Table of Contents

CHAPTER 1

Introduction to Load Balancing Algorithms

1.1 Introduction

The ever-growing use of computer networks is attracting much attention in the design, development and implementation of distributed computing systems. High speed computation is required to solve many real world problems like flight simulation, real-time image processing, parallel processing of robotic computations etc. These motivations led to many changes in the architecture and organization of computing systems but due to limitations at microelectronic level such systems do not go beyond a certain speed barrier and all system resources are not put to maximum use. An option available is to use multiple processors where computational power of a set of processors is harnessed and the given task is distributed on different processors.

In this chapter, first distributed systems are discussed along with their different configurations. Next the requirement of load balancing is gauged as are the different arrangements used to balance the load. After studying the advantages and disadvantages of different existing techniques, the algorithm chosen to be implemented is detailed.

1.2 Distributed Systems

A distributed system comprises of a network of autonomous processors, each having private resources (such as CPU, memory and disks) and all sites share the network resources. The programs compiled at one site in the system can be executed at any other. Distributed systems provide users with access to remote resources spread across a room, a community, or a country.

In *Homogeneous Distributed System*, there are machines having same configurations and architecture. While in *Heterogeneous Distributed system*, the host machines with different architectures and configurations are connected together to form a network.

Distributed systems are divided into two groups: those processors that have shared memory are known as *tightly coupled* systems and those that do not have shared memory are called *loosely coupled* systems. Tightly coupled processors communicate through their shared memory while loosely coupled through message passing. In loosely coupled processors the resources such as network and disks can be shared transparently among the processes at different sites; local processes can only access other resources, such as CPU and memory

As distributed computing systems become increasingly popular, resource sharing among a number of computers connected by communication networks becomes feasible and desirable. Some of the possible resources to share are computing

power, data, and hardware devices. The sharing of computing power is usually in the form of load transfer. The transfer of load among different processors in a distributed system leads to the area of balancing the load among different processors, which is the aim of this project.

1.3 Load Balancing

It is a common observation in a network of autonomous processors that, at some point during execution, at least one processor will be idle while there are multiple tasks queued for execution on other processors. In such a situation, it may be advantageous to move some of the work from the busy processors to the idle ones. This movement of load is referred to as *load balancing*.

As a result of load balancing, job throughput rate and average job response time is likely to improve. Without buying new resources, load balancing as a system service can offer much more resources and better response times to all users, and can effectively avoid disasters that sometimes can arise from bad load distribution.

Load balancing is a critical factor in achieving optimal performance more than ever in applications where tasks are created in a dynamic fashion. It is basically focused on application response time and system resource utilization, measured by task elapsed times, application elapsed times, CPU utilization, paging rates and sometimes disk and network utilization

There are basically two methods adopted for load balancing. First is to balance the load such that every processor has some work at any time. The second is to adapt some technique such that the load must be distributed nearly evenly across all the processors. Both these methods require the system to be monitored for balancing the load. The responsibility of monitoring and balancing the load may be assigned to one particular processor or all the processors may share the responsibility.

As regard to hosting the responsibility for information collection and decision-making, there are two approaches for load balancing:

1.3.1 Centralized Load Balancing

In Centralized approach, status information about the entire system is collected and decision to load balance is made at one location. It provides one central agent for collecting load information and decision-making.

Centralized load balancing usually comes along with central task queuing. Hence load control is possible, because tasks can be queued until enough resources are available. Otherwise, heavy load situations saturate the system and reduce the throughput due to process switching, memory paging and network congestion. Central queuing also enables late decisions, because tasks may not be assigned when they are born but when they are removed from the queue for immediate execution.

This approach has several strong advantages:

- The measurement information and predictions consistently reside in one place and may not be distributed or replicated among the system which would cause message traffic and could lead to outdated, inconsistent information and contradictory load balancing decisions.

- Centralized load balancing can exploit the global knowledge about the system and application behavior and can utilize sophisticated strategies.

1.3.2 Distributed Load Balancing

Distributed load balancing becomes necessary for large parallel and distributed systems. These concepts try to keep load balancing efforts (resource consumption as well as delays) constant regardless to the system size, prevent single points of failure by local decision autonomy and reduce information and task exchange between nodes by simple, load balancing policies.

The main advantages of distributed strategy are:

- The jobs arrive at their nodes, so there is no overhead for sending the jobs to the server node.

- The normal operation of the whole system continues even if one or more nodes crash.

Each of the above approach has its merits and demerits in different situation, thus one or the other cannot simply be ruled out.

Central load balancing can avoid load imbalances before they arise, whereas distributed load balancing always has tasks waiting or running at the node where they originated and tries to defeat the imbalances. This project also has to balance the load on the system keeping in view the condition of the system, thus it also requires an efficient method to keep track of system changes. By distributing this task to all the processors on the system, the overhead of communication is greatly reduced.

1.3.3 Categorization of Load Balancing Algorithms

Every load-balancing algorithm involves the following three steps

- Calculating the load parameters / gathering system information
- Making decisions for task migration
- Practically migrating the tasks

In the above section, the main difference between algorithms on the basis of collection of system information has already been discussed. The effect of the next two factors can be seen by the following two type of algorithms,

1.3.3.1 Static Load Balancing Algorithms

A static load-balancing algorithm assigns a process to a host upon invocation. Policies that use only information about the average behavior of the system, ignoring the current state of the system, are referred to as static policies. Static load balancing policies are generally evaluated at compile-time and cannot adapt to unexpected load distributions.

1.3.3.2 Dynamic Load Balancing Algorithms

A dynamic load-balancing algorithm assigns a process to another host after executing on its current host for some time. Problems that have unpredictable computational requirements are best suited for dynamic load balancing policies. Dynamic policies use the current state of the system to make load-balancing decisions at run-time.

When balancing load dynamically, a distributed system must be able to migrate a process from its current host to a destination host. The migration policy determines how to balance the processing load across the hosts in the distributed system. That is, the migration policy decides when a migration should occur. The migration mechanism extracts a process and its associated context from its source host and establishes the process on its destination host for execution. Since for many applications it is almost impossible to predict how much computation a given sub problem involves, a dynamic load balancing strategy is necessary which is able to keep the processors busy without incurring an undue overhead.

The methods of estimating the load of a processor and determining maintenance policy are of primary importance in designing a dynamic load-balancing algorithm. An estimating function can combine several load indicators, including length of the CPU queue, rate of memory occupancy, rate of CPU utilization, rate of communication, and more. However, it has been shown that, in most multi-threaded systems, the length of the ready queue is a good indicator of processor load.

1.4 Classification of Dynamic Load Balancing Algorithms

There are many algorithms available that are used for the distribution of load in a distributed system. On the basis of different policies used for job transfer, these algorithms are categorized as follows:

1.4.1 Random Algorithm

In this algorithm, when a node load level crosses the threshold, it sends the newly arrived job to a randomly selected node. It has the lowest overhead since no system information collection is needed but has the poorest performance due to large job movements and high percentage of wrong job transfers. There is a possibility that a job may oscillate in the network without being executed at any of the nodes.

1.4.2 Sender Algorithm

In Sender Algorithm, when a node becomes overloaded, it polls for the destination and transfers the job after getting ACCEPT message from the destination. The job is processed locally when either the maximum number of probes is reached or if a polling session is already in progress when the job arrives. This algorithm gives better performance at low to moderate load levels because all the nodes become heavily loaded, it becomes difficult to find an idle or underloaded node.

1.4.3 Receiver Algorithm

When a node becomes underloaded, it polls for the overloaded nodes and gets the job transferred from one of them. This algorithm performs well at heavy load levels but not at low load levels because of the difficulty of finding a heavily loaded node, when all the nodes are underloaded.

1.4.4 Symmetric Algorithm

This algorithm is a combination of the Sender and Receiver algorithm, therefore better than the above mentioned algorithms. It initiates the Sender Algorithm when it is overloaded, and initiates the Receiver Algorithm when it is underloaded giving a better performance over the whole range of load levels. However, it involves a higher number of load balancing messages and job movements. This tends to increase the percentage of CPU utilization significantly and also higher negotiation failures result from the concentration of the probing on the time scale.

During an experimental study, load balancing was implemented on a cluster of diskless SUN-2 workstations, running in a distributed systems laboratory under the SUN/UNIX operating system, connected by an Ethernet and supported by file servers [1-2] and load balancing in such an experimental environment was studied.

During a simulation it had already been observed that transparent, and flexible load balancing at the job level could be achieved at a low cost, and without modifying either the system kernel or any of the existing application programs. It was also proved that load balancing is capable of substantially reducing the mean of the process response times, and their standard deviation, especially when the system is heavily loaded, and/or the instantaneous loads on the hosts are considerably unbalanced. A number of load balancing algorithms using periodic load information exchanges or acquiring such information on demand produced comparable performance improvements. Load balancing can still be highly effective when only a small fraction of the workload can be executed remotely.

In another study, Random Polling has proved to be very efficient in practice for applications like depth first search [1-3]. In a recent study of load balancing, an arbitrary network (graph) was used to model a distributed system [1-4]. It was assumed that the load consists of independent *tokens* that may be processed anywhere. An *adversary* that determines the locations and the number of tokens that are added or deleted at any time controls the arrival and departure of load. The main result obtained was that a simple, locally controlled distributed load-balancing algorithm could

maintain the load of the network within a stable level against this powerful adversary.

1.5 Periodic Symmetrically-Initiated Load Balancing (PSI) Algorithm

After discussing some of the systems based on the above mentioned algorithms it was found that the feasibility of these algorithms does not apply to our project. The Symmetric Algorithm produces the lowest job response time but involves larger number of load balancing messages at heavy load levels. Periodic Symmetrically Initiated (PSI) algorithm is one such algorithm which uses periodic probing of nodes to balance the load in a distributed system [1-1]. The performance level provided by the PSI algorithm and its robustness over a range of system attributes and workload makes it a very promising algorithm.

PSI Algorithm involves a smaller number of load-balancing messages than the Symmetric Algorithm and a spreading of probing messages over time. Probing involves the sequential polling of a set of nodes for communication. It is symmetrically initiated and uses periodic polling of a single remote and random node. This algorithm aims at fixing (reducing) the Lp parameter (the probing limit) to one, and at the same time fixing the frequency of algorithm invocation through a timer parameter Pt. This would result in a spreading of the probing messages over time and limit the overhead at high system loads in comparison to the Symmetric algorithm where the algorithm is invoked each time a job departs or starts.

For every timer period the node load is checked against the threshold T.

- If exceeding the threshold (load.i > T+1), a request is sent to a random node (Lp = 1), the node replies with an ACCEPT message if it is underloaded (load.j < T+1), otherwise it ignores the request. The requesting node transfers a job from its transferable jobs queue as a response to an ACCEPT message, or ignores the request if it is no longer overloaded.

- If below the threshold (load.j < T + 1), a request to receive a job is made to a random node. The chosen node will respond by sending a job from its transferable job queue, or just ignores the message if it is also underloaded (load.j < T+1).

- If the load is normal (load.i = T + 1), no load balancing is attempted.

This algorithm is adaptive in the sense that based on the current load level, it activates either its sender-initiated component or its receiver-initiated component. The main measure of the performance of the load balancing algorithms is the metric, job mean response time. This measures the average time that a job spends in the system.

The loosely coupled distributed system which was used for the simulation of the algorithm consisted of a set of autonomous computers connected by a local area network that exchange information through a message passing mechanism and operate in a cooperative fashion. The nodes were homogeneous with no priority for local jobs over remote jobs of the same category. The

jobs were assumed to being executed on the basis of the First-Come-First-Serve local scheduling discipline.

Simulation results showed that:

- PSI Algorithm produces the highest improvement of the job mean response time and that the algorithm ranking based on the increase in the CPU utilization is in the reverse order of the improvement in job response time.

- A higher value of the threshold was found more appropriate when a large compute/communicate ratio (R) is used (T=2 for R=0.4).

- The performance order of the algorithm remains the same for heterogeneous as for homogeneous jobs.

- The level of performance improvement increases with the system size; the best results were obtained for a 20 nodes system.

- The performance becomes insensitive to the number of nodes if they increase beyond a certain limit.

- There is no significant difference between the performance ordering of the algorithms for the diskless and disk-based models of the base-line system.

- It produces the lowest mean job response time and results in less number of messages and job movements than the Symmetric one.

- The file system structure, communication bandwidth, work-load model and system size have no significant effect on the working of the algorithms but leads to different levels of performance by the algorithm.

- For the case of adaptive load balancing schemes based on dynamic parametric tuning the essential parameters to monitor are the communication delay, system load and the system size, whereas the adjustable parameters are the *threshold* and the *timer period*

The PSI Algorithm has proved to have better performance and lowest job response time than all the other algorithms. This algorithm has been simulated and tested, with one of the important result that it is not necessary to consider the *communication protocols* and the *heterogeneous workload* model. However, only the correct representation of the *communication bandwidth* and the *algorithm parameters* tuning (i.e. T, Pt) is important to get the required results.

1.6 Project Brief

The system specifications on which this algorithm is implemented are quite similar to the one on which it had been simulated. The project involved a system of loosely coupled computers connected by a local area network, which communicate through message passing. The simulation results had proved that the PSI algorithm shows the highest improvement in the job mean response time with the least number of messages and job

movements. Also the performance level increases with the increase in system size upto a certain limit.

This project involved the implementation of PSI algorithm on a distributed system using Windows 95. This operating system support loosely coupled distributed system consisting of autonomous computers. Hence it is best suited since the simulation results had been obtained on a similar system. Windows 95 does not support load balancing and with the implementation of this algorithm, it would be possible to include an important feature of a distributed operating system and hence obtain overall better system performance of the network.

Chapter 2

System Specifications

The introduction to distributed systems and load balancing provides a firm foundation to proceed with the project. In this chapter, an overview of the specifications of the system along with the reasons for this specific implementation are discussed. After an introduction to Win32 API, architecture of Windows 95 is studied. The implementation of processes in Windows 95 as well as their movement across the network is thoroughly reviewed.

2.1 WIN32 API

Win32 defines a set of operating system functions that application programs can use to carry out their work. By writing a program in win32 API (Application Programming Interface), the same executable can be run on any win32 implementation. In theory, win32 implementation should be such that each operating system should gloss over any underlying differences in hardware or low level operating system design. One of the key advantages of the win32 is that it is 32bit and has 32 bit executable file formats (Primary Executable).

There are a number of implementations of the win32 API, out of which Windows NT is just one example. Windows NT is the choice of implementation for powerful high-end machines where

robustness and security are of primary importance. Windows 95 also uses a subset of the win32 API but in systems where security is not of major concern. Another subset of the win32 API is the Win32s. It basically compromises of a collection of DLLs and Virtual Drivers (VXDs) that could be added to an existing win 3.1 machine to enable it to run win32 programs. The Win32s libraries provide some of the API functions that Windows NT & Windows 95 have. However, it does not support many features of the modern operating system such as threads and separate address spaces. Threads are a feature of the advanced operating system that allow more than one portion of the program to execute at once. Presented with these options, Windows 95 seemed the appropriate choice with a combination of 16-bit and 32-bit code. Also using a subset of the win32 API allowed the implementation to be extended onto Windows NT 4.

2.2 Windows 95

The PSI algorithm was simulated and tested on a network of loosely coupled computers communicating through message passing. A system was needed to replicate these features, and thus Windows 95 was chosen for the implementation of the algorithm. Windows 95 is not completely a 32-bit operating system. It strikes a balance between 3 requirements: delivering compatibility with existing applications and drivers; decreasing size of operating system to run on 4MB of RAM; and offering system performance. That is why it uses a combination of 32-bit and 16-bit code. It employs 32-bit code where 32-bit code significantly improves performance without sacrificing application compatibility. Existing 16-bit code is retained where it is required to maintain compatibility

or where 32-bit code would increase memory requirements without significantly improving performance. All of the I/O subsystem, device drivers such as networking and file systems; all memory management and scheduling such as kernel and virtual memory manager are fully 32 bit.

Each process in Windows 95 gets its own address space but all loaded system DLLs are visible to a Windows 95 process, not just the DLLs that the process has loaded itself. This means that parts of DOS, win16, win32 processes all mingle in the same address space. Each 32-bit Windows 95 process is in the CPUs page mapping tables only when that process is the current process. When the scheduler switches to another 32-bit process the private memory of the first process is no longer accessible to any other process. This makes it impossible for one task to scribble on another task's memory. Current 32-bit Windows 95 process can see all the memory in use by 16 bit programs. A 32-bit process cannot see the memory of other 32 bit processes.

The implementation of the PSI algorithm means that a process will need to be shifted from one computer to another and required to execute on the other computer. It is, therefore, imperative to gain an understanding of how the Operating System (Windows 95) stores the processes, maintains their essential information, shifts between processes and changes their states. Different communication techniques available and methods employed for communicating between the processes e.g. Inter Process Communication in Windows 95 also needs to be understood.

2.2.1 Task Scheduling and Multitasking

Windows 95 uses a task scheduler to determine the order and priority of processes running on the computer. These processes run as threads. In Windows 95 and Windows NT, which are switched preemptively. Each thread has its own message queue and a separate input system. It assigns mouse and keyboard events to the appropriate queues. This allows one thread to be as unresponsive and take as much time it wants without affecting other programs.

There are two parts of a scheduler process: Primary and Secondary. The *Primary* evaluates all thread priorities and gives a time slice of execution time to the thread of highest priority. If two or more threads have the same priority, they are stacked. Each stacked thread is granted a time slice of execution in sequence, until no threads have the same priority. The *Secondary* scheduler can boost the priority of non-executing threads. This boost helps threads having low base priority from being blocked of receiving execution time.

Windows 95 is not very smooth in multitasking in the presence of 16 bit programs because 16 bit system DLLs are non reentrant and they do not expect to be switched in between execution. The problem is that there are no real synchronization primitives as the multitasking is non pre-emptive. Win32 programs can expect to be switched at any suitable time

2.3 Windows 95 Processes

A process is a unit of ownership. A process owns memory, file handles that the application code can use to read and write files, threads and a list of DLL modules that have been loaded into the process's memory context.

When Windows 95 creates a new process, it also creates a new memory context for the process's threads to execute in. In addition, Windows 95 creates an initial thread of execution for the process, a file handle table in which the process can keep a list of open handles and a process database to represent the process.

2.3.1 The Windows 95 Process Database (PDB)

A process database is a KERNEL 32 object that contains a vast quantity of information about a process. The process database memory is allocated out of KERNEL 32 shared memory heap, so all process databases are visible to all tasks.

The fields of the process database are as follows:

00h DWORD Type
This DWORD contains 5, the KERNEL32 object type for a process.

04h DWORD cReference
This is the number of things that are currently using the process structure for something.

08h DWORD un1
The meaning of this DWORD is unknown, it appears to always be 0.

0Ch DWORD pSomeEvent

This DWORD is a pointer to an event object (K32OBJ_EVENT).

10h DWORD TerminationStatus

This DWORD is the value that would be returned by calling GetExitCodeProcess.

14h DWORD un2

The meaning of this DWORD is unknown. It appears to always be 0.

18h DWORD DefualtHeap

It contains the address of the default process heap.

1Ch DWORD MemoryContext

This DWORD is a pointer to the process's memory context. A memory context contains the page directory mappings to provide a process with its own private region in the 4GB-address space.

20h DWORD flags

24h DWORD pPSP

This DWORD holds the linear address of the DOS PSP created for this process.

28h WORD PSPSelector

This WORD is a selector that points to the DOS PSP for this process.

2Ah WORD MTEIndex

This WORD contains an index into the global module table (pModuleTableArray).

2Ch WORD cThreads

This field is the number of threads belonging to this process.

2Eh WORD cNotTermThreads

This field holds the number of threads for this process that haven't yet been terminated.

30h WORD un3

The meaning of this WORD is unknown. It appears to always be 0.

32h WORD cRing0Threads

This WORD holds the number of ring 0 threads.

34h HANDLE HeapHandle

This DWORD holds the handle of the HEAP that handle tables belonging to this process should be allocated from.

38h HTASK W16TDB

This DWORD holds the Win16 Task Database (TDB) selector associated with this process.

3Ch DWORD MemMapFiles

A pointer to the head node in the list of memory mapped files in use by this process. A node in the list represents each memory-mapped file. The format of each node is:

DWORD	Base address of the memory mapped region
DWORD	Pointer to next node, or 0

40h PENVIRONMENT_DATABASE pEDB

This DWORD is a pointer to the environment database.

44h PHANDLE_TABLE pHandleTable

This field is a pointer to a process handle table.

48h PPROCESS_DATABASE ParentPDB

This DWORD is a pointer to the PROCESS_DATABASE for the process that created this process.

4Ch PMODEREF MODREFlist

This field points to the head of the process's module list.

50h DWORD ThreadList

A pointer to the list of threads owned by the process.

54h DWORD DebuggerCB

This DWORD appears to be a debugger context block. When a process is being debugged, this field points to a block of memory above 2GB. This block includes a pointer to the debugger's process database.

58h DWORD LocalHeapFreeHead

This DWORD points to the head of the free list in the default heap for the process.

5Ch DWORD InitialRing0ID

The meaning of this DWORD is unknown. It appears to always be 0.

60h CRITICAL_SECTION crst

This field is a CRITICAL_SECTION used by various API functions for synchronizing threads within the same process.

78h DWORD un4[3]

These three DWORDS appear to always be set to 0.

84h DWORD pConsole

If this process uses the console, this DWORD points to the console object (K32OBJ_CONSOLE) used for output.

88h DWORD tIsInUseBits1

These 32 bits represent the status of the lowest 32 TLS (Thread Local Storage) indexex.

8Ch DWORD tIsInUseBits2

This DWORD represents the status of TLS indices 32 through 63.

90h DWORD ProcessDWORD

The meaning of this DWORD is unknown.

94h PPROCESS_DATABASE ProcessGroup

This field is either 0 or points to the master process in a process group. Process Groups are collections of processes that belong together. When the group is destroyed, all processes in that group are destroyed.

98h DWORD pExeMODREF

This field points to EXE's MODREF (module list entry).

9Ch DWORD TopExcFilter

This DWORD holds the “Top Exception Filer” for the process. This is the routine that will be called if no other exception handlers choose to handle an exception.

A0hDWORD BasePriority

This DWORD holds the scheduling priority for this process. Windows 95 supports 32 priority levels, grouped into four classes.

A4hDWORD HeapOwnList

This field points to the head of the linked list of heaps for the process.

A8hDWORD HeapHandleBlockList

Moveable memory blocks in the process heap are managed via movable handle tables embedded within the heap. This field is a pointer to the head of the moveable handle table list within the default process heap.

ACh DWORD pSomeHeapPtr

The exact meaning of this field is unknown. It’s normally 0, but when not, it’s a pointer to a movable handle table block in the default process heap.

B0hDWORD pConsoleProvider

This field is either 0, or a pointer to a KERNEL32 console object (K32OBJ_CONSOLE).

B4hWORD EnvironSelector

This WORD holds a selector that points to the process’s environment.

B6hWORD ErrorMode

This field contains the value set by the SetErrorMode function.

B8hDWORD penvtLoadFinished

This DWORD points to a KERNEL32 Event object (K32OBJ_EVENT).

BDh WORD UTState

The meaning of this field is unknown, it has something to do with Universal Thunks. It's usually set to 0.

2.3.2 The Environment database

At offset 40h in the process database is a pointer to a vital data structure, process environment database that also contains process-related information. Its fields are:

00h PSTR pszEnvironment

This field points to the process environment. The process environment is in the block of memory in the per-process data area, and usually resides just above where the EXE module loads.

04h DWORD un1

The meaning of this DWORD is unknown. It appears to always have a value of 0.

08h PSTR pszCmdLine

This field points to the command line passed to CreateProcess to start this process.

0Ch PSTR pszCurrentDirectory

This field is a pointer to the current directory of the process.

10h LPSTARTUPINFO pStartupInfo

This pointer points to the process's STARTUPINFO structure. A STARTUPINFO structure is passed to CreateProcess to specify the process's window size, standard file handles etc.

14h HANDLE hStdIn

This is the file handle the process uses for the standard input device.

18h HANDLE hStdOut

This is the file handle the process uses for the standard output device.

1Ch HANDLE hStdErr

This is the file handle the process uses for the standard error device.

20h DWORD un2

The meaning of this field is unknown. It seems to always be 1.

24h DWORD InheritConsole

This field indicates whether the process is inheriting the console from its parent process.

28h DWORD BreakType

This field indicates how console events should be handled.

2Ch DWORD BreakSem

Normally this field is 0, but if application calls SetConsoleCtrlHandler, this DWORD points to a KERNEL32 semaphore object (K32OBJ_SEMAPHORE).

30h DWORD BreakEvent
Normally this field is 0, but if application calls SetConsoleCtrlHandler, this DWORD points to a KERNEL32 EVENT object (K32OBJ_EVENT).

34h DWORD BreakThreadID
Normally this field is 0, but if application calls SetConsoleCtrlHandler, this DWORD points to the thread object (K32OBJ_THREAD) of the thread that installed the handler.

38h DWORD BreakHandlers
Normally this field is 0, but if application calls SetConsoleCtrlHandler, this DWORD points to a data structure allocated from the KERNEL32 shared heap. This data structure is a list of the installed console control handlers.

After looking at the details of the process structure, the different methods available for communication were studied.

2.4 Synchronous and Asynchronous Communication

Most distributed applications today use synchronous communication (such as remote procedure calls) and do not use queuing. Communications are synchronous (not queued) when the sender of a request must wait for a response from the receiver of the request before it can proceed to performing other tasks. The

time that the sender must wait is completely dependent on the time it takes for the receiver to process the request and return a response. It's acceptable to use synchronous communication for workgroup applications if using adequate hardware to handle the workload. Though, a large peak workload can require a lot of hardware. One of the limitations of the synchronous approach is the overhead involved in starting the server objects. When a client is done with a service, the server object is destroyed.

With asynchronous communication (queuing), senders make requests to receivers and then immediately move on to other tasks. There is no guarantee that receivers will process requests within any particular period of time, but good real time responsiveness can usually be achieved in all but peak load conditions.

Communication operations can be either blocking or non-blocking. A blocking operation does not return until the resources specified in the call can be reused. When sending a message, this means the message has been copied from the send buffer, and the user is free to reuse this buffer. When receiving a message, this means the message has actually arrived in the specified buffer, and the buffer's contents are available for use. When a blocking operation is called, the entire process is blocked, and does not regain control of the processor until the call has completed.

Non-blocking operations may return before the operation completes, i.e., before the user is free to use resources specified in the call. Non-blocking calls return a handle, which may be used for checking the status of the operation. This handle can be used either to wait for the operation to complete (a form of blocking call), or to test if the operation has completed. The resources specified in

the call should not be reused until either a wait is called or a test returns true. Once the user returns from a wait call, or tests positive for completion, the user may reuse the resources, including the handle, specified in the original non-blocking call.

After reviewing the alternatives available for communicating and the structure of processes, the method for IPC could be decided.

2.5 Inter Process Communication

Typically, cooperating and communicating applications can be categorized as clients or servers. A client is an application or a process that requests a service from some other process. A server is an application or a process that responds to a client request. Many applications act as both a client and a server, depending on the situation. The process of communication between two or more applications, regardless of their client/server status, is called *Inter Process communications* (IPC). The use of threads is strictly related to the Inter Process Communication (IPC) mechanism supported by Win32. The basic idea is that the developer needs to coordinate all the thread activities, protect shared resources, and synchronize access to system devices or memory areas.

Many IPC mechanisms, such as dynamic-link libraries and shared memory, are implemented only on the local system. Other IPC mechanisms, such as distributed COM (DCOM) and remote procedure calls (RPCs), find their primary functionality across networks and between different computer systems. Each IPC mechanism has unique features that singularly describe it, and

which should be considered during the development of an application. Depending upon the actual task win32 provides the following tools:

2.5.1 Semaphores

A semaphore is useful in limiting the number of simultaneous accesses to a counter that is a shared resource. This IPC mechanism is either in the signaled or non-signaled state. Its state depends on the value of the counter. Setting a semaphore counter is an operation that is strictly related to the creation of the synchronization object.

2.5.2 Mutexes

A mutex is an IPC mechanism that allows mutually exclusive access to a shared resource to all waiting threads except one. Writing in a shared memory block is a typical example of effectively using a mutex. When an application must grant access or ownership to a specific resource to a single thread at a time a mutex is the appropriate IPC tool to use. A mutex is the only IPC object that is owned by a thread.

2.5.3 Events

An event is extremely useful in coordinating the activities of two or more threads. A thread can be created in suspended mode, though once activated, there is no other way to re-suspend it unless

invoked. Events offer a valid and more efficient alternative to implementing a thread that is always alive, though the event actually executes only when necessary. For example, several threads have to be created when the application starts, but executes only a few of them. Some threads will block waiting for an event to occur therefore entering a wait state. When the event is signaled, the waiting threads are free to continue execution. An event acts like a signal informing other threads that a specific event has occurred. Events are extremely useful to dynamically suspend and resume the execution of one or more threads, implementing the underlying application design and logic.

2.5.4 Critical Sections

The critical section is a very simple IPC mechanism that protects a set of data in a multithreaded environment. According to Win32 terminology a critical section is a portion of code that accesses some application data that require additional protection because other threads can perform some changes on them. By encapsulating all these portions of code between APIs, it can be ensured that only one thread at a time is allowed to access and modify those data. Critical sections work only inside a multithreaded process. They are the simplest form of IPC because they enforce a rather weak form of thread synchronization.

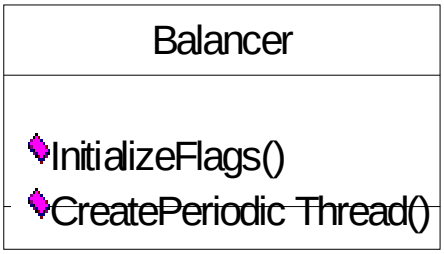
Chapter 3

Software Structure

The implementation of any project is preceded by analysis of how the project will be structured. After reviewing the requirements of the load balancer, the objects, in the form of class diagram, are discussed in this chapter. These objects encapsulate appropriate functions and attributes.

3.1 Balancer

Creates the thread to handle the decision module and the migration of the processes. It contains all the load information and acts as the base class of all the other classes. All the other classes are associated with this during some time of execution. One instance of each class will be associated with the Balancer class.



3.1.1 Public Properties

load : int*

An array to hold all the loads of the computers on the network

3.1.2 Private Properties

IPAddress : char*

An array of the IP addresses of all the computers on the network

STATE : char

State of the computer

ThresholdHistory : int*

Array of integers to hold the threshold values of the past time periods

3.1.3 Public Methods

InitializeFlags (void) : void

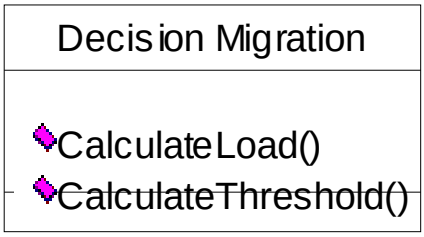
At every time period, all the flags are initialized controlling the operations.

CreatePeriodicThread (lpvoid)

The periodic thread handles the execution of load balancing algorithm periodically.

3.2 Decision Migration

This class incorporates most of the functions, which are required for making the decision relating the course of action taken by the computer. To reach a decision the computer needs to communicate with other computers on the network. It is thus associated with the Communication class. Thus, each instance of the decision migration class will associate with an instance of the Communication class. It will also need to prompt the Migration class to inform it of its decision, as the migration class will have to act upon it.



3.2.1 Private Properties

SelfLoad : char*

A string variable to hold the status of load on the computer.

3.2.2 Public Methods

CalculateLoad (void) : void

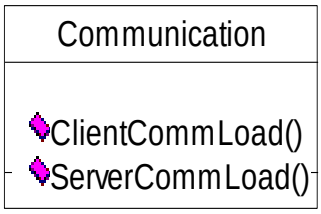
Calculates the total load on the computer

CalculateThreshold (void) void

Calculates the threshold value of the network based on the total load of all the computes on the network.

3.3 Communication

This class performs all the functions of communication required by the load balancer. It is associated with the decision migration class, as it needs to pass any information that it communicates to the decision migration class.



3.3.1 Public Methods

ClientCommLoad (void) : void

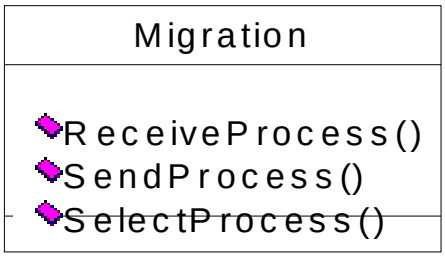
It communicates load with a subset of the network acting as a client in the communication

ServerCommLoad (void) : void

It communicates load with a subset of computers acting as a server to other computers

3.4 Migration

This class performs the functions, which are actually required for migration of a process or reception of a process. It communicates with the MFC classes, which keep a record of the processes, migrated or received.



3.4.1 Private Properties

InProcessCount : int

Total number of the foreign processes present in the system

OutProcessCount : int

Total number of processes sent to other computers on the network

3.4.2 Public Methods

ReceiveProcess (AddrCount : int) : void

This process is invoked when underloaded receives the process from the overloaded computer

SendProcess (void) : void

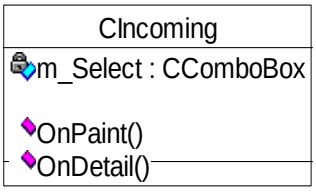
This process is invoked when overloaded and responsible for sending a process to an underloaded computer

SelectProcess (void) : ProcessStructure

Selects one of the processes chosen from the processes present in the ready queue of the computer, to be migrated to the other computer. System processes cannot be migrated.

3.5 *CIncoming*

This is a Dialog Class representing the total number of processes, which are migrated, into that computer from other computers of the network.



3.5.1 Private Properties

m_Select : CComboBox

Gives the option of seeing the detail of any incoming process

3.5.2 Public Methods

OnPaint (void) : void

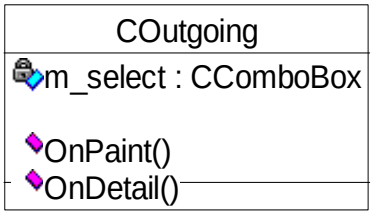
Displays the total number of processes and also gives the option of seeing the detail of any of them

OnDetail (void) : void

The details of the process selected is shown to the user

3.6 *COutgoing*

Descendent of the CDialog Class representing the total number and details of the Outgoing Processes



3.6.1 Private Properties

m_select : CComboBox

Gives the list of the outgoing processes whose details can be seen

3.6.2 Public Methods

OnPaint (void) : void

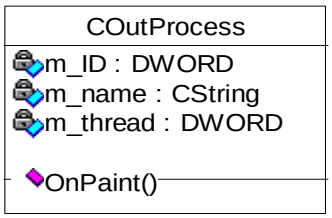
Displays the total number of outgoing processes and also gives the option of seeing the details of them

OnDetail (void) : void

Gives the details of the selected outgoing process

3.7 *COutProcess*

Class representing the details of the migrated processes



3.7.1 Private Properties

m_ID : DWORD

Process identifier

m_name : CString

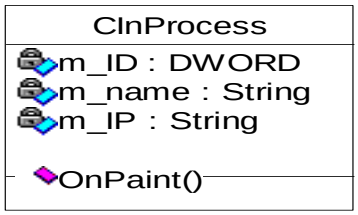
Name of the Process

m_thread : DWORD

Gives the total number of Threads of that process

3.8 *CInProcess*

Class representing the details of the incoming process



3.8.1 Private Properties

m_ID : DWORD

Gives the Process ID

m_name : CString

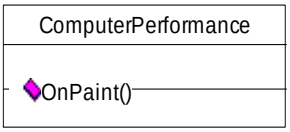
Represents the Name of the Process

m_IP : CString

IP Address of the source computer

3.9 ComputerPerformance

This class is the descendent of CDialog Class of MFC. It is responsible for showing the graph representing the total load on each computer on the network.



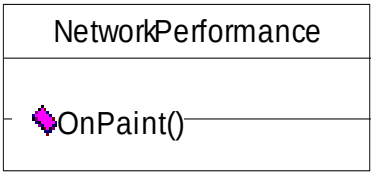
3.9.1 Public Methods

OnPaint (void) : void

The values for the load of all the computers on the network are taken from the Balancer Class

3.10 NetworkPerformance

This class has CDialog as its parent class. It displays the overall network performance by drawing the graph of the threshold value changing with time.



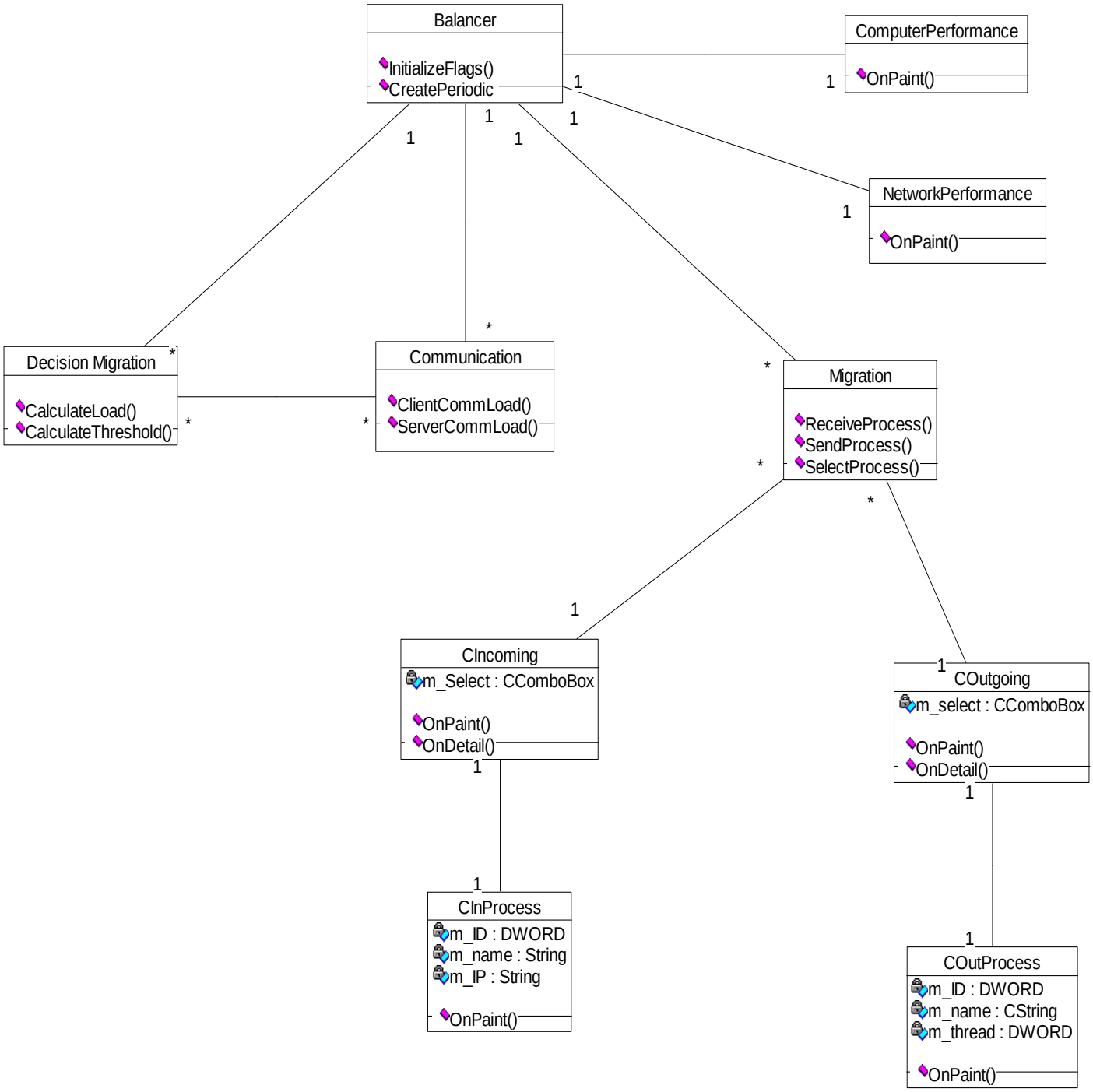
3.10.1 Public Methods

OnPaint (void) : void

It takes the parameter from the Balancer Class for drawing the graph.

3.11 Overall Structure

At first the balancer class is instantiated, it creates instances of all the classes, as each class is associated with the balancer class. It also creates a period thread, which is required because the execution has to be split into two paths. The threads then use the function of each class to communicate load, calculate the threshold, decide the computer's state, and finally send or receive the process


Balancer
InitializeFlags()
CreatePeriodic
ComputerPerformance
OnPaint()
NetworkPerformance
OnPaint()
Decision Migration
CalculateLoad()
CalculateThreshold()
Communication
ClientCommLoad()
ServerCommLoad()
Migration
ReceiveProcess()
SendProcess()
SelectProcess()
CIncoming
m_Select : CComboBox
OnPaint()
OnDetail()
COutgoing
m_select : CComboBox
OnPaint()
OnDetail()
CInProcess
m_ID : DWORD
m_name : String
m_IP : String
OnPaint()
COutProcess
m_ID : DWORD
m_name : CString
m_thread : DWORD
OnPaint()

Chapter 4

Procedural Flow

The whole design can be viewed as the flow of different processes running. These processes running can be best depicted using activity diagram, which takes into account the synchronization as well as the simultaneous running of different threads.

4.1 Design Behaviour

All the activities related to classes in this design can be combined together to form an activity diagram. The whole design of the project is such that one activity is leading to another activity where activity represents the operation on a particular class. The class diagram is used to represent the overall structure, while the activity diagram is used to show the behavior of the design.

4.2 Basic Modules of the Activity Diagram

The activities in the whole design can be divided into two modules:

- Decision
- Migration

The “Decision” module comprises of a number of activities. The load is calculated, it is communicated with other nodes on the network to calculate the threshold value, own state is found and then depending upon the state, either a process is migrated to another node or a process is accepted from some overloaded node.

The “Migration” module is responsible for the successful migration of the process. If a node is overloaded, it randomly selects one of the processes from its ready queue and moves it to the underloaded computer. Care is taken that the process selected is not a system process or the load balancer itself. In case the node is underloaded, it accepts a process from the underloaded node and executes it. The decision of finding a node for communication is also part of this module.

The whole design of the activity diagram can be divided into the following steps:

4.3 Initialization

Having a multithreaded design optimizes the whole process. A thread is the smallest unit of execution and helps to handle several activities in parallel. They are a convenient way to keep various portions of the program running while other portions are waiting for some external action to occur.

Initially a thread is created to handle the periodic execution of the whole algorithm. Flags and variables are initialized in order to avoid wrong values usage in the program. The thread then

calculates the load on the node by taking in to account the number of processes in the ready queue of that node (Fig 4.1).

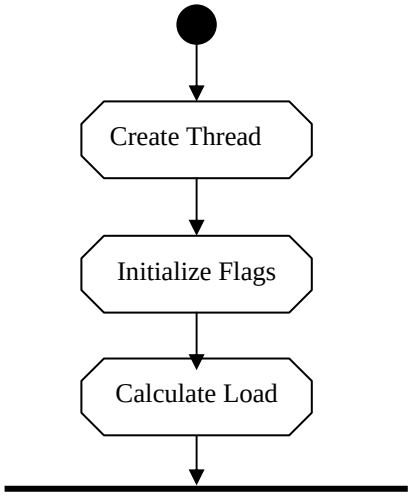


Fig 4.1: Initialization

4.3.1 Initialization of Flags

The flags are used for the synchronization of threads running. Hence for every time period these global flags are initialized for the proper synchronization of the software.

4.3.2 Calculation of Load

The load on each computer is defined in terms of the number of processes running on that computer. The periodic thread at the start of every time period initiates this process. The method "*CalculateLoad*" is defined in the *DecisionMigration* class, which does not take any input parameters nor does it return any value. It uses its own variables to calculate the number of processes running in the system and displays other important details of those

processes like the process identifiers, process size, number of threads owned by the process etc.

4.4 Load Communication

The next phase is to create two threads, which are used to handle the communication with other nodes on the network. Each thread communicates with a subset of the network to find the loads on all the computers on the network. These thread procedures are the *SendThread()* and the *ReceiveThread()*. Both the threads call a separate method for communication of load across the network. This divides the work of each thread.

One thread starts from one end of the network, starting from (n-1) node (if it is the nth node) and sends its load to other computer and receives the other computers load, after which it communicates with (n-2) node (see Appendix A for details).

At the same time the other thread communicates with (n+1)th node and repeats the same process of send and receives, moving onto the (n+2)th node (see Appendix A for details).

This enables the computer to cover the network from both sides at the same time making the process of communication of load twice efficient time-wise. If the network consists of three computers, the process of each thread is executed only once as each thread gets the load of one computer at the same time. Since both of the threads are executing at the same time, the communication of load will be only five times i.e. half the total number of computers, for a network of ten computers.

The process is repeated for every time period as each computer recalculates its load. The process of sending and receiving the loads is through sockets. Using standard Winsock API's a connection is first established and then the load is sent and received on it.

After both of the threads have communicated loads, the next phase is invoked. It is represented in the activity diagram by using {AND} at the synchronization bar (fig 4.2).

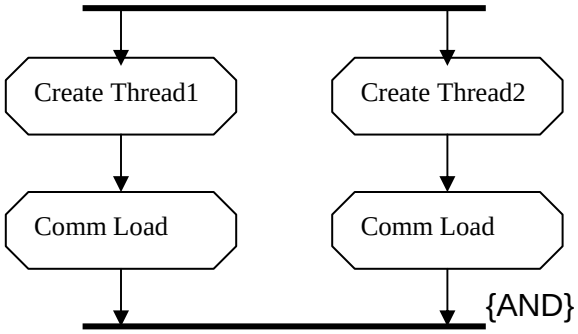


Fig 4.2: Load Communication

4.5 Calculation of Parameters

After obtaining the loads of all the nodes, certain values are to be calculated which are used for the migration of processes (fig 4.3).

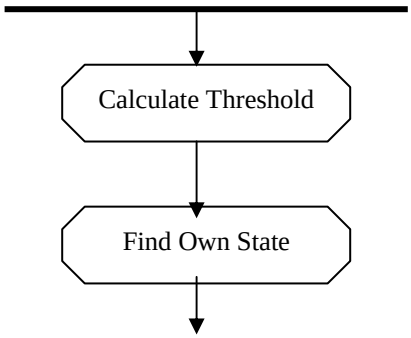


Fig 4.3: Calculation of Parameters

4.5.1 Threshold Value

A similar value of load on all the computers in the network can only be achieved if at any stage an average of the load of the complete network can be computed. The objective is to find such a value from within the network instead of setting a goal for the network without considering the overall state and capabilities of the complete network. This would be the case when the threshold value of a very busy network is set low, which the network will not be able to handle. In that case all the computers of the network would be overloaded, and there would be no computer to receive a process. Similarly, if the threshold of the network is statically set to a high value when the loads on each computer are low. The computers will be below the threshold of the network and will again not migrate any process. Thus by looking at the existing value of loads on each computer of the network, an average value is calculated which the algorithm can execute with.

As each computer has a list of the loads on the network; its own as well as of all the other computers on the network, the average value calculated is the threshold of the network. This

threshold value will differ from network to network depending upon the time of calculation. If the threshold value is being calculated at a time when most of the computers of the network are under utilized, the threshold will subsequently be low. However, if the threshold is calculated for a network, which is being used extensively, the threshold value calculated for the network will be high.

The method “CalculateThreshold” used is from the DecisionMigration class. The average of these loads is calculated on each computer. Since the number of computers is the same as is the load since it is communicated across the network, the value obtained for the threshold is the same on each computer. This is the threshold for load on the network in one period of time. This process is repeated for every time period by the periodic thread as the load continues to change across the network.

4.5.2 Determination of state and decision

Once the threshold is calculated the computers can determine their state on the network with respect to other computers. Each computer compares its own load with the threshold value and sets its state, which can be one of the following:

- Overloaded
- Underloaded
- Balanced

The state of the computer then governs the further actions that the computer takes. If the computer is overloaded, it will migrate a process from its system. If the computer is underloaded, it will look for other computers on the network, which are overloaded and take a process from one of them. If the computer is on the threshold level of the network, it will neither migrate a process from itself nor will it migrate a process from any other computer on the network. In each time period, each of the computers will change its state as the load condition on itself as well the network changes.

Thus, calculating the load on the computer and in the network enables each computer to check its state against the state of the complete network and accordingly make a decision to proceed further. The process of decision making is repeated in each time period to accommodate changes made in the state of the network by running the algorithm once.

4.6 Process Migration

Once each computer has calculated its load, determined its state with respect to the threshold value of load in the network, it either sends or receives a process depending upon the output of the decision making module. After this starts the job of the migration module.

4.6.1 Path Selection

Each computer will execute according to its state calculated. Once the decision to send a process or receive a process has been

taken, execution is divided between two threads, the sending thread and the receiving thread.

The sending thread starts execution on the overloaded computers from which processes have to be migrated. The receiving thread in over loaded computers becomes dormant at this stage. Similarly, the receiving thread starts execution on the under loaded computers which will receive a process. Again the sending thread becomes dormant in the under loaded computer.

Another important issue when the threads take control is which computer is going to send or receive the process. In one time period, the most overloaded computer will transfer a process to the most under loaded computer in the network. The migration thread will determine first whether it is the most overloaded computer or not (see Appendix A for details).

If it is the most overloaded one, it will wait for the most under loaded computer to connect to it, otherwise it will not do anything. The receiving thread, in the under loaded computer, will also verify that it is the most under loaded computer in the network (see Appendix A for details), otherwise it will also not do anything. Else it will determine which is the most over loaded computer in the network, take its IP address and connect to it (see Appendix A for details).

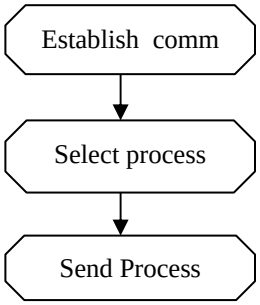


Fig 4.4: Overloaded Node

4.6.2 Overloaded Node

The computers, which are being used extensively, will have a higher number of processes running on them and will have a high value of load (fig 4.4). They will need to move some processes executing on them to other computers, which are running less number of processes. An overloaded computer will simply connect to the most under loaded computer, select a process and move it to another computer but this requires a number of steps, which are explained below.

4.6.2.1 Process Selection

The first step in the migration of a process is to select a process that is to be moved. This is given importance because of a few reasons. Firstly, the scope of the PSI algorithm does not extend to processes, which require interaction with the user during execution as including the user would mean that the time of execution of the process would become user dependent.

Secondly, a system process should not be moved to another computer. This is only because that the system processes do not add to the overall load on the computer. If there is no user process running on any computer in the network, because of similar operating system the number of system processes running on each computer will be the same. Thus each computer will have the same number of processes, the calculation of the threshold of such a network will be easy. The threshold value of the network will be the same as the number of processes running on each computer and

each computer will be on the threshold value. In such a case there will be no need to shift any process across the network.

The method used for this purpose is *SelectProcess()* of the Decision Migration class. It is called from the *SendProcess()* of the Migration class. The sending thread initiates this method in each time period of migration if the state of the computer is over the threshold level of the network. This method returns a structure containing information about the selected process. The method used to actually select the process is quite similar to the method used to calculate the load on a computer. From the list of processes obtained from the running queue of the system, the system processes are separated. Another process that should not be moved is the load balancer, which is the current process. The running processes are thus divided into two groups, those allowed for migration and those not allowed.

4.6.2.2 Sending a Process

The procedure of selection leads to the main purpose of the whole, the actual movement of the process from the overloaded computer to the under loaded one. The main job of this method is to select a process and move it to another computer. The essential points to consider during this method is to establish a connection with an under loaded computer which is to receive the migrating process. The receiving computer also needs certain information about the process being migrated.

The method used is *SendProcess()* of the Migration class. It is invoked every time period of the load-balancing program. The migrating thread calls this method when it is overloaded. It calls the

SelectProcess() of the Decision Migration class. This method in turn returns to it a structure containing necessary information about the process to be shifted. The *SendProcess()* function uses APIs to retrieve the information and manages this information to send it over to the receiving end, where the process is being shifted. This will include the name of the process that is to be shifted, its process identifier and also the size of that process. A connection is established with the other computer on the network using standard Winsock API's. The under loaded computer to which the connection has to be made has already been decided. Once the connection is created, first the information related to the process is shifted and then the complete process is shifted. After the successful migration of the process the process is no longer running on the overloaded computer. The number of processes on that computer is thus one less.

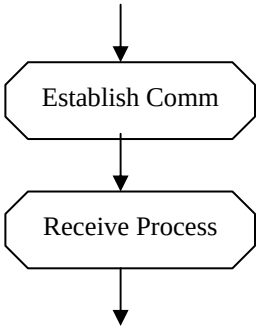


Fig 4.5: Underloaded Node

4.6.3 Underloaded Node

Under loaded computers will allow other computers with heavy job loads to shift their processes to them. Once their state is established as being below the threshold level of the network. They connect to overloaded computers and take a process from them

and execute that process. Resulting in increasing its own load till the process finishes execution (fig 4.5).

4.6.3.1 Receiving a Process

The method *ReceiveProcess()* of the Migration class is used for receiving a process from the overloaded computer. The receiving thread initiates this process at every time period, after it checks that it is the most under loaded. It is provided the address of the most over loaded computer, from which it will receive a process. It connects to that computer.

First, the method requires certain information about the process that it has to receive. It normally takes the name of the incoming process and its size. This method then waits till the complete process has been received. Once it is complete, the process is initiated on the computer so that it starts executing.

4.7 Record updating and Waiting

After the migration of the processes, the structure containing information about the incoming and outgoing processes is updated.

The sending computer updates its information about the processes running on it, in addition to maintaining record of the processes migrated. The CoutGoing class has defined a structure for this purpose. This will include the name, identifier of the process shifted and the computer it is shifted to.

The under loaded computer, after receiving a process from the over loaded computer, updates its record structure. This structure is defined in the class and includes the name of the process, the process identifier and details of the computer from where it was migrated.

All the computers in the network wait for a specific time in every time period, whether over loaded, under loaded or balanced. After executing its required jobs in a particular time period, each computer waits for a pre-defined amount of time. This is the time for the periodic recalculation of the load of all the computers on the network. This time period was calculated through repeated calculations and experimentations with different values of the time period. This is necessary because the next time period must start at the same time for all the computers. If a computer is on threshold level in a time period, it will not have to do anything. Thus at the end of each of the above described processes, the computer will wait for the next time period to begin.

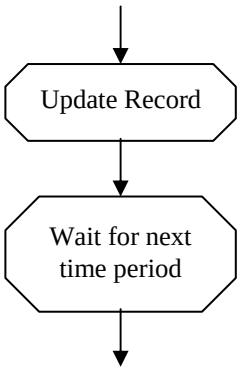


Fig 4.6: Record updating and Waiting

4.8 The Overall Activity Diagram

Figure 4.7 gives the complete activity diagram of the load balancer and shows the complete procedural flow of the balancer. At the start of each time period, flags are initialized which are used for the synchronization of the threads. The periodic thread is created once at the start of the program execution, which handles all the activities in a periodic manner. After initialization of the flags, load is calculated which represents the total number of processes in the ready queue of the system. Then two threads are created for the communication of calculated load with all of the computers on the network. Each thread communicates with a subset of the computers for improved performance. After the communication of load, threshold value is calculated representing the network load. This threshold value is used for the categorization of own load as underloaded, overloaded or balanced. The most overloaded computer on the network selects a process and migrates it to the most underloaded computer on the network. After migration, both these computers update their structure containing record of outgoing and incoming processes. Rest of the computers wait till the start of the next time period.

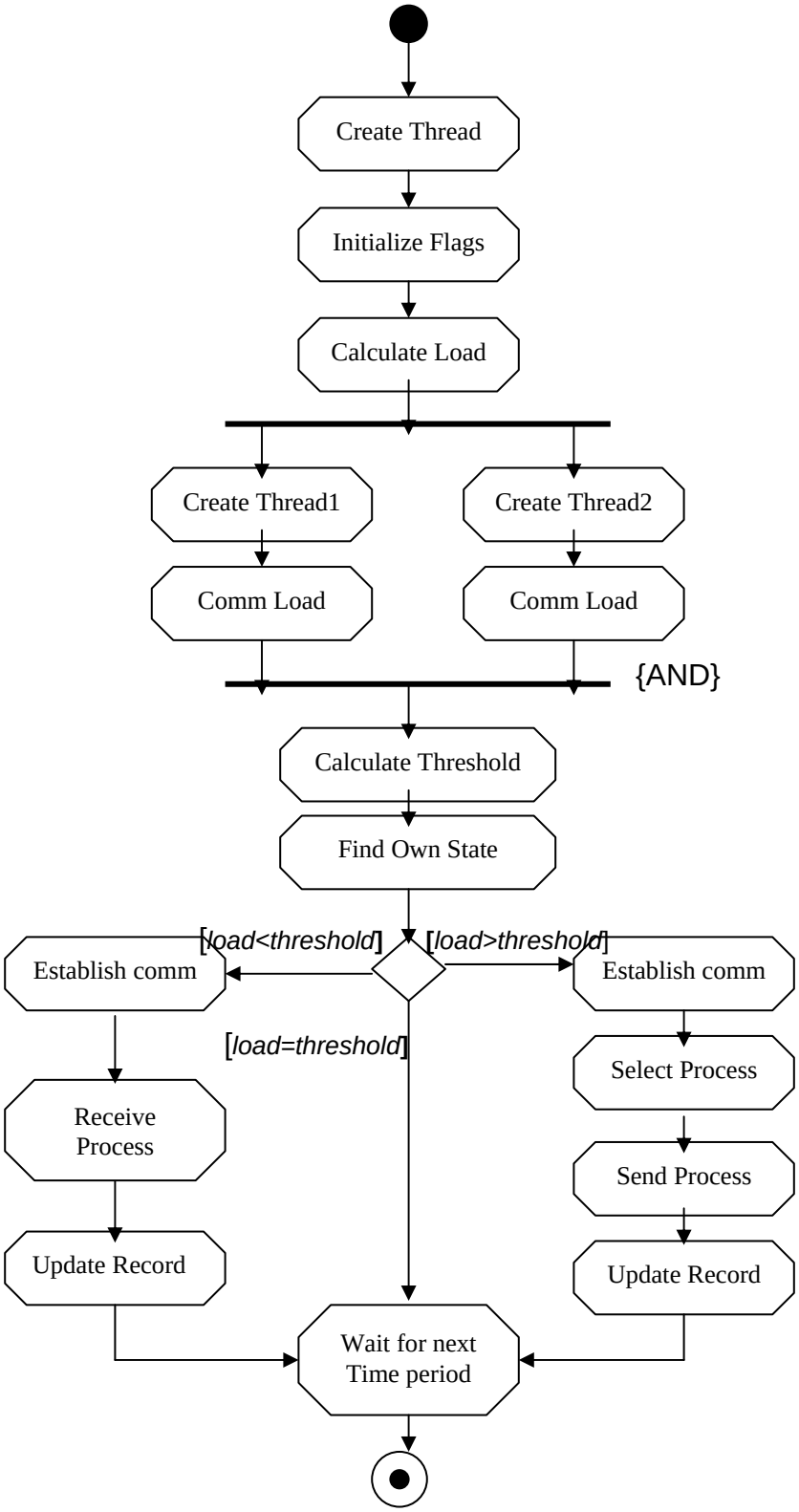


Figure 4.7 Complete Activity Diagram

Chapter 5

User Interface

The movement of processes in the network in user-transparent manner was the main aim of the project. Hence for the user, the interface available is a window showing the performance of the network.

5.1 Load Balancing

When the application starts, main window gives the information about the project and different options are available for the user to choose from the Toolbar as well as the Main Menu (fig 5.2). These options are the source to see and monitor the network as well as the load on the computer.

The graphs give an insight into the overall performance of the network, while the data available is a source of information about all the processes moving across the network.

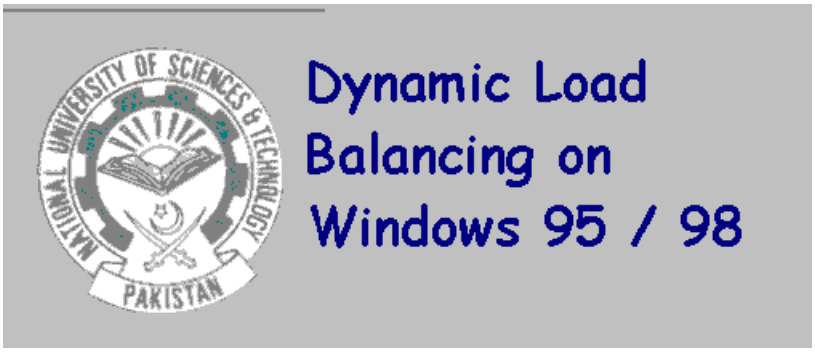


Fig 5.1: Cover Window

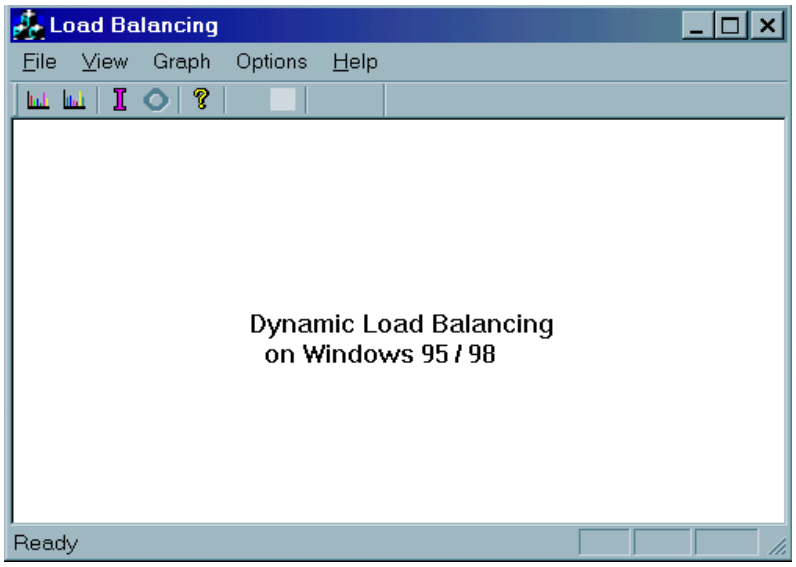


Fig 5.2: Main Window

5.2 Graphs

There are two types of graphs available. The first shows the total load present on each computer while the second gives the network performance by giving the change of threshold value with each time period.

5.2.1 Load on the computers

Taking the computers on the x-axis and the load on them on y-axis shows the load on each computer. For one time period, the graph remains the same and is updated each time period when the load is communicated to other computers on the network (fig 5.3).

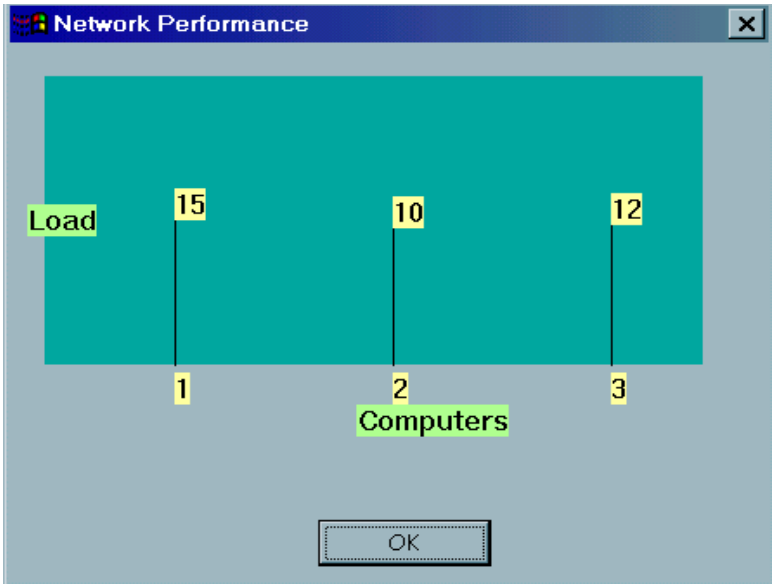


Fig 5.3: Network Performance

5.2.2 Overall Network Performance

Taking the time on the x-axis and the total load on the y-axis shows the overall network performance. This graph is also updated each time interval and gives the overall performance of the network with time. The time interval on the x-axis is taken equal to the time period of the algorithm. Hence by changing the time period, the time interval for the graph is also changed. The graph is drawn with 5 previous values of the total load, hence the load history is maintained for the previous 5 time intervals (fig 5.4).

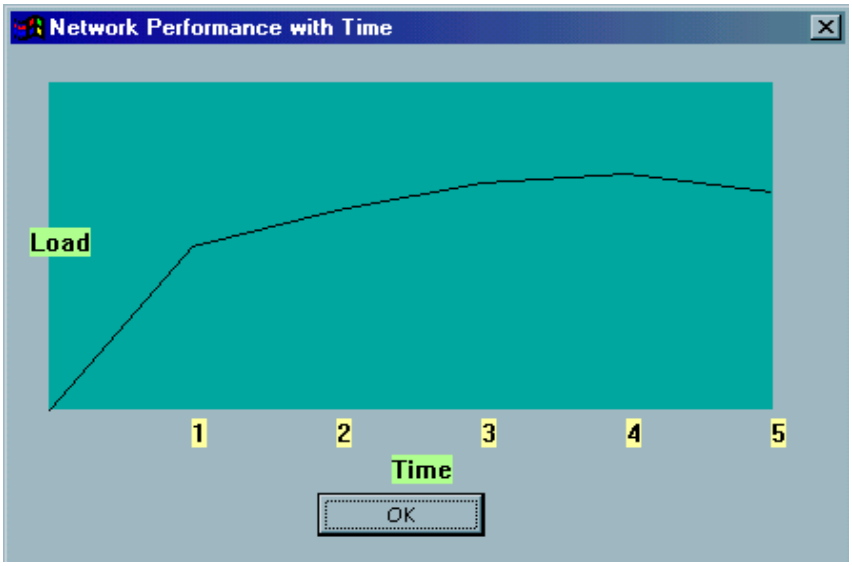


Fig 5.4: Network Performance with Time (time vs. total load)

5.3 *Options*

The various options available are in accordance with the presence of processes in the ready queue of the computer. The processes, which are present as a result of migration from other computers, as well as the processes, which are migrated to other computers, from that particular computer are shown in detail to the user.

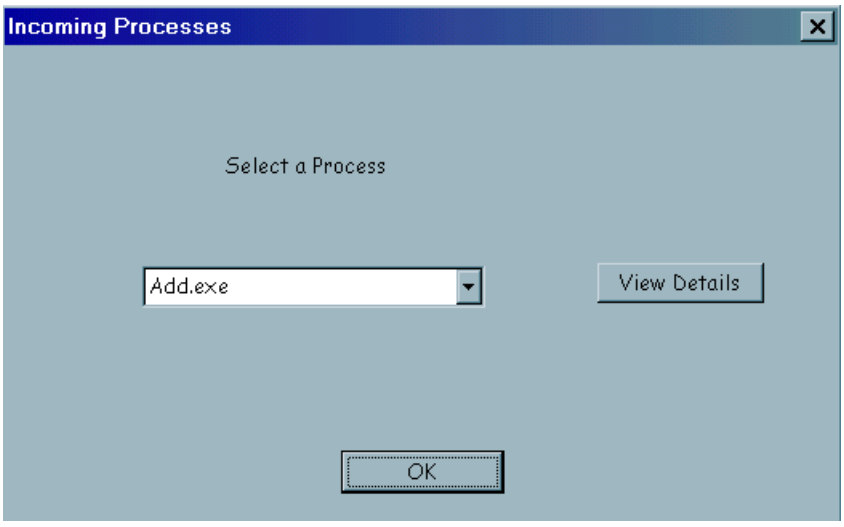


Fig 5.5 Incoming Processes

5.3.1 Incoming Processes

The details of all the foreign processes are kept in a data structure and are available to the user. The main window gives the total number of incoming processes, and their names. The user can also see the detail of any of those processes by selecting the option “View Details” (fig 5.5).

In the detail of each process, Process Name, Process size, IP Address of the computer from where it was migrated and the Process Identifier is shown to the user (fig 5.6).

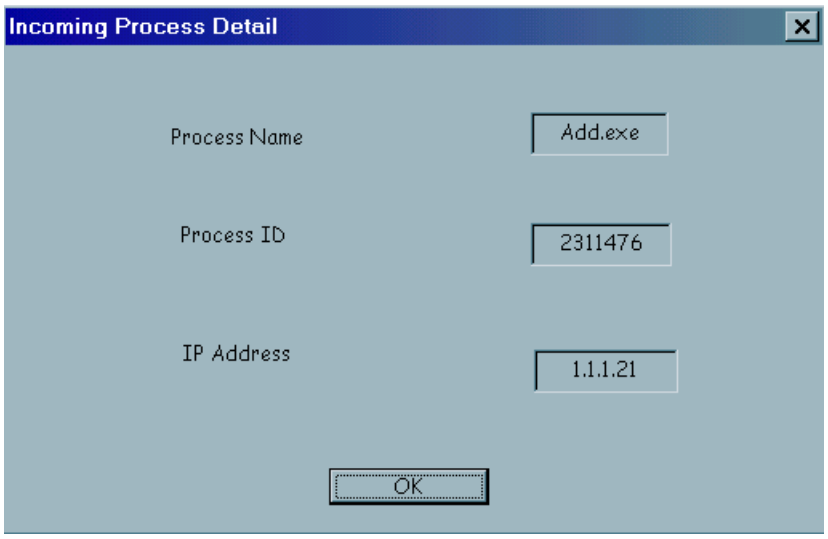


Fig 5.6: Incoming Process Detail

5.3.2 Outgoing Processes

The data of all the processes, which have been migrated to other computers, is also available for the user. If an appropriate option is selected, a window gives the total number of processes and their names (fig 5.7). The user can then select one of the name

and use the option “View Details” to have the details of that process.

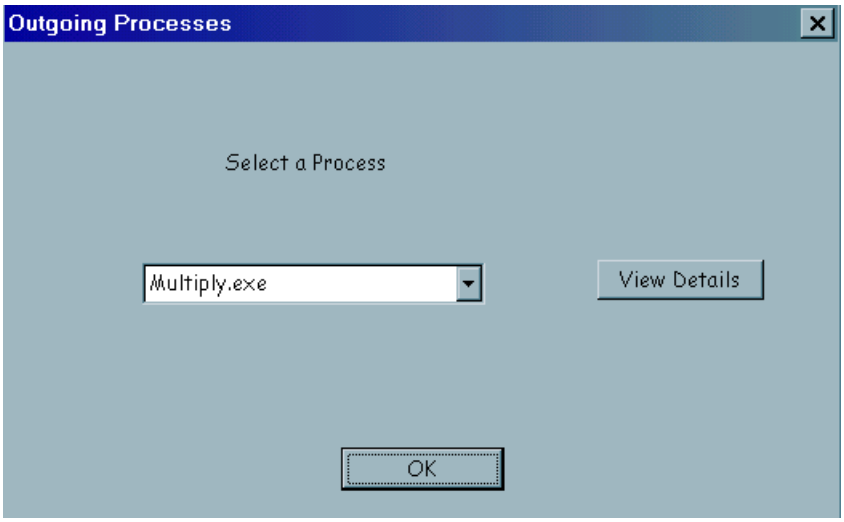


Fig 5.7: Outgoing Processes

The detail of each process contains the name of that process, its Identifier, Size of the process and the total number of threads in that process (fig 5.8).

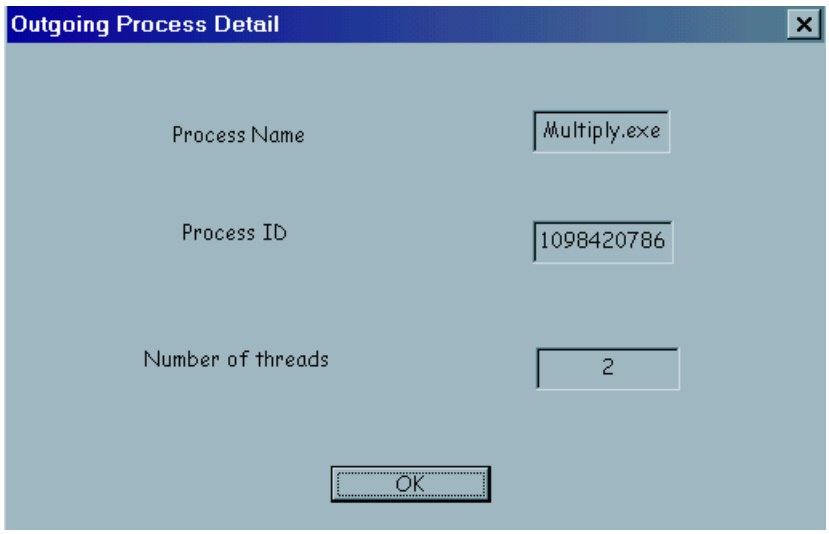


Fig 5.8: Outgoing Process Detail

The only purpose of such type of interface to the user is to display the desired information and also to give the overall performance aquired as a result of the implementation of the load balancing algorithm.

Chapter 6

Results and Further Recommendations

This chapter gives an overview of the results that the project has produced. This includes the aims of the project, its accomplishments, and its performance on a practical system, the difficulties faced in implementation and the limitations of the load balancer.

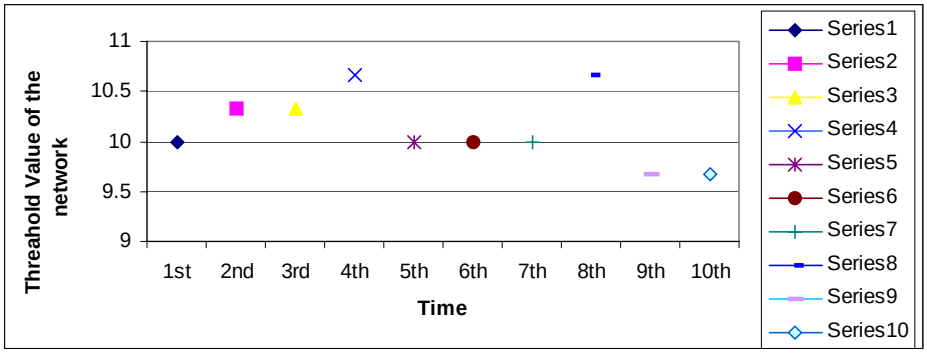


Figure 6.1: Cumulative load on the network changing with time

6.1 Load Balancing

The project was tested for computational jobs only. The graph (figure 6.1) shows the cumulative load on the three computers during each test time period. This gives an overall view of the load; each computer's load is not taken into account separately. The graph shows the execution of the load balancer on a three-computer network. Initially the load is high as a number of processes are initiated (in Series 2, 3 & 4) but when they finish

execution the graph goes down (in Series 5, 6 & 7). The graph peaks as the other processes are started.

6.2 Testing for different time periods

In view of the fact that the algorithm to be implemented was periodic, time period was an important parameter in the load balancer. This is the time period for the load balancer to recalculate the load of each computer and communicate it on the network in order to migrate a process. The time period was to be adjusted keeping in mind that enough time period be adjusted to allow migration to take place entirely. Also the time should not be too large for the migration to become ineffective. Both these conditions were tested on a network of three computers.

The time period was tested at different experimental values to determine the optimal value. The detail of each testing period is discussed below. Each graph shows the load of the computers after 20 milliseconds.

6.2.1 Time period at 15 ms

The time period was initially set to 15 milliseconds. When the time period started, the balancer took some time to calculate the load on the computer, communicate it with the other computers on the network, and then perform the actual migration. Before the migration was complete, the time period expired and the third computer restarted its calculations. The other two computers, which were still shifting a process, could not exchange their new recalculated loads with the third computer. The third computer thus

got abandoned in the process of load communication, as the other computers could not communicate at that time. In another time period, the migration of the process did not take as long as before and all three computers started the next time period to communicate the load successfully. Thus in certain cases the balancer had a possibility of being suspended due to lack of synchronization during communication.

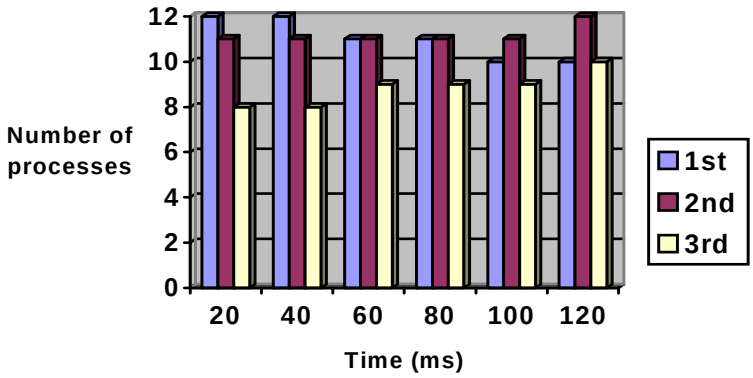
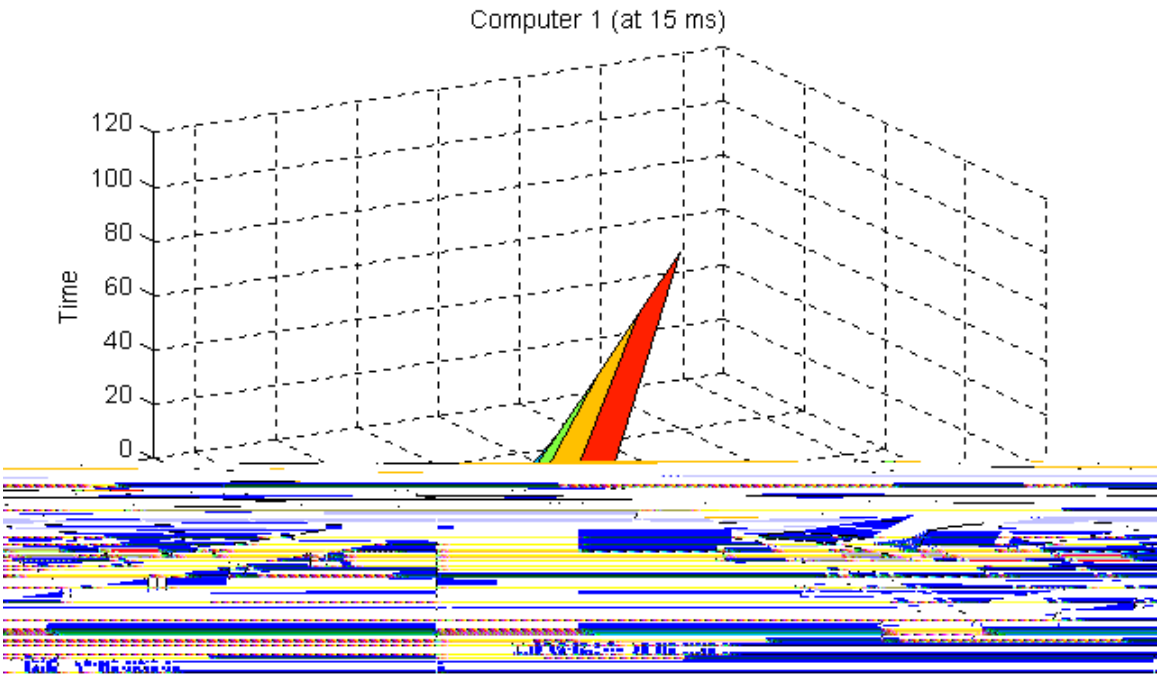
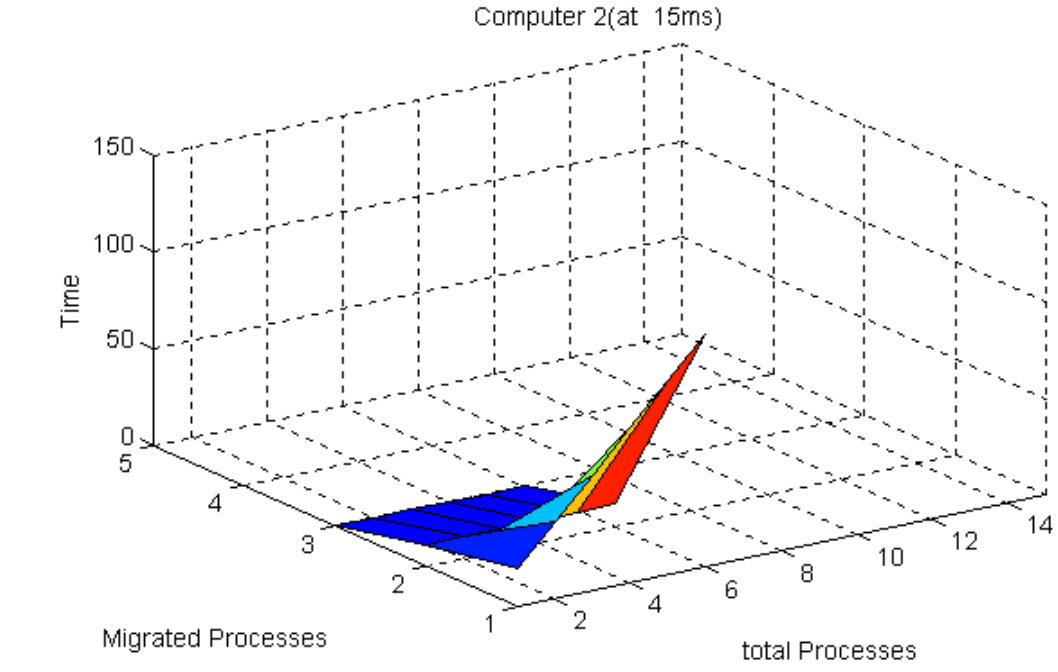
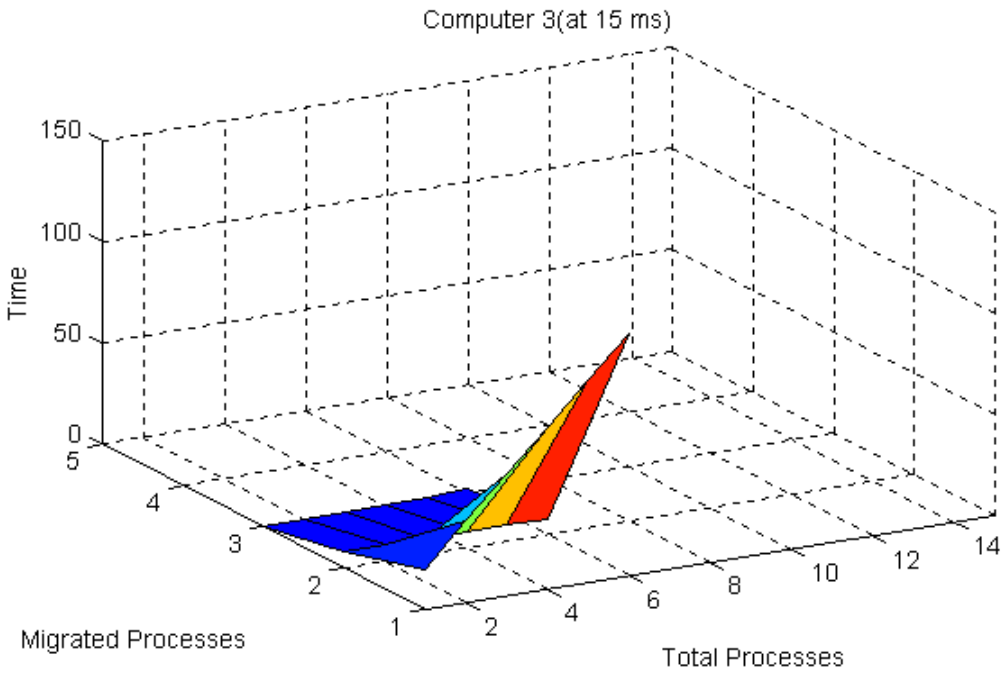


Figure 6.2: Balancer with a 15 ms time period

The graph (in figure 6.2) shows the results of the load balancer at the 15 milliseconds time period. Initially the load on computers 1, 2 and 3 is 12, 11 and 8 respectively. The load is again checked after 20 milliseconds and is the same. This shows that no load balancing took place in the first time period. At 40 milliseconds there is a change in the load, which shows that exchange of process did take place. The next observation in the graph at 60 milliseconds also shows the same load but there is a change at 80 milliseconds. This shows that the time period does not allow a complete migration to take place in every time period. The rest of the graph also shows that a process does not necessarily migrate in every time period.


Computer 1 (at 15 ms)
Time
120
100
80
60
40
20
0


Computer 2(at 15ms)
Time
150
100
50
0
5
4
3
2
1
Migrated Processes
2
4
6
8
10
12
14
total Processes



6.2.2 Time period at 30 milliseconds

The next experiment was carried out for a time period of 30 milliseconds. This was considered to be enough to allow the balancer to complete its process of migration completely, unlike the first time period of 15 milliseconds. This time period gave relatively good performance, it allowed the balancer to completely shift the process and did not make it wait too long to start the next time period. Thus a process was shifted in nearly every time period except when the load of the three computers was near or on the threshold value.

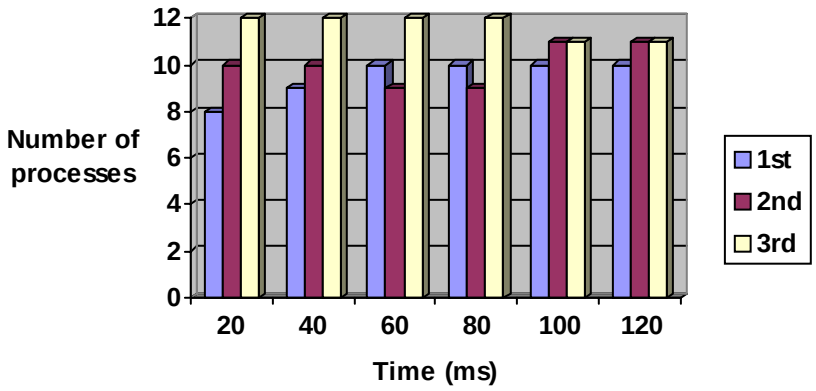
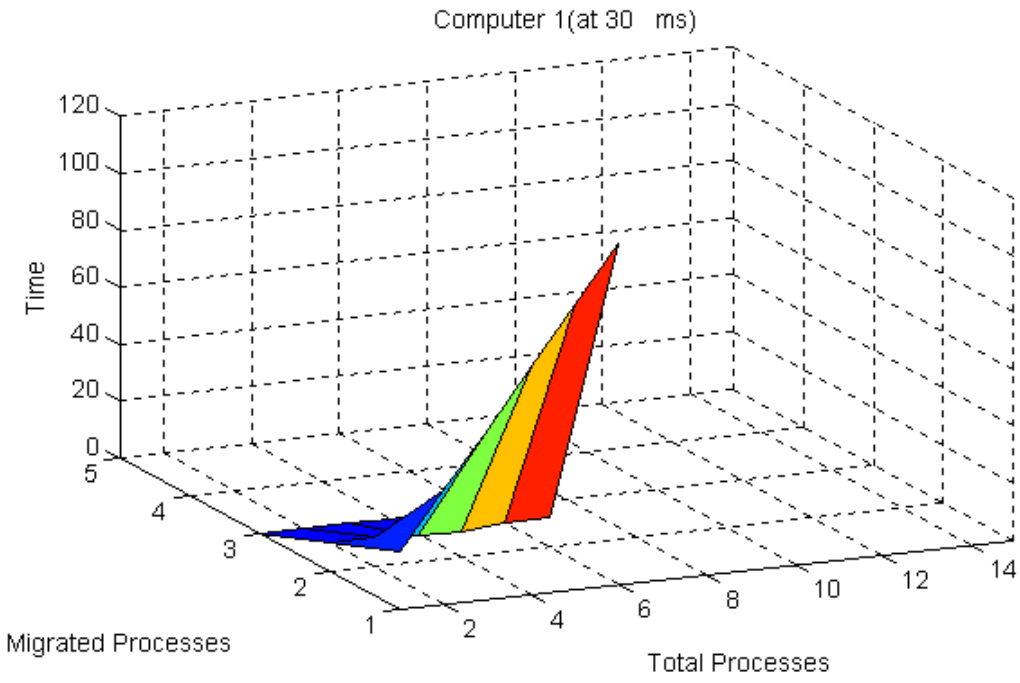
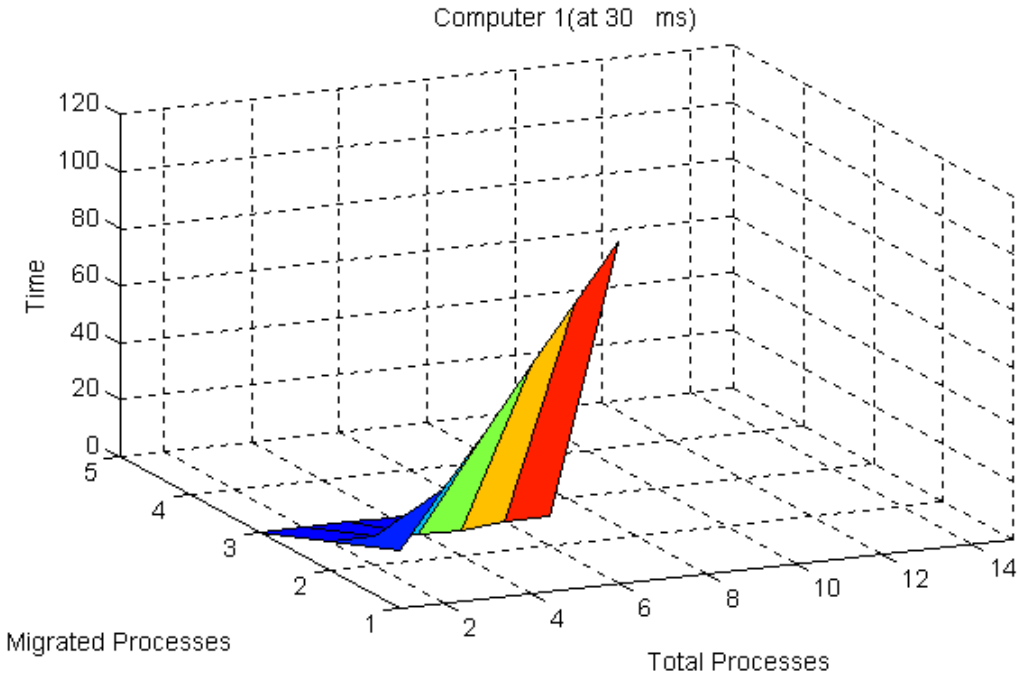
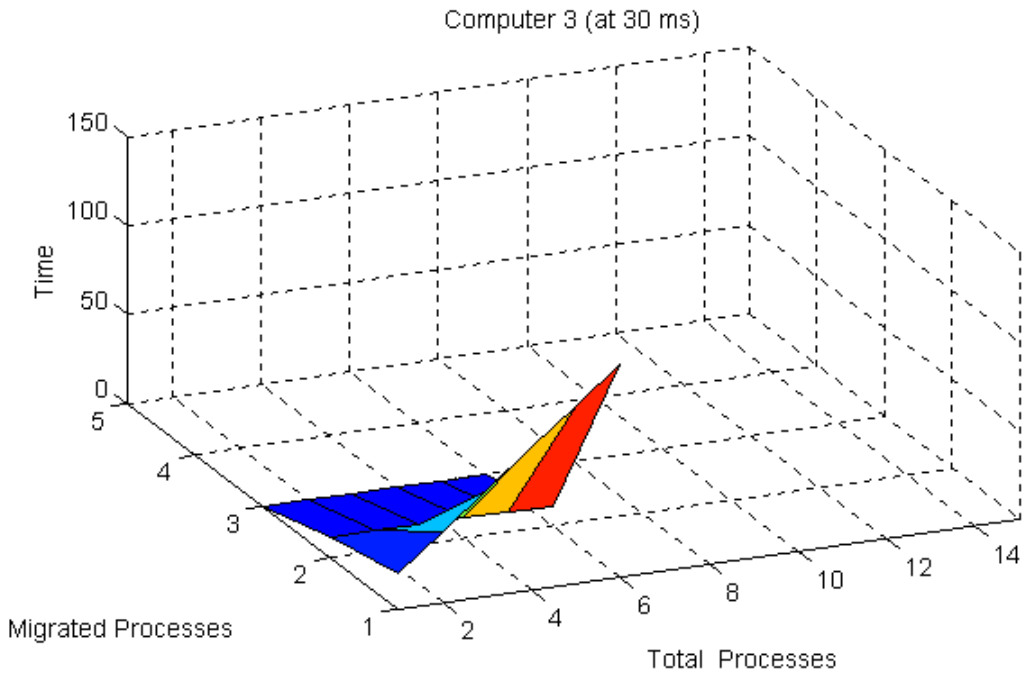


Figure 6.3: Balancer with 30 ms time period

The graph (in figure 6.3) shows the performance of the load balancer at the time period of 30 milliseconds. The first values at 20 millisecond show quite a difference in the loads of Computer 1 and Computer 3, but at 40 milliseconds after the first period of the balancer this difference is bridged a little as a process is shifted from Computer 3 to Computer 1. At 60 milliseconds, which completes the second time period of the balancer another process is shifted from Computer 3 to Computer 1. The load of Computer 3 did not change because more processes were initiated during the execution of the balancer. At 80 milliseconds there is no change in the load as the time period of the balancer is not complete, but at 100 milliseconds a process has been shifted. Thus a process was being shifted for every time period. At 100 milliseconds the loads of the computer were 10, 11 and 11, which meant that there would be no migration. At 120 milliseconds, two processes at computer 1 finished execution and changed the scenario of the load so that a process was again migrated in the next time period.


Computer 1(at 30 ms)
Time
120
100
80
60
40
20
0
5
4
3
2
1
Migrated Processes
2
4
6
8
10
12
14
Total Processes


Computer 1(at 30 ms)
Time
120
100
80
60
40
20
0
5
4
3
2
1
Migrated Processes
2
4
6
8
10
12
14
Total Processes



6.2.3 Time period at 45 milliseconds

The next experimental value for the time period of the load balancer was 45 milliseconds. This did show a difference in execution from the time period at 30 milliseconds as the re-calculation of the load took place after a bit of wait. Even though the balancer migrated a process in every time period but the gap between two migrations increased.

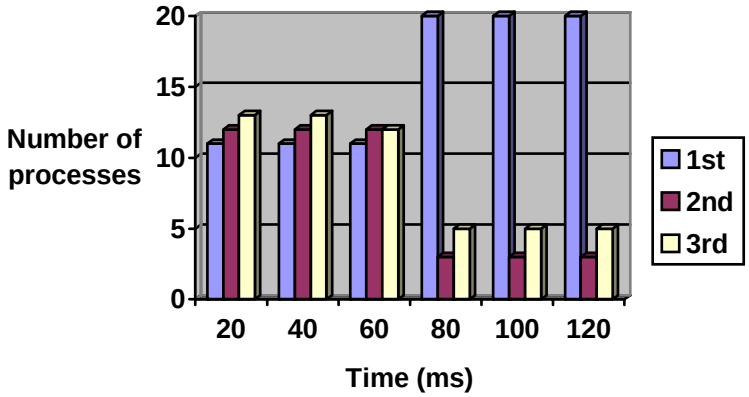
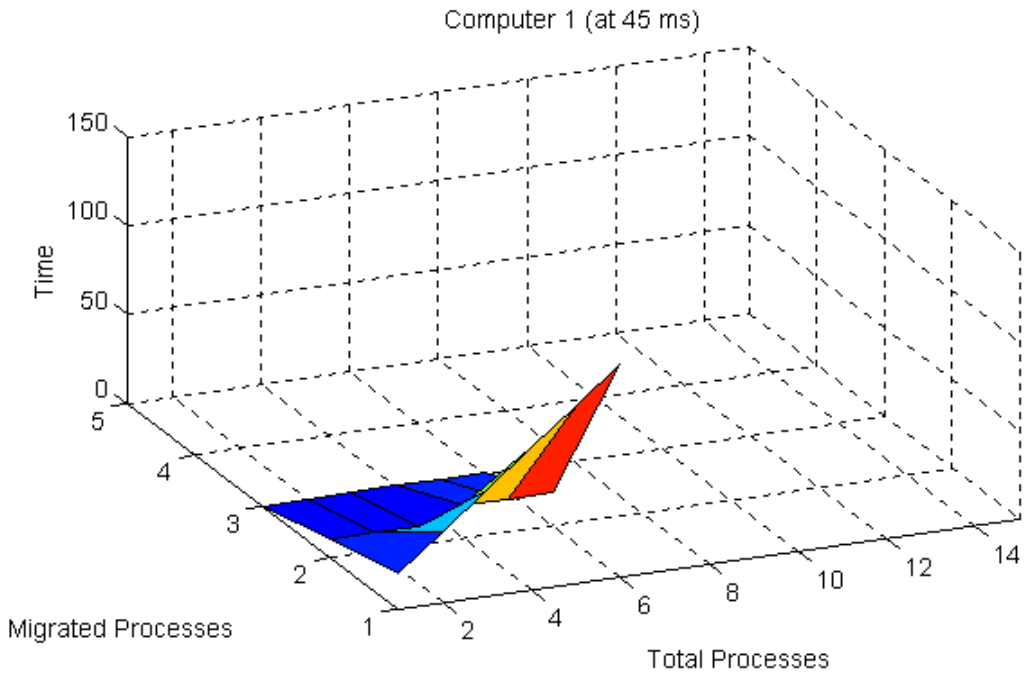
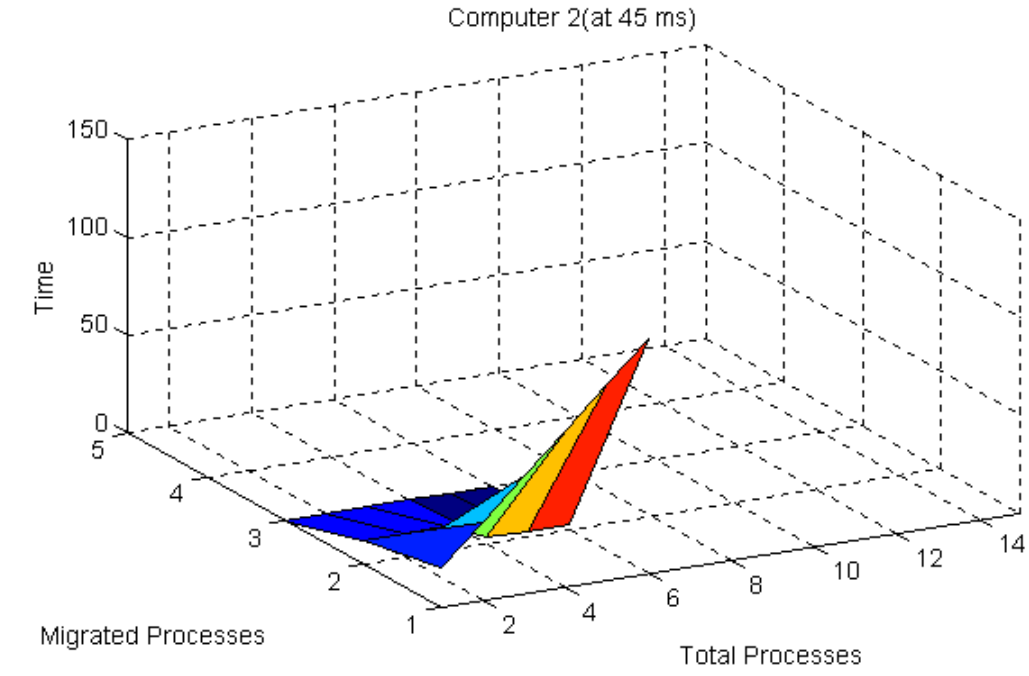
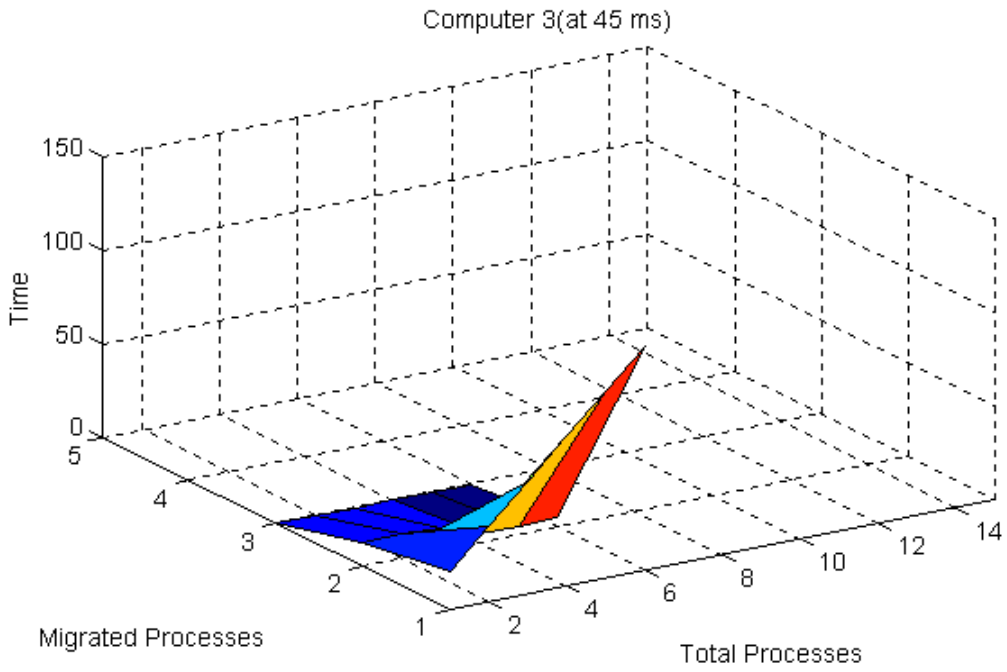


Figure 6.4: Balancer with 45 ms time period

The graph (in figure 6.4) shows the result of the balancer at 45 milliseconds. The first three tests show no change in the load, as the change in load due to migration is only visible when the load has been recalculated after the complete time period. No change in the graph is seen till 60 milliseconds when the next time period had started. When the second time period started, the loads of the three computers did not require any migration. But during that time period the load of Computer 2 changed drastically which the balancer could not detect till it started its next time period at 90 milliseconds. The balancer, thus, did not do anything in the second time period.


Computer 1 (at 45 ms)
Time
150
100
50
0
5
4
3
2
1
Migrated Processes
2
4
6
8
10
12
14
Total Processes


Computer 2(at 45 ms)
Time
150
100
50
0
5
4
3
2
1
Migrated Processes
2
4
6
8
10
12
14
Total Processes



6.2.4 Time period at 60 milliseconds

The next experimental value of the time period was at 60 milliseconds. This again guaranteed that a process would be shifted in every time period only if the states of the three computers was different, overloaded, underloaded or at threshold, at the beginning of the time period. It also did not cater for a sudden change in the load of a computer, which could occur during a time period.

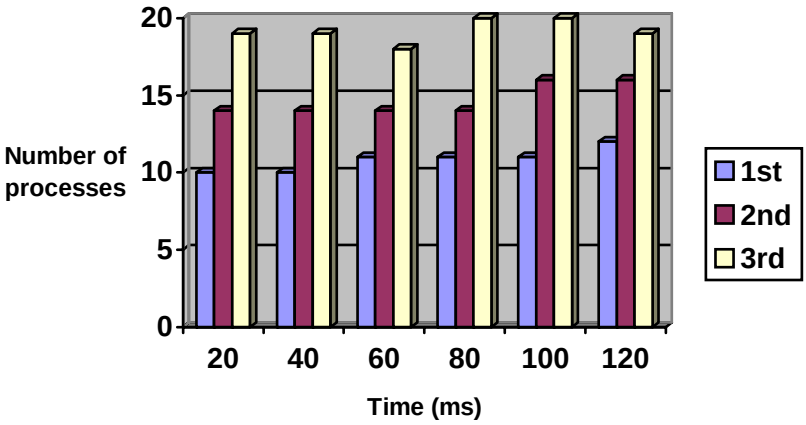
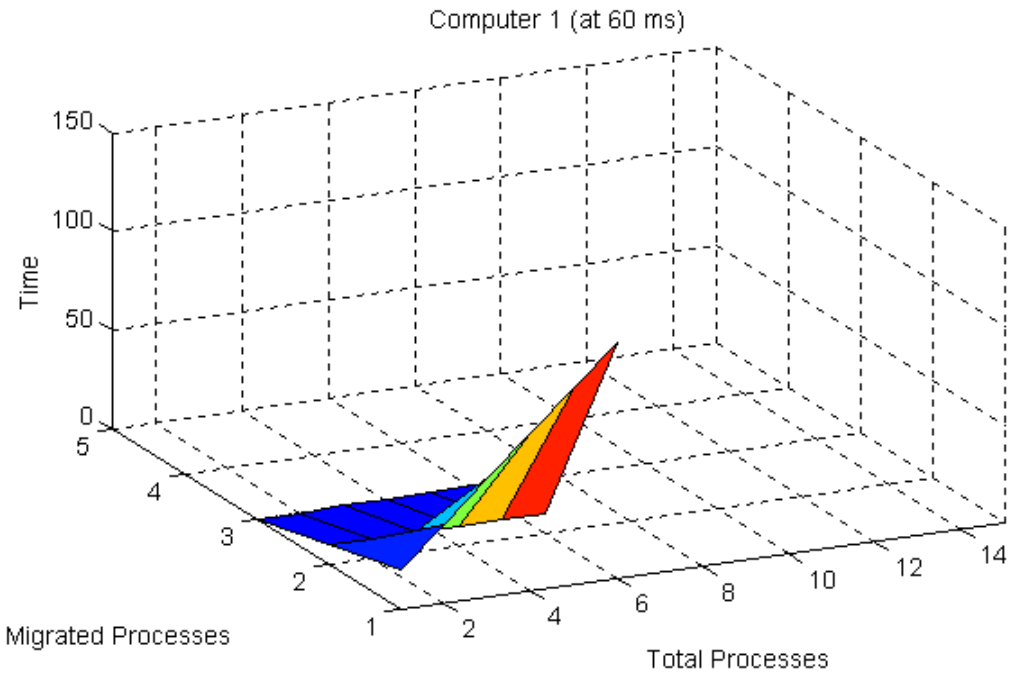
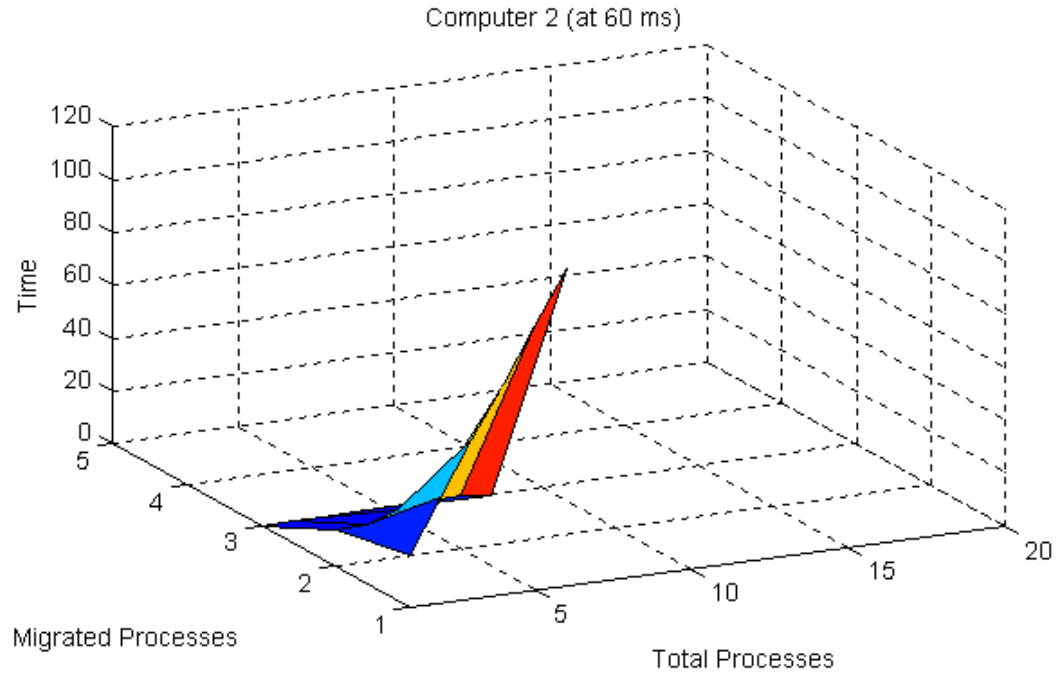
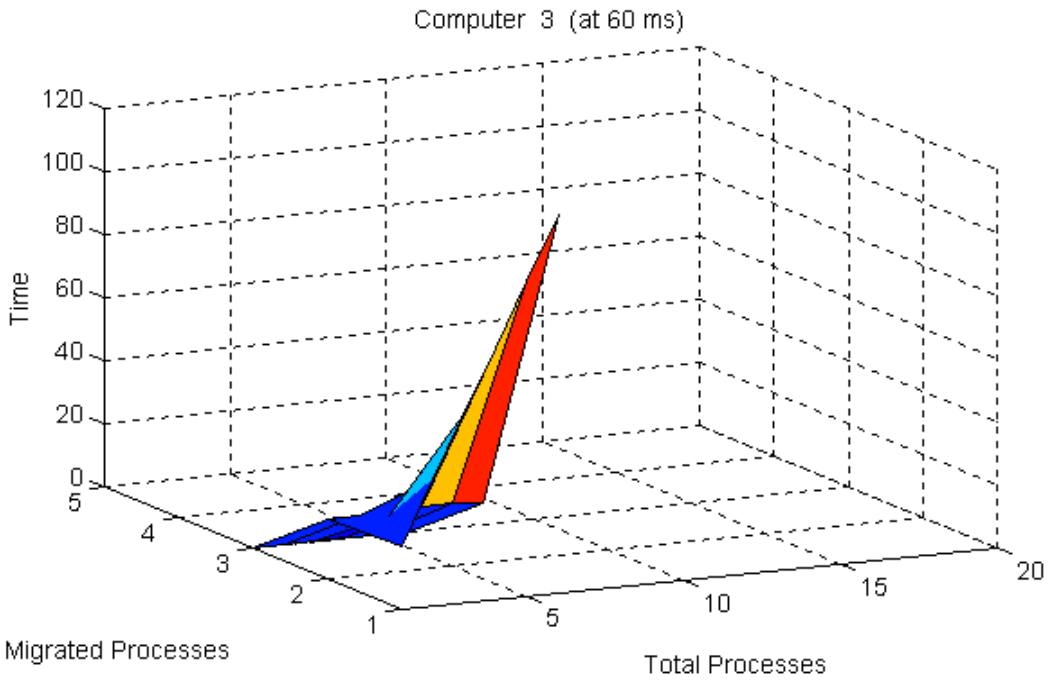


Figure 6.5: Balancer with 60 ms time period

The graph (in figure 6.5) shows the results of the load balancer with the 60 milliseconds time period. As was the case when the balancer had a time period of 45 milliseconds, the graph does not show any change in the loads till the 60 milliseconds mark, when the balancer finishes its first time period. The next period, which continues till 120 milliseconds, also migrated a process between computers 1 and 3, as the load of computer 3 was above the threshold value and computer 1 was below the threshold. The drawback for this balancer was that over a time of 120 milliseconds, only two processes were migrated when this time could have been used for more migrations.


Computer 1 (at 60 ms)
Time
150
100
50
0
5
4
3
2
1
Migrated Processes
2
4
6
8
10
12
14
Total Processes


Computer 2 (at 60 ms)
Time
120
100
80
60
40
20
0
5
4
3
2
1
Migrated Processes
5
10
15
20
Total Processes



6.2.5 Time period at 75 milliseconds

In another case, the time period was set at 75 milliseconds. The load balancer, once started, went through its sequence of events and migrated a process from the highest loaded to the most lightly loaded. When this was done, it waited for the next time period as it did for the time period of 45 milliseconds and 60 milliseconds. Till the start of the next time period, any change in the network could not be detected. This did not lead to any remarkable increase in the average job response time of the network. Also in some cases when the time period started, the load was evenly composed and there was no need of migration, however during the time period one of the processor became heavily loaded. The change in the load could not be communicated to the other computers and nothing could be done about it. Thus the long time period was not beneficial for load balancing.

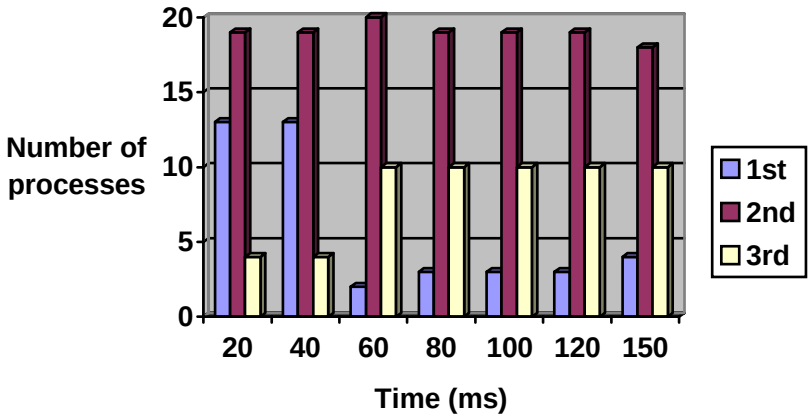
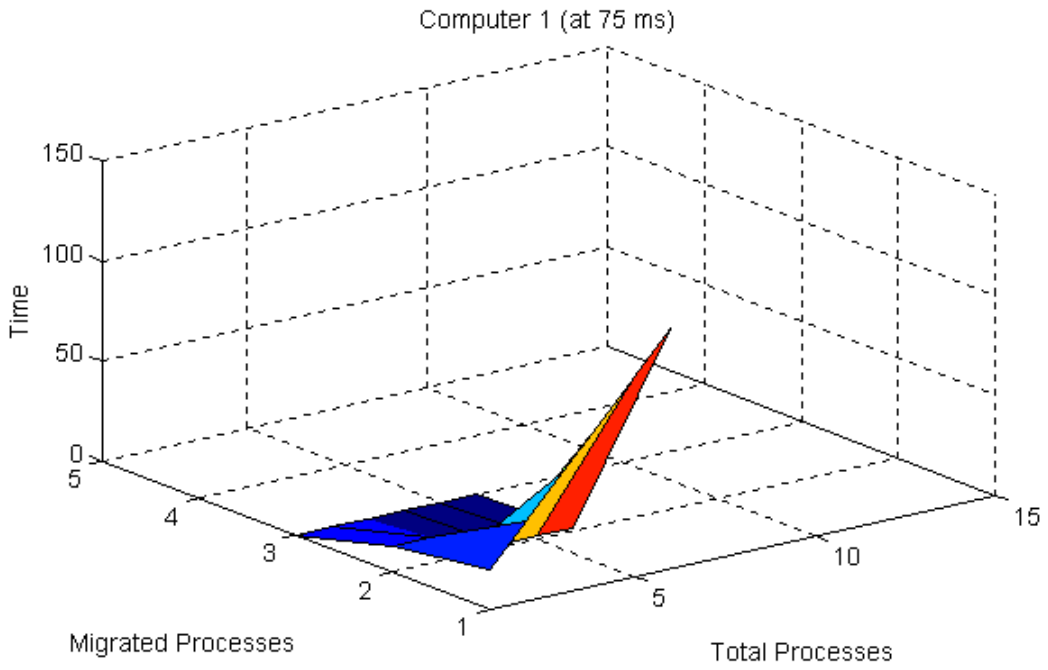
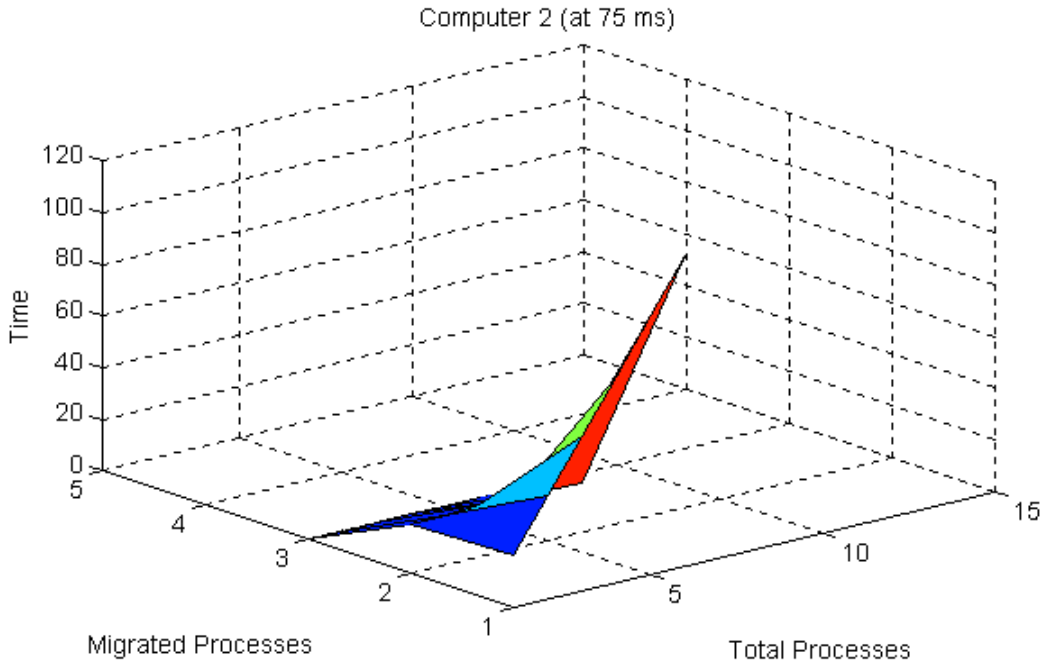
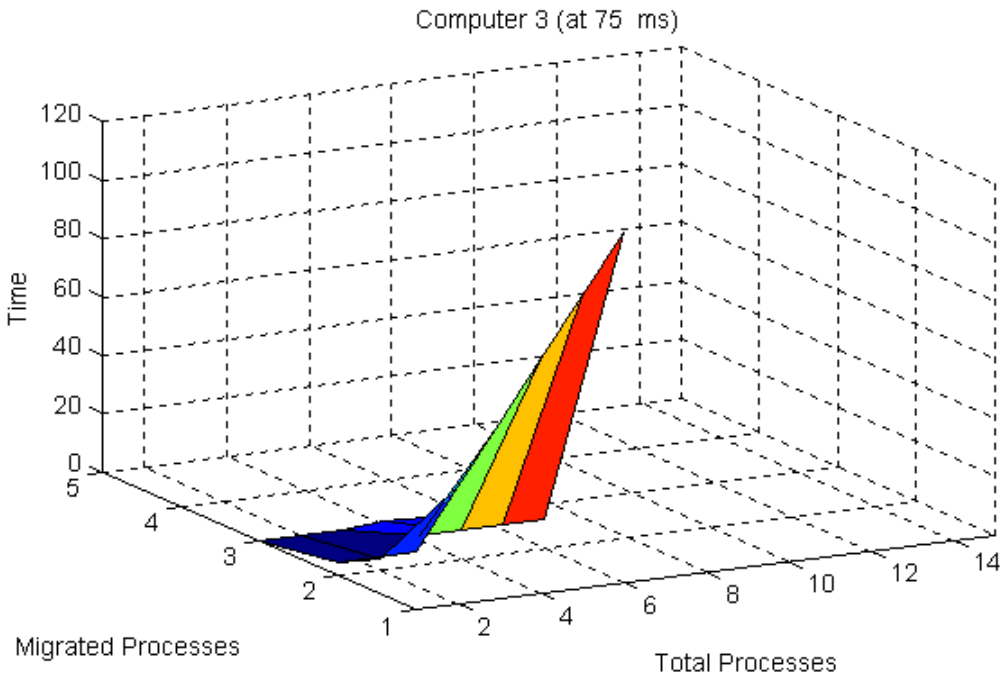


Figure 6.6: Balancer with 75 ms time period

The graph (in figure 6.6) shows the results of the balancer with the time period of 75 milliseconds. As can be seen from the graph the balancer does not change the load condition of the network for a long time. The process is shifted after 80 milliseconds and another one after 150 milliseconds. The effect of the balancing is thus cancelled by the time the balancer has to wait to re-start its loop of re-calculation and communication.


Computer 1 (at 75 ms)
Time
150
100
50
0
5
4
3
2
1
Migrated Processes
5
10
15
Total Processes


Computer 2 (at 75 ms)
Time
120
100
80
60
40
20
0
5
4
3
2
1
Migrated Processes
5
10
15
Total Processes



6.3 Value of Time period

Since the value of the time period of the balancer was very important, it was programmed after much deliberation. The results of all the above experiments were taken into account. The 15 milliseconds time period was not even considered, as it did not allow the balancer to completely migrate a process which at least was possible in all the other time periods. The other difference between the rest of the experiments was, the time the load balancer had to wait after migrating one process, till it could continue with the re-calculation of load for the next. The above experiments showed a definite difference in the load for the 30 milliseconds time period compared to all the rest. Consider each graph after 60 milliseconds; figure 6.3 shows that two processes have been migrated. Figure 6.4 shows that one process has been migrated. Figure 6.5 and 6.6 shows that there is no change in load of the three computers. Thus the maximum transfer of processes

was possible with a time period of 30 milliseconds. Even if the time period is cumulated, considering each graph after 120 milliseconds, the number of processes shifted by the balancer at 30 milliseconds will be the greatest. Thus the load balancer was implemented with a time period of 30 milliseconds.

A few parameters, which would effect the time period being used, are discussed below:

6.3.1 Job Size

One of the parameters on which the execution of the load balancer depends is the size of the processes being shifted. The experiments show that even though the load balancer did not produce desired results with a time period of 15ms, if the processes being shifted are limited to small sizes, the performance of the load balancer increases. The graph (in Figure 6.7) shows the results of the load balancer with a time period of 15 milliseconds, when the size of the processes was limited to below 10KB. The initial state of each computer is seen at 10 milliseconds, then at 30 milliseconds two periods of the load balancer have been executed and two processes have been shifted from computer 2 to computer 3. as the size of the processes is small the process is completely migrated within the same time.

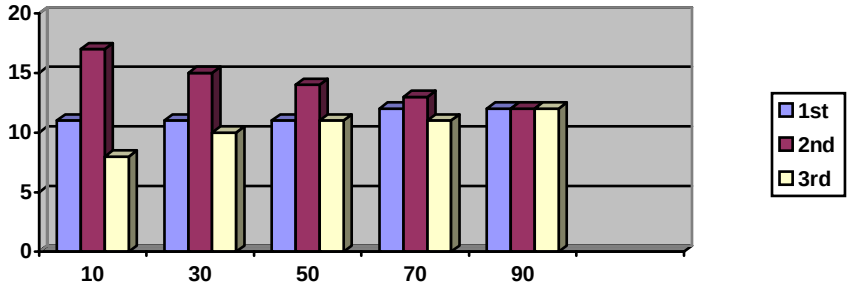


Figure 6.7: With job size less than10kb

Similarly, if the small size processes are used for migration in the load balancer with a time period of 30ms, the delay before the next migration becomes too long. This scenario can be seen with the load balancer having time period of 30 milliseconds (in figure 6.8). In this case the balancing effect is taking longer to be converge as till 30 milliseconds only a single process has been shifted. If this graph (figure 6.8) is compared to the previous one (figure 6.7), at 70 milliseconds (figure 6.7) shows a more balanced network than (figure 6.8). However, it must be remembered that this is only the case while using small processes.

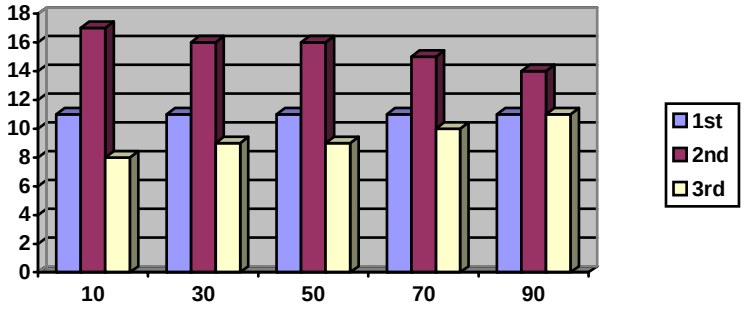


Figure 6.8: With smaller job size at 30 ms

On the other hand, if a processor only runs large size processes, the load balancer at 30 ms becomes inefficient for it and it produces better results with the balancer at 45ms. The graph (in figure 6.9) shows that the load balancer with a time period of 30 milliseconds takes nearly two of its periods to migrate a single process, if the size of the process exceeds 30KB. Thus the shifting of processes does not make complete use of the time period but wastes 60 milliseconds for one migration to take place. Now if the load balancer is run at 45 milliseconds using the same batch of processes, the difference can be observed (in figure 6.10).

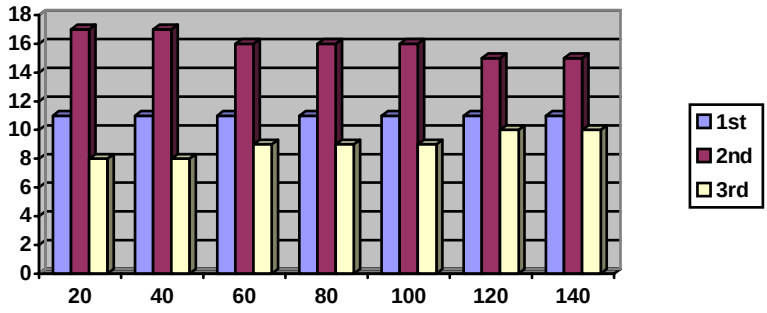


Figure 6.9 With larger job size at 30ms time period

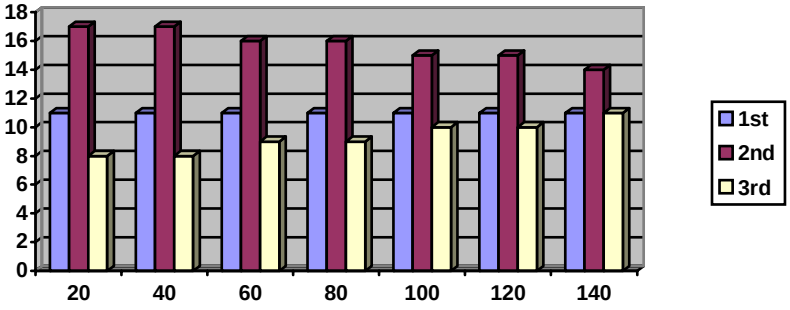


Figure 6.10 with 45ms time period

The graph (in figure 6.10) shows that the first change in load is visible at 60 milliseconds when a complete time period of the balancer has occurred. This is the same as in (figure 6.9) but the difference is that where with the time period of 30 ms the process was migrating in 60 milliseconds ie using two time periods of the balancer. This difference can also be seen when at 140 milliseconds, the graph (in figure 6.10) shows a more balanced state of the network than the graph (in figure 6.9). Therefore, the time period of the load balancer must be kept at an optimum value, or should be modified depending on the nature of work being done in the network.

6.3.2 Network Size

Even though the load balancer was tested for only three computers while calculating the value for the time period. It was observed that if the number of computers were increased in the network the time required for communicating load as well as for migrating processes would also increase. Thus again the time period of 30 ms will not remain optimum, it would need to be recalculated depending upon the number of computers in the network.

6.4 Problems Encountered

The implementation and execution of the load balancer in real time allowed such problems to be visible which were not imaginable during the design or initial planning of the balancer. This section will summarize some of the problems faced during the implementation of the load balancer.

One of the main problems was to synchronize all the computers when starting the load balancer. Since each step of the project was a prerequisite for the next step, until each computer had executed to one point in execution none of the computers could proceed further. Similarly, if the time period of the load balancer was not set with care, it also led to different times of execution on each computer causing any of the computers on the network to be stranded at a point. Selection of a process was another hurdle in the implementation. The design of the project did not allow the migration of such processes that required user input during execution this was a limitation in the project. Another problem with

the implementation was that if a process was shifted which produced or displayed results to the user at the end of its execution, these results could not be shifted back to the originating computer as moving back the results would create a communication overhead as well as create an imbalance in the balance the balancer was trying to achieve. When calculating the load on any computer, the parameter considered was the number of processes in the ready queue of the computer. This parameter did not give the actual utilization of the processor.

6.5 Further Recommendations

This implementation can be considered as a prologue to the subject of load balancing. This reflects on how the simulated results would behave in a real time environment. It is recommended that the limitations of this project be taken as future milestones to be accomplished. This would specially include that the load be calculated considering a number of parameters in addition to the number of processes. The number of processes running on a computer does not realistically report the utilization of a computer unlike stated previously that “in most multi-threaded systems, the length of the ready queue is a good indicator of processor load”. The implementation brought to light the fact that if one process running on a computer uses a lot of system resources whereas five processes running on another system use the same amount of resources. Thus, shifting a process to the computer with the single process running would be very expensive for its resources. Therefore, load balancing can be made more effective by including the system resources, the system utilization and such comprehensive parameters. The load balancer can also be enhanced to work on Windows NT 3.5 with a few changes in

implementation since the load balancer is only incompatible on Windows NT 3.5 because of a few APIs being used. It can also be implemented on a heterogeneous network of Windows 95 and UNIX.

SUMMARY

The Load Balancer is the implementation of the Periodic Symmetrically Initiated (PSI) algorithm. The algorithm was implemented and tested on a network of three computers. Each computer on the network kept a record of the loads on all computers on the network. It decided its state depending on the threshold value of the network and migrated or received a process accordingly. This loop was repeated periodically after a fixed time period. The loop of actions constituted the load balancer, which effectively shifts processes between two computers depending on their states in the network. This decreases the average time taken by a process to be executed on the network.

References

[1-1] K. Benmohammed-Mahieddine, P.M. Dew and M. Kara, *"A Periodic Symmetrically-Initiated Load Balancing Algorithm for Distributed Systems"*, School of Computer Studies, University of Leeds, UK.

[1-2] Songnian Zhou and Domenico Ferrari, "*An Experimental Study of Load Balancing Performance*", Report No. U B/CSD 87l336, January 1987, Computer Science Division (EECS), University of California, Berkeley, California 9204629

[1-3] Peter Sanders, "*Analysis of Random Polling Dynamic Load Balancing*", University of Karlsruhe, Karlsruhe (Germany)

[1-4] S.Muthukrishnan and Rajmohan Rajaman, "*An Adversarial Model for Distributed Dynamic Load balancing*", DIMACS Center, Rutgers University, Piscataway.

General References / Books Consulted

- Oestereich, B., "*Developing Software with UML",* Addison-Wesley, 1999.
- Tony T.Y. Suen and Johnny S.K. Wong, *"Efficient Task Migration Algorithm for Distributed Systems"*, IEEE Transactions on Parallel and Distributed Systems, Vol 3, No 4, July 1992.
- Hart, J. M. and Rosenberg, B., "*Client Server Computing for Technical Professionals",* Addison-Wesley, 1995.

- Platt, D. S., "*Windows 95 and NT Win32 API From Scratch",* Prentice Hall, 1996.
- Maruzzi, S., "*The Microsoft Windows 95 Developer's Guide",* Ziff-Davis Press, 1996.
- Cohen A. and Woodring M., "*Win32 Multithreaded Programming",* O'Reilly & Associates, 1998.
- Bonner, P., "*Network Programming with Windows Sockets",* Prentice Hall, 1996.

Appendix A

Appendix A-1: Load Communication for one thread

Position *is position of computer in the network*
LoopCount *is number of computers in network divided by two*

While 'i' is less than LoopCount
Connect to computer at (position – i)
Send load
Receive load
Increment I

Appendix A-2: Load Communication for second thread

Position *is position of computer in the network*
LoopCount *is number of computers in network divided by two*

While' i' is less than LoopCount
Connect to computer at position + i
Send load
Receive load
Increment 'i'

Appendix A-3: Determining whether most overloaded or not

count *is an index*
number *is the number of computers on the network*
found *is a flag having values TRUE or FALSE*
load*(number) is an array containing loads of computers on the network*
self load *is load of the computer*

```
while(count<(number-1))
{
  if load(count) is greater than self load

      found=TRUE;
      break;
else
      count++;
```

}
if found is not equal to TRUE, node is most over loaded

Appendix A-4: Determining whether most underloaded or not

***count** is an index*
***number** is the number of computers on the network*
***found** is a flag having values TRUE or FALSE*
***load**(number) is an array containing loads of computers on the network*
***self load** is load of the computer*

while(count<(number-1))
{
if load(count) is less than self load
found=TRUE;
else
count++;
}
if found is not equal to TRUE, node is most under loaded

Appendix A-5*:* Finding the most overloaded computer

***i** is an index*
***number** is the number of computers on the network*
***load**(number) is the load of all computers*
***count** will store the index of the computer*

while i is less than the number
{
high = load(0)
If high is less than load(i)
high = load(i)
count = i;
increment i
}
load(count) is the most over loaded