View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Scholarworks@GVSU

Grand Valley State University
ScholarWorks@GVSU

Masters Theses

Graduate Research and Creative Practice

9-29-2004

Determination of Effective Non-preemptive Load Balancing Policies

Murali Rajagopalan
Grand Valley State University

Follow this and additional works at: http://scholarworks.gvsu.edu/theses

Recommended Citation

Rajagopalan, Murali, "Determination of Effective Non-preemptive Load Balancing Policies" (2004). *Masters Theses*. 542.
http://scholarworks.gvsu.edu/theses/542

This Thesis is brought to you for free and open access by the Graduate Research and Creative Practice at ScholarWorks@GVSU. It has been accepted for inclusion in Masters Theses by an authorized administrator of ScholarWorks@GVSU. For more information, please contact scholarworks@gvsu.edu.

Determination of Effective Non-preemptive Load Balancing Policies

By

Murali Rajagopalan

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Computer Science & Information Systems

Grand Valley State University

2004

Committee Members:
Robert Adams, Greg Wolffe, Christian Trefftz

Date: September 29, 2004

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE OF CONTENTS

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGEMENTS

I would like to express my appreciation to Professor Robert Adams my thesis advisor. His initiative and enthusiasm in guidance at Grand Valley State University provides an excellent educational opportunity. In addition, I would like to thank Professor Greg Wolffe and Professor Christian Trefftz for agreeing to be on the thesis committee. Their valuable inputs shaped my efforts while working on the thesis.

Special thanks to my late father Senior Professor and Dean Dr.D.Rajagopalan, Pondicherry Central University, India, who encouraged me throughout my life in every step I took and who advised me to pursue this thesis work. None of this would have been possible without the support and encouragement of my wife Maya who has made many sacrifices that allowed me the time to work on my master's degree.

Faculty Support

Dr. Robert Adams – advisor Professor GVSU

Dr. Greg Wolffe – committee member Professor GVSU

Dr. Christian Trefftz – committee member Professor GVSU

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABSTRACT

We propose to analyze the effectiveness of various non-preemptive Load Balancing Policies that are available in Distributed Computing Systems. The result of the analysis demonstrates the usage of various policies and determining a policy that is effective in the given set of problems. It is not about proposing a new distributed algorithm or distributed system policy. However, it has implemented a prototype software lab comprising various policies to determine the effectiveness of existing load balancing policies. There are several programs implemented to test the lab and to generate data to analyze the effectiveness of the distributed policies.

The prototype software lab (EPLAB) that has been developed, as a part of this thesis is configurable to employ any kind of load balancing policies discussed in this documentation. Any program written in any language (in Linux operating system) can be tested in this lab and the resultant data can be collected to observe the effectiveness of the policies selected in the configuration file. So, this lab can be used as a pedagogical tool to show students how different program respond using different load balancing policies. Two programming solutions used EPLAB to generate data to analyze the effectiveness of the load balancing policies. Many other load balancing policies can be implemented and integrated into EPLAB system and the results of the programs that ran using EPLAB can be analyzed for identifying the efficiency of the policies.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF FIGURES

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF GRAPHS

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. INTRODUCTION

Traditionally, computing problems were solved using single computers. Then multi-processor systems were developed which allowed a problem to be subdivided and solved in parallel. However, the problem remained of contention of resources. The next proposition was developing distributed systems.

A *distributed system* is a collection of hosts interconnected by a communication network in which each host has its own processor, memory, and other peripherals.

Over the past decade, dynamic load sharing and balancing algorithms in distributed computing systems have been an active area of research [6]. Dynamic load balancing algorithms have been based on the following observation; in a network, it is likely to find idle hosts while there are jobs queuing for execution in other hosts. Hence, it would be advantageous to move jobs from heavily loaded hosts to idle or lightly loaded hosts. An early study by Livny and Melman confirmed this hypothesis [10]. Their work has shown that there is indeed a high probability of finding within a distributed system some hosts idle while others have jobs queuing for execution. Subsequently, these findings suggested that load balancing is likely to benefit job response times.

When solving a distributed problem using a distributed system a big problem is ensuring that work is divided evenly between all the hosts. In the worst case, given n hosts all the work will be assigned to 1 host while n-1 hosts remained idle. This situation is no better than a non-distributed approach. Every time a process is started, the completion time of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

every other process on the host is negatively affected (i.e., it will take a longer time to complete each process). This is due to the fact that processes are contending for resources like CPU, memory, disks, etc. Therefore, one major goal of distributed systems is evenly dividing work between available hosts.

A ***resource manager*** is responsible for scheduling processes on available hosts in a distributed system, similar to the way an operating system schedules processes on a single processor. However, a distributed resource manager is responsible for scheduling processes across multiple processors. Like an operating system scheduler, the resource manager attempts to ensure that the processes finish in the quickest possible time. However, a resource manager must contend with factors that a traditional operating system does not. This factor includes resource usage, response time, network congestion and scheduling overhead.

A load balancing algorithm attempts to balance system load across available nodes by transferring processes from heavily loaded nodes to lightly loaded nodes. The idea is that by moving a process to a host that is less busy it will be able to finish quicker than if it were left on the heavily loaded host.

For example, one node might be running a large compilation. If another compilation were started on that node, then compile time could be improved by moving the new compilation to a node that is currently doing nothing.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A resource manager has three primary responsibilities.

1. To decide if a node is heavily or lightly loaded.
2. To decide whether or not to migrate a process to another node.
3. To transfer a process from a heavily loaded node to a lightly loaded node.

In order to perform its tasks a resource manager must implement several policies like load estimation policy, process transfer policy, and state information exchange policy.

1.1 Load Estimation Policy

A Load Estimation Policy determines how the workload of nodes of the system will be estimated (i.e., is the node heavily or lightly loaded). Workload of a node is the utilization of the resources on the load such as memory, processor performance, etc. Many parameters have been proposed to estimate the workload of a node [1]:

1. Number of processes currently executing

   Some nodes may be running many processes at one time while many other nodes may be only bare minimum processes such as operating system specific processes. So, a node A running 10 processes is considered to be busy or highly loaded compared to a node B that is only running 3 processes. In this case, node B is capable of running one more process compared to node A.
2. Maximum memory available in the system and currently available memory

   Some nodes may have more memory than other nodes in a network. For example, newer nodes have much larger memory as the memory hardware components became

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cheaper. Also, newer systems are capable of handling more memory than most of the older systems. Some nodes may be running processes that consume less memory compared to some other nodes in the network, which may be running a different operating system, or processes that may consume more memory. For example, due to various factors, it takes much lesser memory to run Microsoft Visual studio development environment (IDE) compared to running IBM WebSphere developer studio environment.

3. Processor architecture and speed

   Some of the less expensive nodes may be hosting hardware whose processing speed may be lesser than a node that is running a better hardware. For example, nodes carrying Intel 16 bit processors may be running slower than nodes carrying 64 bit processors.

4. Processor utilization

   Due the differences in the process allocation methods within operating systems, some nodes may be under utilizing the processor.

A resource manager uses some or all of the above parameters while estimating the load of a node. For example, a node having the following parameters – 64 bit processor, 512 MB total memory, 300 MB available memory and running 5 processes is definitely capable of running one more process compared to another node in the same network having the following set of parameters – 64 bit processor, 512 MB total memory, 50 MB available memory and running 8 processes.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The workload of the node can vary from time to time. There are three ways in which the workload of all the nodes in the network can be monitored. This is explained in detail in the State Information Exchange Policy (1.3).

1.2 Process Transfer Policy

A Process Transfer Policy determines whether to transfer a process to a different node or leave it in the current node. Most of the load-balancing algorithms use the *threshold policy* to make this decision [2]. The threshold value of a node is the maximum desirable value of its workload. If a node's load is less than the threshold, then the resource manager is free to move processes to that node. If a node's load is above its threshold, the resource manager attempts to move processes to another node to alleviate the load.

1.2.1 Static and Dynamic Policies

Process transfer policies can be categorized into two camps [2]:

1.**Static policy**: Each node has a predefined threshold value defined when the node boots. Furthermore, the threshold does not change over the lifetime of the node.

2. **Dynamic policy**: A node's threshold value is calculated as a function of the average workload of all the nodes and a predefined constant. More importantly, the node's load parameters (used to calculate the threshold value) are updated periodically by having the nodes exchange state information.

3. **Adaptive policy:** This can be considered as a combination of both the static and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

dynamic policies. When the system is started, static policy is used. As the time progresses, more nodes may join the network and more processes may be running in these nodes. At this point, Dynamic policy can be employed which can calculate the workload of all or part of the nodes and updated.

1.2.2 Single and Double Threshold Policies

Most load-balancing algorithms use a single threshold policy [3] and thus have only two states: overloaded or under-loaded (Figure 1). With a single-threshold policy, a node accepts new processes only if its load is below the threshold value, and tries to offload processes if its load is above the threshold value. The use of single-threshold value may lead to thrashing because a node's load may be below the threshold when it decides to accept a process, but then the load increases above the threshold as soon as the process arrives. Therefore, immediately after receiving the new process, the node will again try to transfer one or more of its processes to some other node.

Alonso and Cova [3] state that:

- A node should only transfer one or more of its processes to another node that is lightly loaded if such a transfer greatly improves the performance of the rest of its local processes.
- A node should accept remote processes only if its load is such that the added workload of processing does not significantly affect the service to the local ones.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To reduce the instability of the single-threshold policy and to take care of these two notions, Alonso and Cova proposed a double-threshold policy called the *high-low policy* (Figure 2).

Figure 1, 2: Single & Double Threshold Policies

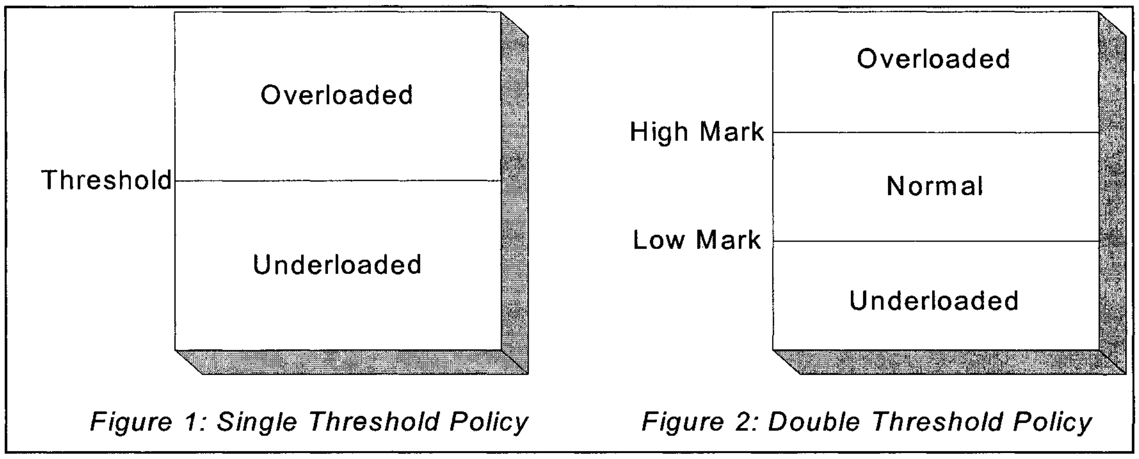


Figure 1: Single Threshold Policy *Figure 2: Double Threshold Policy*

The high-low policy uses two threshold values called the *high mark* and the *low mark,* which divide the space of possible load states into three regions:

- Overloaded – above the high-mark and low-mark values
- Normal – above the low-mark value and below the high-mark value
- Under-loaded – below both the high-mark and low-mark values

A node's load state switches dynamically from one region to another. Depending on the current load status of a node, the decision to transfer a local process or to accept a remote process is based on the following rules:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- When the load of the node is in the overloaded region, new local processes are sent to remote nodes that are lightly loaded and requests to accept remote processes are rejected.
- When the load of the node is in the normal region, new local processes run locally and requests to accept remote processes are rejected.
- When the load of the node is in the under-loaded region, new local processes run locally and requests to accept remote processes are accepted.

1.2.3 Preemptive and non-preemptive process migration policies

A very simple mechanism to achieve system-wide utilization of available resources within a distributed system is Initial Placement or Remote Execution. Remote execution creates the particular process on a remote node prior to execution. Once such a process has started to execute, it can or cannot be stopped and sent to another node to continue its execution. If the process is stopped and sent to another node to continue its execution, it is called Preemptive process migration.

Preemptive Process Migration dynamically relocates a running process to another node in a distributed system after initiating execution on the source node [11]. The amount of information that has to be transferred depends on the employed migration algorithm. Usually it is larger compared to remote execution because the information can consist of the entire process environment including the process' address space. One advantage of preemptive process migration is that after a process has begun execution, the changes of the load of system can be estimated so that load balancing policies can make more

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

efficient decisions. But for the sake of simplicity, this thesis paper deals only with non-preemptive process executions.

1.3 State Information Exchange Policy

The State Information Exchange Policy determines how often host load information is received by the resource manager. The resource manager on each host requires frequent exchange of state information. Many exchange policies have been proposed and are summarized below [2]:

- Periodic Transmission

  State information is transmitted every n units of time

- Transmit when state changes

  State information is transmitted if the state of the node changes (from normal to over-loaded or under-loaded)

- Exchange by Polling

  State information of a node is transmitted only if another node polls for it

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.4 Other Distributed Systems and their load balancing policies

LSF – Load Sharing Facility

Load Sharing Facility from Platform (http://www.platform.com) is a suite of application resource management products that schedule, monitor and analyze the workload for a network of computers.

LSF is distributed load sharing and batch queuing software that transforms the network into a shared computing resource by providing a transparent, single view of all hardware and software resources, regardless of platform differences. LSF works for both interactive and batch jobs, plus it contains support for parallel packages such as PVM and HPF. LSF version 2.0 supports all major UNIX platforms, including DEC OSF/1, ULTRIX, IBM AIX on RS 6000 systems, HP-UX on HP 9000 systems, SGI IRIX, SunOS and Solaris on Sun SPARC stations, and Convex OS on C-series supercomputers.

LSF consists of two servers running on each host, a runtime load sharing library, and load sharing applications built on top of the library.

Load Information Manager (LIM): the policy server of the LSF. Each LIM periodically exchanges load information such as CPU, memory, disk I/O, the number of login sessions with other LIMs.

Remote Execution Server (RES): the mechanisms for transparent remote execution of tasks. The RES accepts remote execution requests for all load sharing applications.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Load-sharing Library (LSLIB): provides a procedural interface between load-sharing applications and LIMs and RESes. Users of LSF can develop load sharing applications using LSLIB.

Message Library (MSGLIB): provides efficient message passing and synchronization for parallel applications (under development).

PANTS Application Node Transparency System

PANTS system is developed by Mark Claypool and David n Finkel [13] at the Department of Computer Science, Worchester Polytechnic Institute, Worchester, MA USA. PANTS is an implementation of Cluster computing on Beowulf clusters that enables transparent distributed computing. PANTS employs a fault-tolerant communication architecture based on multicast communication that minimizes load on busy cluster nodes. Load balancing algorithm used in PANTS is a variation of the multi-leader load-balancing algorithm proposed by Wills and Fenkel.

In this algorithm one of the nodes is required to be the leader. The leader can be any node in the cluster and is chosen randomly from among other nodes. The leader has three basic responsibilities – accept information from a lightly loaded node in the cluster, use that information to maintain a list of available nodes, and return an available node to any client that requests it. An available node is one that is lightly loaded as measured by CPU Utilization. The actual implementation is a variation of the multi-leader policy described in Wills and Fenkel [13] and implemented in Moyer [12].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. POLICY COMPARISON

Each time a new kind of policy is proposed authors present evidence that their policy is better than the policies that existed before. However, testing a policy in isolation is not possible. The total effectiveness of the system depends on all the other policies in effect as well as their interaction.

Given the large number of policies that have been proposed, it is natural to ask which one is the most effective. The purpose of this thesis is to implement a lab environment to determine which combination of load balancing policies is most effective in balancing the load in a distributed system.

Testing policy effectiveness can be done in one of two ways. First is through theoretical analysis using graphs and queuing theory. Unfortunately, this quickly becomes cumbersome, when more and more factors such as operating system policies, machine load, network parameters, etc. are taken into account. Several factors need to be assumed to be ideal when dealing with theoretical analysis.

A second alternative is to do simulation. Using a simulation, load-balancing policies can be tested under "real world" conditions using real world problem, not abstract theoretical conditions using theoretical problems.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3. EPLAB - EFFECTIVE POLICY LAB

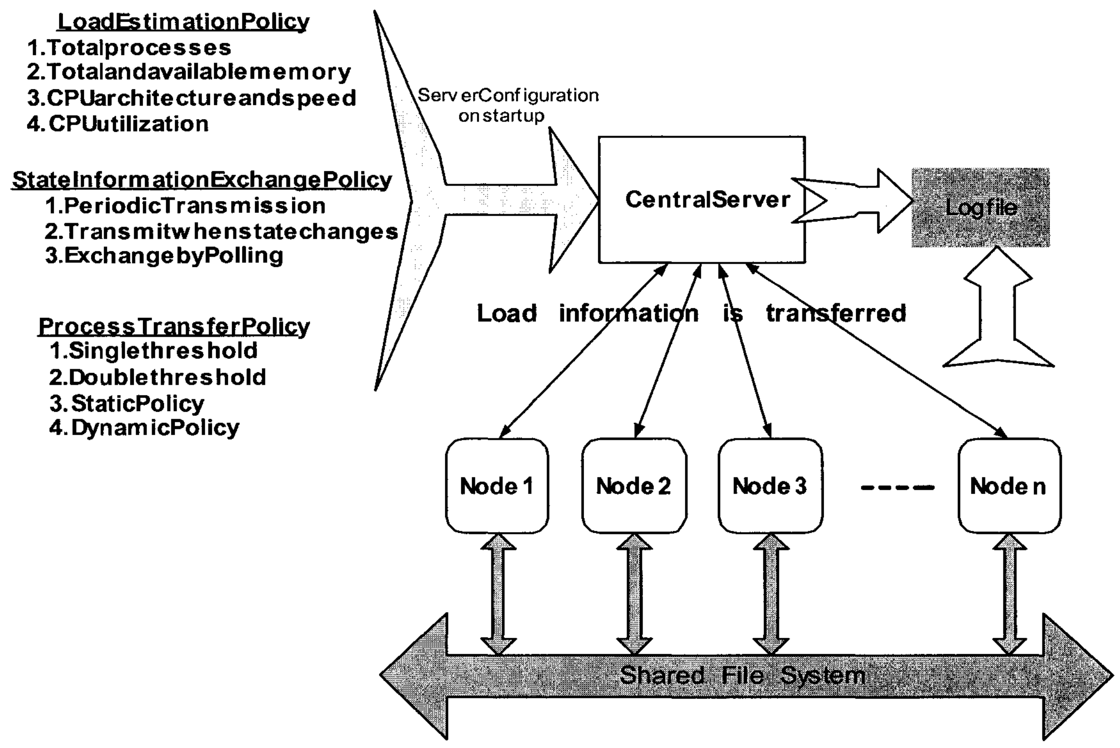
3.1 Overview

The *EPLAB system (Figure 3)* is a modular load balancing laboratory that enables a user to plug-and-play load balancing policies and to monitor their effectiveness. The objective is to provide a framework where one can experiment with combinations of new and existing load balancing policies in order to determine which combination is most effective.

Because of time contraints, the EPLAB system only considers non-preemptive load balancing polices. That is, policies that do not stop a running process, transfer it to another host, and then restart it. In the EPLAB system, a target host is chosen before new processes are started.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure 2: EPLAB System



The EPLAB system has a central server and several distributed client nodes. The central server is responsible for

1. determining which policies will be used during the current run of the simulation
2. sending policy information to clients
3. receiving load messages from clients and
4. making the decision where new processes should be started. This is decided centrally by employing a Process Transfer Policy.

When a node wants to start a new process, it sends a request to the server. The server calculates the load of all the participating nodes, determines the node best able to handle

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

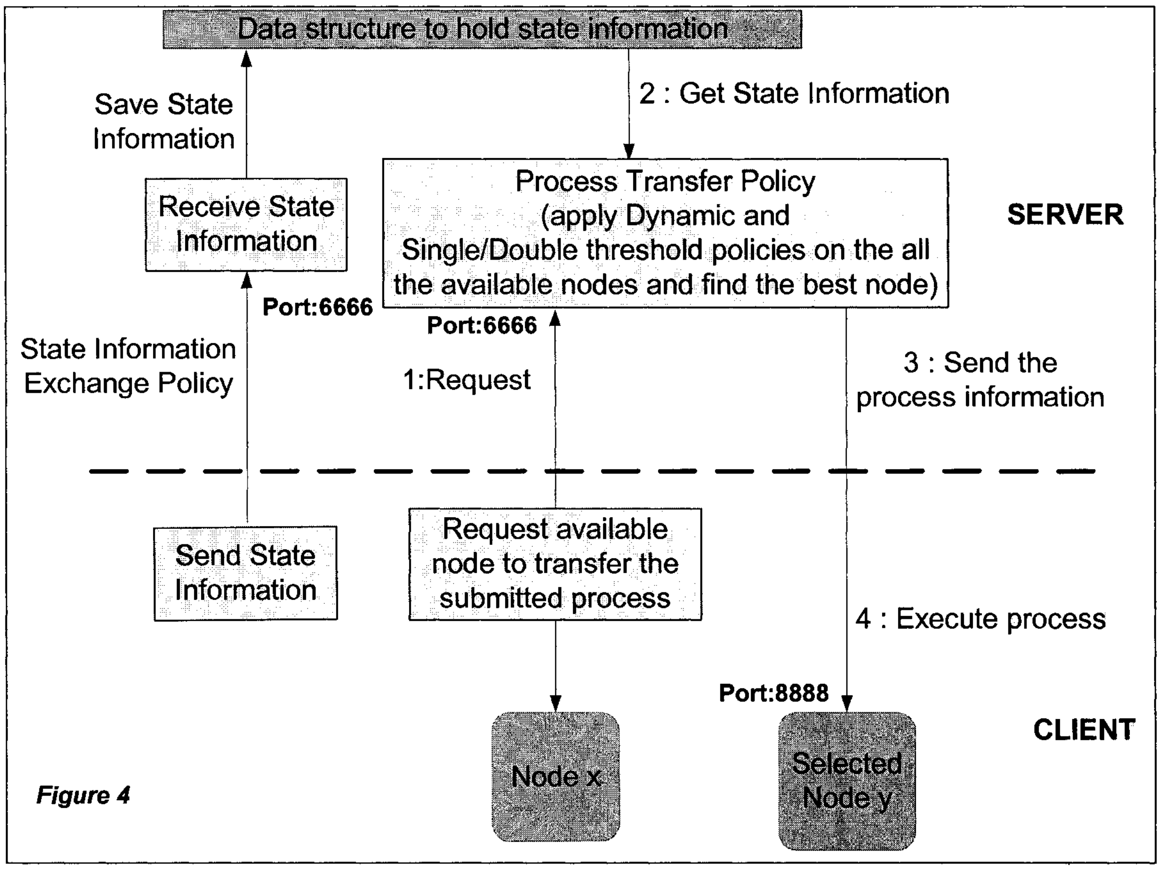
the new process and sends information about the process to the destination node. The destination node executes the process. Client nodes are responsible for accepting requests for starting new processes from the server and of sending their load to the server.

EPLAB server periodically writes the load information it receives from the EPLAB client and the process transfer information to log files. For simplicity, the server and all client nodes share a single file system.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.1 EPLAB System Design & Implementation

Figure 3: EPLAB System Design & Implementation


Data structure to hold state information
Save State Information
2 : Get State Information
Receive State Information
Process Transfer Policy (apply Dynamic and Single/Double threshold policies on the all the available nodes and find the best node)
SERVER
Port:6666
Port:6666
State Information Exchange Policy
1:Request
3 : Send the process information
Send State Information
Request available node to transfer the submitted process
4 : Execute process
Port:8888
CLIENT
Figure 4
Node x
Selected Node y

When the central server *(EPLABServer)* is started, it reads a configuration file *(EPLAB.ini)* that defines which policies should be in effect. After the server is initialized it looks at a well-known port for incoming messages from other nodes.

After the client nodes *(EBLABClient)* are loaded and initialized, they send their load information to the server based on current the State Information Exchange Policy.

The server accepts two kinds of messages from clients: "DATA" messages containing load information from a node and "JOB" messages containing the pathname of the process to be executed in one of the client nodes. Then the server scans the list of node

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

state information and applies the transfer policy to identify which node is relatively free to execute the process. Finally, the server transmits the JOB message to the selected node. Upon receiving a job message a client will execute the process specified in the pathname.

3.1.1 Process Creation

The programs (or jobs) can be submitted using *EPLABSubmit* program that sends a JOB message to the EPLABServer (Figure 5). JOB messages can have either the name of the process to be executed or the name of a text file that has all the processes to be executed sequentially.

For example, a job text file will look like

#comment: This job file solves distributed matrix multiplication

./DisMatMul Results.out 100 200

./DisMatMul Results.out 200 300

./DisMatMul Results.out 400 500

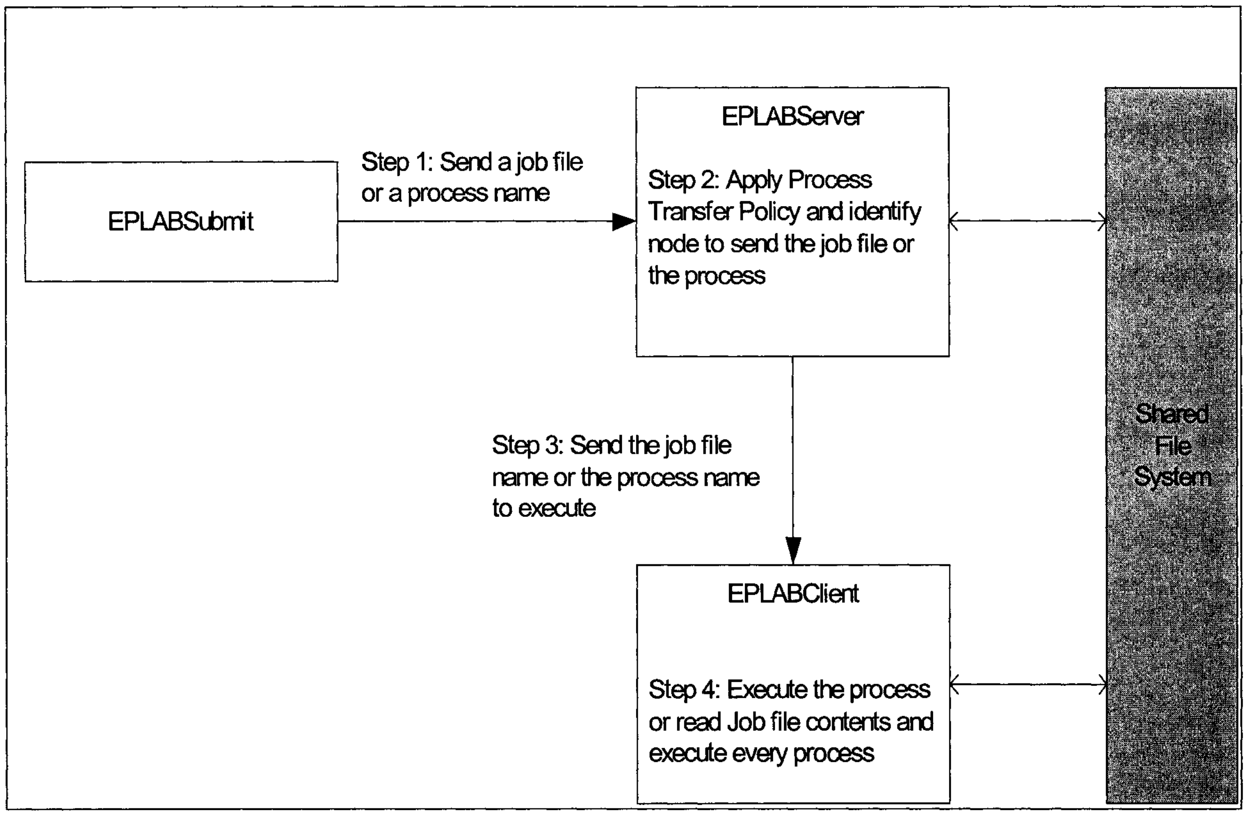
./DisMatMul Results.out 600 700

In the above examples, "DisMatMul" represents the name of the program, "Results.out" is the name of the output file and the next two numbers are the command line arguments that are passed to the program when it starts executing. Each of the line in this JOB text file represents an instance of a program (process).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

If the message submitted to EPLABServer is a JOB message with a Job text file, then it reads the content of the job file and allocates each process in the text file to a client node based on the threshold value. Then the process name is sent to *EPLABClient* on the selected node and it gets executed.

Figure 4: Job Submission Process


EPLABSubmit
Step 1: Send a job file or a process name
EPLABServer
Step 2: Apply Process Transfer Policy and identify node to send the job file or the process
Step 3: Send the job file name or the process name to execute
EPLABClient
Step 4: Execute the process or read Job file contents and execute every process
Shared File System

If jobs are submitted as job text files, then the job arrival rate at the EPLAB system is Poisson. Single instances of programs can also be requested to run the EPLAB system. So in that case, the job arrival is more interactive by nature.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.1.2 Process Logging

The purpose of load balancing is improving performance by reducing the completion time of the jobs. Such an algorithm that completes the assigned job in least time is considered to be highly effective.

EPLAB Server and Client programs write information to the log file every action they take to process the submitted jobs like starting and stopping a job. Each of these message in the log file is timestamped.

Log file example :

```
06/22/2003 07:00:10 SERVER: Received node 2 state information

06/22/2003 07:00:11 SERVER: Received node 3 state information

06/22/2003 07:00:12 NODE 2: Started job ./DisMatMul Results.out 100 200

06/22/2003 07:00:13 NODE 3: Started job ./DisMatMul Results.out 300 400

06/22/2003 07:00:15 NODE 4: Started job ./DisMatMul Results.out 500 700

06/22/2003 07:00:16 SERVER: Received node 5 state information

06/22/2003 07:00:18 NODE 2: Started job ./DisMatMul Results.out 800 900

                          ::::: log file contents continued ::::

06/22/2003 07:02:05 NODE 2: Completed job ./DisMatMul Results.out 100 200

06/22/2003 07:02:15 NODE 3: Completed job ./DisMatMul Results.out 300 400
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Analysis of the log files will indicate the effectiveness of the policies being used in the EPLAB system. If the log files has information that only one client is involved in solving all the submitted process then it may indicate that either the processes completed quickly or the Process Transfer Policy is not effective to solve the kind of problem at hand.

3.1.3 System Implementation

EPLAB system is developed using C programming language on Linux operating system. The communication between the EPLAB server and the clients are done using TCP/IP sockets. The functional behaviour of the system can be explained in the following steps.

Step 1: Start the EPLAB server

When the server is started, it reads the initialization file that contains the system parameters. These parameters will guide the entire system.

Example of an initialization file:

```
[CENTRAL SERVER]

[Load Estimation Policy]

CPU

TotalMemory

AvailableMemory


[State Information Exchange Policy]
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Periodic=120


[Process Transfer Policy]

SingleThreshold

Dynamic
```

Step 2: Start the EPLAB clients

The client hosts get the system parameters from the central server and initialize their properties.

Step 3: Submit job to execute at the hosts

When a process is submitted at a client node, it requests the central server for the best available node to transfer the process. The central node applies Process Transfer Policy and decides the best node to transfer the process to and sends the destination node address to the requesting node. If the destination node is not the requesting node, then the process is transferred to the destination node.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. POLICY EXPERIMENTS & RESULTS

In order to analyze the effectiveness of the policies defined in the EPLAB system, a programmatic solution to several problems were developed and executed several times by changing the policies and the parameters of the EPLAB system.

Collecting the load information of a node does not consume much time, and also it was necessary to collect all the information to compute Process Transfer Policy. So, the Load Estimation Policy remained the same throughout the testing and analysis phase of the implementation. Every node would collect its total memory, free memory and the number of processs running as a part of Load Estimation Policy.

Algorithm To Calculate Single Threshold:

```
For Each Client
     Balance_Memory  = Total_Memory - Free_Memory
     Average_Per_Process_Memory = Balance_Memory  /
Total_Processes
     Client_Threshold = Total_Memory /
Average_Per_Process_Memory
     If Client_Threshold  < ALLOWED_THRESHOLD Then
          Use This Client
     End If
Next client
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Double Threshold Calculation:

```
For Each Client
     Balance_Memory  = Total_Memory - Free_Memory
     Average_Per_Process_Memory = Balance_Memory
                                  / Total_Processes
     Client_Threshold = Total_Memory /
                             Average_Per_Process_Memory

     If Client_Threshold  < LOW_VALUE Then
          Add Client and Client_Threshold To
                         LOW_VALUE_LIST
     Else If Client_Threshold  < HIGH_VALUE Then
          Add Client and Client_Threshold To
                         HIGH_VALUE_LIST
     End If
Next Client

If there are clients in LOW_VALUE_LIST then,
     Lowest Client_Threshold in the LOW_VALUE_LIST
     and use the Client
Else If there are clients in HIGH_VALUE_LIST then,
     Lowest Client_Threshold in the HIGH_VALUE_LIST
     and use the Client
End If
```

Also, to implement the policy of transmitting the state information when the load changes, we need to implement the threshold calculations in the EPLABClient code also. For this implementation, EPLAB considers only Periodic transmission of state information. According to some of the studies, there is a very little impact of using various State Information Exchange Policies on the performance of the distributed problems [3, 7, 10].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.1 Problem 1: Finding average, maximum and minimum value of every row in a large matrix

Problem Definition: Write a program to calculate the average, maximum and minimum of all the elements of every row of a large matrix.

Program Implementation: The program is designed to read all the columns of one row of a matrix and calculate the average, maximum and minimum of the elements and write the results to a text file. The program assumes that the input matrix is represented as a text file of rows delimited by linefeeds and columns delimited by spaces. The name of the matrix file and the output file and row to operate on are passed as command line parameters.

For example: The command **ProcessMatrixRow Matrix.in Results.out 25** will read the 25th row of Matrix.in file, calculate the average, maximum and the minimum and write the results to Results.out file.

After programming this solution, a JOB file called JOB.Matrix was generated. This job file contained sequence of calls to the program to get the results of various rows of the matrix. Then, the ESPSubmit program was used to send JOB.Matrix to ESPServer which then read the contents of the JOB file one line at a time and found out the best possible node (applying Process Transfer Policy) to execute the process and sent the processes to the nodes to execute it. A log file was generated as processes were dispatched to nodes.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

These jobs are homogeneous in nature and they are located in a shared file system. So the time to create these jobs are constant. There were no contention for the job files or the data files.

The following figures contain the summary of the log files after each run of the program.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Test: 1.1

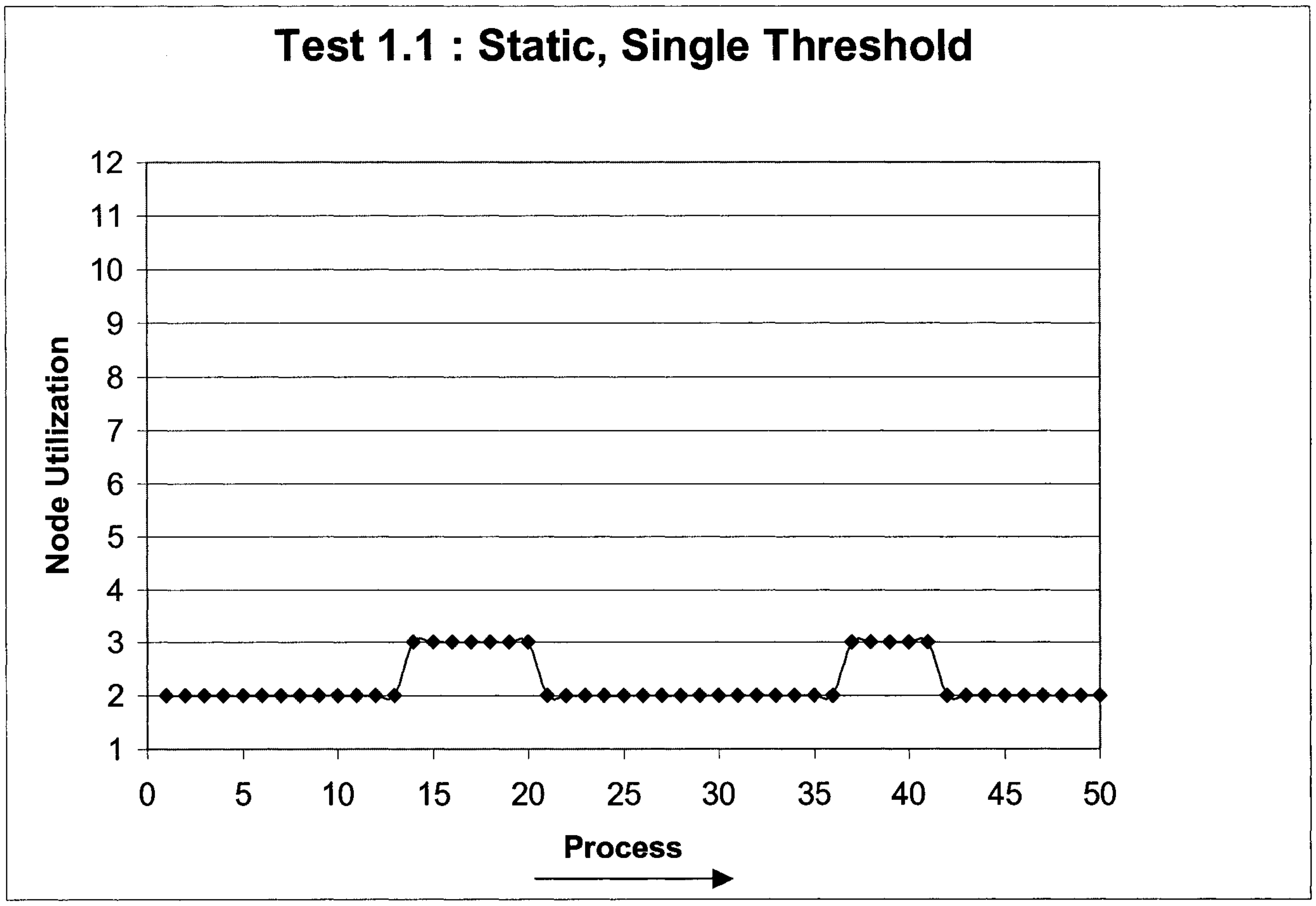
State Information Transfer Policy: Periodic (5 seconds)

Process Transfer Policy: Static, Single Threshold

EPLAB Client Nodes: 2, 3, 4, 5, 6, 7, 8, 9, and 10

EPLAB Server running at Node 1.

Graph 1.1 Static, Single Threshold Test 1.1



Total Running Time: 8 minutes

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Analysis:

From the graph (1.1), it is obvious that the system did not make use of all the available nodes. This is due to the fact that the threshold of nodes 2 and 3 were always less than the maximum threshold. Every time the threshold of all the nodes was calculated, nodes 2 and 3 was completing the jobs in a faster rate and thereby posting a value lesser than the threshold value. The running time of all the processes is approximately 8 minutes. When the system started, the first 13 processes used node 2 and once the threshold limit was then the next node was used the next 7 processes at which point node 2's load went below the threshold and now processes were created at node 2 again. This pattern continued throughout the running time of the system. The system never tried to use the other nodes. This clearly shows the drawback to a static single threshold policy.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Test: 1.2

State Information Transfer Policy: Periodic (5 seconds)

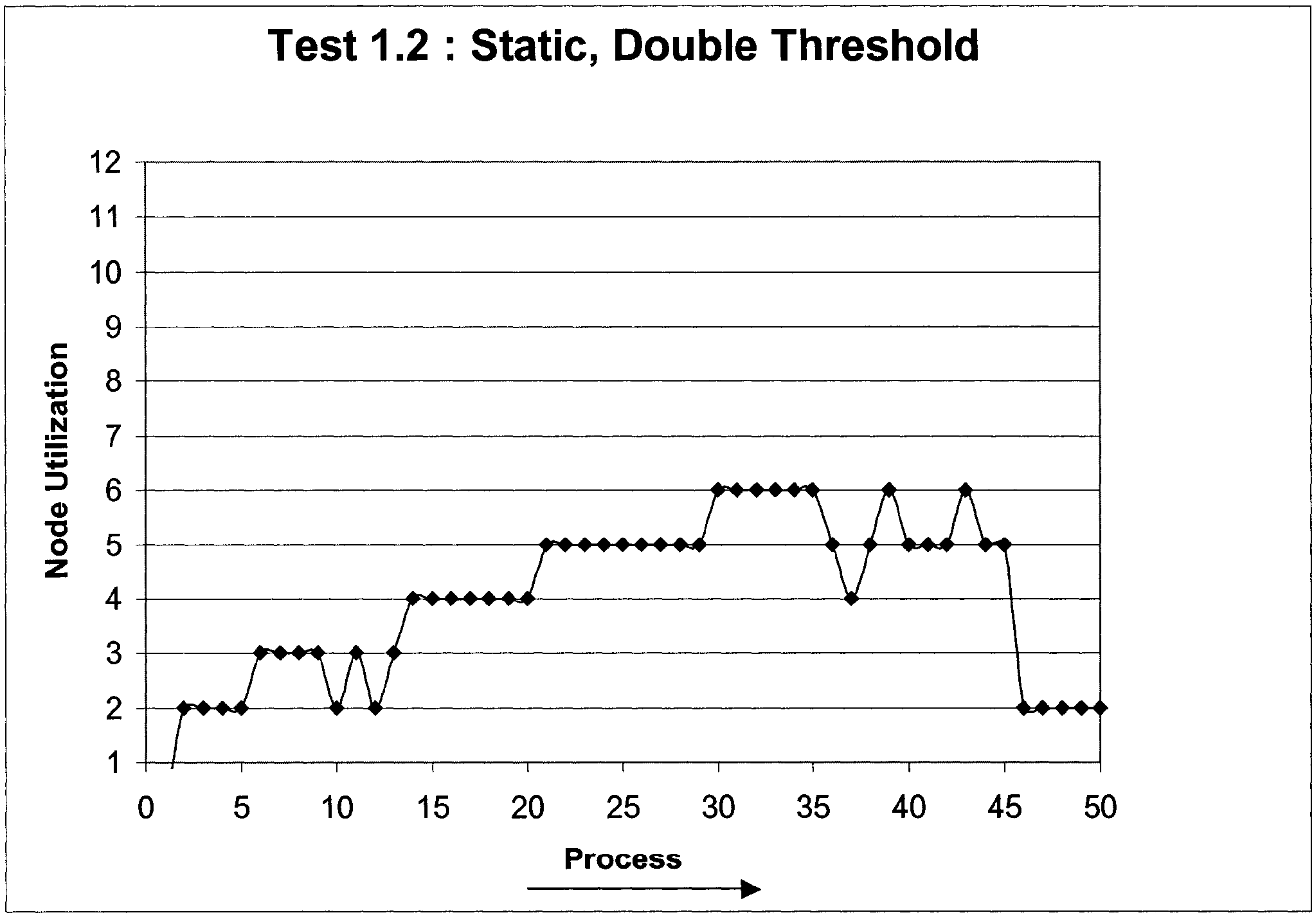
Process Transfer Policy: Static, Double Threshold

EPLAB Client Nodes: 2, 3, 4, 5, 6, 7, 8, 9 and 10

EPLAB Server running at Node 1.

Total Running Time: 5 minutes

Graph 2: Static, Double Threshold Test 1.2



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Analysis:

This test (graph 1.2) is using Static, Double Threshold policy. By static, the threshold limit is not calculated from time to time, it stays fixed. By Double threshold policy, there is a High value and a Low value assigned to the nodes. When this policy is applied more than 2 nodes were used to execute 50 jobs. At the start, the nodes 2 and 3 were processing the jobs until they became busy to accept any more jobs. The initial jobs were more time consuming and that is the reason the nodes 4 and above started processing other jobs. When the job 12 completed at node 2, the next job in the queue was job 46, which was processed by node 2. Total time to process the 50 jobs was only 5 minutes. So, this policy is better than Static, Single Threshold policy.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Test: 1.3

State Information Transfer Policy: Periodic (5 seconds)

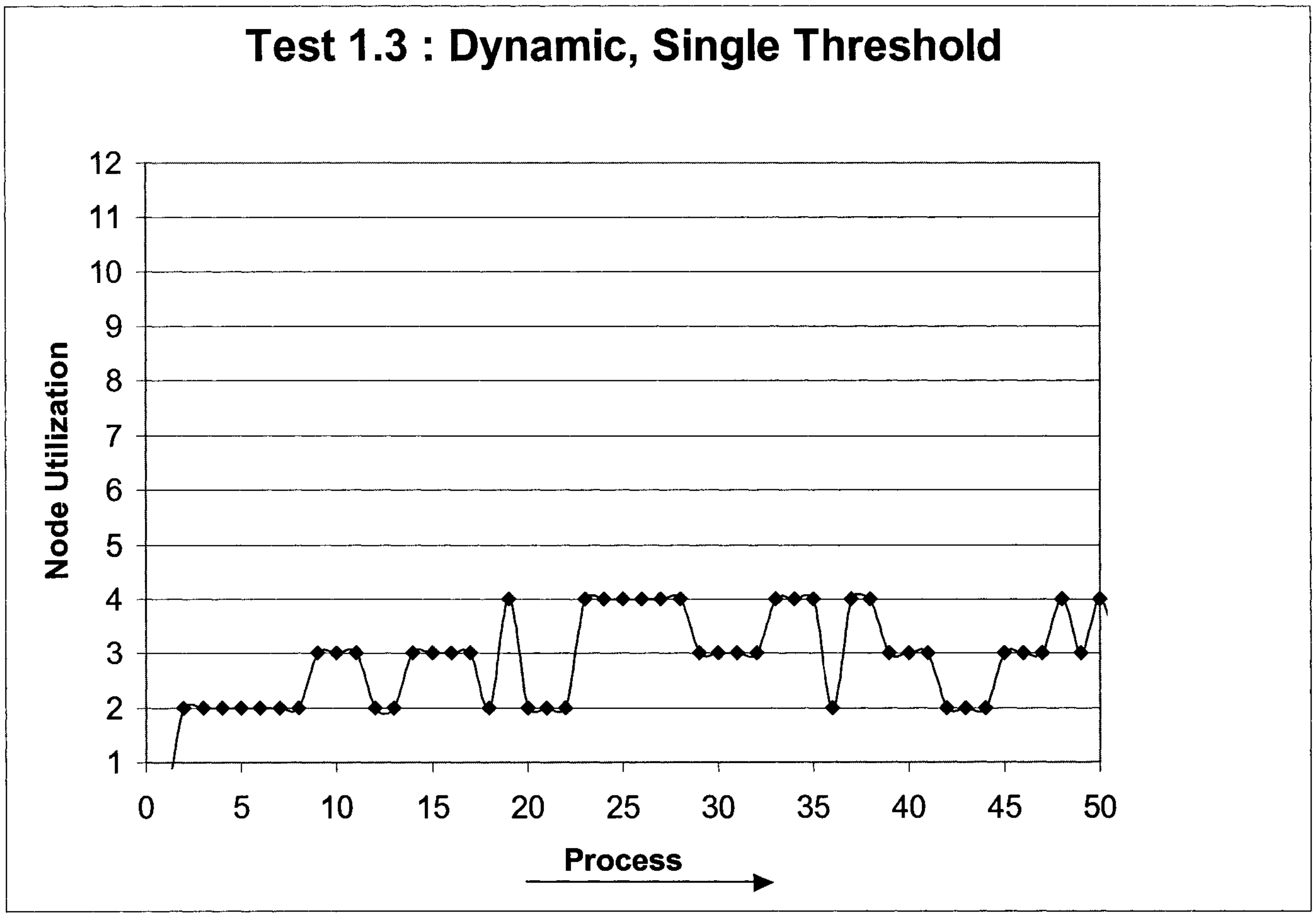
Process Transfer Policy: Dynamic, Single Threshold

EPLAB Client Nodes: 2, 3, 4, 5, 6, 7, 8, 9 and 10.

EPLAB Server running at Node 1.

Total Running Time: 7 minutes

Graph 3: Dynamic, Single Threshold Test 1.3



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Analysis:

The above graph (1.3) depicts the result of using Dynamic, Single Threshold Policy. When this policy was applied, only 3 nodes were used and the total time taken was about 7 minutes. Because of the single threshold policy, there is no upper limit that has to be met. So the jobs were allocated freely to nodes 2, 3 and 4. There was no need for the system to allocate jobs to other nodes.

From tests 1.2 and 1.3, it is obvious that Single Threshold policy does not use all the available nodes to execute processes. This may reduce the performance of the nodes, as it has to work on the jobs allocated by the EPLAB system as well as to work on its other jobs.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Test: 1.4

State Information Transfer Policy: Periodic (5 seconds)

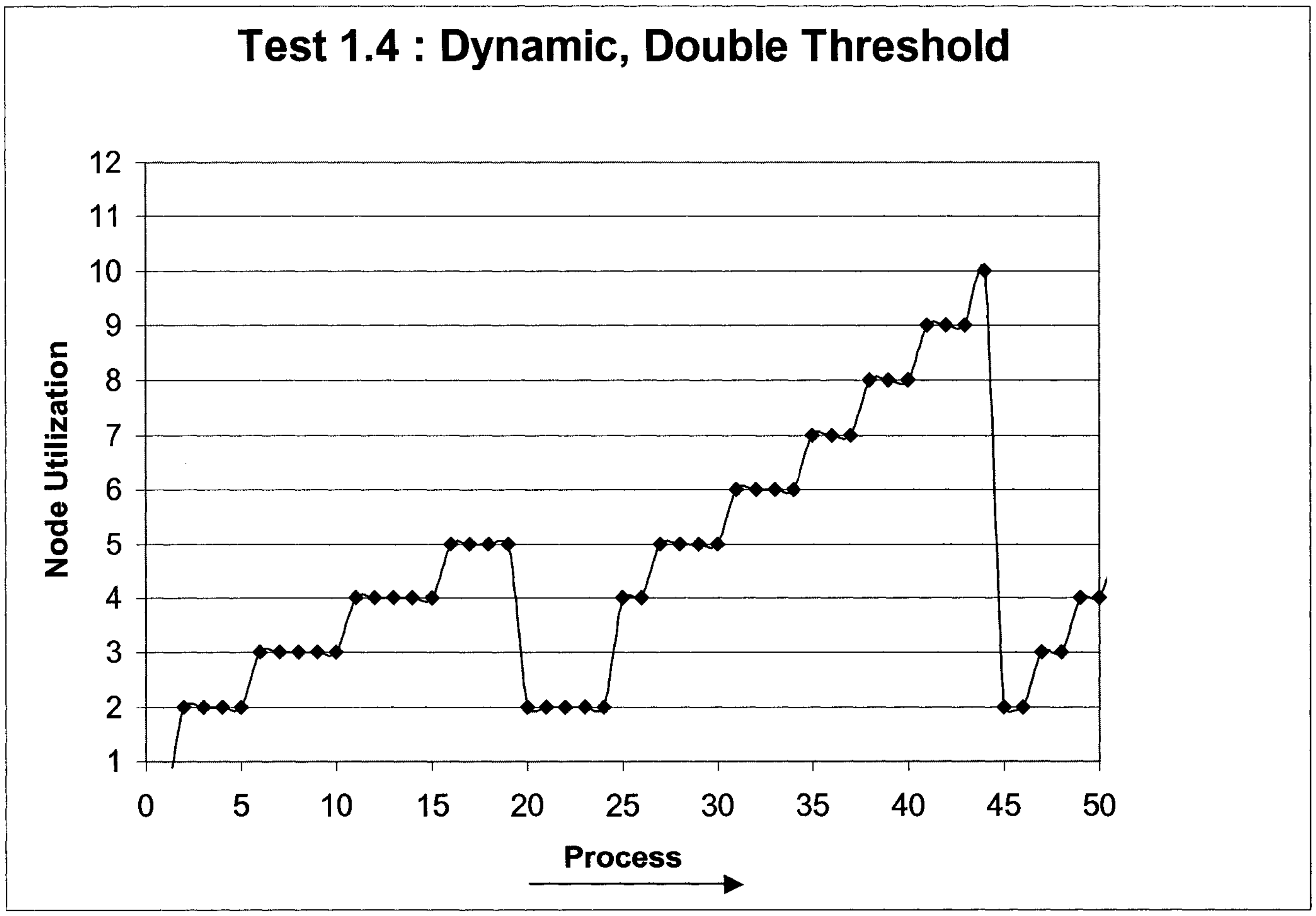
Process Transfer Policy: Dynamic, Double Threshold

EPLAB Client Nodes: 2, 3, 4, 5, 6, 7, 8, 9 and 10.

EPLAB Server running at Node 1.

Total Running Time: 4 minutes

Graph 4: Dynamic, Double Threshold Test 1.4



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Analysis:

From the above graph 1.4, using Dynamic, Double Threshold policy, the total time taken was about 4 minutes, but 9 nodes were used to execute 50 jobs. Because of the Dynamic threshold policy, the nodes quickly resisted to take up more jobs to process. That is the reason for using 9 nodes to complete all the jobs. This method indicates that this can be used for effective utilization of the available nodes. But timewise, there is not much of difference between the tests 1.4 and 1.2.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2 Problem 2: Process Retail Data Transaction Files

Problem Definition: A Point of Sale (POS) component in a Retail System generates data files for every transaction. These data files are periodically transferred to a centralised system to be processed by a program to insert into specific tables in a relational database system. The efficiency of this system depends on how quickly the data files can be processed.

This problem is similar to the previous problem with regard to reading and processing data files, but the difference lies in the amount of data read in this problem.

Program Implementation: The program to process the data files is designed to read one file at a time and look for specific record delimiters and record entries and generates SQL text file that has SQL statements which can be operated on a relational database. As there are several such data files being generated by the POS system, many instances of the program are spawned handle these data files.

The program assumes that the input data file is formatted by records with proper record delimiter and column delimiters. The name of the data file and the output SQL file are passed as command line parameters.

For example: The process command **ProcessDataFile POS.001 SQL.001**
will read the POS.001 data file, extract the transaction records and write the SQL statements to SQL.001 file.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

After programming this solution, JOB file called JOB.POSData was generated. This job file contained sequence of calls to the program to process the data files. Then, the ESPSubmit program was used to send JOB.POSData file to ESPServer which then read the contents of the JOB file one line at a time and found out the best possible node (applying Process Transfer Policy) to execute the process and sent the processes to the nodes to execute it. A log file was generated as processes were dispatched to nodes.

The following tables contain the summary of the log files after each run of the program.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Test: 2.1

State Information Transfer Policy: Periodic (5 seconds)

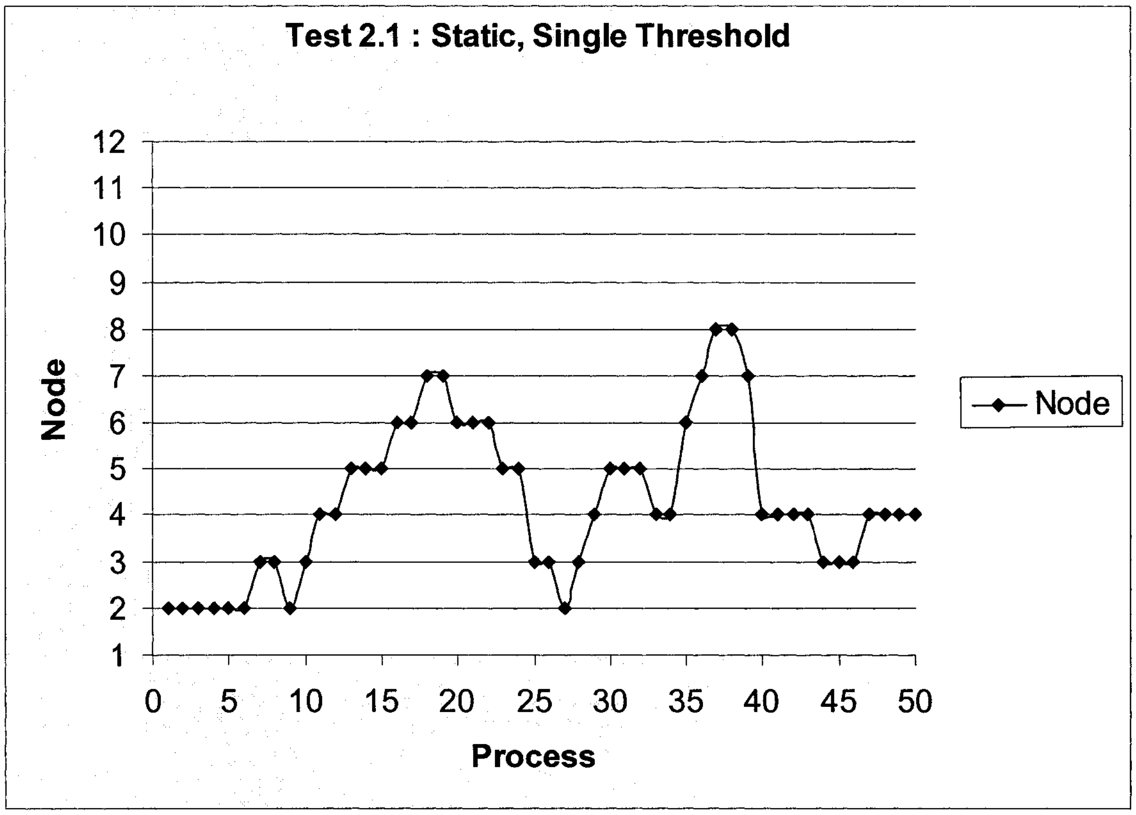
Process Transfer Policy: Static, Single Threshold

EPLAB Client Nodes: 2, 3, 4, 5, 6, 7, 8, 9, 10

EPLAB Server running at Node 1.

Total Running Time: 4 minutes

Graph 5: Static, Single Threshold Test 2.1



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Analysis:
From the graph (2.1), we can observe that the system is using up to 8 nodes to process the data files. The running time of all the processes is approximately 4 minutes. When the system started, the first 5 processes used node 2. Initially the data files were very big. That is the reason that the node 2 was tied up for much more time to process those data files. Once the threshold limit exceeded, the system started using other nodes. This policy efficiently uses almost all the nodes to process.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Test: 2.2

State Information Transfer Policy: Periodic (5 seconds)

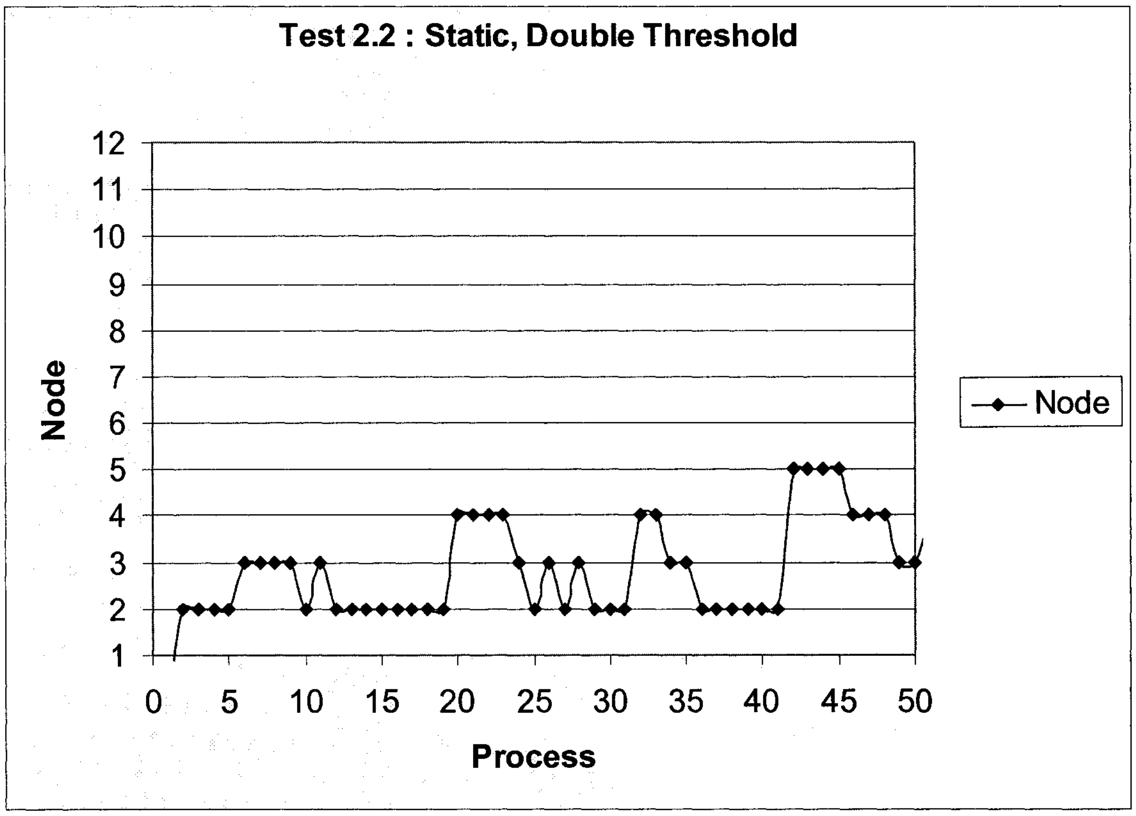
Process Transfer Policy: Static, Double Threshold

EPLAB Client Nodes: 2, 3, 4, 5, 6, 7, 8, 9, 10

EPLAB Server running at Node 1.

Total Running Time: 6 minutes

Graph 6: Static, Double Threshold Test 2.2



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Analysis:
From the graph (2.2), we can observe that the system is only using up to 5 nodes to process the data files. The running time of all the processes is approximately 6 minutes. This policy can be used if we have limited nodes to process. This test proves that this cannot be an effective set of policies as this test did not complete in less time and did not use more nodes than test 1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Test: 2.3

State Information Transfer Policy: Periodic (5 seconds)

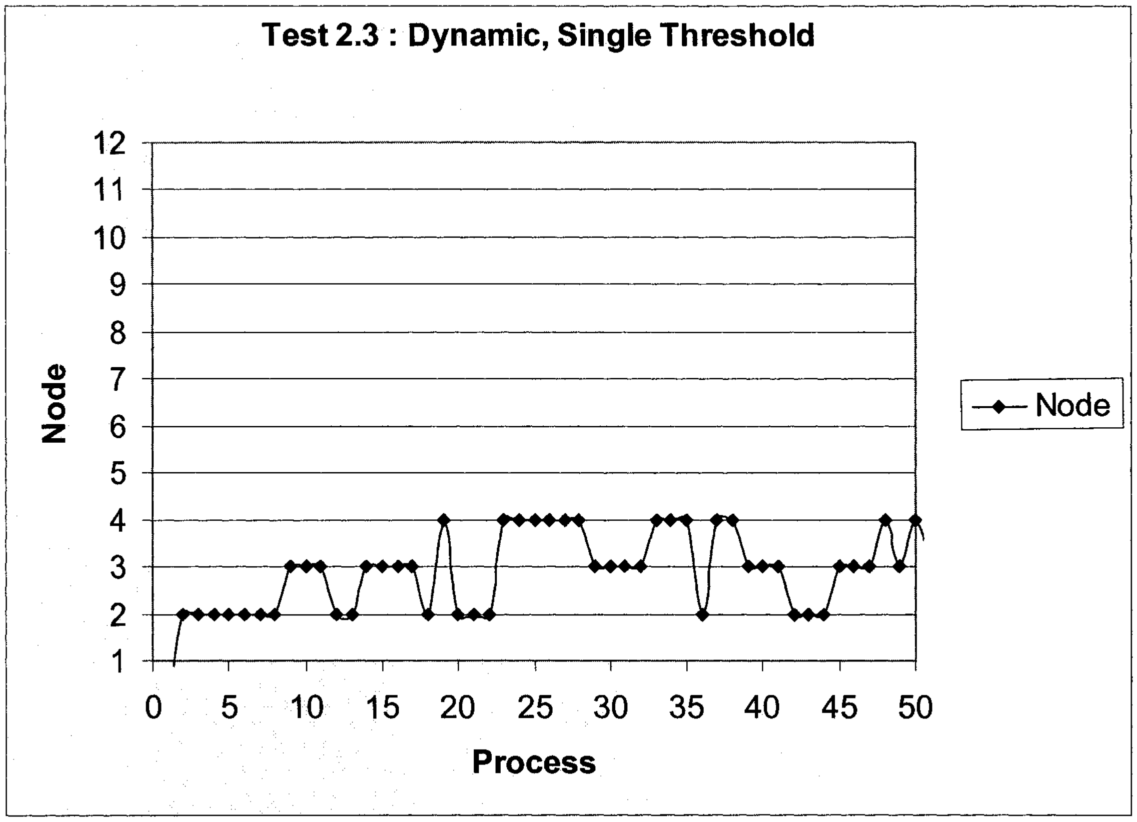
Process Transfer Policy: Dynamic, Single Threshold

EPLAB Client Nodes: 2, 3, 4, 5, 6, 7, 8, 9, 10

EPLAB Server running at Node 1.

Total Running Time: 11 minutes

Graph 7: Dynamic, Single Threshold Test 2.3



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Analysis:
From the above graph (2.3), we observe that the system is only using up to 4 nodes to process the data files. The running time of all the processes is approximately 11 minutes. Though this policy uses much less nodes, it may not be efficient as it took more than 10 minutes to process the data files that was processed much quicker using the other policies.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Test: 2.4

State Information Transfer Policy: Periodic (5 seconds)

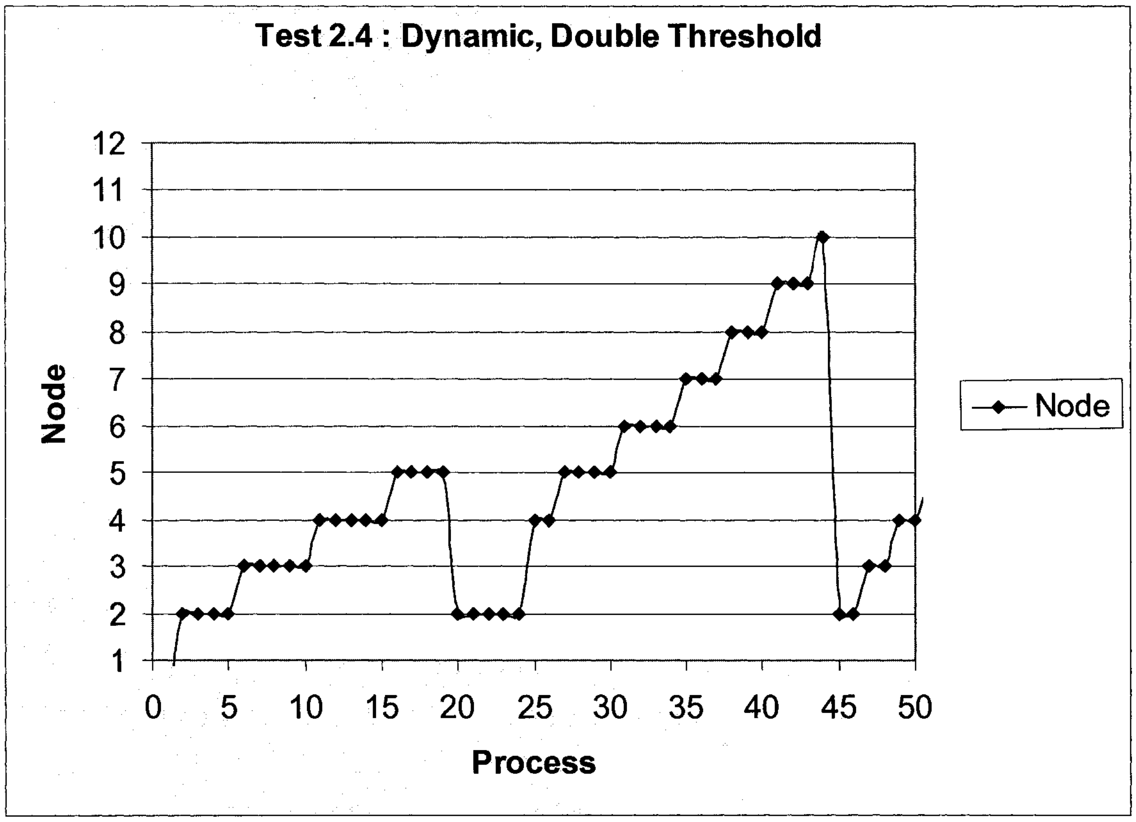
Process Transfer Policy: Dynamic, Double Threshold

EPLAB Client Nodes: 2, 3, 4, 5, 6, 7, 8, 9, 10

EPLAB Server running at Node 1.

Total Running Time: 5 minutes

Graph 8: Dynamic, Double Threshold Test 2.4



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Analysis:
From the above graph (2.4), we observe that the system uses all the available nodes and the total time to process all the data files is only 5 minutes. This policy is the most efficient of all the policies to process the program both in terms of using the available nodes and also completing the task faster.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5. CONCLUSION

In the course of this thesis several books and research papers relating to the concept of load balancing in distributed operating systems were studied and analyzed. There are several policies available for state information exchange, process transfer and load estimation of a node. But not many talk about the combination of these policies or try to deduce the effective combination to solve distributed computing problems.

By running these programs several times and observing the results, the combination of Dynamic and Double Threshold Policies is very effective in using all the available nodes to solve a distributed problem. But using Static and Single Threshold policies used many nodes available but completed the distributed problem in the least time.

EPLAB is a non-preemptive system where the running jobs are never revoked to run on another system. Also, the system assumes the existence of a shared file system where the jobs exist as executable programs.

As an extension to the thesis, many more distributed system policies could be implemented and several problems can be solved using EPLAB system to find out the effectiveness in solving different categories of problems like database programs, message queuing systems etc. to name a few.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A dynamic load-balancing policy is one that uses run-time state information in making scheduling decisions. There are two kinds of dynamic policies: adaptive and non-adaptive. The latter always use the same (fixed, load-dependent) policy; the former may adjust policy parameters in order to gradually improve their performance.

The key point is that while non-adaptive policies use only the information about the run-time state (`load'), adaptive policies use that plus information about current performance (`speed-up'). In adaptive policies, the rules for adjusting policy parameters may be static or dynamic. An example of the former will be: "shift to a conservative migration rule when system-wide load patterns are varying too rapidly." An example of the latter will be: "increase sender-side threshold when migrated jobs cause slowdown rather than speed-up." Some researchers refer to the performance-driven adaptation exhibited by the second policy as "learning."

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6. REFERENCES

[1] Distributed Operating Systems, Chapter 7: Resource Management Section 7.4.2, Page 359 Load Estimation Policies.

[2] Distributed Operating Systems, Chapter 7: Resource Management Section 7.4.2, Page 360 Process Transfer Policies.

[3] Alonso R. and Cova L. "Sharing Jobs Among Independently Owned Processors", Proceedings of the 8th International Conference on distributed Computing Systems, IEEE, New York, pp 282-288 (June 1988).

[4] Process Migration, Dejan S.Milojicic, Fred Douglis, Yves Paidaveine, Richard Wheeler and Songnian Zhou, ACM Computing Surveys, Vol. 32, No. 3, September 2000, pp 241-299.

[5] Distributed Operating Systems, Chapter 7:Resource Management, Page 357

[6] D L Eager, E D Lazowska and J Zahorjan. Adaptive load sharing in homogeneous distributed systems, IEEE Transactions on Software Engineering, SE 12 no 5:662 675, May 1986

[7] G Bernard, D Steve, and M Simantic. A survey of load sharing in distributed systems. Distributed Systems Engineering Journal, December 1993.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[8] S Chowdury. The greedy load sharing algorithm. Journal of Parallel and Distributed Computing, 9:93-99, 1990.

[9] B Shirazi and A R Hurson, Special issue on scheduling and load balancing. Journal of Parallel and Distributed Computing, 16 no 4, December 1992.

[10] M Livny and M Melman. Load balancing in homogeneous broadcast distributed systems. Proceedings of the Computer Network Performance Symposium, pages 47-55, April 1982.

[11] Comparative Evaluation of Process Migration Algorithms
Mathias Noack, Dresden University of Technology, Operating Systems Group, 21th July 2003.

[12]
MOYER, PANTS Application Node Transparency System online at
http://segfault.dhs.org/ProcessMigration/

[13]
Wills and Finkel, Scalable approaches to Load sharing in the presence of Multicasting Computer Communications, September 1995

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.