communicate with the processes. Broadcast messages are not acceptable for tracing a process because it creates unnecessary overhead for all other nodes in the system except the target node. Such a solution is also non-scalable.

An important issue in distributed Unix process management is whether the unmodified existing binaries can run in the system. The process ids also have an impact on this issue since some programs store process ids in their local variables. These programs will not work correctly if the new process id size is larger than the reserved space of any variable in the existing binary which is used to store process ids. Hence, for complete binary compatibility, size of the process ids must not be larger than sizes of native process ids of any node in the load sharing system.

Process groups are needed to send signals to a set of related processes. In the uniprocessor case, the process group id is typically kept in the process table entries. For sending a signal, the kernel would merely search the table and find out the processes in that group. But in a distributed environment the member processes may run in any node in the system.

Hence such a search would again be infeasible. What is required is a process group structure which will contain the member process ids and processes interested in sending a signal will simply receive this list of members from the node having the group structure. Like the process id, the node holding the process group structure should also be easily found from the process group id.

2.8 Signals

Unix uses signals to notify processes of asynchronous events. Signals thus are an abstraction of interrupts. A process can choose to ignore a signal, supply a handler for it, or let the kernel take default action for the signal when the signal is received. Further, signals can be blocked. At any instant of time, the signal mask of a process shows what signals are currently blocked. When a process calls exec, the disposition of all the signals that it was handling is changed OT default, since the signal handler function may not exist in the new program that the process will not execute.

When a Unix process migrates during the exec call, the information about how it will handle signals, which signals are currently blocked etc., needs to be sent to the target machine.

Signal numbers need to be standardized because certain signal values represent different signals for different operating systems. For example, SIGSTOP is 17 in 4.3 BSD, but 23 in SVR4. Moreover, some system also use non-standard custom signals. The signal SIGSTKFLT in Linux is an example of a non-standard signal which is sent to a process when a stack fault occurs., the mechanism should also be flexible enough to handle such non-standard customer signals.

2.9 Process Relationships and Wait

In Unix, processes are organized into a tree with a special process, called init, at the root. Every process in the tree has a parent and may have child process(es). The parent child relationship is set up when a process calls fork to create a new process. The caller of fork

is called parent of the newly created child process. Typical Unix implementations use pointers to process table entries to indicate the parent child relationships [5,6,7].

Maintaining the parent-child relationship of Unix process across the nodes in a network of workstations is a challenge to the designer of a load sharing mechanism. The problem here is that parent the child processes can run in different nodes. Hence, the normal approach of keeping pointers to process structure does not work for keeping the process relationship information. Moreover, failures can cause loss of relationship information, thereby damaging the process hierarchy.

The fact that the parent and children can run in different nodes in the system, makes implementation of the wait system call harder. The wait system call is used by a parent process to wait for its children processes to die. The parent process can use this call to obtain the exist status, and other information about its expired children. In (29) two methods have been proposed for cross-node wait implementation. In one method, called the pull model, the parent request information from the child when it does a wait. If the child process has not exited so far, a callback is established with the child so that the parent is informed when the child dies. In the second method, called push model, the children inform the parent about every state change (process exit or stopped while being traced by the parent). In the pull model no message is exchanged until parent does a wait, but setting up and tearing down the callbacks can be expensive if the parent has many children, since many callbacks will have to be created and revoked. The second method is more efficient, as practically child process exit is the only event of interest to the parent.

2.10 File Descriptions

There are four major abstractions supported by Unix for interaction of a process with the outside world- files, devices, sockets and pipes. These objects are accessed by a process by using open file descriptors. In the following subsections, we consider each of these abstractions separately.

2.10.1 Files

The files that were open in a process before migration should be reopened in the target machine. Hence, state information of open files has to be passed to the target machine including an identification filed to locate the files. The pathname of a file would be sufficient to locate the file under the assumption of existence of a global shared file system. But there are two problems associated with pathnames. First, the system does not store the file name anywhere after opening the file and second, it is not always possible to determine the path of a file given a file descriptor. A solution could be to save the path into some variable while opening or creating a file or duplicating a file descriptor. This technique has been followed by all the systems that support process migration at the user level. But this does not wear away the problem entirely. There could be some files that do not have any path in the file system. For example, a process can remove an open file. Such a file is accessible to the process though it does not have any name in the file system.

Another alternative to pathname for locating a regular file could be the file systems internal identification for the file. For example, in NFS, the file handle can uniquely identify a file in a given file system. If we append this file handle with the host id and an identification for the file system, we get a unique identification for the file.

Another problem unique to files is that in Unix a child process shares the read/write offset for the open files that it inherits from its parent process. Now if the child process migrates to another machine, it is not possible to maintain this sharing except at an exorbitant cost. Fortunately, such usage pattern is very rare. The reason for not using this feature in practice is the potential race condition between the two processes sharing the offset, resulting in unpredictable file contents. Thus the semantics of fork and exec usually need to be watered down a bit, i.e., the parent and child do not share the file offset for inherited open files once they execute in different machines. As mentioned, this is unlikely to cause any existing programs to stop working since this feature is rarely used.

2.10.2 Devices

Devices have an advantage over files that these always have names in the file system. But the problem with devices is that they are inherently stultify. Thus, the stateless distributed file systems, like NFS, usually do not support remote access to devices. Even if the underlying distributed file system does not support device access from a remote machine, it is possible to provide remote access to devices by using device servers. OSF/1 AD TNC and Solaris MC take this approach. The state of an open device not only includes its identification, but also the mode in which it is opened and other parameters needed to open the device. For example, a terminal device can be opened in three different modes (cooked, break or raw mode).

2.10.3 Sockets

A socket is a communication endpoint. Typically socket are used for communicating with processes on other machines on the network using TCP/IP. A socket is bound to a transport address, that in the context of TCP/IP is a combination of IP address of the machine and a port number. Since the IP address is fixed for a machine, transferring an open socket to another machine does not make sense in the existing infrastructure.

The shadow process approach, as used in the Condor system, can be used to handle this problem. In this approach a process, called the shadow, is kept in the originating machine of the migrated process. this process takes care of the local socket connections, devices and local file accesses of the migrated process. The migrated process communicates with its shadow process for communication, I/O etc. This approach uniformly handles all kinds of file descriptors, but is very inefficient. For example, consider a case where a process from machine A is transferred to machine B for remote execution while the process had been communicating with another process in machine B. Now, every message sent from the process to its peer (also in machine B) will first go to machine A and then bounce back. To overcome this problem what we need is a transparent transport address which is not attached to a particular machine. Such techniques are described in (13) and (29). A socket server and virtual socket based approach, as described in (27), can also be used for this purpose.

2.10.4 Pipes

Pipes are special kind of files that implement FIFO semantics for readers and writers. They are heavily used for communication among processes on the same machine. It has been observed that mostly single reader and single writer pipes are used in Unix systems and the Unix shell contributes to most of the pipe usage. The implementation of pipes differs from system to system. For example, earlier Unix systems used the file system for implementing pipes whereas BSD uses Unix domain sockets.

Pipes can be implemented in a distributed system using different approaches, viz. Using file system, using shadow process or over distributed socket implementation. Locus (25) and Sprite (20) implemented single reader single writer pipes in the shared file systems. Locus uses special protocol for this restricted pipe support. But modern day distributed file systems for Unix such as NFS (9) and AFS (10) do not support pipes. The shadow process approach can of course deal with pipes in exactly the same manner as for all other kinds of file descriptors. Special protocols are also used in Solaris MC and OSF/1 AD TNC to implement and migrate distributed pipes. In OSF/1 AD TNC, pipes are implemented over distributed Unix domain socket-pairs. A socket-pair representing a pipe is placed in the node executing a reader process. This reader process is called primary reader. The pipes are migrated by the system under certain conditions. For example when primary reader migrates, the pipe is also migrated along with it. Pipes can also migrate when the primary readers die.

2.11 Time

Unix processes can read current time stored in a machine through time or get time of day system calls. These system calls are obviously location dependent, i.e. the output of these calls depend upon the node on which these are executed. The system times of two different machines in a distributed system are always different because of the physical limitations of clock synchronization (32) in distributed systems. As a result, a process might experience that the time has flown backwards, when it migrates to a node whose clock lags behind the clock of the node from which the process has migrated. Condor and Sprite have

solved this problem by executing all the location dependent calls only in the home machine of the process so that all time values that the process perceives are relative to the clock in the home machine.

In systems supporting migration only during exec system call, the old process space (and hence all the local variables holding time values) is overlaid by the new program during migration. Hence, the process usually does not know the old time values seen by the process before migration. Although there is a possibility that the time value seen by the process before exec is saved in a known place (for instance in a file) to be read in by the new program, such a use is very rare. This means that a process migrated during exec usually would not see time moving backwards.

2.12 Standardization

Heterogeneity of operating system versions in a load sharing Unix environment require standardization of certain parameters so that they can be transferred across different architecture or operating system versions and the values from different machines can be compared. Parameters such as signal values, resource usage, nice value, disk quota, resource limit, timer values, etc. are examples of values that need to be standardized. As an example, BSD assigns a priority between 0 and 127 to each process whereas older versions of System V use a priority range of 0 to 31. Hence, the nice value of a migrating process has to modified to a standard scale before transferring the state to target node.

2.13 Fault Tolerance

Fault tolerance is an area where the earlier systems have not concentrated enough. Only a few systems consider node failures and recovery. Solaris MC provides failure recovery and defines node failure semantics for processes. It assumes that the processes that were in a node which has been crashed are dead and does cleanup operations. But most of the existing systems do not handle network failures. Since node failure and network failures are both quite common, the load sharing system should be prepared to handle both these kinds of failures.

Several issues crop up when one considers failure recovery. For example, how does a node detect that a failure has occurred, how does it determine the type of failure, what action should be taken when a failure is detected, and what should be done when the fault is finally removed from the system? When a node failure occurs, a node loses information regarding all processes which were running on the node. These processes might have had relationship was with process may be executing on another node and may do a wait. If the node that was executing the child process has become inaccessible, how long should the parent process wait to receive information about its child process? When the crashed node comes back up, the parent process should be informed that the child has died. When a crashed node comes back up, care has to be taken to ensure that it does not assign the same process id to a new process as it assigned to an old process that had migrated before the crash. Similarly, when partitions merge after recovery from a network failure, information needs to be exchanged so that no problems arise later.

CHAPTER 3 The Distributed Process Scheduling and Migration Protocol

In this chapter, The proposed protocol for load sharing (DPMP) and the conventions are described in details. The chapter starts with the design goals of the protocol. The protocol is currently at a premature stage and it has a few restrictions. These are discussed next. Then the conventions and the messages in the protocol are described. Load balancing algorithm of the protocol is discussed in the next chapter.

3.1 Goals

The design of the load sharing protocol and conventions are based upon the following objectives.

Remote Execution We wish to support only non-preemptive process migration in order to enable transparent process migration across heterogeneous nodes and also to reduced the overhead and complexity of the implementation of the load sharing mechanism.

Transparency The primary goal of our work is to support transparent remote execution of processes in a cluster of heterogeneous Unix workstations. The mechanism should also provide transparent access to every system resource (files, devices, sockets, pipes, etc.) and try to preserve the single system look of the system as far as possible.

Heterogeneity The protocol must support architectural, operating system and configuration heterogeneity in the system nodes. Processes should be able to migrate across heterogeneous machines.

Minimal Semantic Changes As mentioned earlier, a few changes are required in the Unix process semantics for improved efficiency and ease of implementation. The protocol should make minimal changes to the Unix system call semantics so that most of the existing programs can work without any modifications.

Scalability The protocol should scale well to a larger number of nodes. Thus, for example, broadcast messages and centralized components should be avoided as far as possible.

Fault Tolerance The protocol should consider and suitably handle both node and network failures. The semantics of the system calls in the face of failures should be reasonable and well defined.

Simplicity The load sharing protocol should be simple, so that, it is easy to implement and incorporate to any existing Unix system. We would like to use existing software and protocols to the extent possible. For example, we shall assume the existence of a standard distributed file system protocol for distributed file access rather than designing ourselves. Ideally, the protocol should be built on top of well accepted standard protocols such as TCP/IP, NFS etc.

Independence from Load Sharing Polices The mechanisms for process migration should not dictate the policies for load sharing. One should be able to use any load sharing policy with the protocol.

Portability As mentioned earlier, the user level implementations are portable while the implementations requiring kernel modifications are not. Hence, the protocol should mandate minimal (if possible no) changes to the Unix kernel so that most of it can be implemented at the user level.

3.2 Limitations

The current version of our protocol for load sharing does not migrate a process if it has a device, socket or pipe open at the time of migration. But the process can open devices, sockets or pipes after it has migrated to the new node. At the moment we have not designed protocols for transparent access to remote distributed devices, sockets or pipes. For transparent access to files, we use NFS as the underlying distributed file system. This NFS dependency has simplified the implementation of the protocol in user level implementation, as all the jobs may be easily accessible on each node of our distributed network. In the proposed protocol the victim process selection for the migration has not been

discussed, as it is a separate issue for the research, we are targeting on latest process for this purpose only.

Unix uses a session abstraction to represent user login sessions. A session is a collection of processes in those process groups to a terminal device, called the controlling terminal. A session may have a controlling terminal associated with it and all the processes in a session can read from or write to the controlling terminal. The controlling terminal assumes that the session is managed by a session leader process (typically the Unix shell) which has created the session and whenever its terminal driver detects a connection failure with the controlling terminal device, it sends a SIGHUP signal to the session leader. The leader is supposed to know every process group in the session and kill all the processes in the session by sending SIGHUP. In addition to help in session management, the controlling terminal also has special role in job control[(6] [7)] [8]. Jobs are essentially the process group under a session. Process groups of a session are divided into a foreground process group and other background process groups. At any instance, the processes in the foreground process group have exclusive access to the controlling terminal. They can read from or write to the controlling terminal of the session and the terminal signals generated by the controlling terminal are also sent to the processes in the foreground group only. Processes in the background groups, on the other hand, are stopped by the controlling terminal by sending SIGTTIN (SIGTTOU) signal, when those try to read from (write to) the controlling terminal. The session leader can change the foreground process group of a session by modifying the process group id parameter stored in the controlling terminal device.

Since the protocol currently does not support remote device access, a restriction is imposed that a process cannot access its controlling terminal when it is away from the node holding the controlling terminal. But such a process will receive the signals sent to processes in its session or process group. No support for remote devices also means that the session leaders cannot migrate when these have controlling terminals associated with them. This is because the session leader is the only process in a session which can set the foreground process because it does not have the access to the remote controlling terminal.

Essentially, under the current protocol only the processes with input and output redirected to files can be migrated across machines.

In Unix, programs are typically debugged either using the ptrace system call or through the/proc file system [6,8]. The ptrace system call can be used by a parent process to trace the execution of its child processes. Current version of the load sharing protocol does not allow a process to be traced by its parent through ptrace system call when they run in different machines. Further, standard distributed file system protocols like NFS do not allow mounting the /proc file system of a remote machine. Hence, a process cannot trace the execution of another remotely executing process in the current infrastructure.

The other restriction is related to the number of times a process can migrate. The protocol assumes that a process can migrate only once in its lifetime. Although this seems a severe restriction, in practice majority of the Unix processes call exec at most once in their lifetimes. Hence, this restriction will not cause any loss of load sharing opportunity in practice. On the other hand, this restriction greatly simplifies the protocol.

The protocol assumes that, the cluster of nodes participating in load sharing is formed statistically, and each machine knows all its peers in the system. A list containing the nodes in the cluster is kept in every machine. This list is called access list. For the time being the protocol does not support dynamic configuration of the cluster.

The load sharing protocol also assumes that the node failures are fail-stop in nature. Byzantine failures are not handled by the protocol. The protocol also assumes the existence of the connection oriented protocol like TCIP on top of which it is implemented. This assumption of having a connection oriented protocol simplifies the detection of failures.

3.3 Conventions

Some of the issues regarding process migration presented in the last chapter have to be resolved before describing the actual protocol for load sharing. Providing solution to these problems gives rise to a few convention and choices for implementation.

3.3.1 Uniform File System View

To preserve the single system look, the protocol assumes a more of less uniform view of the file system in every node of the system as well as same user id and group id for every user in the system. The NFS remote file system mounting facility would be used to mount the same remote file systems at the same mount points on all machines, so that a reasonably uniform view of the name space can be achieved on different machines. The Network Information Service (31) (NIS) can be used to maintain same user and group in al the machines in the cluster.

3.3.2 Pathname and Heterogeneity

We have seen in Section 2.6 that for migration across heterogeneous machines there is a need to access different executable files by the same pathname depending upon the node where the pathname is parsed. To achieve this, a convention is used that defines four read only kernel variables, $ARCH, $ARCHVER, $OS AND $OSVER corresponding to the architecture, architecture version, operation system and its version respectively. These variables are initialized by appropriate values corresponding to local architecture and operating system in every node in the cluster. For example, a Dec machine running OSF/1 can have $ARCH= DECAND $OS=OSF1 and an Intel 80386 PC running linux can have $ARCH= ï386 and $OS=Linux. While parsing a filename, all occurrences of these variables in the path should be replaced by the appropriate local values. Hence, a filename /home/usr1/a.dec.osf1.out in the former machine described above and to /home/usr1/a.i386. linux.out in the latter machine. This means that if a process calls exec (/home/usr1/a.$ARCH/$OS.out), the process can be transparently remote executed to any of the above two machines provided.

3.4 Building Blocks of the Proposed Process Migration System

(1). The Scheduler Daemon

The scheduler/server daemon. Only few scheduler daemons are needed to run only in the NIS servers. It will be an eternal process. Referred as *scheduler* in many places in this document.

(2). The IdleHost Daemon

One per machine. This is the implementation of execution subsystem as well as the load measurement system, which will be discussed in next sections. This will run in the form of a eternal process.

(3). The User Interface

This will be the common front-end given to the user, by which he can submit his job to the system. It will be referred as *initiator* sometimes.

(4). The SessionManager

One per JSMT session. Runs in the server machines. Stops when the session ends. Referred as the *backend* and *session manager* in some places in this document. Mostly it's name is used.

3.4.1 Functions of the daemons

- **initiator**

  Display the GUI, collects all user inputs and prepares a schedule file.

  Converses with *scheduler* to start the session corresponding to a schedule or to find out a *sessionManager* for job monitoring.

  Talks to *sessionManager* for job monitoring purposes.

- **sessionManager**

  Talks to *scheduler* to get a idle host for job scheduling.
  Communicates with the *idleHost*s of the idle hosts for scheduling and monitoring of jobs.
  Registers the status of all the jobs in a status file which can be used later for restarting an interrupted session.

- **idleHost**

  Measures the load of the machine periodically. Registers itself with it's scheduler whenever the load is less.

  Deregisters itself whenever the load crosses a threshold.

Sends ‘Migrate’ request, whenever it’s load reaches to the MIG_LOAD.

Receives job scheduling requests from the *sessionManager*s and either accepts or rejects them.

Schedules a new job / migrated job.

Receives job monitoring requests from *sessionManager*s and informs them about the status of the job(s).

- **scheduler**

  Receives registration/deregistration/migrate requests from all the *idleHost*s under it, updates it’s idle hosts’ list and informs it’s peers in the other NIS servers.

  Receives idle host request from it’s clients (i.e *sessionManager*s) and supplies the necessary information.

  Receives session start requests from the *initiator*s, starts them and shares the session information with it’s peers.

  Writes the session info. in a file once it receives information about the completion of the session.

  Receives session location requests from the *initiators*, locates the session and informs them.

  Start the *process migration* activity.

3.5 Strategy

In this strategy we have multiple schedulers running in the system. They run in the NIS servers of our network and will be serving the hosts in that domain. **initiators** started in a particular domain will contact only the scheduler in their NIS server for anything. The advantage in this method is that every **idleHost** and **initiator** need not know the name of the machine on which a scheduler is running. They can find this out by querying their own host which knows it’s NIS server. Moreover, if one of the NIS servers go down, only a part of the network is affected. The **idleHost**s periodically check the load of their hosts. They send *Register/Deregister/Migrate* messages to their schedulers based on the load value. Whenever **user** wishes to start a scheduling session he may invoke the **initiator** that con-

tacts it's scheduler with the necessary information of the job to be processed. The scheduler starts a **sessionManager** which will take care of the session. The **sessionManager** will generate idle host requests which will be added to a queue of the scheduler. The scheduler selects requests from this queue and matches them with a appropriate host in it's idle host list and informs the **sessionManager** about this which then send the job to the **idleHost** in that idle host. **idleHost** will schedule the job in it's machine. The *Idle Host List* and the *Process Migration Mechanism* are described in the next two sections.

3.6 The Idle Host List

The *idle host list* (**ihlist** henceforth) is the data structure in which a scheduler i.e **scheduler** stores information about all the idle hosts in it's NIS domain. Each host will have a weight associated with it. Initially, when a host registers itself, this weight will be having a value which will indicate that the host is idle (0 maybe). The weight changes as the host gets jobs for scheduling. The **ihlist** will be sorted based on this weight. As the responsibility of deregistration is with the client machines, this sorting will prevent the same host from being selected repeatedly for job scheduling and will distribute the jobs fairly. This sorting ensures that when a host is selected for scheduling, it will be the least loaded among it's kind.

3.7 The Process Migration Mechanism

In the proposed system, all the user jobs will be submitted by the initiators only, so that they can be monitored, processed and migrated from one host to other idle host depending upon the load of the machine. So this must be a stateful implementation where the details of each process will be kept by the scheduler in its internal data structures. This information will have a mapping of executing process with the host where the process is being executed.

For each user job/ group of jobs, a unique sessionManager is created, which requests for the hosts, keep the updated information of each process and help in the migration of process from one idleHost to other idleHost. The requests for idle hosts given by *sessionMan-*

agers are kept in a queue by *scheduler*. Selecting a job from a logical queue is based on the priority of the job which is calculated by the initiating *sessionManager* and sent as part of the job request. The scheduler gives the name of the idleHost for the job execution depending upon the Idle Host List maintained by it. Thus a session Manager submits the job on the specified idleHost and waits for the completion of the process on that idleHost.

Each and every idleHost, continuously, measures the load of the machine (the Load Measurement Algorithm, will be discussed in next chapter) while executing the user process in different thread. While executing the process, if the load the machine crosses the MIG_LOAD parameter than the idleHost sends the Migrate message to the scheduler, requesting it to migrate the process to another available idleHost. On receiving Migrate message, the scheduler reschedule the process on a different idleHost with the process state received in the migrate message. The sessionManager now resubmit the migrated process on the new idleHost with the current process state and continues the process till the end or till further migration. The whole migration activity is kept transparent to the user and is implemented in the proposed system itself.

3.8 Messages in Proposed Protocol

Messages used in the proposed system can be classified into the following types

(1). C2SCHAN Messages

Client to Scheduler Channel messages. These are exchanged between the schedulers in the NIS servers and their clients. Clients could be the job scheduling ones i.e *idleHost*s or the job initiating clients - *initiator*s and *sessionManager*s.

(2). CCHAN Messages

Client Channel messages. These are exchanged between the clients.

(3). ACHAN Messages

Administration Channel Messages. These are exchanged by the schedulers among themselves.

3.8.1 C2SCHAN Messages

The abbreviations in parentheses are used later in this document

(1). initiator to scheduler

1. Session Initiation Request (sinitreq)
2. Session Location Request (slocreq)
3. Session Restart Request (sessrestart)

(2). scheduler to initiator

1. Session Initiated (sinitok)
2. Session Initiation Failed (sinitfail)
3. Session location reply (slocreply)

(3). sessionManager to scheduler

1. Session finished (sessfin)
2. Idle host request (ihreq)
3. I am Alive (iamalive)

(4). scheduler to sessionManager

1. Idle host reply (ihreply)
2. New Idle host allocation for migration (ihreply_migrate)

(5). idleHost to scheduler

1. Idle host registration (ihregn)
2. Idle host deregistration (ihdregn)
3. Idle host migration request (proc_migrate)

3.8.2 CCHAN Messages

(1). sessionManager to initiator

1. Session Over (sessover)
2. Job status info. (jstinfo)
3. I am alive (iamalive)
4. Action failed (actionfail)

(2). sessionManager to idleHost

1. Job scheduling request (jsreq)
2. Job suspension request (jsuspreq)
3. Continue a suspended job (jcontreq)
4. Job termination request (jtermreq)

(3). idleHost to sessionManager

1. Job status info (jstinfo)
2. Job scheduling request rejected (jsreject)
3. Job completed successfully (jover)
4. Job terminated with error (jfail)
5. I am alive (iamalive)

6. No such job (nosuchjob)

3.8.3 ACHAN Messages

(1). scheduler to scheduler

1. Idle host registration (ihregn)
2. Idle host deregistration (ihdregn)
3. Idle host Migration request (proc_migrate)
4. Session start (sstart)
5. Session over (sessover)
6. Availability Metric Modification (AvmModMsg)
7. Idle Host List Updation Request (ihlUpdMsg)

There may be some more messages esp. **ACHAN** ones. We will identify them when we decide if it will be possible to use IP Multicasting. We may be able to do away with some of these messages by using IP Multicasting in some places.

3.9 The idle host registration/deregistration/migration process

1. If **idleHost** has just been started it does the following thing(s)
    1. Checks if there is another idleHost already running. If yes, refuses to start.
    2. Finds it's NIS server.
2. **idleHost** measures the load of the machine.
3. Checks where the load value lies - in the *idle range*, *busy range or migrate range*.
4. If this is the first time the load is being measured and if it is in the idle range a ***ihregn*** message is sent to the *scheduler.*
5. If this is not the first time, it checks the value of the previous measurement. If both the values lie in the same range, either *idle*, *busy or migrate range*, nothing is done except for replacing the previous measurement's value with the current value. However, if the

values lie in different ranges, a message is sent to the **scheduler** in the NIS server. In the following table **plv** stands for *previous load value* and **clv** stands for *current load value*.

PLV's Range	CLV's Range	Message Sent
idle	idle	-
busy	busy	-
migrate	migrate	-
busy	migrate	migrate_proc
busy	idle	ihregn
idle	migrate	migrate_poc
idle	busy	ihdregn
migrate	busy	-

TABLE 1. Load values and messages

6. The next measurement is done after N units of time.

3.10 The Scheduling Session

1. The user may submit his job by invoking initiator.
2. **initiator** becomes a background process and gives the prompt back to the user. *It may not relinquish the controlling terminal*.
3. It then displays the start-up window.
4. Once the jobs have been defined, **initiator** obtains scheduling related information from the user and validates it.
5. The real scheduling starts now. Information related to the jobs is with **jsmt** in it's memory (or may be in a file).

6. **initiator** finds the NIS server of the host in which it is running. It then requests the scheduler in that server to start a session for it's schedule (**sinitreq**). It will supply the necessary information about the session.

7. The scheduler (**scheduler**) checks if it is possible to start a new session. If it can't, it sends the message **sinitfail**. If it can, it starts the backend of jsmt which is called **sessionManager**. It makes a note of this session, informs it's peers and the initiator (**sinitok**).

8. One of the inputs given to **sessionManager** while starting it will be the name of the host from which the initiation request came. This is necessary for scheduling jobs with the scheduling type **Local**. No idle host request is sent to the scheduler for these jobs. They are sent to the initiating machine.

9. The user is free to stop the initiator as **sessionManager** will take over. If the user does not stop **initiator**, it enters the monitoring mode so as to monitor it's session. It sends a job monitoring request (**jmreq**) to the **sessionManager**. This will make **sessionManager** to send the status information of the jobs (**jstinfo**) to the initiator. If the user stops **initiator** during monitoring, **sessionManager** is informed about it (**jmterm**) so that it can stop sending status information.

10. **sessionManager** reads the information about the jobs, assigns priority to each job.

11. **sessionManager** selects the first job to be scheduled. Selection is based on the priority. **inititator** checks the requirements of the job and sends a request for an idle host (**ihreq**).

12. **scheduler** receives this request and adds it to the job queue.

13. **scheduler** selects a request from the request queue.

14. **scheduler** searches it's idle hosts list for a hosts which will be able to satisfy the requirements of the request. If such a host is found, it informs **sessionManager** (**ihreply**) and removes the request from the queue. If there is no suitable host, **scheduler** turns it's attention to the job of next higher priority.

15. When **sessionManager** receives the name of the idle host, it sends the complete details of the job to be scheduled to the **idleHost** of the idle host (**jsreq**).

16. The target host checks it's current load. If it can accept the job request, it schedules the job and sends the status of the job to the source host (**jstinfo**). Otherwise, it sends **jsreject**.

17. If the request is rejected, **sessionManager** contacts it's scheduler once again. On the other hand, if the request is accepted, **sessionManager** changes the state of the job from *Waiting* to *Running*.

18. When **sessionManager** has successfully scheduled all it's parallel jobs, it waits.

19. **sessionManager** sends an **iamalive** message periodically to it's scheduler and to all of it's monitoring **initiator**s. If this message is not received for a certain amount of time, the scheduler assumes that the session has been interrupted and registers this information in it's session info file. The monitoring **initiator**s can time out and inform the user and close the connection.

20. When a job is completed successfully, the **idleHost** informs **sessionManager** through the message **jover**. If the job stops due to some error the message **jfail** is sent instead.

21. **idleHost** sends **ws_iamalive** periodically to the source host's **sessionManager**.

22. When all the jobs are over, **sessionManager** sends the message **sessover** to the initiator, if it is still monitoring and deregisters it's session from it's scheduler (**sessfin**).

23. When the **idleHost** reaches to its migrate range, it get the current process state and fill the process state data structure.

24. idleHost constructs the the message **migrate_proc**, fills it with the updated process state to be migrated and stops the execution of that process.

25. The migrate_proc message has been sent from the idleHost to the scheduler.

26. On receiving the migrate_proc message the scheduler searches its Idle Host List data structure for the next available host.

27. **Scheduler** constructs the message **ihreply_migrate** with the stopped process current state and the new idleHost where the process can be resubmitted.

28. The **ihreply_migrate** message is sent to the sessionManager, the sessionManager resubmit the job on the new idleHost and updates its state with the *process state* data structure.

29. User's job is continued to be executed as it was earlier executing before the migration taken place. At the end **jover** message is sent back to the **sessionManager**.

30. It the initiator was monitoring the session, **sessionManager** sends **sessover** message to the initiator, acknowledging it about the completion of the job.

Once the scheduling starts, the user is free to close all the windows, including the start-up window. **sessionManager** will continue to write the session details in the disk.

3.11 A Job Monitoring Session

User can monitor the jobs submitted by him. It can be a part of the initiator or the user may have options for the job monitoring as well.

1. The user starts **initiator.**
2. The start-up window will be displayed and the user will click the '**Monitor Jobs**' button.
3. **initiator** gets all the relevant inputs from the user. It then requests it's scheduler to locate the session (**slocreq**).
4. The scheduler checks it's memory to see if any such session is active anywhere in the network. If yes, it sends the name of the host who started the session (**slocreply**). If there is no information in the memory, the scheduler checks it's history file and, gets the name of the session's log file and sends it back (**slocreply**). If no such session had existed, the scheduler send the message (**slocreply**).
5. If the session is not active, **initiator** reads the log file and displays it in the monitoring window. It also informs the user that the session is inactive.
6. If the session is active, **initiator** sends a monitoring request (**jmreq**) to the **sessionManager** who started the session.

7. **sessionManager** first sends information about all the jobs which have completed and then forwards all the **jstinfo** messages which it gets from it's target **idleHost**s.
8. If the user wants to terminate the monitoring process, **sessionManager** is informed about it so that it can stop forwarding monitoring information (**jmterm**).
9. When the session is over and if the monitoring **initiator** is still listening, **sessionManager** sends information about the completion of the session (**sessover**)

3.12 Rescheduling (jobs)

Rescheduling is done when the session is active and has stopped due to errors in one or more jobs. This can be done from the monitoring window. After rectifying the error(s), the user is required to specify a ***Rescheduling Start Point*** which is nothing but the job from which scheduling is to be continued. This can be done by simply clicking the job in the monitoring window. The message **sessresched** is sent to **sessionManager**.

3.13 Job manipulation

The job manipulation facilities are limited at present.

(1). A running job can be suspended (jsuspreq)

Obviously not all jobs can be suspended. This will be applicable to executables only.

(2). A suspended job can be continued (jcontreq)

(3). A running job can be terminated (jtermreq)

All these manipulations can be done on a job only by the user who started the session or by any privileged user. If the target **idleHost** is unable to find the job running in it's memory, it sends back a message to the effect (**nosuchjob**). If the specified action cannot be performed, the message **actionfail** is sent back.

3.14 Restarting (a session)

A session is restarted if it has been interrupted. Every **sessionManager** records the status of a session continually in a file. The session can be restarted by supplying the name of this file. This file can contain the following information

(1). All the scheduling information given by the user

(2). The session schedule computed by sessionManager

(3). The status of the jobs

The message **sessrestart** is sent by **jsmt** to the **Scheduler**.

3.15 JSMT Administration

Currently, the ***JSMT Administration Process (JAP),*** is limited to the dissemination of session and idle host information only. The following events/information is sent by a scheduler to it's peers.

(1). Idle Host Registration

(2). Idle Host Deregistration

(3). Session Start

(4). Session Over

3.16 The Possibility of Using IP Multicast

IP Multicast can be used for the following things

3.16.1 For exchanging messages between the schedulers

This is how we propose to do this. A common multicast host group address will be known to every scheduler. It will join this group as soon as it comes up. Every **ACHAN** message will be sent to this group which will automatically be forwarded to the members of the

group. The advantage here is that a scheduler need not know the names of the machines in which the other schedulers are running.

3.16.2 For sending and receiving session monitoring messages

This could be done in the following way. Every scheduler will have a set of multicast addresses with it. Whenever, a **sessionManager** is started, it is assigned one of these addresses. From the viewpoint of the **sessionManager** this is the address of the group of **jsmt**s who are interested in monitoring the session owned by **sessionManager**. Every **jsmt** which wishes to monitor a session is informed about this address and it has to join that group in order to receive status information.

3.17 Drawbacks

- The source directory of the jobs can, at present, only be a NFS mounted directory. Local disk directories are not supported.
- When a job is interrupted due to shutdown, power failure etc., it will only be restarted, not continued.
- For the current draft of the process migration protocol, the latest process which sends the idleHost to the migrate load range, is selected for the migration purposes. A proper process selection algorithm is to be devised which can select the appropriate process to be migrated. It is another topic of the research, so i am finding it difficult to cover this in the present thesis.

CHAPTER 4 Load Measurement Algorithm

4.1 Building blocks of Load Measuring System

In the current section it shows the design aspects of a distributed system where the load balancing is attempted to provide reasonable solution to measure the load of the machine in a parameterized formula form. The overall system may be divided into following sub-systems:

4.2 Load Balancing System

The load balancing subsystem is to be running on each and every workstation connected in LAN. Its main task is to get the system parameters and compute the load of the system using a proper Load Measurement Algorithm. Depending upon the load, the machine can be registered or de-registered for the further remote execution of the users' processes. The core of this system is Load Measurement Algorithm which will be discussed in next section. Following are the main components of the Load Balancing System:

4.2.1 Information Subsystem (idleHost)

The information subsystem is in charge of collecting and maintaining global information about each available node in the network and its load condition. This can be assumed to be a centralised resource for the informations about the runnable ready workstations with their respective system loads.

4.2.2 Decision Making Subsystem (scheduler)

The decision making subsytem is meant to keep the records of currently available hosts that can be treated as being Idle. The list of all available idlehosts is created by the registration & de-registration messages. The responsibility of this subsystem is to decide the most appropriate host for the requested program execution. This subsystem also schedules the users' jobs on the remote idle hosts.

4.2.3 Execution Subsystem

The execution subsystem has the responsibility to execute the user's job on the remote idle host being selected by the decision making subsystem and return the execution status of the process back to the user. This subsystem will use fork() & exec() system calls of the unix operating system to execute the remote process. Also it must provide a facility to initiate a process in a remote machine. This process interacts with the user and access to data files in a transparent manner, as it would be executing in the user local machine.

The Load Balancing System design and implementation are proposed to be incrementally developed. At the first the basic load formula will be discussed which may provide the base for any load balancing system. During this stage this system collects statistical data about the values of relevant variables under diverse user-selected distribution policies based on diverse metric approaches. These data base can be used to get the load value of any machine at the given time of experimentation.

4.3 Parameters for the Load Measurement Algorithm

There my be sufficient performance degradation due to the unbalanced load distribution. This may convey lower system throughput and increased response time for arriving jobs. We can devise a mechanism to determine a highly convenient, not necessarily optimum, execution site for an incoming job.

For such an algorithm to work following steps are to be taken care of:

1.What system parameters?

2.Formula for the load?

3.How system parameters can be collected?

4.The policy for the load collection?

4.4 Performance Metric System Parameters

A workstation in a LAN has many parameters that must be considered for its load measurement. These parameters may include the cpu idle percentage, memory statistics, input/ output statistics and the number of processes waiting in the run-queue. Any well-behaved operating system will give a CPU-bound process as much CPU as it has available, provided that the processing needs of all other processes are met as well. Because most systems use only a small fraction of their processing power, there is usually more than 90% free CPU available at any time. Thus CPU cycles will be wasted unless utilized. An extremely sensitive correlation exists between I/O traffic in the node and the response time for the I/O bound jobs. Similarly a correlation exists between the number of I/O packets being transferred by the network interface of the host and the completion time of the jobs on the node. So the total number of packets being transferred should be taken into account while calculating the load of the node. Depending upon the load of a workstation its memory statistics may change significantly. A heavier process may tend to result in more memory faults. So the average number of memory faults should also be considered for the load calculation of the machine. A loaded node may have bigger process run-queue. So the length of system's run-queue will be taken into account for the measurement of the load value.

4.4.1 CPU Busy Percentage

Abbreviation of central processing unit, and pronounced as separate letters. The CPU is the brains of the computer. Sometimes referred to simply as the processor or central processor, the CPU is where most calculations take place. In terms of computing power, the CPU is the most important element of a computer system.

The processor plays a significant role in the following important aspects of your computer system:

Performance: The processor is probably the most important single determinant of system performance in the PC. While other components also play a key role in determining per-

formance, the processor's capabilities dictate the maximum performance of a system. The other devices only allow the processor to reach its full potential.

Software Support: Newer, faster processors enable the use of the latest software. In addition, new processors such as the Pentium with MMX Technology, enable the use of specialized software not usable on earlier machines.

Reliability and Stability: The quality of the processor is one factor that determines how reliably your system will run. While most processors are very dependable, some are not. This also depends to some extent on the age of the processor and how much energy it consumes.

Energy Consumption and Cooling: Originally processors consumed relatively little power compared to other system devices. Newer processors can consume a great deal of power. Power consumption has an impact on everything from cooling method selection to overall system reliability.

Motherboard Support: The processor you decide to use in your system will be a major determining factor in what sort of chipset you must use, and hence what motherboard you buy. The motherboard in turn dictates many facets of your system's capabilities and performance.

CPU busy percentage is a critical parameter to judge the load of the CPU, at the run-time. The CPU

Cpu_busy_percentage = (CPU_BUSY_SEC * 100) / CPU_ONLINE_SEC. (EQ 1)

4.4.2 Number of Processes in the run queue

A process is an execution stream in the context of a particular process state. It is an execution stream as a sequence of instructions. Process state determines the effect of the instructions. It usually includes (but is not restricted to):

+ Registers

+ Stack

+ Memory (global variables and dynamically allocated memory)

+ Open file tables

+ Signal management information.

In multiprogramming systems like unix multiple processes run at a time. It allows system to separate out activities cleanly. Multiprogramming introduces the resource sharing problem - which processes get to use the physical resources of the machine when? One crucial resource: CPU. Standard solution is to use pre-emptive multitasking - OS runs one process for a while, then takes the CPU away from that process and lets another process run. Must save and restore process state. Fairness is a key issue it must ensure that all processes get their fair share of the CPU.

For accounting reasons the operating system keeps track of how much time is spent in each user program. It also keeps a running sum of the total amount of time spent in all user programs. Two threads increment their local counters for their processes, then concurrently increment the global counter. Their increments interfere, and the recorded total time spent in all user processes is less than the sum of the local times. So there are various activities are being carried by the operating system to handle the various processes in unix like multiprogramming environment. So the number of simultaneously executing process gives a parameter for load measurement.

4.4.3 Total number of input output packets: IO statistics

A workstation which is connected to a LAN also undergoes with a lot of layering for the IO traffic control. It may receive various input packets at the same time it may send messages to the remote host. So there are two important concepts to study the load for the IO operations:

- Sending a message on the network

An outgoing message begins with an application system call to write data to a socket. The socket examines its own connection type and calls the appropriate send routine (typically INET). The send function verifies the status of the socket, examines its protocol type, and sends the data on to the transport layer routine (such as TCP or UDP). This protocol creates a new buffer for the outgoing packet (a socket buffer, or struct sk_buff skb), copies the data from the application buffer, and fills in its header information (such as port number, options, and checksum) before passing the new buffer to the network layer (usually IP). The IP send functions fill in more of the buffer with its own protocol headers (such as the IP address, options, and checksum). It may also fragment the packet if required. Next the IP layer passes the packet to the link layer function, which moves the packet onto the sending device's xmit queue and makes sure the device knows that it has traffic to send. Finally, the device (such as a network card) tells the bus to send the packet.

- Receiving a message from the network

  An incoming message begins with an interrupt when the system notifies the device that a message is ready. The device allocates storage space and tells the bus to put the message into that space. It then passes the packet to the link layer, which puts it on the backlog queue, and marks the network flag for the next ‘‘bottom-half’’ run; The bottom-half is a system that minimizes the amount of work done during an interrupt. Doing a lot of processing during an interrupt is not good precisely because it interrupts a running process; instead, interrupt handlers have a ‘‘top-half’’ and a ‘‘bottom-half’’. When the interrupt arrives, the top-half runs and takes care of any critical operations, such as moving data from a device queue into kernel memory. It then marks a flag that tells the kernel that there is more work to do - when the processor has time - and returns control to the current process. The next time the process scheduler runs, it sees the flag, does the extra work, and only then schedules any normal processes.

There are lot of processed involved for sending a message on the network and receiving a message from the network. So as the number of input and output packets received by the workstation on a LAN increases its load characteristics also get changed. So the total number of input and output packets gives a valuable parameter for the load measurement.

4.4.4 Memory Page Faults: Memory statistics

Virtual memory is divided up into pages, chunks that are usually either 4096 or 8192 bytes in size. The memory manager considers pages to be the atomic (indivisible) unit of memory. For the best performance, we want each page to be accessible in Main memory as it is needed by the CPU. When a page is not needed, it does not matter where it is located. A page fault occurs when the CPU tries to access a page that is not in main memory, thus forcing the CPU to wait for the page to be swapped in. Since moving data to and from disks takes a significant amount of time, the goal of the memory manager is to minimize the number of page faults. Where a page will go when it is "swapped-out" depends on how it is being used.

Writing a page to disk need not wait until a page fault occurs. Most modern UNIX systems implement pre-emptive swapping, in which the contents of changed pages are copied to disk during times when the disk is otherwise idle. The page is also kept in main memory so that it can be accessed if necessary. But, if a page fault occurs, the system can instantly reclaim the pre-emptively swapped pages in only the time needed to read in the new page. This saves a tremendous amount of time since writing to disk usually takes two to four times longer than reading. Thus pre-emptive swapping may occur even when main memory is plentiful, as a hedge against future shortages.

Since it is extremely rare for all (or even most) of the processes on a UNIX system to be in use at once, most of virtual memory may be swapped out at any given time without significantly impeding performance. If the activation of one process occurs at a time when another is idle, they simply trade places with minimum impact. Performance is only significantly affected when more memory is needed at once than is available. This swapping and associated page faults can be used to measure the load of the machine.

4.5 Parameterized formula for the Load Measurement

As a result of above presented hypothesis about the load of a workstation it may be concluded that the load(L), can be considered as follows:

L = f(cpu_busy_percentage, mem_faults, io_stat, proc_run_queue); (EQ 2)

where,

L = Load of the workstation

cpu_busy_percentage = system given cpu busy percentage

mem_faults = memory faults generated by cpu at the time of load measurement

proc_run_queue = total number of runnable processes at the time of load measurement

This association between the parameters have to be formulated. The attempt to get the association among them lead to devise following algorithm.

1. Decide about the time interval for the collection of data. The above parameters are interval specific, so a proper time interval is needed to get their values. The time interval t, should be as minimum as possible. (Ideally, t -> 0)
2. Get a scale to measure the load of the workstation. Since the time taken in the execution of a cpu bound time consuming process is directly proportional to the system load. So the execution time of that process can be a good indicator of the load of a machine in the units of time.
3. Collect the samples of the load value as an individual function of the parameters defined above (cpu_busy_percentage, mem_faults, io_stat, proc_run_queue).
4. Plot the samples to get the graph between load vs parameters values.
5. Using the best fit strategy a smooth curve can be drawn (with minimum deviations) that will give the load as the function of each parameter individually.
6. Take the weighted average of the various load values to get the parameterized formula for the Load Measurement of the machine. (These weights have to be assigned heuristically)

The various plots are being present below for the Dec Alpha Server (RISC Architecture, running digital OS, named as OSF). Here the load of the machine is calculated as the time taken to execute a significantly long process and the load parameters are collected at time intervals of 10 seconds.

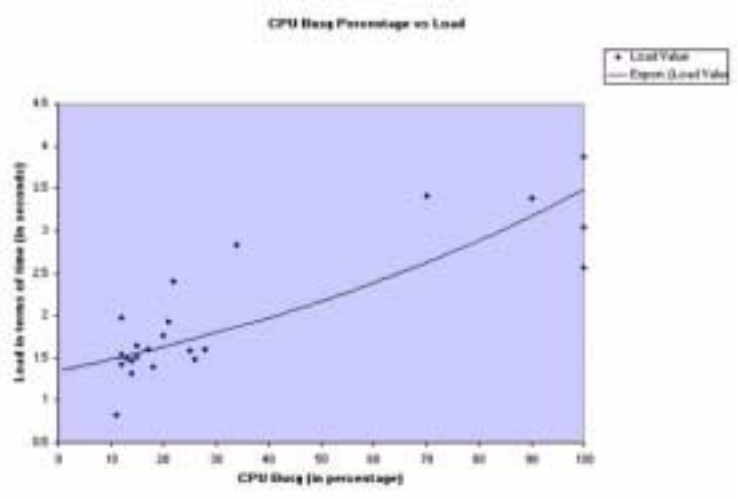

FIGURE 1. Graph between Load of a machine and CPU Busy Percentage

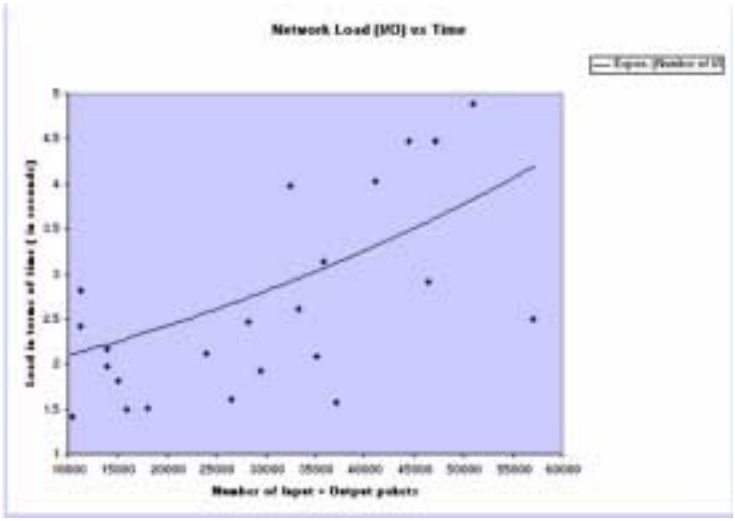

FIGURE 2. Graph between Load of a machine and I/O statistics

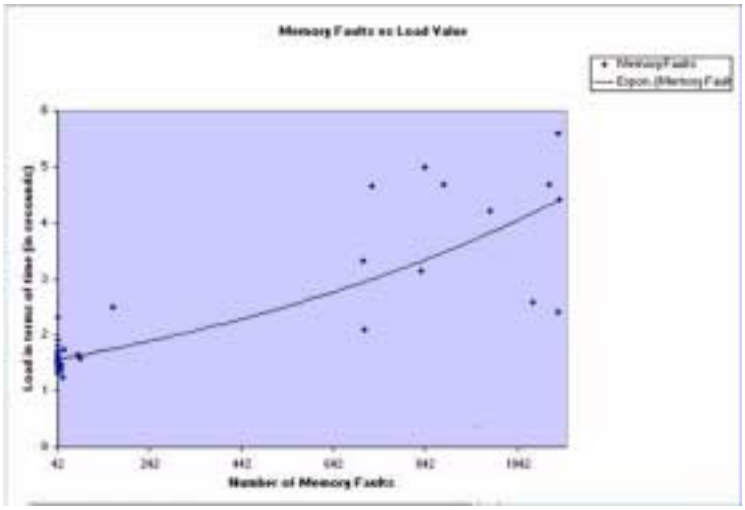

FIGURE 3. Graph between Load of a machine and Memory Faults

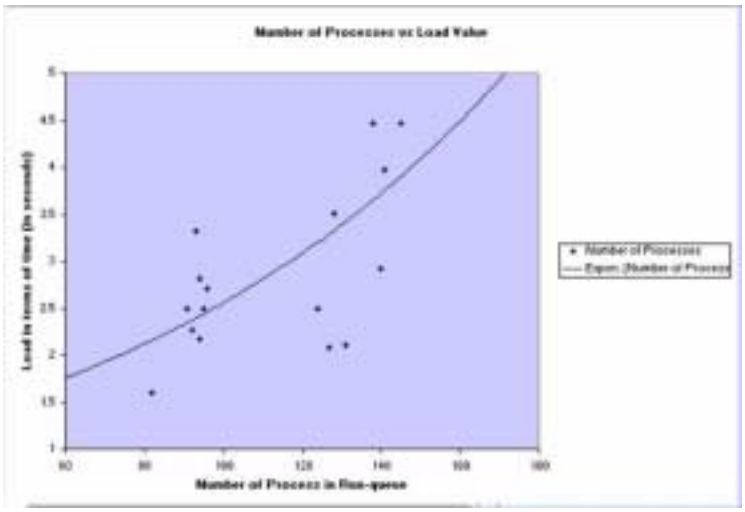

FIGURE 4. Graph between Load of a machine and Length of Run-queue

The samples for the above graphs have been collected using the output of 'vmstat' command of unix. These samples have been collected at different points of time to get the vast distribution of the domain value. The major events when data collection was done were:

- During **link generation time**, more cpu intensive work was going on.
- During **back-up time**, more network processing was going on.
- During **normal processing time**, number of processes in the run-queue was high.
- During normal procession time, when number of users were working, generating lot of **page faults**.

The graphs with the best fitted strategy have been used to get the trend of these observations. These graphs have been plotted with minimum standard deviation. It was found that the exponential graph fitting provides least deviation, so for all the observations exponential graphs have been plotted.

Let lc, ln, lf and lp are the load values corresponding to the load parameters as cpu_idle_percentage, network_io_packets, memory_faults and number_of_process in run-queue respectively. So from above graphs it can be shown that:

lc = mc * exa*cpu_busy_per + Cc; **(EQ 3)**

Similarly, load measured for network_io_packets is:

ln = mn* exn*network_io_packets + Cn; **(EQ 4)**

Load value corresponding to memory page faults is:

lf = mf* exf*memory_faults + Cf; **(EQ 5)**

whereas the load value for the number of processes in run-queue is:

lp = mp * exp*number_of_process + Cp; **(EQ 6)**

In above equations, Cc, Cn, Cf and Cp are the constants that correspond to the minimum load value of the workstation for the respective parameters. From above four equations the

net load of the workstation can be calculated with some weighted average of these values. If L denotes the net load of the machine then:

L = lc + ln + lf + lp; **(EQ 7)**

So the Load Value of a machine is:

L = mc*exa*cpu_busy_per + mn*exn*network_io_packet + mf* exf*memory_faults + mp* exp*number_of_process + C **(EQ 8)**

mc, mn,mn and mp are named as multiplication factor and the exponents xa, xn, xp and xf are named as exponent factors. Their values are shown in following table, which are taken from the inclinations of the above graphs.

TABLE 2. Values of Multiplication Factor and Exponent Factor for DEC Alpha Server (RISC Architecture)

Load Parameter	Multiplication Factor (m)	Exponential Factor (x)
cpu_busy_percentage	1.32	9.1x10-3
Network_io_pkts	1.8	1.5x10-5
Number_of_processes	1.04	6.3x10-3
Tot_mem_faults	1.51	9.45x10-4

At any instant depending upon the system parameters, the load value of the machine can be calculated. This load value is a relative term, it only gives the value for the posterior analysis. So depending upon some threshold value, the host that can be considered as idle, may be selected for the user's job submission.

Lmin = 1.32 + 1.047 + 1.514 + 1.80 = 5.56 (EQ 9)

A machine may be treated as a loaded one, when it's load value is more than Lthreshold. Using some heuristic it could be a sufficiently good approximation that Lthreshold will be a value at which it assumes 50% of the fully loaded value. In this way Lthreshold can be determined.

Lthreshold = 2.4 + 2.8 + 3.3 + 3 = 11.5 (EQ 10)

Heuristically we can defined the migrate range, let it be 70% of fully loaded value so, from the above formula we can find.

$$L_{migrate} = 3.2 + 3.36 + 4.1 + 4.3 = 14.96 \quad \text{(EQ 11)}$$

The simulation and results for the above derivation can be seen in the Appendix B.

CHAPTER 5 Implementation Design and Message Structures

Distributed systems are based on message passing semantics. There is no concept of shared memory as all the nodes have their own different system image. So the communication among the nodes is carried out by the messages between them. These messages may be the control message or they may be the information sharing messages. In this chapter some of the important messages for the scheduling and migration of the processes have been discussed.

5.1 Common Structures

In this section, we list some of the structures which are used in more than one message.

5.1.1 Message Header

All the messages have a header. At present, it is a very simple one and is of the following structure

```
typedef struct
{
   int   whatMesg;
} MsgHeader;
```

The value of the member **whatMesg** will indicate the type of the message and will aid the receiver to decipher the rest of it.

5.1.2 Session Details

```
typedef struct
{
   char  title[];
   char login[];
```

```
    char type[];
    char release[];
    char file[];
} SessionInfo;
```

This structure is used in some of the session related messages.

5.1.3 Host Details

```
typedef struct
{
    int nProcesses;
    int maxProcesses;
    char mountedDirs[][];
    char otherFacilities[][];
} HostDetails;
```

This is used in messages related to hosts. The member **otherFacilities** will contain a list of tools or libraries or anything else which is unique to a particular host or type of hosts.

5.2 Messages

5.2.1 Session Initiation Request

```
typedef struct
{
    MsgHeader mhead;
    SessionInfo sinfo;
} SessionInitReq;
```

This message is sent by **jsmt** to it's scheduler at the start of a session. The scheduler will extract information from this and store it in it's memory for future references and initiate a session.

5.2.1.1 Members

- **mhead.whatMsg**

  Will contain the value **SESS_INIT**.

- **sinfo.title**

  The title of the session as given by the user.

- **sinfo.login**

  The login from which the session was started.

- **sinfo.type**

  The type of the session. Presently, the following types are supported

  (1). ***LGEN*****:** Link Generation

  (2). ***PGEN*****:** Patch Generation

  (3). ***EXEC*****:** Executable

- **sinfo.release**

  In **LGEN** and **PGEN** sessions this will contain the release.

- **sinfo.file**

  The PERT file.

5.2.2 Session Initiation Successful

```
typedef struct
{
    MsgHeader mhead;
    int port_for_JM_reqs;
} SessionInitOk;
```

This message is sent by the scheduler to a jsmt when a session has been successfully started and registered.

5.2.2.1 Members

- **mhead.whatMsg**

  Will contain the value **SESS_INIT_OK**

- **port_for_JM_reqs**

  The port in which the **sessionManager** created for managing this session will be listening for job monitoring requests.

5.2.3 Session Initiation Failed

```
typedef struct
{
    MsgHeader mhead;
    char reason[];
} SessionInitFail;
```

This message is sent by the scheduler to a jsmt when a session initiation attempt fails.

5.2.3.1 Members

- **mhead.whatMsg**

  Will contain the value **SESS_INIT_FAIL**.

- **reason**

  The reason for failing.

5.2.4 Session Restart

This message is sent by **jsmt** to it's scheduler when it wants to restart an interrupted session. The structure of this message identical to that of *Session Initiation Request*.

5.2.4.1 Members

- **mhead.whatMsg**

  Will contain the value **SESS_RESTART**.

- **sinfo.file**

  The name of the status file.

5.2.5 Session Finished

```
typedef struct
{
    MsgHeader mhead;
    int sid;
} SessionFin;
```

This message is sent by **sessionManager** to it's scheduler when a session is over. The scheduler, on receiving this, will remove the session information from the memory and store it in the disk. This is also sent by **sessionManager** to a monitoring **jsmt**.

5.2.5.1 Members

- **mhead.whatMsg**

  Will contain the value **SESS_FIN**.

- **sid**

  The session ID provided by the scheduler. This will be a '*don't care*' value when the message is sent by **sessionManager** to a monitoring **jsmt** as it will be monitoring only one session and will inherently know which session is over.

5.2.6 Session Reschedule

The jobs of a session are rescheduled after error rectification. This is done on active sessions. This message is sent by **jsmt** to **sessionManager**.

```
typedef struct
{
    MsgHeader mhead;
```

```
    char startPoint[];
} SessionResched;
```

5.2.6.1 Members

- **mhead.whatMsg**

  Will contain the value **SESS_RESCHED**.

- **startPoint**

  The ID of the job from which rescheduling should start.

5.2.7 Session Location Request

```
typedef struct
{
    MsgHeader mhead;
    SessionInfo sinfo;
} SessionLocReq;
```

This message is sent by any **jsmt** who wishes to do job monitoring to it's scheduler.

5.2.7.1 Members

- **mhead.whatMsg**

  Will contain the value **SESS_LOC_REQ**.

- **sinfo.login**

  The user who initiated the session.

5.2.8 Session Location Reply

```
typedef struct
{
    MsgHeader mhead;
    char title[];
```

```
        char host[];
        int active;
        union
        {
         int port_for_JM_req;
         char sessionLog[];
       }input;
    } SessionLcnReply;
```

This message is sent by **scheduler** to **jsmt** in response to the **Session Location Request** message and contains information about the host on which the requested session was started.

5.2.8.1 Members

- **mhead.whatMsg**

  Will contain the value **SLR_SESSION** if a matching session is found, **SLR_NOSUCHSESSION** otherwise.

- **title**

  The title of the session.

- **host**

  The host in which the session was started.

- **active**

  This flag will be **True** if the session is still running and **False** if it has already been completed.

- **input.port_for_JM_req**

  If the flag **active** is **True**, this will contain the port number in the host machine to which job monitoring requests are to be sent.

- **input.sessionLog**

If the session is over, this will contain the name of the log file which contains the details of the session.

5.2.9 Idle Host Request

```
typedef struct
{
    MsgHeader mhead;
    char jid[];
    int hostType;
    char hostName;
    HostDetails resourceReqs;
} IdleHostRequest;
```

This message is sent by **sessionManager** to it's scheduler for obtaining a host for a job.

5.2.9.1 Members

- **mhead.whatMsg**

  Will contain the value **IHREQ**.

- **jid**

  The identity of the job for whom a host is being requested.

- **hostType**

  Type of the required host. It will be either a combination of **HP**, **Digital**, and **Sun** or the value **Any**.

- **hostName**

  The name of a host in case a specific host is required. If both **hostType** and **hostName** have valid values, they should be matching.

- **resourceReqs.mountedDirs**

  The list of directories which should be accessible in the required host.

- **resourceReqs.otherFacilities**

The list of facilities which should be provided by the required host.

5.2.10 Idle Host Allocation

```
typedef struct
{
    MsgHeader mhead;
    char host[];
    char jid[];
} IdleHostAlloc;
```

This message is sent by the scheduler in response to a idle host request and is received by **sessionManager**.

5.2.10.1 Members

- **mhead.whatMsg**

  Contains the value **IHA_HOST**.

- **host**

  The name of the idle host.

- **jid**

  The job for which the host has been allocated.

5.2.11 Idle Host Registration

```
typedef struct
{
    MsgHeader mhead;
    int hostType;
    char hostName[];
    HostDetails availableResources;
} IdleHostRegn;
```

This is sent by the **idleHost** of an idle machine to it's scheduler.

5.2.11.1 Members

- **mhead.whatMsg**

  Will contain the value **IHREGN**.

- **hostName**

  May be redundant.

- **availableResources.mountedDirs**

  Directories which are accessible.

- **availableResources.otherFacilities**

5.2.12 Process Migration Request

```
typedef struct
{
    MsgHeader mhead;
    int hostType;
    char hostName[];
    ProcessDetails procState;
} ProcessMigrationReguest;
```

This is sent by the **idleHost** of an idle machine to it's scheduler.

5.2.12.1 Members

- **mhead.whatMsg**

  Will contain the value **PROCMIGRATION**.

- **hostName**

  May be redundant.

- **procSatate.procName**

Name of the process.

- **procState.procState**

  Complete process state that is to be transferred to another machine.

5.2.13 Process Scheduling Request

```
typedef struct
{
    MsgHeader mhead;
    char pid[];
    char  pname[];
    char srcDir[];
    char logFile[];
    char cwd[];
    int   processType;
    char commandLine[];
    uid_t puid;
    gid_t pgid;
} ProcSchedRequest;
```

This is the message sent by **initiator** to a **idleHost** when it wants to schedule a job.

5.2.13.1 Members

- **mhead.whatMsg**

  Will contain the value **PROCSCHED_REQ**

- **pid**

  The ID of the job to be scheduled.

- **pname**

  The name of the job.

- **srcDir**

The directory in which the job (file) is present.

- **logFile**

  The log file in which the output of the job is to be stored if the job is a background one.

- **cwd**

  The directory in which the job is to be executed.

- **commandLine**

  The command line of the job.

- **puid**

  The user ID with which the job is to be executed.

- **pgid**

  The group ID with which the job is to be executed.

5.2.14 Process Status Information

There are two different structures representing this message. The following structure is used when the message is sent by a **idleHost** to a scheduling **sessionManager**.

```
typedef struct
{
    MsgHeader mhead;
    char pid[];
    int status;
} ProcStatus_jw2jb;
```

5.2.15 IamAlive, Session Over and Idle Host Deregistration

These four messages have the following simple structure

```
typedef struct
{
```

```
    MsgHeader mhead;
};
```

The value of **mhead.whatMsg**

Message	Value
I am Alive	IAMALIVE
Session Over	SESSOVER
Idle Host Deregistration	IHDREGN

The message **iamalive** is sent by **idleHost** to a **sessionManager** and by **sessionManager** to it's scheduler and it's monitoring **jsmt**s periodically. The **sessover** message is sent by **sessionManager** to a **jsmt** when the session is completed. The idle host deregistration is sent by **idleHost** to it's scheduler when the machine load increases beyond a tolerable limit. **jmterm** sent by a monitoring **jsmt** to the **sessionManager** which provides the status information to terminate the monitoring session.

5.2.16

```
typedef struct
{
    MsgHeader mhead;
    char pid[];
};
```

This simple structure is used by the following messages:

Message	Sender	Receiver
PROC_REJECT	idleHost	sessionManager
PROC_SUCC	idleHost	sessionManager
PROC_FAIL	idleHost	sessionManager
NOSUCHPROC	idleHost sessionManager	sessionManager jsmt

5.2.17 Action Failed

```
typedef struct
{
    MsgHeader mhead;
    int action;
    char reason[];
};
```

Sender	Receiver
idleHost	sessionManager
sessionManager	initiator

This message is sent when an action requested by the user cannot be done.

CHAPTER 6 System Modelling - Analysis and Design

This chapter describes the **Analysis** part of OMT. We haven't written a separate problem statement and picked objects from that. We have picked our objects from the description of the architecture of the system.

6.1 Object Modelling

6.1.1 The initial object list

- Scheduler
- Idle Host List
- Idle Host Info.
- Idle Host Request List
- Idle Host Request
- Session List
- Session Info.
- Idle Host
- Job Server

  The daemon which accepts external jobs and schedules them in the local host.
- Job Initiator
- Session
- Job

The association among the above object can de seen in following diagram:

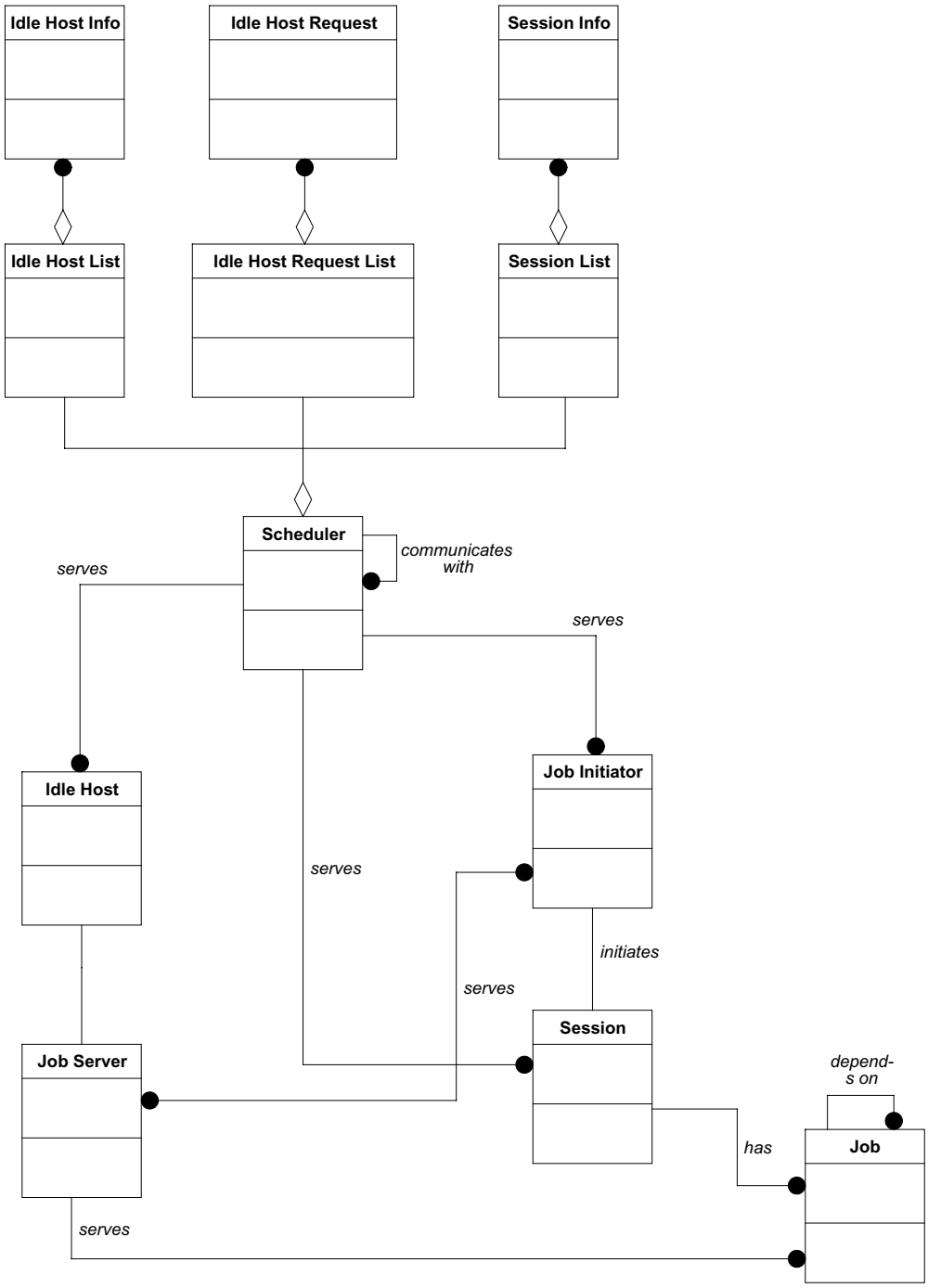

FIGURE 5. Association of various object in the proposed protocol

6.1.2 Data Dictionary

- **Job Initiator**

The front end which takes input from the user and requests the scheduler for the start of a session.

- **Job Monitor**

  That part of the front end which helps the user to monitor the jobs of a session; converses with the **Scheduler** and **sessionManager** for this.

- **Session**

  A collection of jobs; initiated by a **Job Initiator** and assigned by the **Scheduler** to a **sessionManager** which will manage it till the end.

- **Status File**

  Contains information about the jobs of a session.

- **Job**

  The unit of JSMT which is scheduled in an **Idle Host**. A job's scheduling can depend on that of zero or more of others.

- **sessionManager**

  The backend of jsmt which starts and manages a scheduling session; also takes part in the monitoring session.

- **Idle Host**

  A machine which is ready for hosting jobs. It keeps track of it's load and informs the **Scheduler** about it's willingness to accept jobs. Once a job is scheduled, this object informs the backend about the status of the job.

- **Scheduler**

  The central object of the JSMT system. This keeps track of all the idle hosts in the network, receives requests for idle hosts from **sessionManager**s and allocates hosts for them. It also keeps track of all the sessions of the network - both active and completed ones. All the schedulers share their information.

- **Idle Host Info**

  Details which an idle host send about itself.

- **Idle Host List**

  A collection of **Idle Host Info** objects maintained by the **Scheduler**.

- **Idle Host Request**

  The request for an idle host.

- **Idle Host Request List**

  A collection of **Idle Host Request** objects from which the **Scheduler** selects requests one by one and allocates hosts from the **Idle Hosts List**.

- **Session Info**

  Details of a session - title, initiator etc., sent by a **Job Initiator** to the **Scheduler** at the beginning of a session.

- **Session List**

  A collection of **Session Info** objects maintained by the **Scheduler** for monitoring purposes. Information about currently alive jobs are maintained in the memory of the **Scheduler** and information about completed ones are stored in a file for future reference.

6.1.3 Associations

1. The front end can be a job initiator or a job monitor.
2. A job initiator talks with only one scheduler.
3. The job initiator initiates a session.
4. Each session is made up of many jobs.
5. A job may (or may not) depend on other jobs.
6. A session is started and managed by a sessionManager.
7. Each session has a status file.
8. The status file is generated by the sessionManager handling the session.
9. The monitor converses with a sessionManager for getting the status of the jobs of a session.

10. The Scheduler has three types of information with it - the Idle Hosts List, the Idle Hosts Requests' List and the Session List.
11. The Idle Host List contains information about the all the idle hosts who have reported their idleness to the server.
12. The Idle Hosts Requests' List has all the requests for idle hosts received by the Scheduler.
13. The Session List is made up of information about all the sessions in the network.
14. The Scheduler shares information with it's peers.
15. The Scheduler creates many sessionManagers for managing sessions.
16. The Scheduler keeps track of many idle hosts in it's domain.
17. The Scheduler also talks with many job initiators and job monitors.

6.1.4 Attributes

6.1.4.1 Idle Host Info

- name
- current number of processes
- maximum processes that can be supported
- mounted directories
- other facilities

6.1.4.2 Idle Hosts List

- number of idle hosts

6.1.4.3 Idle Host Request

- ID of the job for which the host is required
- Type of the job

- Type of the required host
- Name of the required host, if known
- Maximum number of processes created by the job at any time
- The priority of the job
- Required directories
- Required facilities

6.1.4.4 Idle Host Request List

- Number of idle host requests

6.1.4.5 Session Info

- Session title
- Login of the initiator
- Session type
- Release, if needed
- PERT file
- Session ID

6.1.4.6 Session List

- Number of alive sessions

6.1.4.7 Scheduler

- Number of sessionManagers
- Details of each sessionManager - process ID, listening port.

6.1.4.8 sessionManager

- Name of the PERT file

6.1.4.9 Idle Host

- Name
- Type
- Load (average)
- Current number of processes
- Maximum number of processes that can be supported
- Mounted directories
- Other facilities

6.1.4.10 Process

- Process ID
- Process Name
- Source Directory
- Log File
- Working Directory
- Command line
- User ID
- Group ID
- Scheduling type
- Execution host type
- Execution host, if needed
- Expected execution time

6.1.4.11 Session

- Session title
- Initiator’s login

- Session type
- Release
- Name of the PERT file

6.1.4.12 Status file

- Name

6.1.4.13 Job Initiator

- Login
- User ID
- Group ID

6.1.4.14 Job Monitor

- Login
- User ID
- Group ID
- The multiplicity of the association *has* between the classes **Session** and **Job** on the latter's side has been modified from zero or more to one or more. This is because a session comes into existence only if there is atleast one job.
- A similar modification has been done in the association *serves* between the classes **Idle Host** and **sessionManager** in the former's side.
- The class **Scheduler** was previously having a aggregate relationship with the classes **Idle Host Info**, **Idle Host Request**, and **Session List**. This has been converted to three one-to-one associations named *maintains* between the **Scheduler** and the three classes.

6.2 Dynamic Model

6.2.1 Scenarios and Event Trace Diagrams

6.2.1.1 Process Scheduling

- The **Job Initiator** sends a session start request to the **Scheduler**.
- The **Scheduler** spawns a **sessionManager** to handle the new session.
- The **Scheduler** updates the **Session List** and informs the **Job Initiator** about the successful start of the session.
- The newly spawned **sessionManager** starts sending idle host requests for it's jobs.
- The **Scheduler** adds these requests to the **Idle Host Request List**.
- The **Scheduler** selects a request from the list and asks the **Idle Host List** for a matching host.
- When the **Scheduler** gets the idle host's name, it passes it on to the **sessionManager**.
- **sessionManager** sends a job scheduling request to the **Idle Host**.
- The **Idle Host** schedules the job and send it's status to **sessionManager** which updates the **Status File**.
- When the last job is completed successfully, **sessionManager** updates the status file and informs the **Scheduler** and the **Job Initiator**.

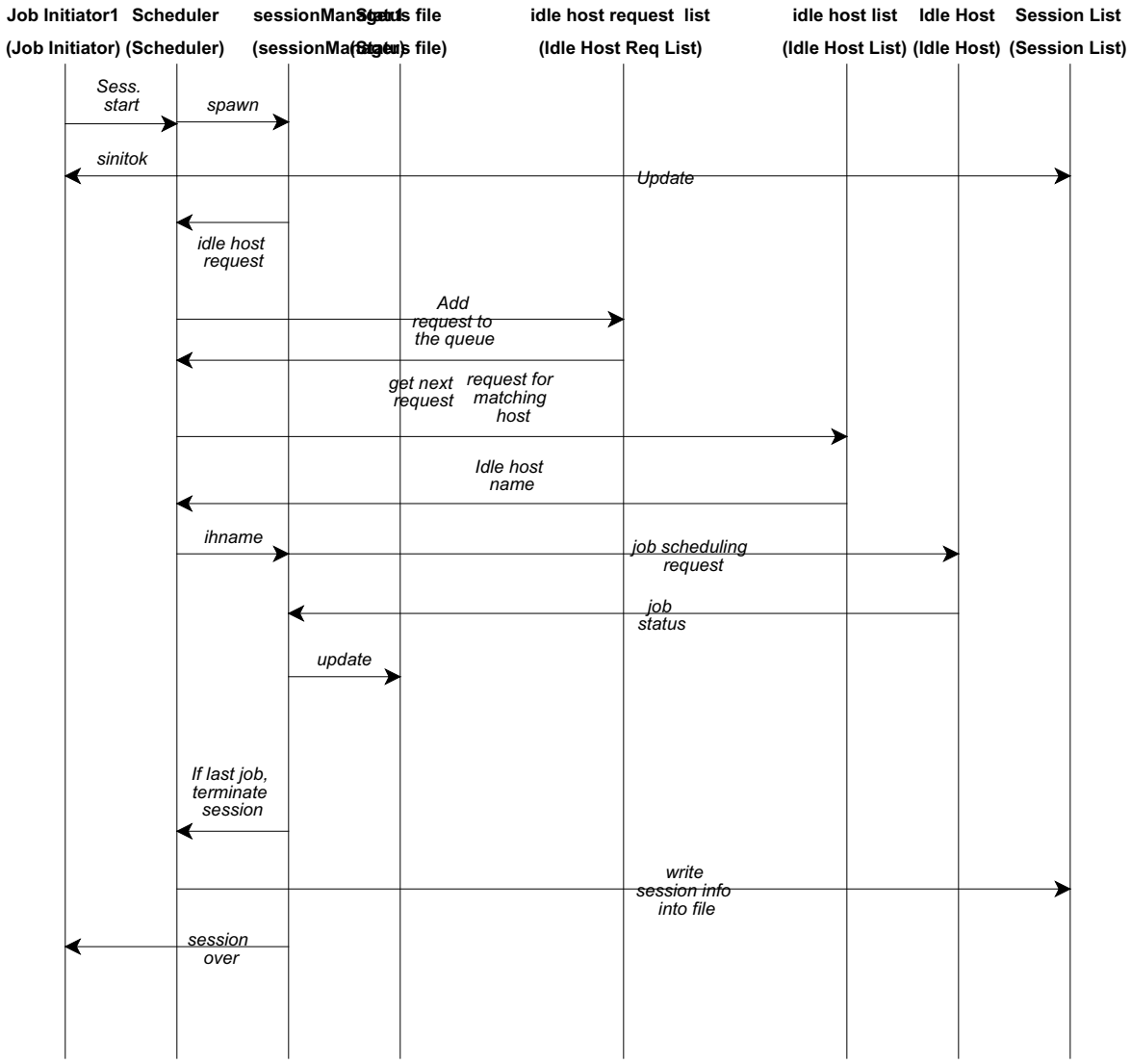

FIGURE 6. Event Trace Diagram for Process Scheduling

6.2.1.2 Session Restart

- The **Job Initiator** sends a session restart request to the **Scheduler**.
- The **Scheduler** spawns a **sessionManager** to handle the session and updates it's **Session List**.

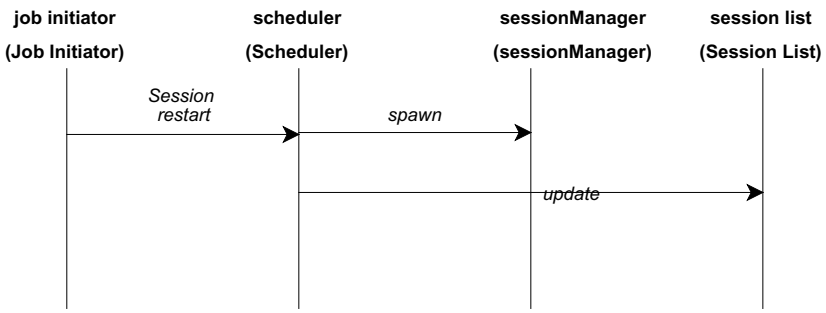

FIGURE 7. Session Restart

6.2.1.3 Idle Host Registration and Deregistration

- The **Idle Host** sends a registration or deregistration request to the **Scheduler**.
- The **Scheduler** asks the **Idle Host Info List** to either add or remove the **Idle Host Info**.

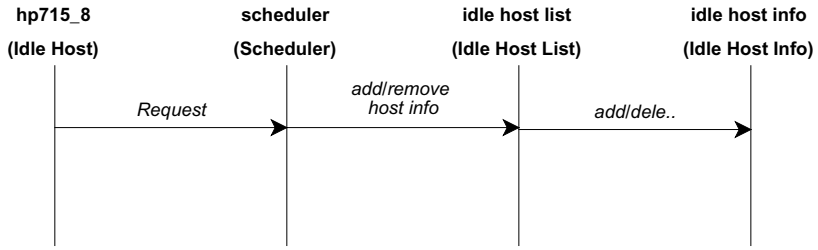

FIGURE 8. IdleHost Registration and Deregistration flow

6.2.1.4 Session Start Failed

- The **Job Initiator** sends a session start request to the **Scheduler**.
- The **Scheduler** tries to spawn a sessionManager and fails.
- It then informs the **Job Initiator** about the failure.

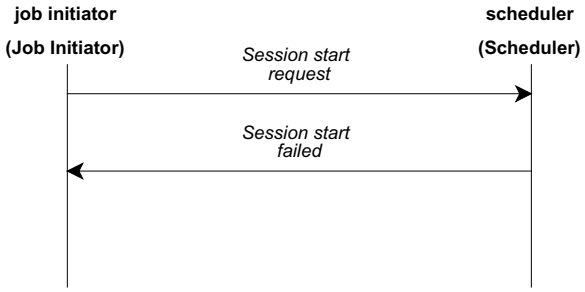

FIGURE 9. Session Start Failed scenario

6.2.1.5 Job Scheduling - Request Rejected

- The backend (**sessionManager**) sends a job scheduling request to the **Idle Host**.
- The **Idle Host** rejects the request as it can't host any more jobs.
- The backend sends a fresh idle host request for the rejected job.

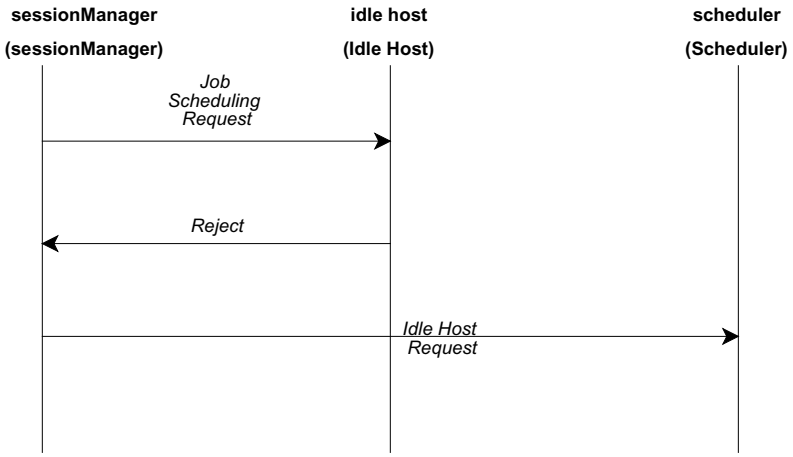

FIGURE 10. Job Scheduling Request Reject

6.2.2 State Diagrams

We will be drawing the state diagrams of the following classes only

(1). Scheduler

(2). Idle Host

(3). sessionManager (backend)

(4). Job Initiator

(5). Job Monitor

(6). Idle Host Request List

6.2.2.1 Scheduler

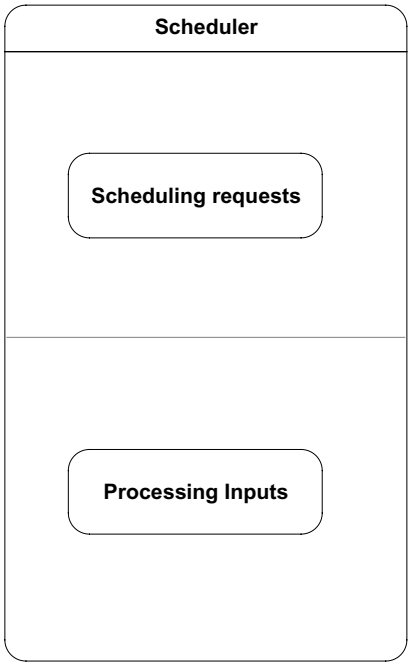

FIGURE 11. Object model of the Scheduler

The **Scheduler** does two activities simultaneously.

- *Scheduling Requests*

  The **Scheduler** schedules requests from the idle host request list. The expanded state diagram of this activity is included later in the document.

- *Processing Inputs*

  Simultaneously, the **Scheduler** accepts inputs from the outside world. This activity too is expanded later in the document.

Peer Messages

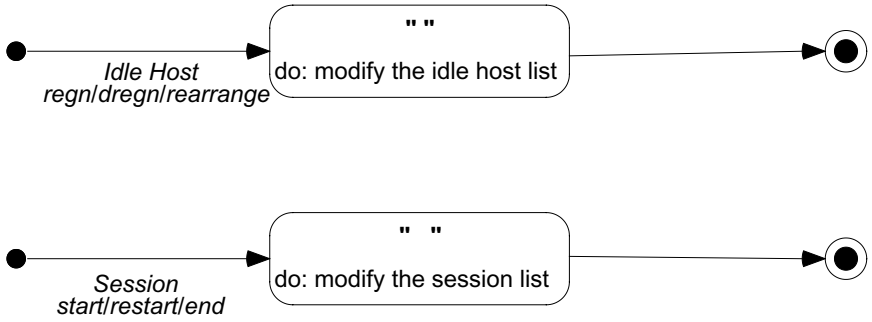

FIGURE 12. Messages from the peer

Client Messages

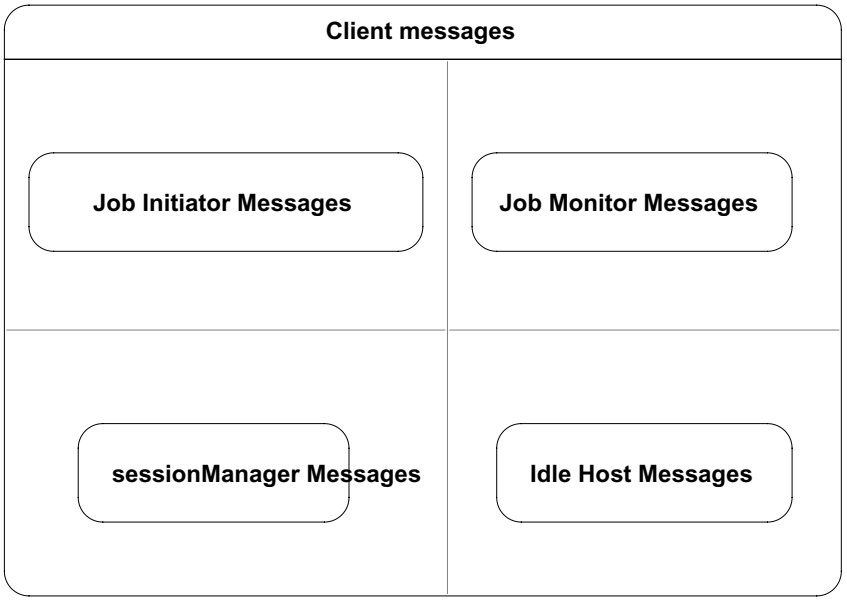

FIGURE 13. Messages from the client

The scheduler can receive and handle four kinds of client messages simultaneously.

Job Initiator Messages

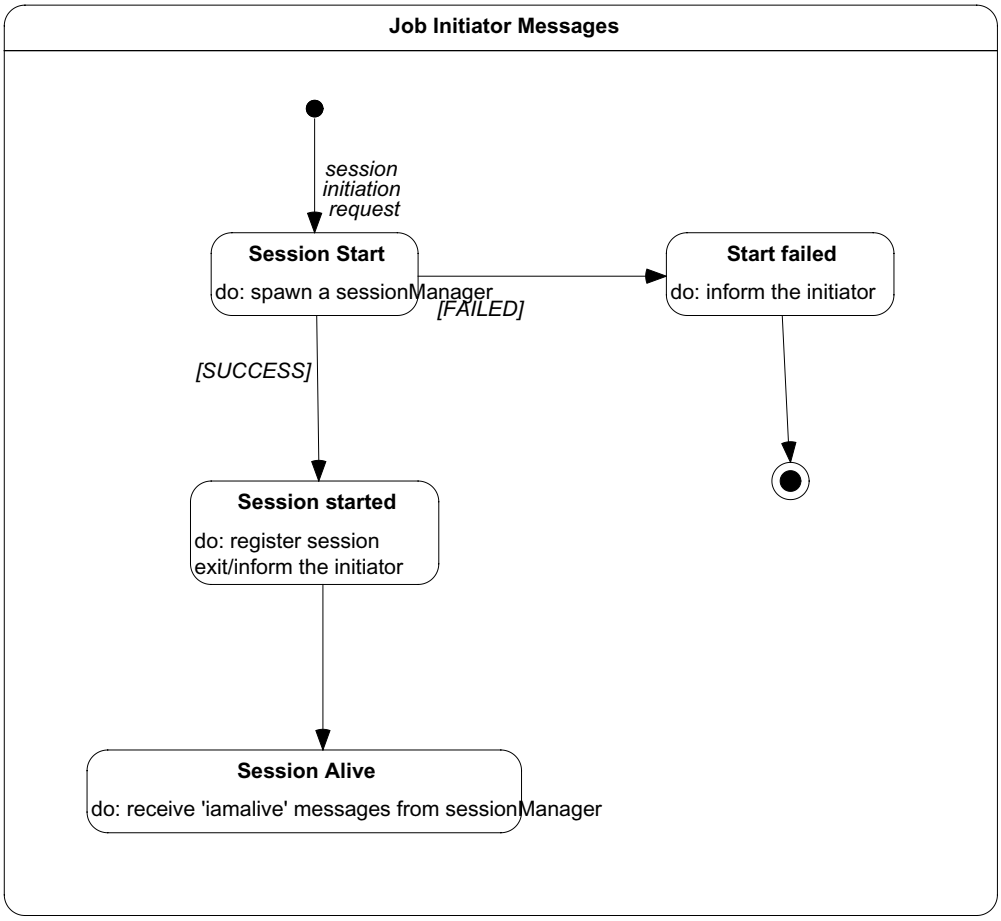

FIGURE 14. Process Initiation Request

This contains the activities that take place in the beginning of a job scheduling session. The session initiation request comes from the job initiator. Once a session is started, the scheduler starts receiving ‘iamalive’ messages from the session managers i.e sessionManagers. If the message is not received for a prolonged period of time, the session is assumed to be over. This is explained in a different state diagram later.

sessionManager Messages

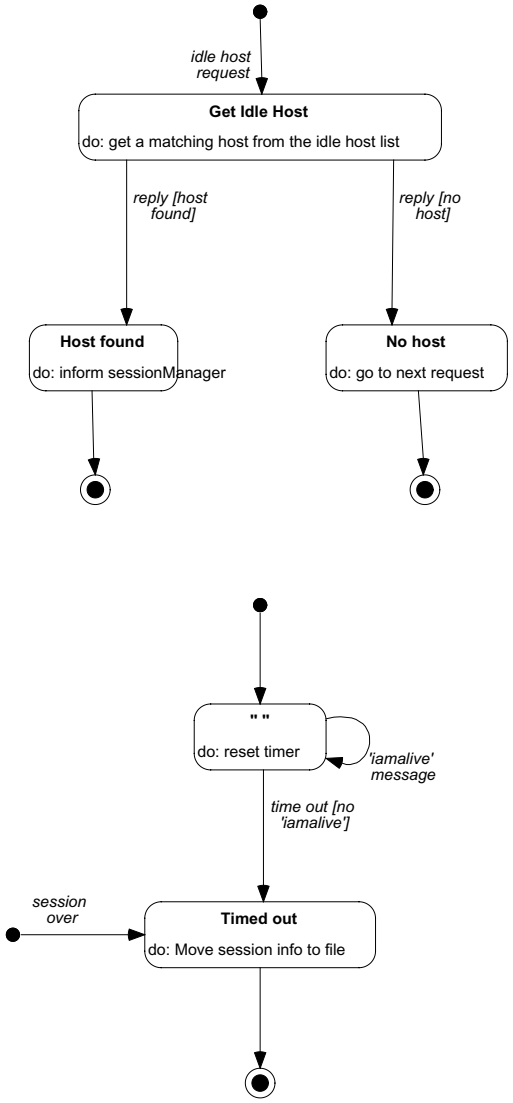

FIGURE 15. SessionManager Messages

The second diagram in this set depicts the handling of the ‘iamalive’ messages sent by the session manager.

Idle Host Messages

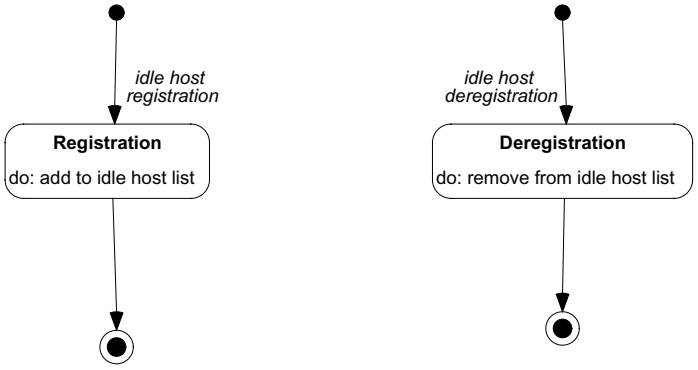

FIGURE 16. IdleHost Messages

6.2.2.2 Idle Host

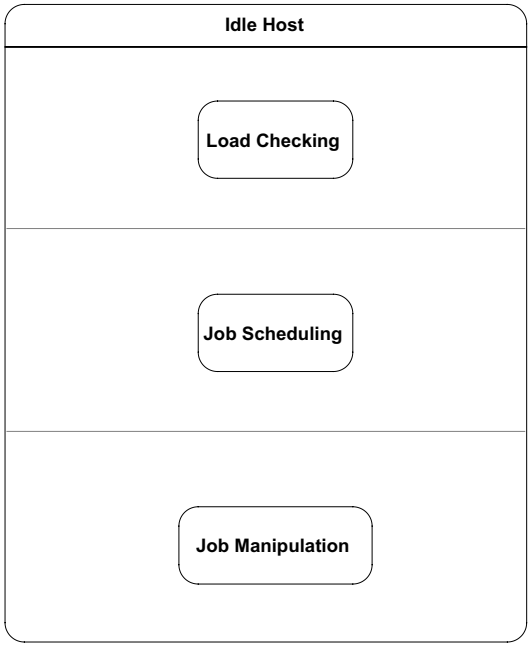

FIGURE 17. IdleHost Object

The **Idle Host** objects do three activities simultaneously.

- *Load Checking*

  Keeps track of the load of the machine.

- *Job Scheduling*

  Receives job scheduling requests and handles them.

- *Process Migration*

  Once jobs are scheduled, it entertains requests for migrating them. For the different kinds of manipulation facilities currently supported see section **Job Migration**.

Load Checking

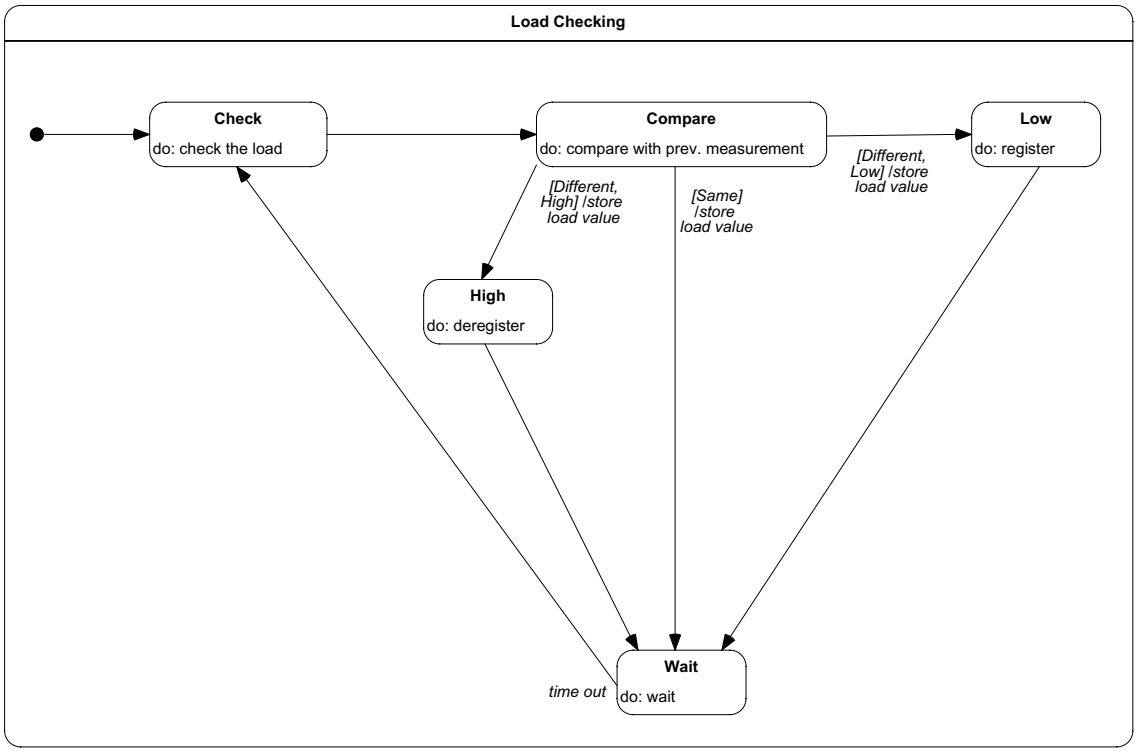

FIGURE 18. Load Checking Activity of IdleHost

Job Scheduling

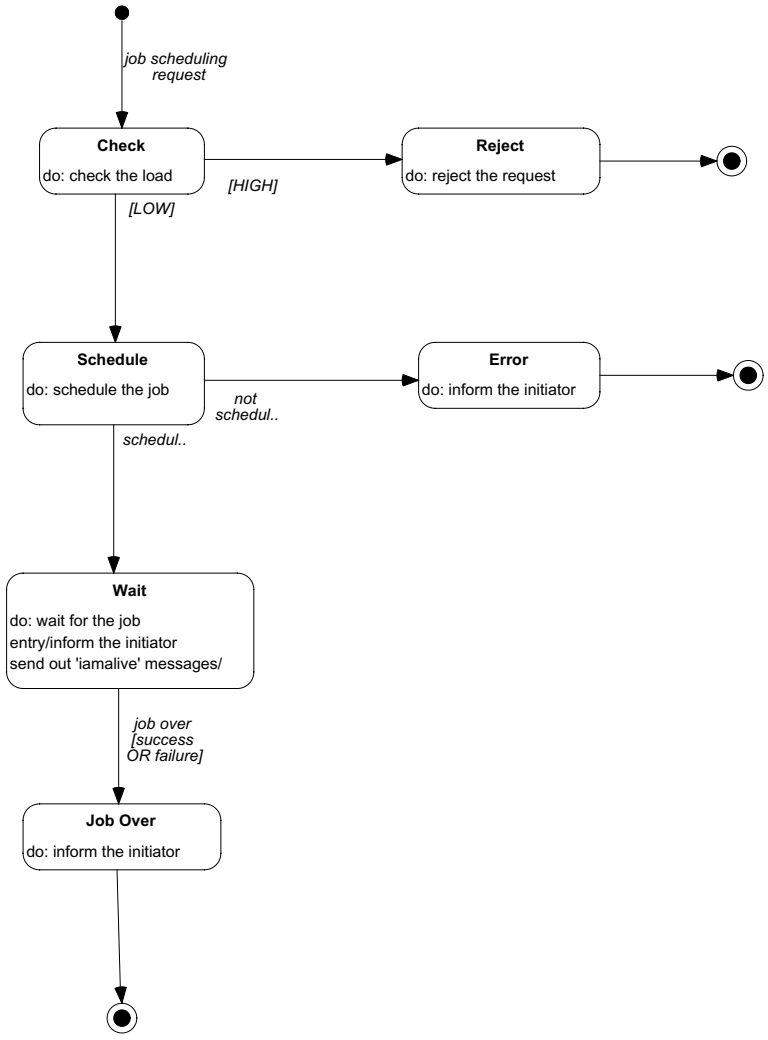

FIGURE 19. Job Scheduling by IdleHost

6.2.2.3 sessionManager (The 'Backend')

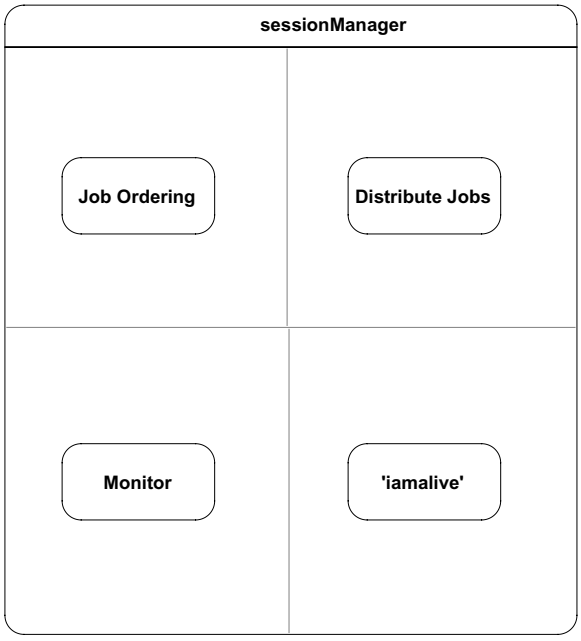

FIGURE 20. SessionManager Object

Also called the **Session Manager**, this object is responsible for a session and performs the following activities simultaneously

- **Process Ordering**

  This refers to the planning of a schedule for the jobs in the session. Priorities of the jobs are decided. This need not be static. This could be changed dynamically as the session progresses.

- **Process Distribution**

  Jobs are distributed based on the current schedule.

- **Session Restart**

A user can restart a session if it is terminated abruptly. The session will actually be continued in this case i.e jobs which have been completed successfully will not be rescheduled.

- **Process Monitoring**

  The session manager also accepts monitoring requests from clients and services them.

- **Sending ‘iamalive’ messages**

  The ‘iamalive’ messages are sent to the scheduler and to all the monitoring clients.

6.2.2.4 Process Initiator

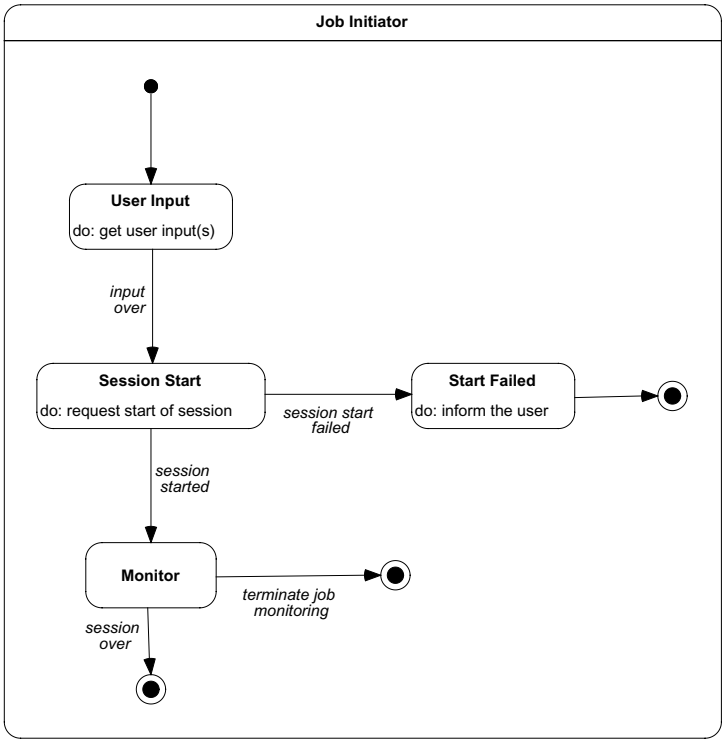

FIGURE 21. Process initiation

6.2.2.5 Idle Host Request List

The idle host request list contains the requests sent in by the session managers for idle hosts. It is used by the scheduler and there are two kinds of activities

- **Request Selection**
- **Queue Selection**

Request Selection

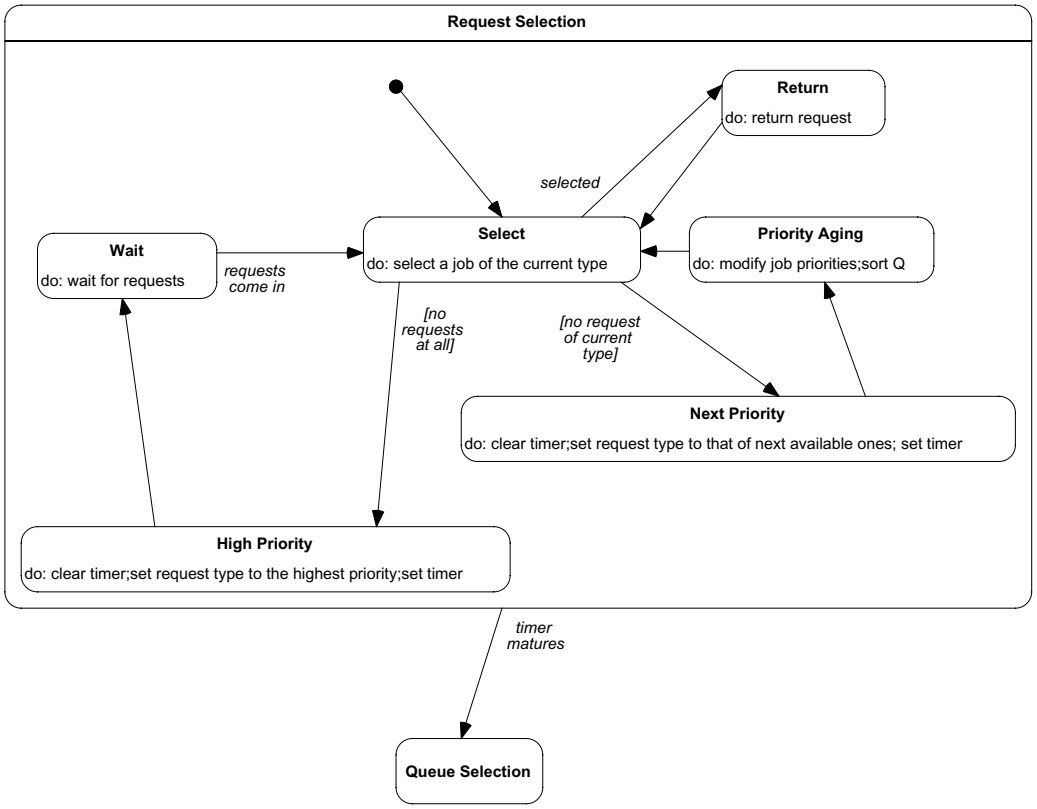

FIGURE 22. Request Selection Diagram

Queue Selection

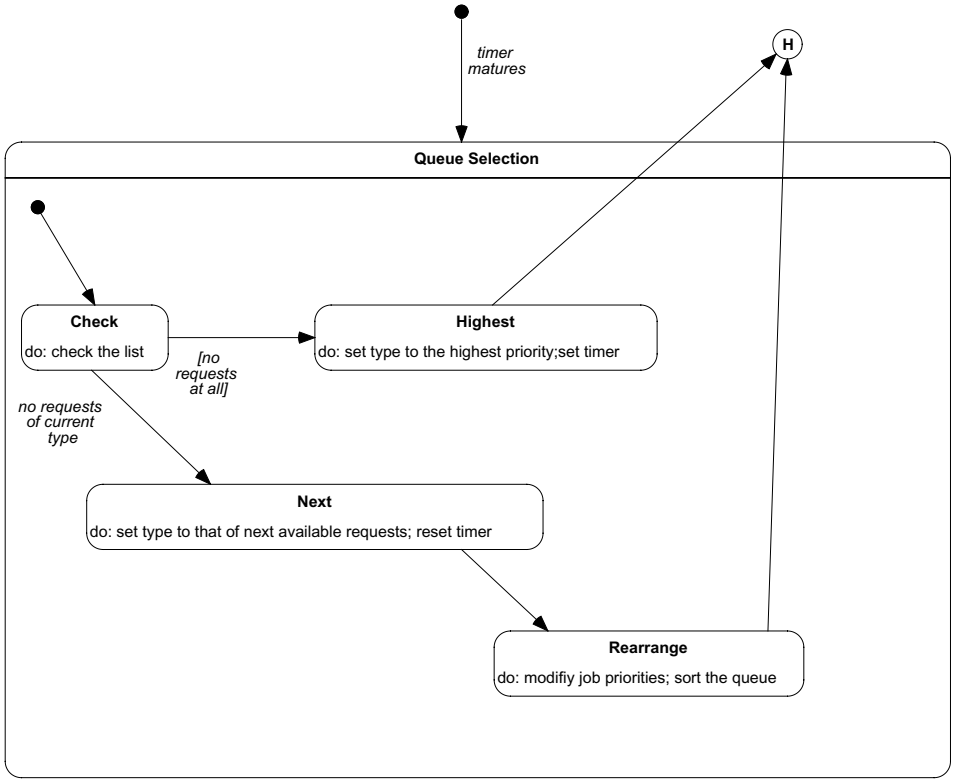

FIGURE 23. Introducing priority queues for the job scheduling

The history state **H** has been used to indicate that control returns to the state in which the object was when the timer matured.

6.3 Operations

We give a class-wise list of operations. This may not be the complete list. To quote *James Rumbaugh* et. al. from **Object-Oriented Modelling and Design**

> *"The list of potentially useful operations is open-ended and it is difficult to know when to stop adding them."*

We haven't listed the operations of all the classes; only the major players have been covered. The operations listed here correspond to the actions and activities in the state diagrams.

6.3.1 Scheduler Operations

- *schedule()*
- *receiveInputs()*
- *spawnSessionManager()*
- *handleJobInitiatorMsg()*
- *handleJobMonitorMsg()*
- *handleIdleHostMsg()*
- *handleSessionManagerMsg()*
- *handlePeerMsg()*

6.3.2 Idle Host Operations

- *periodicLoadCheck()*
- *getLoadRange()*
- *informScheduler()*
- *receiveInputs()*
- *scheduleJob()*
- *sendIAmAlive()*
- *manipulateJob()*

6.3.3 Session Manager (i.e) sessionManager Operations

- *readPERTFile()*
- *readStatusFile()*
- *prepareSchedule()*

- *runSession()*
- *receiveInputs()*
- *handleIAmAlive()*
- *sendIAmAlive()*
- *handleStatusInfo()*
- *sendStatusInfo()*
- *addListener()*
- *removeListener()*
- *sendManipulationRequest()*

6.3.4 Job Initiator Operations

6.3.5 Job Monitor Operations

6.3.6 Idle Host Request List Operations

- *getNextRequest()*
- *ignoreRequest()*
- *modifyPriorities()*
- *sort()*
- *queueSelection()*

6.3.7 Session List Operations

- *registerSession()*
- *deregisterSession()*
- *matchSession()*
- *moveSession2History()*
- *matchSessionInHistory()*

6.3.8 Idle Host List Operations

- *getHost()*
- *registerHost()*
- *deregisterHost()*
- *modifyHost()*

We have done a fair amount of system designing before we even started the object modelling of the system. This has been explained in the chapter **JSMT System Architecture**.

6.4 Identifying Subsystems

We have done a fair amount of system designing before we even started the object modeling of the system. Following building blocks of the protocol implementation can be designed:

(1). User Interface

Constituent Objects: *Job Initiator*, *Job Monitor.*

(2). Scheduler

Constituent Objects: *Scheduler*, *Idle Host List*, *Idle Host Request List*, *Session List*, *Session Info*, *Idle Host Request*, *Idle Host Info*.

(3). Session Manager

Constituent Objects: *Session Manager (sessionManager)*, *Session*, *Job*, *Status File*.

(4). Host

Constituent Objects: *Idle Host*.

6.4.1 System Topology

Following is the complete architecture of the proposed system. It depicts all the major messages for the process scheduling and the various nodes to be designed into the system. In the following figure, it shows the non-preemptive process migration scenario.

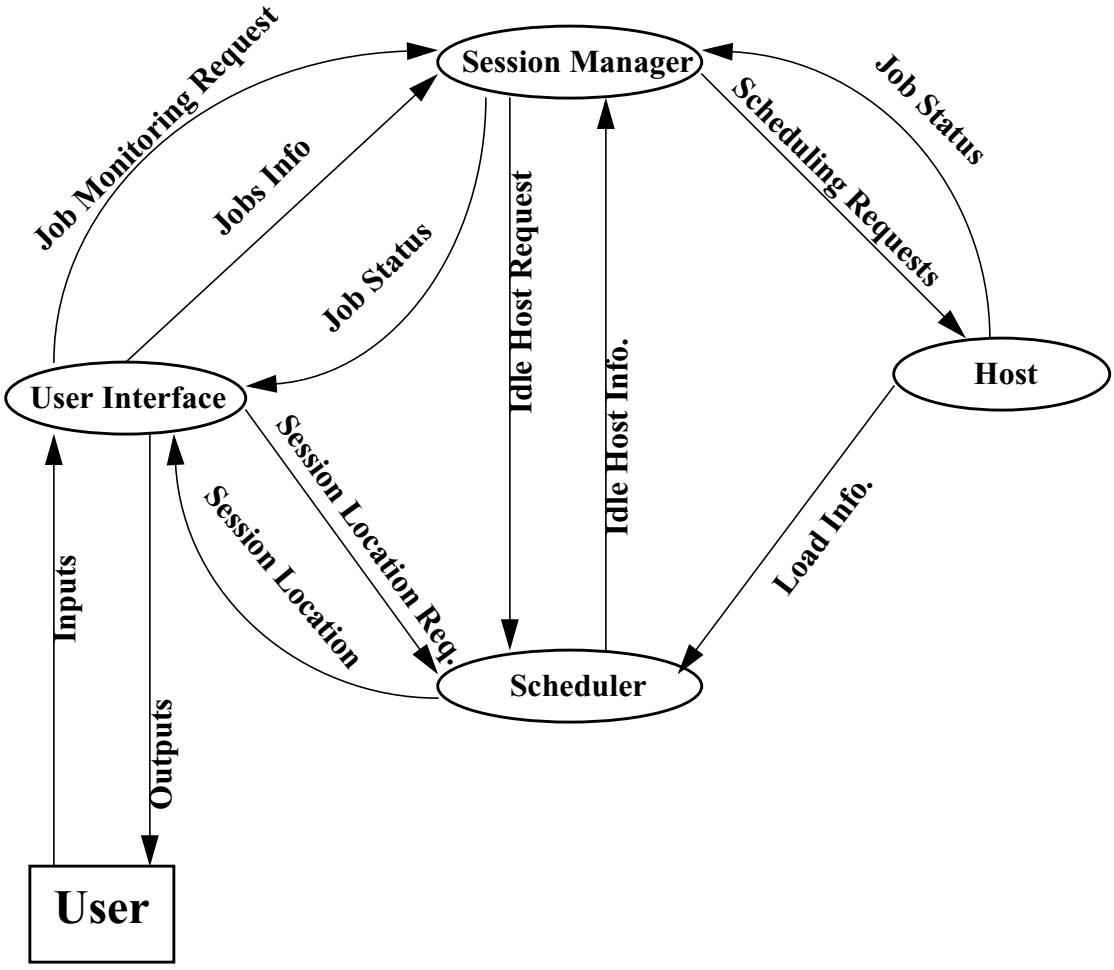

FIGURE 24. The Architecture of the proposed system

6.5 Physical locations of the subsystems

Since our application is a distributed system, we have decided to allocate subsystems to processors i.e physical locations before attempting to identify concurrency. There are three different physical locations for the subsystems.

- **Server**

  All the subsystems can be present in the server. The subsystems *Scheduler* and *Session Manager* will (and should) be present. Since a server too can be used for starting sessions and hosting jobs, the subsystems *User Interface* and *Idle host* can be present here.

- **Idle Host**

  The machine which hosts jobs. This will contain the subsystem *Idle Host.* All the machines covered by our application will contain this subsystem. The user can use this for starting sessions too.

- **User's Machine**

  Used for starting sessions with the subsystem *User Interface*. Since it can host jobs too, the subsystem *Idle Host* too can be present.

CHAPTER 7 Conclusions and Future Work

In this thesis we have proposed a distributed process scheduling and process migration protocol to enable transparent load balancing and load sharing in heterogeneous Unix clusters, the machines in the cluster may have different architectures and may run different flavors of Unix operating system. But all the machines must implement the protocol. The protocol supports remote execution or non-preemptive process migration because the processes can migrate across heterogeneous machines.

We also define a set of conventions that each machine must follow. For example, the UNIX process ids have to be mapped with protocol specific ids in a certain manner, the scheduling process must be on NFS or it can't be a process residing on the local machine, etc. the conventions specify the requirements of the protocol without dictating the way they are implemented. Distinguishing the conventions from the protocol has the benefit that different implementations can implement the conventions differently, but as long as they are implemented, the protocol works fine.

The protocol tries to provide as single system image to the cluster of machines. But this cannot be achieved unless the other resources like files, devices, sockets, etc., of all the nodes in a distributed system are transparently accessible from any node in the system. Conventions are also defined for maintaining more of less uniform views for files on all machines.

It is seen that a few Unix process management semantics are impossible to extend over a distributed environment without heavy overhead. Hence, the protocol has not been implemented in the kernel level mode, but in the user level mode only. The user level implementation of the protocol provides more flexibility and reduced threats towards the Unix kernel. The protocol may be implemented in the kernel also, but it requires a lot of homework with the existing behavior of system calls implemented in Unix. Linux may be a better operating system, which may be used for the implementation of the protocol in Linux kernel.

The protocol works with any load sharing policy and also try to use existing standard protocols as much as possible. For example, it does not design a new distributed file system protocol for file sharing. Instead, it uses the existing standard protocol NFS for preserving uniform file system view in the nodes of a distributed system.

The protocol handles both the node and network failures. For node failure, the job or the process may be rescheduled on different coherent node. Network failures are made transparent in the system in the sense that these only delay the message exchange among the machines. But top the user, this looks only a reduced performance. For failure detection the protocol assumes existence of a connection oriented protocol. This simplifies the design of the protocol.

A possible user level implementation architecture is also proposed in this thesis. Even it seems that a completely user level implementation will not be possible with the kind of transparency the protocol is trying to achieve with the desired performance and efficiency The proposed implementation architecture defines some layers over the exiting Unix kernel, that may be used for user level process scheduling and process migration and don't requires and modification in the existing Unix kernel. In the protocol, a group of user's job can also be scheduled simultaneously, which introduces the concept of sessionManager. Distributed load measurement algorithm has been discussed which may be used in heterogeneous Unix networks. This separates the information subsystem from the execution subsystem for the greater system efficiency.

7.1 Future Work

Although we have designed the process scheduling and migration protocol for Unix for a heterogeneous cluster, a complete implementation is required to measure the correctness and performance of the protocol. The kernel level implementation of the protocol can be done for the greater extend of the performance and efficiency. This can be tried out on Unix implementations for which source code is available (for example, Linux and NetBSD). After experience with these implementations, the protocol can be reevaluated and

suitably modified. Process selection algorithm can be added to get the complete view of process migration system.

BIBLIOGRAPHY

[1] A.S. Tanenbaum, R.V. Renesse, H.V. Staveren, G.J. Sharp, S.J. Mullender, J. Jansen, and G.V. Rossum. Ëxperiences with the Amoeba Distributed Operating System. Communications of the ACM, 33, December 1990.

[2] David R. Cheriton. The V Kernel: A Software Base for Distributed Systems. IEEE Transactions on Software Engineering, 1(2), April 1984.

[3] David R. Cheriton. The V. Distributed System. Communications of the ACM, 31(3), March 1988.

[4] Yeshayahu Artsy and Raphael Finkel. Designing a Process Migration Facility: The Charlotte Experience. IEEE Computer, 22(9), September 1989.

[5] Maurice J. Bach. The Design of the UNIX Operating System. Prentice Hall., 1986.

[6] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S, Quarterman. 4.3 BSD UNIX Operating system. Addision Wesley, May 1989.

[7] Uresh Vahalia. UNIX Internals: the new frontiers. Prentice Hall, 1996.

[8] Marshall K. Mckusick, Keith Bostic, Michael J. Karles, and John S. Quarterman. The Design and Implementation of the 4.4 BSD Operating System. Addision Wesley, 1996.

[9)] Russel Sandberg, David Goldberg, Steve Klliman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In Proceedings of the USENIX Summer Conference. USENIX, May 1985.

[10] J. Morris, M. Satyanarayan, M.H. Conner, J.H. Howard, D.S. Rosenthal, and F.D. Smith. Ändrew: A Distributed Personal Computing Environment. Communications of the ACM, 29(3), March 1986.

[11] Peter Smith and Norman C. Hutchinson. Heterogeneous Process Migration: The Tui System. Technical Report 96-40, Department of Computer Science, University of British Columbia, Vancouver, B.C., V6T 1X4, Canada, 1996.

[12] Derek E. Eager, Edward D. Lazowska, and John Zahorjan. The Limited Performance benefits of Migrating Active Processes for Load Sharing. ACM SIGMETRICS, 16, May 1988.

[13] S. Petri and H. Langendorfer. Load Balancing and Fault Tolerance in Workstation Clusters Migrating Groups of Communicating Processes. ACM SIGOPS, May 1995.

[14] Songnian Zhou, Jingwen Wang, Xiaohu Zheng, and Pierre Delisle. Ütopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. Technical Report CSRI 257, Computer Systems Research Institute, University of Toronto, Toronto, M5S 1A1, Canada, April 1992.

[15] Michael J. Litzkow. Remote Unix: Turning Idle Workstations into Cycle Servers. In Proceedings of the USENIX Summer Conference. USENIX, June 1987.

[16] Michael Litzkow and Marvin Solomon. Supporting Checkpointing and Process Migration Outside the Unix Kernel. In Proceedings of the USENIX Winter Conference. USENIX, January 1992.

[17] Michael Litzkow, Todd Tanenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report 1346, Department of Computer Science, University of Wisconsin-Madison, April 1997.

[18] Amit M. Vahdat, Douglas P. Ghormley, and Thomas E. Anderson. Efficient, Portable and Robust Extension of Operating System Functionality. Technical Report CS-94-842, Computer Science Division, University of California at Berkeley, December 1994.

[19] Marcin M. Theimer, Keith A. Lantaz, and David R. Cheriton. Preemptable Remote Execution Facility for the V-System. In Proceedings of the 10th Symposium on Operating System Principles. ACM SIGOPS, March 1985.

[20] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Wetch. The Sprite Network Operating System. IEEE Computer, February 1988.

[21] Fred Doughlis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. Software- Practice and Experience, August 1991.

[22] Annon Barak and Oren Laádam. The mosix Multicomputer Operating System for High Performance Cluster Computing, Journal of future generation computer systems, 13(4-5), March 1998.

[23] MOSIX Homepage. http://www.cs.huji.ac.il/mosix/.

[24] Berkeley Software Division. BSD/OS. http://www.bsdi.com/.

[25] Gernald J. Popek and Bruce J. Walker. The LOCUS Distributed System Architecture. MIT Press, 1985.

[26] Bruce Walker, Gerald Popek, Robert English, Charles kline, and Greg Thil. The LOCUS Distributed Operating System. ACM, SIGOPS, 17(5), October 1983.

[27] Roman Zajcew, Paul Roy, David Black, Chris Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Lei bensperger, Michael Barnett, Fa ramarz Rabii, and Dur riya Netterwa la. Än OSF/1 Unix for Massively Parallel Multicomputer. In Proceedings of the USENIX Winter Conference. USENIX, January 1993.

[28] M. Accetta, R. Baron, D. Golub, P. Rashid, A. Tevanian, and M. Young. Mach Ä New Kernel Foundation for Unix Development. In Proceedings of the USENIX Summer Conference, USENIX, 1986.

[29] Ken Shririff.Building Distributed Process Management on An Object Oriented Framework . USENIX, 1996.

[30] Yousef A. Khalidi, Jose M. Bernabeu, Valada Matena, Ken Shirriff, and oti Thadani. Solaris MC: A multi Computer OS. In Proceedings of the USENIX Winter Conference. USENIX, January 1996.

[31] Andrew S. Tanenbaum. Distributed Operating Systems. Prentice Hall, 1995.

[32] Sun Mircosystems. Network Programming Manual 1988.

Appendix A(1)

- Following is a sample Unix shell script, which is being called by the idleHost in the Load Measurement Code for load sharing purposes. Following script measures the total number of input and output packets (tot_io_pkts), cpu_busy_percentage, total number of memory faults (tot_mem_flts) and total number of processes in run-queue (tot_proc_exe) in the machine in the last duration of sampling.

```
#!/bin/ksh
#################################################################################
#   File Name                       : GetParam.Script                           #
#   Author                          : Milan Saxena                              #
#   Last Modified by                :                                            #
#   Last Modified                   :                                            #
#   Brief Description               : This is the shell script made to find the cpu load #
#                                     parameters, as cpu_per_busy, faults, tot_io_pkts,  #
#                                    and no of processes in executio.           #
#                                                                                #
#################################################################################

net_file="/tmp/.netst_result"
vms_file="/tmp/.vmstat_result"
NetLoad.Script $net_file &
sleep 5
vmstat 10 2 > $vms_file

proc=0
max_proc=0
per_cpu_idle=0

#get the name of the operating system
ostype=` uname|cut -c1 `

#let us calculate the values for netstat

        #for digital unix
        if [ "$ostype" = "O"  ]
        then
                max_proc=64
                last_ipkts=`awk '{new=NF-4; if(NR == 4){print $new}}' $net_file`
                last_opkts=`awk '{new=NF-2; if(NR == 4){print $new}}' $net_file`
        fi

        #for sun
        if [  "$ostype" = "S"  ]
        then
                max_proc=1909
```

```
                last_ipkts=`awk '{new=NF-4; if(NR == 4){print $new}}' $net_file`
                last_opkts=`awk '{new=NF-2; if(NR == 4){print $new}}' $net_file`
        fi

        # for hp
        if [  "$ostype" = "H"  ]
        then
                max_proc=76
                last_ipkts=`awk '{new=NF-4; if(NR == 4){print $new}}' $net_file`
                last_opkts=`awk '{new=NF-2; if(NR == 4){print $new}}' $net_file`
        fi

        #for linux
        if [  "$ostype" = "L"  ]
        then
                max_proc=64
                last_ipkts=`awk '{new=NF-4; if(NR == 4){print $new}}' $net_file`
                last_opkts=`awk '{new=NF-2; if(NR == 4){print $new}}' $net_file`
        fi

echo `expr $last_ipkts + $last_opkts`

#Calculate the virtual memory statistics fo all the three flavours

#vmstat for sun
if [ "$ostype" = "S" ]
then
        per_cpu_idle=`awk ' {if(NF > 8 && $NF!="id" && NR == 4){print $NF} }' $vms_file`
        fault=`/usr/xpg4/bin/awk '{if(NF > 8 && NR == 4){ sub(/M/,"000",$7); {print $7}}}' $vms_file`

echo $per_cpu_idle
echo $fault
fi


#vmstat for digital
if [ "$ostype" = "O" ]
then
        per_cpu_idle=`awk ' {if(NF > 8 && $NF!="id" && NR == 5){print $NF} }' $vms_file`
        fault=`awk '{if(NF > 8 && $14!="sy" && NR == 5){ sub(/M/,"000",$7); {print $7}}}' $vms_file`

echo $per_cpu_idle
echo $fault
fi


#vmstat for hp
```

```
if [ "$ostype" = "H" ]
then
        per_cpu_idle=`awk ' {if(NF > 8 && $NF!="id" && NR == 4){print $NF}}' $vms_file`
        fault=`awk '{if(NF > 8 && $14!="sy" && NR == 4){ sub(/M/,"000",$7); {print $7}}}' $vms_file`

echo $per_cpu_idle
echo $fault
fi

#vmstat for sun
if [ "$ostype" = "L" ]
then
        per_cpu_idle=`awk ' {if(NF > 8 && $NF!="id" && NR == 4){print $NF} }' $vms_file`
        fault=`awk '{if(NF > 8 && NR == 4){ sub(/M/,"000",$7); {print $7}}}' $vms_file`

echo $per_cpu_idle
echo $fault
fi

#finding percentage of processes executing
proc=`ps -e|wc -l|awk '{print $1}'`
echo $proc

rm -f $net_file
rm -f $vms_file
```

Appendix A (2)

- Following is a sample Unix script NetLoad.Script, which is being called by the script GetParam.Script defined in Appendix A (1). Following script mesures the total number of input and output packets, send or recieved by the machine during the last interval of sampling.

```
#!/bin/ksh
################################################################################
#   File Name                       : GetParam.Script                          #
#   Author                          : Milan Saxena                             #
#   Last Modified by                :                                           #
#   Last Modified                   :                                           #
#   Brief Description               : This is the shell script made to find the cpu load #
#                                   parameters, as cpu_per_busy, faults, tot_io_pkts,  #
#                                   and no of processes in execution.          #
#                                                                              #
################################################################################

# The following code was added during Sun porting
grep=grep
awk=awk
id=id
ostype=`uname|cut -c1`
#set up the aliases and shell variables
#for sun
if [ "$ostype" = "S" ]
        then
                grep="/usr/xpg4/bin/grep"
                awk="/usr/xpg4/bin/awk"
                id="/usr/xpg4/bin/id"
        fi
#code ends here

#get the type of OS
if [ "$ostype" = "O" ]
then
          /usr/sbin/netstat -i 10 >> $1 &
else
          netstat -i 10 >> $1 &
fi
sleep 12
job_count=`jobs -l | wc -l`
if [ ${job_count} -ne 0 ]
then
          netstat_pid=`jobs -l | awk '{print $3}'`
```

```
        kill -9 ${netstat_pid}
        if [ $? -ne 0 ]
        then
                while [ 1 ]
                do
                kill -0 ${netstat_pid} > /dev/null 2>&1
                if [ $? -eq 0 ]
                then
                                kill -9 ${netstat_pid}
                                break
                        fi
                done
        fi
fi
exit
```

Appendix B

1.) SIMULATION AND RESULTS

For Dec Alpha Server (RISC Architecture), load values are shown as below. These results were taken by a load measurement script which in given in the appendix A.

TABLE 3. Load Parameters and the measured Load Value

I+O PKTS	CPU_BUSY	MEM_FAULTS	PROCESSES	LOAD_VAL
45926.000000	18.000000	339.000000	115.000000	9.377328
18068.000000	17.000000	372.000000	114.000000	8.215099
19492.000000	21.000000	379.000000	114.000000	8.336581
17269.000000	15.000000	326.000000	116.000000	8.095818
21399.000000	18.000000	326.000000	116.000000	8.284259
24834.000000	19.000000	370.000000	115.000000	8.500853
23662.000000	17.000000	327.000000	114.000000	8.328816
26860.000000	20.000000	344.000000	114.000000	8.528652
27189.000000	20.000000	370.000000	115.000000	8.607431
25044.000000	30.000000	327.000000	115.000000	8.588831
33545.000000	24.000000	408.000000	107.000000	8.902819
31303.000000	25.000000	344.000000	109.000000	8.717259
30212.000000	21.000000	326.000000	111.000000	8.603562
30554.000000	23.000000	371.000000	112.000000	8.749929
31952.000000	26.000000	370.000000	110.000000	8.825301
24866.000000	20.000000	426.000000	109.000000	8.551944
37666.000000	25.000000	389.000000	113.000000	9.144222

I+O PKTS	CPU_BUSY	MEM_FAULTS	PROCESSES	LOAD_VAL
31385.000000	26.000000	349.000000	111.000000	8.772418
33043.000000	29.000000	440.000000	114.000000	9.119063
32213.000000	31.000000	438.000000	113.000000	9.096849
40745.000000	29.000000	386.000000	112.000000	9.332512
38031.000000	32.000000	338.000000	113.000000	9.167611
45094.000000	30.000000	370.000000	114.000000	9.561434
27931.000000	20.000000	326.000000	113.000000	8.522588
38988.000000	28.000000	338.000000	116.000000	9.190659
41007.000000	33.000000	405.000000	117.000000	9.516909
21866.000000	17.000000	333.000000	120.000000	8.356977
27527.000000	26.000000	463.000000	121.000000	8.990629
23666.000000	19.000000	371.000000	107.000000	8.351097
13910.000000	12.000000	339.000000	107.000000	7.846696
16740.000000	12.000000	22.000000	88.000000	7.168590
21050.000000	19.000000	338.000000	88.000000	7.954191
15551.000000	12.000000	326.000000	88.000000	7.641700
27302.000000	19.000000	345.000000	89.000000	8.217996
6621.000000	8.000000	326.000000	94.000000	7.379263
8000.000000	9.000000	326.000000	95.000000	7.445283
1265.000000	35.000000	310.000000	99.000000	34.152638
74.000000	36.000000	429.000000	109.000000	92.374321
894.000000	57.000000	298.000000	110.000000	31.563347
16087.000000	15.000000	509.000000	108.000000	8.335574

I+O PKTS	CPU_BUSY	MEM_FAULTS	PROCESSES	LOAD_VAL
6360.000000	11.000000	503.000000	120.000000	8.125178
5741.000000	14.000000	509.000000	123.000000	8.204270
0.000000	100.000000	0.000000	100.000000	8.581940
69620.000000	63.000000	1018.000000	97.000000	13.292508
45175.000000	99.000000	10.000000	101.000000	10.292423
80710.000000	65.000000	1017.000000	95.000000	14.206496
0.000000	100.000000	0.000000	107.000000	8.671421

2.) SIMULATION AND RESULTS

For SUN Solari (RISC Architecture sparc SUNW,Ultra-5_10), load values are shown as below. These results were taken by a load measurement script which in given in the appendix A.

TABLE 4. Load Parameeters and the measured Load Value

CPU_BUSY	PROCESSES	MEM_FAULTS	I+O PKTS	LOAD_VAL
4.00	34.00	24.00	253.00	21.83
45.00	36.00	586.00	3047.00	34.25
5.00	36.00	105.00	257.00	22.04
4.00	35.00	24.00	267.00	21.84
1.00	36.00	34.00	313.00	21.34
1.00	36.00	24.00	206.00	21.29
2.00	36.00	24.00	275.00	21.50
4.00	35.00	105.00	203.00	21.82
6.00	37.00	107.00	876.00	22.54
95.00	40.00	41.00	771.00	62.01
72.00	42.00	334.00	1137.00	45.74
71.00	40.00	582.00	2496.00	45.96
82.00	42.00	636.00	1995.00	52.65
79.00	42.00	548.00	1897.00	50.55
76.00	43.00	680.00	3304.00	49.49
64.00	42.00	577.00	4464.00	43.44
59.00	42.00	241.00	5166.00	41.45
58.00	42.00	575.00	5387.00	41.20

CPU_BUSY	PROCESSES	MEM_FAULTS	I+O PKTS	LOAD_VAL
76.00	40.00	556.00	2114.00	48.73
81.00	42.00	525.00	2590.00	52.29
75.00	50.00	421.00	3912.00	49.25
82.00	42.00	517.00	14443.00	64.56
61.00	42.00	777.00	4608.00	42.09
2.00	36.00	34.00	473.00	21.60
1.00	36.00	24.00	228.00	21.30
2.00	36.00	24.00	327.00	21.52
3.00	36.00	105.00	230.00	21.66
1.00	36.00	24.00	239.00	21.31
81.00	42.00	670.00	3697.00	53.01
61.00	50.00	42.00	7251.00	44.06
74.00	50.00	61.00	11254.00	55.12
91.00	41.00	129.00	27804.00	105.94
61.00	50.00	48.00	6618.00	43.53
68.00	42.00	520.00	8386.00	48.71
46.00	42.00	423.00	6879.00	37.38
74.00	42.00	133.00	366.00	46.49
62.00	42.00	277.00	3777.00	41.93
89.00	42.00	396.00	2243.00	57.92
58.00	42.00	728.00	4447.00	40.56
55.00	42.00	550.00	5287.00	39.78
79.00	42.00	618.00	2852.00	51.12

CPU_BUSY	PROCESSES	MEM_FAULTS	I+O PKTS	LOAD_VAL
50.00	41.00	581.00	5799.00	38.08
67.00	42.00	826.00	2864.00	44.02
74.00	42.00	401.00	2210.00	47.54
78.00	39.00	728.00	2511.00	50.27
72.00	40.00	398.00	3012.00	46.82
64.00	39.00	535.00	2237.00	42.00
81.00	39.00	725.00	1731.00	51.81
80.00	39.00	605.00	2617.00	51.63
69.00	39.00	520.00	3871.00	45.68
81.00	39.00	403.00	2798.00	52.38
67.00	39.00	510.00	3764.00	44.51
70.00	39.00	565.00	1450.00	44.78
61.00	37.00	747.00	2759.00	40.86
39.00	39.00	295.00	3241.00	32.24
56.00	39.00	310.00	1136.00	37.55
76.00	39.00	402.00	5381.00	50.84
99.00	38.00	35.00	256.00	65.34
53.00	39.00	639.00	6064.00	39.52
65.00	39.00	996.00	2905.00	43.01
85.00	38.00	118.00	5654.00	57.10
72.00	38.00	233.00	4551.00	47.79
43.00	39.00	417.00	4375.00	34.38
47.00	39.00	451.00	3685.00	35.5