INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Adaptive load distributing in distributed systems

Shivaratri, Niranjan Gubbi, Ph.D.

The Ohio State University, 1994
ADAPTIVE LOAD DISTRIBUTING IN DISTRIBUTED SYSTEMS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Niranjan G. Shivaratri, B.S.E.E., M.S.

* * * * *

The Ohio State University

1994

Dissertation Committee:

Dr. Mukesh Singhal
Dr. Ming T. Liu
Dr. Phillip Krueger

Approved by

Adviser

Computer and Information Science Department
© Copyright by
Niranjan G. Shivaratri

1994
ACKNOWLEDGEMENTS

I would like to express my sincere thanks to my advisor Prof. Mukesh Singhal for his encouragement, support, and guidance right from the first day to the last day of my graduate studies. I would like to express my gratitude and thanks to Prof. Phillip Krueger for his extensive guidance and advise throughout my research work. Also, thanks go to Prof. Liu, Prof. Sadayappan, and Prof. Jayasimha for their support and guidance in my graduate studies. I would like to thank Mahendra Ramachandran and Manas Mandal who have helped me at various stages in my graduate studies. I would also like to thank my wife Sumathi for her support during my studies. Finally, thanks to my parents whose support and encouragement brought me to the stage where I am today.
VITA

September 21, 1958 ......................... Born - Krishnarajanagar, INDIA

1979 .................................................. B.S. Electrical Engineering
Mysore University
Mysore, India.

1983 .................................................. M.S. Computer Science
Villanova University
Villanova, PA 19085.

1987-1992 ........................................... Graduate Teaching Associate,
The Ohio State University.

1/93-12/93 ........................................... Presidential Fellowship,
The Ohio State University.

Publications


Fields of Study

Major Field: Computer and Information Science

Studies in:

- Distributed Systems  Prof. Mukesh Singhal
- Performance Evaluation  Prof. Phillip Krueger
- Computer Architecture  Prof. D. N. Jayasimha
# Table of Contents

ACKNOWLEDGEMENTS ................................................................. ii

VITA ........................................................................................................ iii

LIST OF TABLES ................................................................................... viii

LIST OF FIGURES ................................................................................ x

CHAPTER PAGE

I Introduction ..................................................................................... 1

  1.1 Background .................................................................................. 1
  1.2 Motivation .................................................................................... 1
  1.3 Problem Statement and Goals .................................................... 3
  1.4 Summary of Results ................................................................... 8
  1.5 Organization of the Thesis ......................................................... 10

II Taxonomy ....................................................................................... 12

  2.1 Introduction .................................................................................. 12
  2.2 Issues in Load Distributing ....................................................... 12
  2.3 Example Algorithms .................................................................. 21
    2.3.1 Sender-Initiated Algorithms ............................................. 21
    2.3.2 Receiver-Initiated Algorithms ........................................... 25
    2.3.3 Symmetrically-Initiated Algorithms .................................. 28
  2.4 Performance Comparison ......................................................... 28
  2.5 Real-time Systems ....................................................................... 32
  2.6 Priority Systems .......................................................................... 33
  2.7 Summary ...................................................................................... 34
III  Adaptive Location Policies ................................................................. 36
   3.1  Introduction .................................................................................... 36
   3.2  System Model .................................................................................. 39
   3.3  Symmetrically-Initiated Adaptive Location Policy ......................... 40
       3.3.1  Data Structure at a Node ..................................................... 41
       3.3.2  Sender-Initiated Negotiation .............................................. 42
       3.3.3  Receiver-Initiated Negotiation .......................................... 43
       3.3.4  An Efficient Scheme for Maintaining the Lists .................... 44
       3.3.5  Discussion ............................................................................. 46
   3.4  Performance Comparison ............................................................... 47
       3.4.1  Global Scheduling Algorithms ............................................ 48
       3.4.2  Results ................................................................................ 49
   3.5  Sender-Initiated Adaptive Location Policy ....................................... 57
       3.5.1  Data Structure at a Node ..................................................... 58
       3.5.2  Initialization ......................................................................... 58
       3.5.3  Sender-Initiated Negotiation .............................................. 58
       3.5.4  Receiver-Initiated Negotiation .......................................... 59
       3.5.5  Performance Comparison .................................................... 59
   3.6  Summary and Conclusions ................................................................ 62

IV  A Load Index And A Transfer Policy for Global Scheduling Tasks With Deadlines ............................................................... 65
   4.1  Introduction .................................................................................... 65
   4.2  System Model .................................................................................. 68
       4.2.1  A Local Scheduling Policy .................................................. 68
   4.3  Free-Time: A Load Index ............................................................... 69
   4.4  A Transfer Policy .......................................................................... 73
   4.5  Performance Study ......................................................................... 75
       4.5.1  Global Scheduling Algorithms ............................................ 76
       4.5.2  Performance Results ............................................................ 79
   4.6  Related Work ................................................................................ 97
   4.7  Summary and Conclusions ............................................................. 100

V  Distributed Scheduling of Tasks with Priority ....................................... 102
   5.1  Introduction .................................................................................... 102
List of Tables

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Homogeneous workload, THRESHOLD = 1, LOWERTHRESHOLD = 1, UPPERTHRESHOLD = 1, Response time at Offered system load of 50%.</td>
</tr>
<tr>
<td>2</td>
<td>Homogeneous workload, THRESHOLD = 1, LOWERTHRESHOLD = 1, UPPERTHRESHOLD = 1, Response time at Offered system load of 70%.</td>
</tr>
<tr>
<td>3</td>
<td>Homogeneous workload, THRESHOLD = 1, LOWERTHRESHOLD = 1, UPPERTHRESHOLD = 1, Response time at Offered system load of 90%.</td>
</tr>
<tr>
<td>4</td>
<td>Homogeneous workload, THRESHOLD = 1, LOWERTHRESHOLD = 1, UPPERTHRESHOLD = 1, Percentage of Priority Tasks = 50%, Response time at Offered system load of 50%.</td>
</tr>
<tr>
<td>5</td>
<td>Homogeneous workload, THRESHOLD = 1, LOWERTHRESHOLD = 1, UPPERTHRESHOLD = 1, Percentage of Priority Tasks = 50%, Response time at Offered system load of 70%.</td>
</tr>
<tr>
<td>6</td>
<td>Homogeneous workload, THRESHOLD = 1, LOWERTHRESHOLD = 1, UPPERTHRESHOLD = 1, Percentage of Priority Tasks = 50%, Response time at Offered system load of 90%.</td>
</tr>
<tr>
<td>7</td>
<td>Homogeneous workload, THRESHOLD = 1, LOWERTHRESHOLD = 1, UPPERTHRESHOLD = 1, Percentage of Priority Tasks = 20%, Response time at Offered system load of 50%.</td>
</tr>
</tbody>
</table>
Homogeneous workload, THRESHOLD = 1, LOWERTHRESHOLD = 1, UPPERTHRESHOLD = 1, Percentage of Priority Tasks = 20%, Response time at Offered system load of 70%.

Homogeneous workload, THRESHOLD = 1, LOWERTHRESHOLD = 1, UPPERTHRESHOLD = 1, Percentage of Priority Tasks = 20%, Response time at Offered system load of 90%.

Heterogeneous workload, THRESHOLD = 1, Response time when No. of load generating nodes = 30.

Heterogeneous workload, THRESHOLD = 1, Response time when No. of load generating nodes = 20.

Heterogeneous workload, THRESHOLD = 1, Response time when No. of load generating nodes = 10.
# List of Figures

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>3</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>6</td>
<td>52</td>
</tr>
<tr>
<td>7</td>
<td>53</td>
</tr>
<tr>
<td>8</td>
<td>56</td>
</tr>
<tr>
<td>9</td>
<td>60</td>
</tr>
<tr>
<td>10</td>
<td>62</td>
</tr>
</tbody>
</table>
11 Mean Response Time of SEND and ADAPT-SEND vs. Probeflimit.
Offered System Load = 0.75, Threshold = 1, Mean Service Time = 1

12 Calculation of free-time .................................................. 72

13 Mean Miss Ratio vs. Offered System Load. ............................. 81
   X = 0.2-S, LTFT = 0.8-S, UTFT = 1.05-S,
   W = 0.9-(X + S).

14 Mean Miss Ratio vs. Laxity. LTFT = 0.8-S, UTFT = 1.05-S,
   W = 0.9-(X + S), U = 0.7.

15 No of Anticipatory Transfers vs. Offered System Load/Laxity. LTFT = 0.8-S,
   UTFT = 1.05-S, W = 0.9-(X + S).

16 Mean Miss Ratio vs. UTFT. W = 0.9-(X + S), .......................... 89
   U = 0.7, LTFT = 0.8-S.

17 Mean miss ratio vs. LTFT. W = 0.9-(X + S), .......................... 90
   U = 0.7, UTFT = 1.05-S.

18 Mean Miss Ratio vs. Window Factor. ................................. 92
   U = 0.7, LTFT = 0.8-S, UTFT = 1.05-S

19 Mean Miss Ratio vs. Number of Load-Generating Nodes. U = 0.7, LTFT = 0.8-S,
   UTFT = 1.05-S, X = 0.2-S, W = 0.9-(X + S)

20 Mean Miss Ratio vs. Offered System Load. ............................. 96
   X = 0.2-S, LTFT = 0.2-S, UTFT = 1.05-S,
   W = 0.9-(X + S)
CHAPTER I

Introduction

1.1 Background

The availability of low-cost processors and advances in communication technologies (local area networks) have spurred considerable interest in locally-distributed systems. The primary advantages of these systems are high performance, availability, and extensibility at low cost. To realize these benefits, however, system designers must overcome several problems. One such problem is that of allocating the considerable processing capacity available in a locally distributed system so that it is used to its fullest advantage. This thesis focuses on the problem of judiciously and transparently redistributing the load of the system among its computers so that overall performance is maximized.

1.2 Motivation

A ‘locally distributed system’ consists of a collection of autonomous computers, which are connected by a local area communication network. Users submit tasks at their host computers for processing. The need for load distributing arises in such environments because due to the random arrival of tasks and their random service time requirements, there is a good possibility that several computers are heavily loaded
and hence suffering from performance degradation, while other computers are idle or lightly loaded (Figure 1).

Clearly, if the workload at some computers is typically heavier than that at others, or if some processors execute tasks at a slower rate than others, this situation is likely to occur more often. The usefulness of load distributing is not so clear in systems in which all processors are equally powerful, and over the long term, all have equally heavy workloads. Livny and Melman [26] have shown that even in such a homogeneous distributed system, because of statistical fluctuations in the arrivals of tasks to computers and task service time requirements, there is a large probability that at least one computer is idle while other computers are heavily loaded. Therefore, even in a
homogeneous distributed system, system performance can potentially be improved by appropriately transferring a portion of the workload from heavily loaded computers (senders) to idle or lightly loaded computers (receivers). This raises the question of what we mean by performance. A widely used performance metric is the ‘average response time’ of tasks. The response time of a task is the time elapsed between its initiation and its completion. Minimizing the average response time is often the goal of load distributing in general type of systems. In real-time systems, tasks are associated with deadlines and they must complete by their deadlines. Therefore in real-time systems, maximizing the number of tasks that complete within their deadlines is the goal. In systems that allow multiple priorities, providing better service to higher priority tasks is the goal.

1.3 Problem Statement and Goals

General Systems

Two important components of a global scheduling algorithm are its transfer policy and its location policy. The transfer policy at a node determines whether a task should be transferred either to it or from it, by identifying the node as a suitable sender of a task, a suitable receiver, or neither. The location policy at a node, on the other hand, determines to where or from where a task should be transferred. Location policies can be broadly characterized as sender-initiated (in which receivers are found for senders), receiver-initiated (in which senders are found for receivers), and symmetrically-initiated (in which complementary nodes are found for both senders and receivers).
In searching for a complementary node for a sender or a receiver, many location policies *probe* nodes to determine their current status. In an analytic study of homogeneous systems (identical nodes having identical workload characteristics) that assumed the processor cost of probing to be negligible, Eager, et al. [11] found that the type of location policy that performs best depends on the workload. Their receiver-initiated policy performed best at high system loads, since at such loads, busy nodes that can transfer tasks are easily found. On the other hand, their sender-initiated policy outperformed their receiver-initiated policy at low to medium system loads since at such loads, idle nodes that can accept work are easily found. Both policies used a simple method of probing, in which nodes were probed randomly until a complementary node was found, or until *Probelimit* (a parameter of the algorithm) nodes had been probed.

Krueger [18], in a comparison of sender, receiver, and symmetrically-initiated policies, showed that the situation is more serious when the processor cost of probing is taken into consideration. At high system loads, both sender-initiated and symmetrically-initiated policies cause system instability for the following reason: At such loads, few nodes in the system are likely to be lightly loaded (i.e., good receivers), and hence the probability that an overloaded node (sender) will succeed in finding a receiver is very low. However, the probing activity due to sender-initiated searches increases as the task arrival rate increases, eventually reaching a point where the cost of probing is greater than its benefit. At some more extreme point, the workload that cannot be offloaded from a node, together with the overhead incurred by probing,
exceeds the CPU capacity of the node, and instability results. Clearly, the cause of
this instability is the failure of the location policy to adapt its activity to the global
system load (even though the actions may be justified according to the local load at
the individual nodes). Indiscriminate probing wastes computing capacity at times
when such waste cannot be afforded.

Another problem occurs when workload arrival rates are not homogeneous. Under
non-homogeneous workloads, the problem is that inefficient probing (as occurs when
probing is random) considerably reduces the range of conditions under which global
scheduling can deliver acceptable performance. Measurement studies, including those
by Mutka and Livny [27] and by Krueger and Chawla [19], have found such workloads
to be common in existing systems.

The goal of this thesis is to develop location policies that capture the advantages
of both receiver-initiated and sender-initiated location policies, that probe efficiently,
and that are simple enough to be practical. We propose two adaptive policies: a
symmetrically-initiated policy that is suitable for systems that have the capability
to transfer both new tasks and partly-executed tasks, and a sender-initiated policy
that is suitable for systems where only newly-arrived tasks can be transferred. Two
key features of our policies are: First, to allow adaptivity, the proposed policies use
data gathered during probing which is normally discarded by non-adaptive policies; no
extra messages are required to collect system state information. Second, the proposed
location policies are general, and can be used with a broad range of transfer policies.
Real-time Systems

Real-time distributed systems have deadline constraints associated with tasks. A global scheduling algorithm improves performance through load distributing by taking advantage of the statistical fluctuations in the load at different nodes. By improving the system performance, it is meant that the number of tasks that complete by their deadlines is maximized.

Over the past several years, many global scheduling algorithms have been proposed for real-time distributed systems to schedule tasks with deadline constraints. These algorithms try to transfer tasks whenever task's deadlines cannot be met locally [7, 22, 32, 33, 35], and/or when there are more than $T$ (threshold) tasks in the queue [7, 35].

There are three problems with these approaches:

1. A queue length $< T$ does not necessarily mean that the local schedule will be able to meet a new arrival's deadline due to the following reasons:

   - A task with a high service demand can keep a node busy beyond the latest starting time of a waiting or an arriving task.
   - A scheduling policy based on the queue length neither takes into account a task's service demand nor task's deadline information in making decisions. Hence queue length is not an appropriate load index to use.

2. Initiating task transfers only after realizing that the local schedule cannot meet a task's deadline will limit the scheduler's ability to satisfy deadlines (especially tight deadlines). This is because, not only the task selected for transfer is
delayed due to communication delays, other tasks that are waiting are also
delayed due to the overhead incurred by the CPU to carry out the task transfer
activity. Any delay in starting the execution of a task increases the probability
of that task missing the deadline.

3. A fundamental limitation of the previous approaches [7, 22, 32, 33, 35] is that,
the actions taken by the scheduling algorithms are corrective in nature, i.e.,
corrective actions (e.g., task transfers) are initiated only after an undesirable
condition has occurred (e.g., when a deadline cannot be met, local load is high,
etc.).

In view of the above problems, our goals are as follows:

1. Define a load index that gives a more precise information than the queue length
   about a node's ability to satisfy deadlines.

2. Construct a mechanism which takes:

   (a) Preventive measures by looking ahead, thereby, averting potential deadline
       misses.

   (b) Corrective measures if necessary to avert a deadline miss.

Priority Systems

Distributed systems have been proposed for applications such as control systems, in-
dustrial process control, automated manufacturing, air-traffic control, program trading, etc. [38]. In these systems, it is important that certain critical tasks complete
within certain timelimits. These tasks can be viewed as high priority tasks which must get better service than other type of tasks. Moreover, in a general computing facility, certain users/tasks may receive higher priority than others because of project deadlines [28], importance of the tasks [28], the user is paying higher fees for better service, etc.

In the final part of the thesis, we address scheduling tasks with priorities in distributed systems. The objective is to ensure that when a task is scheduled at a node, that task has the highest priority among all the tasks waiting for service in the entire system. This objective is difficult to realize as the state information—tasks with what priority are waiting at which nodes—is not readily available in distributed systems. To realize the above objective in distributed systems, whenever a node is ready to schedule a task, it is necessary to make the up-to-date state information available to it. It should be ensured that the state information is made available with minimum delay and by exchanging minimum number of messages. Otherwise, the overhead due to scheduling may outweigh the benefit of scheduling.

1.4 Summary of Results

For general systems, we have presented two adaptive location policies, namely, symmetrically-initiated adaptive location policy and sender-initiated adaptive location policy, for global scheduling algorithms. By increasing the efficiency of negotiation, we found that these policies do both improve the performance and reduce the threat of system instability relative to non-adaptive policies, yet add little complexity to global scheduling. A key feature of the proposed policies is that they are general, and can
be used with a broad range of transfer policies.

The symmetrically-initiated adaptive location policy presented maintains stability and out-performs all the non-adaptive policies studied at all system loads under both homogeneous and heterogeneous workloads. The sender-initiated adaptive location policy performs better than its non-adaptive counterpart under all conditions studied. A limitation of the sender-initiated adaptive policy is that it is not stable under extreme workload heterogeneity. While not matching the performance of the symmetrically-initiated adaptive policy, the sender-initiated policy has the advantage of not requiring transfers of partly-executed tasks.

For real-time systems, we have proposed a load index and a transfer policy. The load index "free-time" gives a more accurate indication of a node's ability to satisfy the deadlines of future arrivals than just the "queuelength" at the node. The key features of the new transfer policy are: (a) It takes preventive measures by doing anticipatory transfers in addition to corrective measures, thereby, reducing the number of potential deadline misses. (b) Using free-time, it adapts much better to the system state by looking-ahead and does not require the transfer of partially executed tasks. (c) It is not very sensitive to minor variations of the free-time thresholds and window size, and in this sense the new transfer policy is robust. We have showed that "Least-Laxity First" is a better local scheduling policy to satisfy deadlines when compared to "Least Processing Time First". A simulation study reveals that synergism resulting from Least-Laxity-First local scheduling policy, free-time, and our transfer policy, reduces the number of deadline misses significantly over a wide range of system parameters.
For priority systems, we proposed a token-based algorithm for scheduling tasks with priority in distributed systems. In the token-based algorithm, each node forwards the state information regarding task arrivals and their priorities for deposition in the token. Nodes that are ready to schedule their next task, obtain the token and use the state information available in the token to schedule the highest priority task waiting in the system. We also modified the list-based algorithm proposed for general systems to schedule priority tasks. We compare the token-based algorithm’s performance with an ideal algorithm and show that the new algorithm collects, disseminates, and updates the entire system state efficiently and quite accurately. In addition, a system employing the token-based algorithm is shown to perform better than the list-based algorithm and substantially better than a system that performs no load distributing at all.

Finally, each of these protocols except symmetrically initiated adaptive location policy is designed to use non-preemptive transfers exclusively. As a result, all carry relatively low overhead, both in terms of software development and maintenance time, and in terms of execution overhead. The proposed algorithms and protocols are simple enough to be practical, yet effective over a wide range of system conditions.

1.5 Organization of the Thesis

In Chapter II, we discuss several key issues in load distributing for general purpose systems, including the design tradeoffs for load distributing algorithms and the classification of these algorithms. In addition, we describe several load distributing algorithms previously proposed and compare their performance.
In Chapter III, the details of the two adaptive location policies, is described. The performance of these policies compared to the previously proposed policies is also presented.

In Chapter IV, the details of the new load index and the new transfer policy which makes use of the load index is presented. The performance of this combination is compared to the previously studied algorithms.

In Chapter V, the details of the token-based and the list-based algorithms for scheduling tasks with priority is presented. The performance of these algorithms is compared with that of an ideal algorithm and the results are presented.

In Chapter VI, we present the conclusions and the directions for future research.
CHAPTER II

Taxonomy

2.1 Introduction

In this chapter, we discuss several key issues in load distributing for general purpose systems, including the motivation for load distributing and design tradeoffs for load distributing algorithms. In addition, we describe several load distributing algorithms and compare their performance.

2.2 Issues in Load Distributing

In this section, we discuss several central issues in load distributing, which will help in understanding its intricacies. In the following, we use the terms computer, processor, machine, workstation, and node interchangeably.

Motivation: A 'locally distributed system' consists of a collection of autonomous computers, which are connected by a local area communication network. Users submit tasks at their host computers for processing. The need for load distributing arises in such environments because, due to the random arrival of tasks and their random service time requirements, there is a good possibility that several computers are heavily loaded and hence suffering from performance degradation, while other computers
are idle or lightly loaded.

**Static, Dynamic, and Adaptive Algorithms:** Load distributing algorithms can be broadly characterized as static, dynamic, or adaptive. *Dynamic* load distributing algorithms use system state information (the loads at nodes), at least in part, to make load distributing decisions, while *static* algorithms make no use of such information.

In static load distributing algorithms, decisions are hard-wired in the algorithm using a priori knowledge of the system. For example, under a simple 'cyclic splitting' algorithm, each node assigns the $i^{th}$ task it initiates to node $i \mod N$, where $N$ is the number of nodes in the system. Alternatively, a probabilistic algorithm assigns a task to node $i$ with probability $p_i$, where the probabilities are determined statically according to factors such as the average task initiation rate and execution rate for each node. A weakness in both of these algorithms is that each can potentially make poor assignment decisions. Because the states of the nodes are not considered when making such decisions, a task initiated at an otherwise-idle node may be transferred to a node having a serious backlog of tasks.

Dynamic algorithms have the potential to outperform static algorithms by making use of system state information to improve the quality of their decisions. For example, a simple dynamic algorithm might be identical to one of the above static algorithms, except that an arriving task would not be transferred if the node it arrived to was idle. A more sophisticated dynamic algorithm might also take the state of the receiving node into account, possibly transferring a task to a node only if the receiving node was idle. Dynamic algorithms can also improve performance relative
to static algorithms by exploiting short-term fluctuations in the system state. For example, a still more sophisticated dynamic algorithm might transfer an executing task, if it was sharing a node with another task and some other node became idle. While dynamic algorithms incur more overhead than their static counterparts, due to collecting, storing and analyzing state information, this overhead is often well-spent. Most recent load distributing research has concentrated on dynamic algorithms, and that will be our focus for the remainder of this paper.

Adaptive load distributing algorithms are a special class of dynamic algorithms, in that they adapt their activities by dynamically changing the parameters of the algorithm, or even the policy itself, to suit the changing system state. For example, if some load distributing policy performs better than others under certain conditions, while another policy performs better under other conditions, a simple adaptive algorithm might choose between these policies based on observations of the system state.

As another example, a dynamic algorithm might continue operating (and incurring overhead) even when the system is uniformly so heavily loaded that no performance advantage can be gained by transferring tasks. To avoid overloading such a system, an adaptive algorithm might instead reduce its load distributing activity when it observes this condition.

**Load:** A key issue in the design of dynamic load distributing algorithms is identifying a suitable *load index*. The goal of a load index is to predict the performance of a task if it is executed at some particular node. Therefore to be effective, the readings taken from a load index when tasks initiate should correlate well with task response times.
Load indices that have been studied and used include the length of the CPU queue, the average CPU queue length over some period, the amount of available memory, the context-switch rate, the system call rate, and the CPU utilization. Researchers have consistently found significant differences in the effectiveness of such load indices, and that simple load indices are particularly effective. For example, Kunz [21] found that the choice of a load index has considerable effect on performance, and that the most effective of the indices we have mentioned is the CPU queue length. Furthermore, Kunz found no performance improvement over this simple measure when combinations of these load indices were used. Finally, it is crucial that the mechanism used to measure load is efficient and imposes minimal overhead.

**Preemptive vs. Non-preemptive transfers:** Preemptive task transfers involve transferring a task that is partially executed. This operation is generally expensive, since collecting a task's state (which can be quite large and/or complex) is often difficult. Typically, a task state consists of a virtual memory image, a process control block, unread I/O buffers and messages, file pointers, timers that have been set, etc. Non-preemptive task transfers, on the other hand, involve only the tasks which have not begun execution, and hence do not require transferring the task's state. In both types of transfers, information about the environment in which the task will execute must be transferred to the remote node. This information may include the user's current working directory, the privileges inherited by the task, etc. Non-preemptive task transfers have also been referred to as *task placements*. Detailed discussions on issues in preemptive task transfers can be found in [2, 9].
Centralization: Dynamic load distributing algorithms differ in their degree of centralization. Algorithms may be centralized, hierarchical, fully decentralized, or some combination of these. Algorithms in which some components are centralized are potentially less reliable than fully decentralized algorithms, since the failure of a central component may cause the entire system to fail. A solution to this problem is to maintain redundant components, which can become active when the previously active component fails. A second weakness of centralized algorithms is not so easily cured: A central component is potentially a bottleneck, limiting the amount of load distributing activity that can occur in the system. While hierarchical algorithms can alleviate both of these problems, the complete solution lies in fully decentralized algorithms.

Components of a Load Distributing Algorithm: Typically, a dynamic load distributing algorithm has four components: (1) A *transfer* policy that determines whether a node is in a suitable state to participate in a task transfer either as sender or a receiver of a task. (2) A *selection* policy that determines which task should be transferred. (3) A *location* policy that determines to which node a task selected for transfer should be sent. (4) An *information policy* which is responsible for triggering the collection of system state information.

*Transfer policy:* A large number of the transfer policies that have been proposed are *threshold* policies [11, 12, 26, 36]. Thresholds are expressed in units of load. When a new task originates at a node, the transfer policy decides that the node is a *sender* if the load at that node exceeds a threshold $T_1$. On the other hand, if the load at a
node falls below $T_2$, the transfer policy decides that the node can be a receiver for a remote task. Depending on the algorithm, $T_1$ and $T_2$ may or may not have a same value.

Alternatives to threshold transfer policies include relative transfer policies. Relative policies consider the load of a node in relation to loads at other nodes in the system. For example, a relative policy might consider a node to be a suitable receiver if its load is lower than that of some other node by at least some fixed $\delta$. Alternatively, a node might be considered a receiver if its load is among the lowest in the system. Relative transfer policies require more information than threshold policies, and may thus incur more overhead in gathering that information.

Selection Policy: A selection policy selects a task for transfer, once the transfer policy decides that the node is a sender. Should the selection policy fail to find a suitable task to transfer, the node is no longer considered a sender.

The simplest approach is to select one of the newly originated tasks that caused the node to become a sender. Such a task is relatively cheap to transfer, since the transfer is non-preemptive.

Factors to be considered in selecting a task are: (1) The overhead incurred by the transfer should be minimal. For example, a task of small size carries less overhead. (2) The selected task should be long lived so that it is worthwhile incurring the transfer overhead. (3) The number of location-dependent system calls made by the selected task should be minimal. Location-dependent calls are system calls that must be executed on the node where the task originated, because they use resources such
as windows, clock, or the mouse, that exist only at that node [9, 19].

**Location policy:** The responsibility of a location policy is to find suitable a 'transfer partner' (sender or receiver) for a node, once the transfer policy has decided that the node is a sender or receiver.

A widely used decentralized policy finds a suitable node through *polling*. Under polling, a node polls another node to find out whether it is a suitable node for load sharing [11, 12, 26, 36]. Nodes can be polled either serially or in parallel (e.g., multicast). A node can be selected for polling either randomly [11, 12], based on the information collected during the previous polls [26, 36], or on a nearest-neighbor basis. An alternative to polling is to broadcast a query seeking to find out if any node is available for load sharing.

In a centralized policy, a node always contacts one specified node called a *coordinator* to locate a suitable node for load sharing. The coordinator collects information about the system (which is the responsibility of the information policy) and this information is used by the transfer policy at the coordinator to select receivers.

**Information Policy:** The information policy is responsible for deciding when information about the states of other nodes in the system is to be collected, where it is to be collected from, and what information is collected. Most information policies are one of the following types:

1. Demand-driven: In this decentralized policy, a node collects the state of other nodes only when it becomes either a sender or a receiver, making it a suitable
candidate to initiate load sharing. Note that a demand-driven information policy is inherently a dynamic policy, as its actions depend on the system state.

Demand-driven policies may be sender-initiated, receiver-initiated, or symmetrically-initiated. In *sender-initiated* policies, senders look for receivers to transfer their load. In *receiver-initiated* policies, receivers solicit load from senders. A *symmetrically-initiated* policy is a combination of both where load sharing actions are triggered by the demand for extra processing power or extra work.

2. Periodic: In this class of policy, which may be either centralized or decentralized, information is collected periodically. Based on the information collected, the transfer policy may decide to transfer tasks. Periodic information policies generally do not adapt their rate of activity to the system state. For example, the benefits due to load distributing are minimal at high system loads because most of the nodes in the system are busy. Nevertheless, overheads due to periodic information collection continue to increase the system load and thus worsen the situation.

3. State-change-driven: Under state-change-driven policies, nodes disseminate information about their states whenever their states change by a certain degree. A state-change-driven policy differs from a demand-driven policy in that its goal is to disseminate information about the state of a node, rather than to collect information about other nodes. State-change-driven policies may be either centralized, in which nodes send state information to a centralized collection point, or decentralized, in which nodes send information to peers.
Stability: We first informally describe two views of stability. First, the *queuing theoretic perspective*: When the long-term arrival rate of work to a system is greater than the rate at which the system can perform work, the CPU queues grow without bound. Such a system is termed unstable. For example, consider a load distributing algorithm performing excessive message exchanges to collect state information. The sum of the load due to the external work arriving and the load due to the overhead imposed by the algorithm can become higher than the service capacity of the system, causing system instability.

On the other hand, an algorithm can be stable but still cause a system to perform worse than the system without using the algorithm. Hence, we would like to impose a more restrictive criteria for evaluating algorithms, and we use the *effectiveness* of an algorithm as the evaluating criterion. A load distributing algorithm is said to be effective under a given set of conditions, if it improves performance relative to a system not using load distributing. Note here that, while an effective algorithm cannot be unstable, a stable algorithm can be ineffective.

The *algorithmic perspective*: If an algorithm may perform fruitless actions indefinitely with non-zero probability, the algorithm is said to be unstable. For example, consider *processor thrashing*: The transfer of a task to a receiver may increase the receiver’s queue length to the point of making it overloaded, necessitating the transfer of that task to yet another node. This process may repeat indefinitely. In this case, a task is being moved from one node to another in search of a lightly loaded node without ever receiving any service.
Casavant [5] showed that the existence of algorithmic instability does not necessarily result in poor performance. He showed that processor thrashing is likely to occur at low system loads, but not at moderate and high system loads. Processor thrashing is likely to occur only at low system loads because, as system load increases, the difference in loads at nodes increases and therefore, transfer of tasks is not likely to cause further imbalance to occur. At low system loads, on the other hand, the difference in loads at nodes is small, and hence, the likelihood that a task transfer will cause further load imbalance is much higher. Discussions on various types of algorithmic instability can be found in [5].

2.3 Example Algorithms

During the past decade, a large number of load distributing algorithms have been proposed. In this section, we describe several representative load sharing algorithms that have appeared in the literature. The algorithms selected illustrate how the components of load sharing algorithms fit together, cover a variety of location policies, and bring out their effect on system stability. We discuss the performance of these algorithms in ‘Performance Comparison’ section.

2.3.1 Sender-Initiated Algorithms

Under sender-initiated algorithms, load distributing activity is initiated from an overloaded node (sender) which is trying to send a task to an underloaded node (receiver). The following three simple, yet effective fully-distributed, sender-initiated algorithms
were studied by Eager, Lazowska, and Zahorjan [12].

**Transfer policy:** Each of the algorithms uses the same transfer policy, a threshold policy based on the CPU queue length. A node is identified as a sender, if a new task originating at the node makes the queue length exceed a threshold \( T \). A node identifies itself as a suitable receiver for a task transfer if accepting the task will not cause the node’s queue length to exceed \( T \).

**Selection Policy:** All three algorithms consider only newly-arrived tasks for transfer.

**Location Policy:** These algorithms differ only in their location policy:

*Random:* Random is a simple dynamic location policy which uses no remote state information. A task is simply transferred to a node selected at random, with no information exchange between the nodes to aid in making the decision. A problem with this approach is that useless task transfers can occur when a task is transferred to a node that is already heavily loaded (i.e., its queue length exceeds \( T \)). An issue in this policy is how a node should treat a transferred task. If a transferred task is treated as a new arrival, then it can again be transferred to another node if the local queue length exceeds \( T \). Eager et al. have shown that if such is the case, then irrespective of the average load of the system, the system will eventually enter a state in which the nodes are spending all of their time transferring tasks, and none executing them. A simple solution to this problem is to limit the number of times a task can be transferred. In spite of its simplicity, Eager, et al. found this location policy to provide substantial performance improvements over systems not making use
of load distributing. We will see examples of this ability to improve performance in
'Performance Comparison' section.

![Flowchart](image)

Figure 2: Sender-Initiated Load Sharing Using Threshold Location Policy.

**Threshold:** The problem of 'useless task transfers' can be avoided by polling a node
(selected at random) to determine whether transferring a task would make its queue
length exceed T (see Figure 2). If not, the task is transferred to the selected node
which must execute the task regardless of its state when the task actually arrives.
Otherwise, another node is selected at random and is polled. The number of polls is
limited by a parameter called the *PollLimit* to keep the overhead low. If no suitable
receiver node is found within the PollLimit polls, then the node at which the task
originated must execute the task. By avoiding useless task transfers, Eager. et al.
found that the threshold policy provides a substantial performance improvement over
the random location policy. Again, we will examine this improvement in 'Performance
Comparison' section.

**Shortest:** The two previous approaches make no effort to choose the best destination
node for a task. Under the Shortest location policy, a number of nodes (PollLimit)
are selected at random and are polled to determine their queue length. The node
with the shortest queue length is selected as the destination for task transfer, unless
its queue length ≥ T. The destination node will execute the task regardless of its
queue length at the time of arrival of the transferred task. Eager, et al. found that
the performance improvement obtained by using the shortest location policy over the
threshold policy to be marginal, indicating that using more detailed state information
does not necessarily result in significant improvement in system performance.

**Information Policy:** When either the Shortest or the Threshold location policy is
used, the polling activity is started when the transfer policy identifies a node as the
sender of a task. Hence, the information policy can be considered to be of demand-
driven type.

**Stability:** Sender-initiated algorithms using any of the above three location policies
cause system instability at high system loads. At such loads, no node is likely to
be lightly loaded, and hence the probability that a sender will succeed in finding
a suitable destination node is very low. However, the polling activity in sender-
initiated algorithms increases as the task arrival rate increases, eventually reaching a
point where the cost of load sharing is greater than its benefit. At some more extreme
point, the workload that cannot be offloaded from a node, together with the overhead incurred by polling, exceeds the CPU capacity of the node, and instability results. Thus, the actions of sender-initiated algorithms are not effective at high system loads, and cause system instability, because they fail to adapt to the system state.

2.3.2 Receiver-Initiated Algorithms

In receiver-initiated algorithms, load distributing activity is initiated from an underloaded node (receiver) which is trying to get a task from an overloaded node (sender). In this section, we describe an algorithm studied by Livny and Melman [26] and Eager, Lazowska, and Zahorjan [11] (see Figure 3).

**Transfer Policy**: The transfer policy used is a threshold policy, where the decision is based on the CPU queue length. The transfer policy is triggered when a task departs. If the local queue length falls below the threshold $T$, then the node is identified as a receiver for obtaining a task from a node (sender) to be determined by the location policy. A node is identified to be a sender if its queue length exceeds the threshold $T$.

**Selection Policy**: This algorithm considers all tasks for load distributing, and can make use of any of the approaches discussed in the ‘selection policy’ portion of ‘Issues in Load Distributing’ section.

**Location Policy**: Under this policy, a node selected at random is polled to determine if transferring a task would place its queue length below the threshold level. If not, then the polled node would transfer a task. Otherwise, another node is selected
Figure 3: Receiver-Initiated Load Sharing

at random, and the above procedure is repeated until either a node that can transfer a task (i.e., a sender) is found or a static PollLimit number of tries have failed to find a sender.

A problem with the above location policy is that, if all polls fail to find a sender, then the processing power available at a receiver is completely lost by the system until another task originates locally at the receiver (which may not happen for a long duration). The above problem especially has a severe impact on performance in systems where only a few nodes generate most workload of the system and random polling by receivers can easily miss them. This problem can be easily taken care of as follows: If all the polls fail to find a sender, then the node waits until another task departs or for a predetermined period before reinitiating the load distributing
activity, provided the node is still a receiver[36].

**Information Policy:** The information policy is demand-driven, since the polling activity starts only after a node becomes a receiver.

**Stability:** Receiver-initiated algorithms do not cause system instability because, at high system loads, there is a high probability that a receiver will find a suitable sender to share load within a few polls. Consequently, polls are increasingly effective with increasing system load, and little waste of CPU capacity results.

**A Drawback:** Under the most widely used CPU scheduling disciplines (such as Round-Robin and its variants), a newly-arrived task is quickly provided a quantum of service. In receiver-initiated algorithms, the polling starts when a node becomes a receiver. However, it is unlikely that these polls would be received at senders after new tasks have arrived at them, but before these tasks have begun executing. Consequently, a drawback of this receiver-initiated algorithm is that most transfers are preemptive, and therefore expensive. Sender-initiated algorithms, on the other hand, are able to make greater use of non-preemptive transfers since they can initiate load distributing activity as soon as a new task arrives.

An alternative to this receiver-initiated algorithm is the Reservation algorithm proposed by Eager, et al [11]. Rather than negotiate an immediate transfer, a receiver requests that the next task to arrive be non-preemptively transferred. Upon arrival, the ‘reserved’ task is transferred to the receiver if the receiver is still a receiver at that time. While this algorithm does not require preemptive task transfers, it was found
to perform significantly worse than the sender-initiated algorithms.

2.3.3 Symmetrically-Initiated Algorithms

Under symmetrically-initiated algorithms [20], both senders and receivers initiate load distributing activities for task transfers. These algorithms have the advantages of both sender and receiver initiated algorithms. At low system loads, the sender-initiated component is more successful at finding underloaded nodes. At high system loads, the receiver-initiated component is more successful at finding overloaded nodes. However, these algorithms may also have the disadvantages of both sender and receiver-initiated algorithms. Like sender-initiated algorithms, polling at high system loads may result in system instability. Like receiver-initiated algorithms, a preemptive task transfer facility is necessary.

A simple symmetrically-initiated algorithm can be constructed by combining the transfer and location policies described for sender-initiated and receiver-initiated algorithms.

2.4 Performance Comparison

The objective of this section is to discuss the general performance trends of some of the example algorithms described in the previous section. In addition, we compare their performance with a system which performs no load distributing (i.e., a system composed of n independent M/M/1 systems) and an ideal system which performs perfect load distributing (no unshared states) without incurring any overhead in doing so (i.e., an M/M/K system). The results we present are from simulations of a
distributed system containing 40 nodes, interconnected by a 10 Megabit/second token ring communication network. In this system, task interarrival times and service demands are assumed to be independently exponentially distributed, and the average task CPU service demand is assumed to be one time unit. The size of a polling message is assumed to be 16 bytes, and the CPU overhead to either send or receive a polling message is assumed to be 0.003 time units. A non-preemptive task transfer is assumed to incur a CPU overhead of 0.02 time units, and a preemptive transfer incurs a 0.1 time unit overhead. Transfer overhead is assumed to be divided evenly between the sending and the receiving nodes. The amount of information that must be communicated for a non-preemptive transfer is assumed to be 8 Kbytes, while a preemptive transfer is assumed to require 200 Kbytes. While the specific performance values we present are sensitive to the above assumptions, the performance trends we observe are far less sensitive, and can be expected to hold across a wide range of distributed systems. Errors in the results we present are less than 5% at 90% confident level.

The notation used in Figure 4 corresponds to the algorithms as follows: RECV: receiver-initiated algorithm; RANDOM: sender-initiated algorithm with random location policy, assuming that a task can be transferred at most once; SEND: sender-initiated algorithm with threshold location policy; SYM: symmetrically-initiated algorithm (SEND and RECV combined); M/M/1: a distributed system that performs no load distributing; M/M/K: a distributed system that performs ideal load distributing without incurring overhead for load distributing. A fixed threshold of \( T=UT=LT=1 \)
was used for each of the algorithms.

For these comparisons, a small fixed PollLimit (5) was assumed. We can see why such a small limit is sufficient by noting that, if $P$ is the probability that a particular node is below threshold, then the probability that a node below threshold is first encountered on the $i^{th}$ poll is $P(1-P)^{i-1}$ [12]. For large $P$, this expression decreases rapidly with increasing $i$; however, as $P$ is large, the probability of succeeding on the first few polls is high. For small $P$, the quantity decreases more slowly. However, since most nodes are above threshold, the improvement in system-wide response time

---

1This result assumes that nodes are independent, which is valid if the poll limit is small relative to the number of nodes in the system.
that will result from locating a node below threshold is small; quitting the search after the first few polls does not carry a substantial penalty.

**Main Result:** The ability of load distributing to improve performance is intuitive when work arrives at some nodes at a greater rate than others, or when some nodes have faster processors than others. Performance advantages are not so obvious when all nodes are equally powerful and have equal workloads, over the long term. Figure 4 plots the average task response time versus offered system load for such a *homogeneous* system under each of the above load distributing algorithms. By comparing M/M/1 with RANDOM, one can observe that, even this simple load distributing scheme provides a substantial performance improvement over a system that does not employ load distributing. Considerable further improvement in performance can be gained through simple sender-initiated and receiver-initiated load sharing schemes. The best performance is obtained under M/M/K, though this optimistic lower bound can never be reached, since it assumes no load distribution overhead.

**Receiver-initiated vs. Sender-initiated Load Sharing:** One can observe from Figure 4 that the sender-initiated algorithm (SEND) performs marginally better than the receiver-initiated algorithm (RECV) at light to moderate system loads, while the receiver-initiated algorithm performs substantially better at high system loads (even though the preemptive transfers used in RECV are much more expensive than the non-preemptive transfers used by SEND). Receiver-initiated load sharing is less effective at low system loads because load sharing is not initiated at the time that one of the few nodes becomes a sender, and thus load sharing often occurs late.
Regarding the robustness of the policies, the Receiver-initiated policy has an edge over the Sender-initiated policies. The Receiver-initiated policy performs acceptably over the entire system load spectrum, whereas the sender-initiated policy requires an adaptive location policy to perform acceptably at high loads. At such loads, the receiver-initiated policy maintains system stability because its polls generally find busy nodes, while polls due to the sender-initiated policy are generally ineffective and waste resources in efforts to find underloaded nodes.

Symmetrically-Initiated Load Sharing: This policy takes advantage of its sender-initiated load sharing component at low system loads, its receiver-initiated component at high system loads, and both at moderate system loads. Hence, its performance is better or matches that of the sender-initiated policy at all levels of system load, and is better than that of receiver-initiated policy at low to moderate system loads. Nevertheless, this policy also causes system instability at high system loads, due to the ineffective polling activity of its sender-initiated component at such loads.

2.5 Real-time Systems

Distributed real-time systems have been proposed for real-life applications such as control systems [10] and industrial process control [17, 34]. In such systems, tasks have deadline constraints associated with them, and it is critical that they finish by their deadlines. Real-time systems have been further classified as hard real-time and soft real-time systems. In hard real-time systems, results generated by tasks that have missed deadlines have no value. In soft real-time systems, on the other hand,
results generated by tasks that have missed deadlines have depreciated value after their deadlines. In other words, they can give higher priority to those tasks that can meet their deadlines and thus, maximize the number of tasks that meet their deadlines while not suffering from critical failure due to some tasks missing their deadlines [6].

In real-time distributed systems, a global scheduling algorithm minimizes the number of tasks that miss deadlines through load distributing by taking advantage of the statistical fluctuations in the load at different nodes. Ramamritham and Stankovic [32, 33] are the first to propose distributed scheduling for hard real-time systems. Distributed scheduling schemes for soft real-time systems can be found in [6, 22].

2.6 Priority Systems

Priority systems encounter tasks with different priorities which indicate their criticalness or importance. For example, distributed systems have been proposed for real-time applications such as control systems, industrial process control, automated manufacturing, air-traffic control, program trading, etc [38]. In these systems, it is important that certain critical tasks complete within certain timelimits. These tasks can be viewed as high priority tasks which must get better service than other type of tasks. Moreover, in a general computing facility, certain users/tasks may receive higher priority than others because of project deadlines [28], importance of the tasks [28], the user is paying higher fees for better service, etc. Therefore, priority systems can be considered as a special class of general systems or real-time systems. So far, only Chang [6] has proposed distributed scheduling schemes for priority systems.
The major drawback of his schemes are that they allow only two priority classes and therefore are not general.

2.7 Summary

Over the past decade, the mode of computing has shifted from mainframes to networks of computers, which are often engineering workstations. Such a network promises higher performance, better reliability, and improved extensibility over mainframe systems. The total computing capacity of such a network can be enormous. However, to realize the performance potential of such a system, a good load distributing scheme is essential.

Load distributing algorithms are broadly classified as static and dynamic algorithms depending on whether they make use of system state information. Dynamic algorithms encompass a special class, namely adaptive algorithms. Adaptive algorithms adapt their activities by dynamically changing the parameters of the algorithm, or even the policy itself, to suit the changing system state.

Load distributing algorithms are further classified as sender-initiated, receiver-initiated, and symmetrically-initiated algorithms. Under sender-initiated algorithms, load distributing activity is initiated from an overloaded node (sender) which is trying to send a task to an underloaded node (receiver). In receiver-initiated algorithms, load distributing activity is initiated from an underloaded node (receiver) which is trying to get a task from an overloaded node (sender). Under symmetrically-initiated algorithms [20], both senders and receivers initiate load distributing activities for task transfers.
Typically, a dynamic load distributing algorithm has four components: (1) A transfer policy that determines whether a node is in a suitable state to participate in a task transfer either as sender or a receiver of a task. (2) A selection policy that determines which task should be transferred. (3) A location policy that determines to which node a task selected for transfer should be sent. (4) An information policy which is responsible for triggering the collection of system state information.

The goal of load distributing algorithms depends on the nature of the system employing these algorithms. In the case of general systems, minimizing the average response time is often the goal of load distributing. In real-time systems, tasks are associated with deadlines. Tasks must complete before their deadlines expire. Therefore, in real-time systems, maximizing the number of tasks that complete within their deadlines is the goal. In systems that allow multiple priorities for tasks, providing better service to higher priority tasks is the goal.
3.1 Introduction

Over the past several years, global scheduling has been shown to be an important part of job scheduling for distributed systems [4, 13]. The goal of global scheduling is to improve the performance of a distributed system by appropriately transferring work from heavily loaded computers to idle or lightly loaded computers. Many studies, including [9], [12], [20], and [26], have shown global scheduling to have considerable potential to improve performance.

Two important components of a global scheduling algorithm are its transfer policy and its location policy. The transfer policy at a node determines whether a task should be transferred either to it or from it, by identifying the node as a suitable sender of a task, a suitable receiver, or neither. The location policy at a node, on the other hand, determines where a task should be transferred. Location policies can be broadly characterized as sender-initiated (in which receivers are found for senders), receiver-initiated (in which senders are found for receivers), and symmetrically-initiated (in which complementary nodes are found for both senders and receivers).
In searching for a complementary node for a sender or a receiver, many location policies probe nodes to determine their current status. In an analytic study of homogeneous systems (identical nodes having identical workload characteristics) that assumed the processor cost of probing to be negligible, Eager, Lazowska, and Zahorjan [11] found that the type of location policy that performs best depends on the workload. Their receiver-initiated policy performed best at high system loads, since at such loads, busy nodes that can transfer tasks are easily found. On the other hand, their sender-initiated policy outperformed their receiver-initiated policy at low to medium system loads, since at such loads, idle nodes that can accept work are easily found. Both policies used a simple method of probing, in which nodes were probed randomly until a complementary node was found, or until Probelimit (a parameter of the algorithm) nodes had been probed.

Krueger [18], in a comparison of sender, receiver, and symmetrically-initiated policies, showed that the situation is more serious when the processor cost of probing is taken into consideration. At high system loads, both sender-initiated and symmetrically-initiated policies cause system instability for the following reason: At such loads, few nodes in the system are likely to be lightly loaded (i.e., good receivers), and hence the probability that an overloaded node (sender) will succeed in finding a receiver is very low. However, the probing activity due to sender-initiated searches increases as the task arrival rate increases, eventually reaching a point where the cost of probing is greater than its benefit. At some more extreme point, the workload that cannot be offloaded from a node, together with the overhead incurred by probing,
exceeds the CPU capacity of the node, and instability results. Clearly, the cause of
this instability is the failure of the location policy to adapt its activity to the global
system load (even though the actions may be justified according to the local load at
the individual nodes). Indiscriminate probing wastes computing capacity at times
when such waste cannot be afforded.

We will illustrate a second problem that occurs when workload arrival rates are
not homogeneous. Measurement studies, including those by Mutka and Livny [27]
and by Krueger and Chawla [19], have found such workloads to be common in existing
systems. We show that, under such workloads, inefficient probing (as occurs when
probing is random) considerably reduces the range of conditions under which global
scheduling can deliver acceptable performance.

The goal of this study is to develop location policies that capture the advantages
of both receiver-initiated and sender-initiated location policies, that probe efficiently,
and that are simple enough to be practical. We propose two adaptive policies: a
symmetrically-initiated policy that is suitable for systems that have the capability
both to transfer new tasks and to migrate partly-executed tasks, and a sender-initiated
policy that is suitable for systems where only newly-arrived tasks can be transferred.
Two key features of our policies are: First, to allow adaptivity, the proposed policies
use data gathered during probing which is normally \textit{discarded} by non-adaptive poli-
ties; no extra messages are required to carry system state information. Second, the
proposed location policies are general, and can be used with a broad range of transfer
policies.
Using simulation, we show the following:

- Our symmetrically-initiated adaptive location policy performs as well or better than non-adaptive policies and maintains system stability across all system conditions examined.

- Our sender-initiated adaptive location policy outperforms a non-adaptive sender-initiated location policy under all conditions examined and considerably increases the range of conditions under which system stability can be maintained.

The remainder of the paper is organized as follows: In the next section, we present the system model that we have assumed. Section 3.3 presents our symmetrically-initiated adaptive location policy. We evaluate the performance of this location policy, by comparing it with three previously-studied non-adaptive location policies, in Section 3.4. In Section 3.5, we present a sender-initiated adaptive location policy and evaluate its performance. Section 3.6 summarizes results and presents conclusions.

3.2 System Model

The class of distributed systems that we address in this study is characterized as a collection of autonomous general-purpose computers which are fully connected (eg. Ethernet, token-ring, etc.). As is common in modern systems, we assume that individual nodes use a timesharing local scheduler that provides an initial burst of service to newly-arrived tasks. Tasks are assumed to arrive independently at individual nodes and to execute independently. Finally, we assume that no deterministic a priori information about task resource requirements is available.
3.3 Symmetrically-Initiated Adaptive Location Policy

In this section, we propose a symmetrically-initiated location policy that adapts to the overall system load, provides better probing efficiency than under random probing and reduces global scheduling overhead, yet is quite simple.

Combining the receiver-initiated and sender-initiated approaches, the purpose of a symmetrically-initiated location policy is to find a lightly loaded node (receiver) when a heavily loaded node (sender) wishes to transfer tasks, and to find a sender when a receiver wishes to acquire tasks. Our adaptive policy seeks to accomplish these goals at least as successfully as the best non-adaptive policy for a given system load, and do so without threatening the stability of the system. The proposed location policy is initiated by the transfer policy and returns the first suitable node found to the transfer policy. (This location policy can be easily extended to find the best complementary node, such as the receiver with shortest queue length or fastest CPU, the sender with longest queue length, etc.)

The cause of instability in a location policy is ineffective sender-initiated probing. Our goal is to guide probes towards the nodes that are most likely to be suitable for load sharing without incurring significant additional overhead. To accomplish this objective, our location policy utilizes the information gathered by probing (instead of discarding it as was done by previous policies) to keep track of the recent state of each node in the system. A node's state (either 'sender', 'receiver', or neither('OK')) is identified by its transfer policy. When some other node's location policy probes that node, this state is provided as the response. This state information is stored in a data
structure at the probing node (such a data structure is maintained independently at each node). Using this information, the location policy decides which nodes to send probing messages to and in what order, thus avoiding indiscriminate and ineffective probing. The data structure at a node changes with the state of the system, as the node receives replies to its probes, as well as probes from other nodes. We will show that maintaining this data structure requires negligible overhead.

3.3.1 Data Structure at a Node

The information maintained at a node $i$ consists of three ordered lists. The first list, $Slist_i$, contains the IDs of nodes that have identified themselves (either in a probe or in a reply to a probe) to node $i$ as potential senders of tasks. The second list, $Rlist_i$, contains the IDs of nodes that have identified themselves as potential receivers. Finally, $OKlist_i$, contains the IDs of nodes that are neither senders nor receivers.

Initialization: Initially all nodes assume that every other node is idle, and therefore is a potential receiver. For a node $i$ in a system containing $n$ nodes: $Rlist_i = i + 1, i + 2, \ldots, n, 1, \ldots, i - 1$; $OKlist_i = \text{null}$; $Slist_i = \text{null}$. This order is not required for correctness, but it helps in initially dispersing negotiation activity (probing and replying) among the nodes.
3.3.2 Sender-Initiated Negotiation

When the transfer policy at node \(i\) determines that node \(i\) is a sender, the location policy at node \(i\) does the following to find a suitable receiver:

1. Probe the node at the head of \(Rlist_i\), say \(j\), to determine whether \(j\) is a receiver.

2. If \(j\) replies that it is a receiver, return its ID to the transfer policy and STOP. If \(j\) is not a receiver, remove it from \(Rlist_i\) and add it to the head of either \(Slist_i\) if \(j\) is a sender, or \(OKlist_i\) if \(j\) is OK.

3. STOP negotiation and return FAILURE to the transfer policy if either (a) \(i\) has probed \(Probelimit\) nodes (where \(Probelimit\) is a parameter of the algorithm) without success, (b) \(Rlist_i\) is empty, or (c) \(i\) is no longer a potential sender (due to task completion at the node). Otherwise go to Step 1.

At probed node \(j\):

1. Since this probe has identified node \(i\) as a sender, remove \(i\) from whatever list it is in and add it to the head of \(Slist_j\). (Note: If \(i\) was already in \(Slist_j\), this action simply changes its position in the list thereby reflecting the system state more accurately.)

2. Send a reply message to \(i\) indicating whether \(j\) is a receiver, sender, or OK (determined by the transfer policy at node \(j\)).

As we will discuss shortly (Section 3.3.4), no list search is necessary to remove a node from whatever list it is in; the data structure used by this algorithm allows this
operation to be done in constant time.

### 3.3.3 Receiver-Initiated Negotiation

To find a suitable sender, the location policy at a receiver node $i$ does the following:

1. Probe a selected node, say $j$, to determine whether $j$ is a sender. The selected nodes are chosen in the following order:

   (a) From the head of $Slist_i$ (most up-to-date information is used first).

   (b) If $Slist_i$ is empty, then from the tail of $OKlist_i$ (most out-of-date information is used first in the hope that the state has changed from OK to sender).

   (c) If $OKlist_i$ is also empty, from the tail of $Rlist_i$ (again the most out-of-date information is used first).

Note that the contents of the lists may change as negotiation proceeds, because of negotiations initiated by other nodes occurring concurrently with this negotiation session. To avoid probing a node that has indicated that it is not a sender very recently, nodes that join $OKlist_i$ or $Rlist_i$ after a session has begun are not probed in that session.

2. If $j$ replies that it is a sender and that it will remain a sender after transferring a task, add it to the head of $Slist_i$, return its ID to the transfer policy, and STOP.

   If $j$ replies that it is a sender but will not remain a sender after transferring a task, remove it from whatever list it is in and add it to the head of $OKList_i$, 


return its ID to the transfer policy, and STOP. If j replies that it is either a receiver or OK, remove it from whatever list it is in and add it to the head of either Rlist_i (if it is a receiver), or OKList_i. Again, as we discuss shortly, the overhead required for these list operations is small and independent of the number of nodes in the system.

3. STOP negotiation and return FAILURE to the transfer policy if either i is no longer a receiver (due to a task arrival), if i has probed Probelimit nodes without success, or if all the nodes that were in OKList_i and Rlist_i have been probed.

At probed node j:

1. If j is not a sender, return a message informing i of j’s state (receiver or OK).
   Since i’s probe has identified i as a receiver, remove i from whatever list it is in and add it to the head of Rlist_j.

2. If j is a sender, migrate a task to i. Inform i of j’s state after the migration.

3.3.4 An Efficient Scheme for Maintaining the Lists

Our adaptive policy requires list manipulation with the receipt of nearly every message. The following is a scheme which will efficiently manipulate the lists even for distributed systems containing very large number of nodes. At any moment, each node is in a single state: either ‘sender’, ‘receiver’, or ‘OK’. Hence a node is in only one list at any time. We make use of an array of size n (where n is the number of nodes in the system) to store each node’s state, along with ‘next’ and ‘previous’
pointers to represent its position in a list. Figure 5 illustrates this array. Since this array can be randomly accessed by node ID, and because each node record indicates that node's state, it is not necessary to traverse the lists to find which list a node belongs to. It is also not necessary to traverse a list in order to move a node to the head of the list, or to move a node from one list to another. The node's record can be accessed directly by its ID, and its pointers (as well those of its neighbors in the list) modified to reflect its new position. Overhead due to each list-manipulation action is constant and very small; it is independent of the number of nodes in the distributed system.
3.3.5 Discussion

Whenever a node receives a probe or a reply, it updates its lists, thereby updating its knowledge of the global system state. By selecting nodes at the head or tail of appropriate lists for probing, our location policy:

- Makes use of its knowledge of the system state to guide searches towards the nodes suitable for load sharing.
- Exploits the currency of the system state as known to an individual node by using the most recent (and hence most accurate) information first.

By using this state information, our location policy is able to curtail sender-initiated probing at high system loads, avoiding system instability. To see how this occurs, note that a node is placed in another node’s $Rlist$ only when it probes that node asking for a task transfer and no transfer happens. At high system loads, such occurrences are rare since (1) few nodes are receivers, so there are few receiver-initiated probes and (2) many of the receiver-initiated probes that do occur result in task transfers, since many nodes are senders. As a result, the $Rlists$ empty as the system load becomes high. Since sender-initiated probes are sent only to nodes found in $Rlists$, sender-initiated probing is cut off at sufficiently high system loads, and the system remains stable.

In contrast to sender-initiated probing, receiver-initiated probing is not cut off at low system loads. At such loads, excess processing capacity is available, so instability is not a danger. Furthermore, the receiver-initiated probing has the beneficial effect
of updating the $Rlists$ at the probed nodes, allowing receivers to quickly be found if some node becomes overloaded or if the system load conditions change.

The state information gathered by our location policy is also particularly useful when workload arrivals are heterogeneous. For example, consider a system in which a very few nodes are heavily overloaded, while the remainder of the system is lightly loaded. Under random probing, the heavily-loaded nodes would be difficult to find and would often be missed. Under our location policy, however, the information contained in the $Slists$ guides probes to these nodes; because the overloaded nodes send more sender-initiated probes than other nodes, they tend to be found toward the heads of the $Slists$. A similar situation occurs when there are a few nodes having no workload arrivals; such nodes are easily located by the senders.

Under all other workload conditions, we can also expect our location policy to result in improved performance. Since probes are guided to the nodes that are most likely to be suitable counterparts, the time required to locate a counterpart is reduced and the likelihood of successfully finding a counterpart is increased.

3.4 Performance Comparison

In an earlier study, Eager, Lazowska, and Zahorjan [11] compared the performance of sender-initiated and receiver-initiated global scheduling algorithms. These algorithms share the same transfer policy, but differ in their location policies. Using analysis, the relative performance of these algorithms was shown to depend on the system load. We will show that relative performance also depends on the level of heterogeneity in the workload. In this section, we replace the location policies used
in these algorithms with the symmetrically-initiated adaptive policy described in the previous section, but retain their transfer policy. Using simulation, we then compare the performance of this new adaptive algorithm with those of the original non-adaptive algorithms. For completeness, we also include a non-adaptive symmetrically-initiated algorithm in our comparison, which uses a composite of Eager, et al's sender-initiated and receiver-initiated location policies.

We begin by describing the algorithms that will be studied in this section. Following these descriptions, we present the results of our performance study.

3.4.1 Global Scheduling Algorithms

All the algorithms studied in this section share the same transfer policy: A node identifies itself as a suitable sender whenever its queue length exceeds some threshold $T$, as a receiver whenever its queue length is less than $T$, and as OK whenever its queue length $= T$.

The three non-adaptive algorithms we consider have similar location policies: Nodes are probed randomly until either a complementary node is found (a receiver if the initiator is a sender, or a sender if the initiator is a receiver) or Probelimit probes have completed without a complementary node being found, where Probelimit is a parameter of the algorithm. These location policies differ only in the conditions under which a search for a complementary node is initiated. The sender-initiated policy, used by the SEND algorithm, initiates a search only when a task originating at the node leaves it in the sender state. Conversely, the receiver-initiated policy, used by the RECV algorithm, begins searching only when the completion of a task leaves the
node in the receiver state. For this study, we have modified the RECV algorithm slightly. As originally presented [11], RECV is subject to failure when workload arrivals are not homogeneous, particularly when some nodes have no arrivals for long periods of time (as often occurs in practice [19, 23]). Such a node, when it initially becomes idle, will fail to find a suitable sender within $Probelimit$ probes with non-zero probability. After such a failure, the node will remain idle until tasks are once again initiated at the node. To avoid this problem, we extend the RECV location policy so that, if a search for a sender fails, the search is re-initiated after a timeout period ($Receiver$-$timeout$). Combining these location policies, we also consider a symmetrically-initiated policy, used by the SYM algorithm, which begins its search whenever either of the above events occurs.

We compare the performance of these three non-adaptive algorithms with that of an adaptive algorithm (ADAPT-SYM), constructed by replacing the location policy of the SYM algorithm with the adaptive location policy described in section 3.3.

3.4.2 Results

The simulation results presented in this section are based on the assumption that the distributed system contains 40 functionally identical nodes, interconnected by a 10 Megabit/second token ring communication network. In this system, task interarrival times and service demands are assumed to be independently exponentially distributed, and the average task CPU service demand is assumed to be one time unit. The size of a polling message is assumed to be 16 bytes, and the CPU overhead to either send or receive a polling message is assumed to be 0.003 time units. A non-preemptive task
transfer is assumed to incur a CPU overhead of 0.02 time units, and a preemptive transfer incurs a 0.1 time unit overhead. Transfer overhead is assumed to be divided evenly between the sending and the receiving nodes. Since we assume local CPU scheduling to follow a timeslicing discipline, with newly-arrived tasks quickly given a burst of service, receiver-initiated transfers are assumed to be preemptive (since it is unlikely that a receiver would probe a node at nearly the moment that a new task arrived), and sender-initiated transfers are assumed to be non-preemptive (sender-initiated policies can easily select newly-arrived tasks for transfer, since such policies are triggered by the arrivals of new tasks). For the simulation results, timeshared local scheduling is approximated by the Processor Sharing [15] discipline. The amount of information that must be communicated for a non-preemptive transfer is assumed to be 8 Kbytes, while a preemptive transfer is assumed to require 200 Kbytes.

While the specific performance values we present are sensitive to the above assumptions, the performance trends we observe are far less sensitive, and can be expected to hold across a wide range of distributed systems. Errors in the simulation results we present are less than 4% at the 90% confidence level.

**Heterogeneity:** Several studies, including those for the Condor [23] and Stealth [19] projects, have found heterogeneous workload arrivals to be typical in many existing distributed systems; homogeneous workload arrivals appear to be rare. Of the algorithms we have studied (see Figure 6), RECV is the least tolerant of heterogeneous workloads under our assumptions whose performance degrades significantly when
NLG exceeds 5 nodes. The reason RECV performs poorly under such workloads is that its random method of probing is inefficient at finding a sender when only a few nodes are senders. Consequently, RECV does not support a task transfer rate high enough to maintain stability at the LG nodes.

SEND is slightly more tolerant of heterogeneous workloads. Similar to RECV, SEND's random probing mechanism is too inefficient at finding receivers to support a sufficiently high task transfer rate to maintain stability when the task arrival rate is high.

Of the non-adaptive algorithms, SYM is most tolerant of heterogeneity, whose performance degrades significantly when NLG exceeds 25 nodes. With both senders and receivers searching for partners, the task transfer rate is increased, maintaining stability for a broader range of conditions than RECV or SEND. At sufficiently high levels of heterogeneity, however, where a very high transfer rate is necessary to maintain stability, SYM's inefficient probing fails to achieve a sufficiently high rate, and instability results.

From Figure 6, we observe that only ADAPT-SYM maintains system stability under all heterogeneous conditions. In fact, in contrast to the non-adaptive algorithms, the performance of ADAPT-SYM improves under such conditions. The reason for this improvement is that, since no tasks arrive at NLG nodes except as transfers, these nodes are always in either the receiver state or the OK state, never in the sender state. Consequently, only the LG nodes appear in the Slists. Because all the system load originates at these few nodes, they rarely change state, and are nearly always
in the sender state. As a result, the states stored in the lists reflect reality quite accurately. Hence, nodes are more successful at finding complementary nodes, and do so with fewer tries. As the number of LG nodes decreases (increasing heterogeneity), ADAPT-SYM remains stable and its performance actually improves.

In summary, ADAPT-SYM both performs better than the best of the non-adaptive algorithms and maintains system stability across the range of workload arrival heterogeneity. As the level of heterogeneity increases, ADAPT-SYM's performance improves, rather than degrades. Consequently, ADAPT-SYM is able to take advantage of the workload heterogeneity present in many practical distributed systems.
Offered System Load: Figure 7 plots simulation results for mean task response time against offered system load. To compare our results with those of previous studies [11], these results assume homogeneous workload arrivals. From this figure, we can see that ADAPT-SYM performs as well or better than the best of the non-adaptive algorithms across the range of system loads. Furthermore, ADAPT-SYM maintains system stability, regardless of the load.

From Figure 7, we see that RECV maintains stability in a homogeneous system under all system loads, but both SEND and SYM cause instability at high offered loads.
At high loads, both SEND and SYM become unstable for the following reason: At such loads, few nodes in the system are likely to be receivers, hence the probability that an overloaded node will succeed in finding a receiver is low. However, the level of sender-initiated search activity increases as the task arrival rate increases, since most nodes become heavily loaded, eventually reaching a point where the cost of load sharing is greater than its benefit. At some more extreme point, the workload that cannot be offloaded from a node, together with the overhead incurred by polling, exceeds the CPU capacity of the node, and instability results.

Under high loads, however, RECV remains stable. Under such load conditions, a potential receiver can easily find a sender, since many nodes are overloaded. This results in productive receiver-initiated probes and very little waste of scarce CPU capacity.

For ADAPT-SYM at high loads, an initial high rate of unsuccessful sender-initiated probes results in the removal of nodes from the Rlists. Unless unsuccessful receiver-initiated probes of these nodes occur, which are unlikely at such loads, the Rlists remain empty, thus preventing future sender-initiated probes. As a result, sender-initiated probing is cut off at high loads and only receiver-initiated probing is performed, which does not lead to instability.

At low system loads, a potential sending node has little trouble finding a receiver, since many nodes are idle. A potential receiver, however, is unlikely to find a sender, since few nodes are overloaded. Even though many receivers exist that could give immediate service, a task whose arrival causes a node to become overloaded may wait
considerable time before a receiver finds it and transfers it. As a result, SEND and
SYM both perform better than RECV at low loads.

For ADAPT-SYM at low loads, receiver-initiated negotiation generally fails, but
has the positive effect of updating the *Rlists* at other nodes. These failures do not
adversely affect performance, since extra processing capacity is available at low system
loads. Since the *Rlists* reflect the system's state fairly accurately, sender-initiated
negotiation generally succeeds with very few probes.

At intermediate loads, both potential senders and potential receivers are able to
find partners, so both SYM and ADAPT-SYM perform better than either SEND
or RECV. Because ADAPT-SYM probes more accurately and more efficiently than
SYM, it also provides better performance.

In summary, ADAPT-SYM performs as well or better than any of the non-adaptive
algorithms, and maintains stability across the spectrum of system loads. ADAPT-
SYM achieves this performance through more efficient probing, and by relying on
sender-initiated negotiation at low loads, receiver-initiated negotiation at high loads,
and both at moderate loads.

**Probelimit:** Figure 8 plots simulation results for mean response time against *Probelimit* for a fixed offered system load of 0.75, again assuming homogeneous workload
arrivals. This figure shows that ADAPT-SYM has an optimal value of 4 for *Probelimit*,
beyond which the probes cause wasteful overhead. All the non-adaptive algorithms
require more probes than this to achieve their best performance (6 for SYM, and
more than 10 for RECV and SEND), resulting in transfer delays and greater overhead. The fact that increasing ADAPT-SYM's Probelimit from 2 to 4 results in only marginal benefit shows that its lists accurately reflect the system state. Because most of its scheduling attempts are successful within 2 probes, overhead and network traffic due to probes are low, allowing ADAPT-SYM to outperform all the non-adaptive algorithms.

Summary: In summary, the algorithm using our adaptive location policy performs at least as well as or better than the best of the non-adaptive policies, and remains stable across all system loads and all level of workload heterogeneity. Network utilization by the adaptive policy is low relative to the non-adaptive policies, because it succeeds
in finding suitable counterparts with fewer probes.

3.5 Sender-Initiated Adaptive Location Policy

Transferring a task that has already begun execution is likely to be much more expensive than transferring a newly-arrived task, because the task state, which must accompany it to its new node becomes much more complex after execution begins [2, 9, 39, 31]. For some systems, gathering the state of a task that is executing may be extremely difficult. As a result, a global scheduler may be considerably simplified if it transfers only newly-arrived tasks.

Under our assumption that local CPU scheduling gives an early burst of service to newly-arrived tasks (as is typical in practical timesharing systems), receiver-initiated task transfers nearly always involve tasks that are executing, since it is unlikely that a receiver would probe a sender after a new task arrival at the sender, but before it began executing. Sender-initiated location policies, however, allow newly-arrived tasks to be transferred, since such location policies are triggered by new arrivals.

In this section, we propose a new location policy that can be used in systems that are unable to migrate partly-executed tasks, and that is stable under a very wide range of conditions.

Our approach recognizes that, since the same location policy is used at all nodes, the list-manipulating actions that take place as the result of probes and their replies are common knowledge. This common knowledge is exploited in the following way to attract future task transfers from senders: each node keeps track of which list it belongs to at other nodes. This knowledge requires no additional message exchanges.
When a node becomes a receiver, other nodes must be informed. To avoid excessive and wasteful messages, a new receiver informs only those nodes whose Rlists do not contain it.

In describing our sender-initiated adaptive location policy, since it is very similar to the symmetrically-initiated policy of the previous section, we point out only the differences.

### 3.5.1 Data Structure at a Node

In addition to the data structure described in section 3.3.1, each node maintains a state vector (SV) of size \( n \) (where \( n \) is the number of nodes in the system), whose content implies the following knowledge:

\[
SV_i[j] = \begin{cases} 
0 & \text{node } i \text{ knows that } i \in Rlist_j \\
1 & \text{node } i \text{ knows that, } i \in OKlist_j \\
2 & \text{node } i \text{ knows that, } i \in Slist_j 
\end{cases}
\]

### 3.5.2 Initialization

Initialization is identical to that of section 3.3.1 with the following addition: At each node, all entries in \( SV \) are initially are set to 0, to indicate that each node initially belongs to the Rlists of all other nodes.

### 3.5.3 Sender-Initiated Negotiation

Sender-initiated negotiation is identical to that of Section 3.3.2 with the following additional steps:
At sender node $i$: After Step 1 from Section 3.3.2, $SV_i[j]$ is set to value of 2, i.e., $i$ knows that it is a member of node $j$'s $Slist$.

At probed node $j$: Carry out the step below after Step 2 of section 3.3.2.

$$SV_j[i] = \begin{cases} 
0 & \text{if $j$ replied that it is a receiver.} \\
1 & \text{if $j$ replied that it is a OK.} \\
2 & \text{if $j$ replied that it is a sender.}
\end{cases}$$

3.5.4 Receiver-Initiated Negotiation

In order to avoid excessive state-change messages, a node on becoming a receiver informs only those nodes which are now misinformed about its state.

At receiver node $i$:

$$\forall j, \ i \neq j \text{ do, if } SV_i[j] \neq 0, \text{ then inform } j \text{ that } i \text{ is a potential receiver, and set } SV_i[j] \text{ to zero.}$$

At informed node $j$:

On receipt of state-change message from $i$, remove $i$ from whatever list it is in, and add $i$ at the head of $Rlist_j$ (Note: If $i$ was already present in $Rlist_j$, this action simply changes its position in the list).

Note that there is no migration of partly-executed task. Probing is initiated only on the arrival of a new task.

3.5.5 Performance Comparison

To compare our new policy with non-adaptive location policies, we construct an adaptive algorithm (ADAPT-SEND) by replacing the location policy of SYM from
Figure 9: Mean Response Time of SEND and ADAPT-SEND vs. Number of NLG nodes. Offered System Load = 0.85, Probelimit = 5, Threshold = 1, Mean Service Time = 1

section 3.4.1 with this sender-initiated adaptive location policy. The assumptions of this section are identical to those of section 3.4.2.

**Heterogeneity:** Figure 9 plots mean response time against the number of non-load-generating nodes at an offered system load of 0.85. This figure shows that, ADAPT-SEND is stable at higher levels of heterogeneity than SEND, but becomes unstable under extreme conditions. The reason for this instability is the absence of migration of partly-executed tasks. This constraint can result in a sender remaining a sender for an extended duration if its initial probing fails to find a receiver, even though the sender learns about potential receivers immediately after probing has failed. As
a result, a transfer rate high enough to offload all the excess work arriving at load generating nodes cannot be maintained, resulting in instability.

**Offered System Load:** Figure 10 plots mean response time against offered system load under homogeneous workload arrivals. ADAPT-SEND is stable across all system loads for the same reasons that ADAPT-SYM remains stable (discussed in section 3.4.2). Receiver-initiated state-change messages update the Rlists at all misinformed nodes, thereby accurately reflecting the system state. These accurate Rlists help senders locate receivers within a few tries. On the other hand, random probing by the non-adaptive location policy of SEND may fail to locate a receiver even when one exists, or may need a large number of probes to locate a suitable receiver. The quick success in finding receivers by its adaptive location policy explains why ADAPT-SEND performs better than SEND.

The poorer performance of ADAPT-SEND compared to ADAPT-SYM at system loads greater than 0.85 occurs because a sender can transfer a task only upon the arrival of a new task. As a result a sender may remain in the sender state for an extended duration, even after learning about a potential receiver.

**Probelimit:** Figure 11 plots mean response time against Probelimit at a constant offered system load of 0.75 under homogeneous workload arrivals. This figure shows that the performance of ADAPT-SEND improves with increasing Probelimit, with decreasing benefits for Probelimits beyond 5. This result indicates that
ADAPT-SEND probes more efficiently than SEND, though not as efficiently as ADAPT-SYM (see figure 8). The improved efficiency relative to SEND allows ADAPT-SEND to achieve better performance.

3.6 Summary and Conclusions

Ineffective negotiations hurt the performance of a distributed system. In fact, when workload arrival rates are heterogeneous (as is common in practice) or when the system load is high, we have shown that inefficient and ineffective probing can result in system instability. In this paper, we have presented two adaptive location policies for global scheduling algorithms. By increasing the efficiency of negotiation,
we have found that these policies both improve performance and reduce the threat of system instability relative to non-adaptive policies, yet add little complexity to global scheduling. A key feature of the proposed policies is that they are general, and can be used with a broad range of transfer policies.

The symmetrically-initiated adaptive location policy presented maintains stability and out-performs all the non-adaptive policies studied at all system loads under both homogeneous and heterogeneous workloads. The sender-initiated adaptive location policy performs better than its non-adaptive counterpart under all conditions studied. A limitation of the sender-initiated adaptive policy is that it is not stable under extreme workload heterogeneity. While not matching the performance of the

---

Figure 11: Mean Response Time of SEND and ADAPT-SEND vs. Probelimit. Offered System Load = 0.75, Threshold = 1, Mean Service Time = 1
symmetrically-initiated adaptive policy, the sender-initiated policy has the advantage of not requiring transfers of partly-executed tasks.

In conclusion, the proposed policies are simple enough to be practical, yet effective over the wide range of workload conditions that are likely to occur in practice.
CHAPTER IV

A Load Index And A Transfer Policy for Global Scheduling Tasks With Deadlines

4.1 Introduction

Distributed real-time systems have been proposed for real-life applications such as control systems [10] and industrial process control [17, 34]. In such systems, tasks have deadline constraints associated with them, and it is critical that they finish by their deadlines. In real-time distributed systems, a global scheduling algorithm minimizes the number of tasks that miss deadlines through load distributing by taking advantage of the statistical fluctuations in the load at different nodes.

Two important components of a global scheduling algorithm are its transfer policy and its location policy [11, 12]. The transfer policy at a node determines whether a task should be transferred by identifying the node as a suitable sender of a task, a suitable receiver, or neither. The location policy at a node, on the other hand, determines where a task should be transferred. Location policies can be broadly characterized as sender-initiated (in which receivers are found for senders), receiver-initiated (in which senders are found for receivers), and symmetrically-initiated (in which complementary nodes are found for both senders and receivers).
Over the past several years, many global scheduling algorithms have been proposed for real-time distributed systems to schedule tasks with deadline constraints. These algorithms try to transfer tasks whenever task’s deadlines cannot be met locally [7, 22, 33, 35], and/or when there are more than $T$ (threshold) tasks in the queue [7, 35]. These approaches have the following shortcomings:

1. A queue length $< T$ does not necessarily mean that the local schedule will be able to meet a new arrival’s deadline due to the following reasons:

   - A task with a high service demand can keep a node busy beyond the latest starting time of a waiting task or an arriving task.
   - A scheduling policy based on the queue length neither takes into account a task’s service demand nor task’s deadline information in making decisions. Hence the queue length, by itself, is not an appropriate load index to use.

2. Initiating task transfers, only after realizing that the local schedule cannot meet a task’s deadline limits the scheduler’s ability to satisfy deadlines (especially tight deadlines). This limitation is because, not only the task selected for transfer is delayed due to communication delays, other tasks that are waiting are also delayed due to the overhead incurred by the CPU to carry out the task transfer activity. Any delay in starting the execution of a task increases the probability of that task missing the deadline.

3. A fundamental limitation of the previous approaches [7, 22, 33, 35] is that the actions taken by them are corrective in nature: corrective actions (task
transfers) are initiated only after an undesirable condition has occurred (when a deadline cannot be met or local load is high).

In view of the above problems, the goals of this thesis are as follows:

1. Define a load index that gives a more precise information than the queue length about a node's ability to satisfy deadlines.

2. Construct a mechanism which takes:

   (a) Preventive measures by looking ahead, thereby, averting potential deadline misses.

   (b) Corrective measures if necessary to avert a deadline miss.

We propose free-time in a schedule at a node as the new load index and present a new dynamic transfer policy (which makes use of free-time). Key features of the transfer policy are: (1) It is general and can be used in conjunction with a broad range of existing location policies. (2) It does not require the capability to transfer partially executed tasks. (3) It adapts better to the system state by looking-ahead.

By substituting the transfer policy of a previously studied global scheduling algorithm [7] with the new transfer policy, we show the following: Significant reductions in the number of deadline misses can be realized by taking preventive measures alone compared to previously proposed algorithms over a wide range of system loads and laxities.  

---

1 Laxity of a task is the maximum time the task can wait before receiving service and can still complete by the deadline.
The remainder of the paper is organized as follows: In the next section, we present the system model. In Section 4.3, we present a new load index called the free-time. In Section 4.4, we present the new transfer policy. In Section 4.5, the performance of a global scheduling algorithm using the proposed transfer policy is compared to that of three global scheduling algorithms previously proposed [7, 33]. Finally, Section 4.7 summarizes the results and presents conclusions.

4.2 System Model

The class of distributed systems we address in this study is characterized as a collection of autonomous computers which are connected by a communication network (e.g., Ethernet, token-ring, etc.). As in the previous studies [7, 22], we assume that each task's service time requirement, deadline, and laxity are known at the time of its arrival. This assumption is not unreasonable since real-time application tasks perform specific functions and their requirements can be determined in advance through profiling. A task can be in one of the following three states: on-time, to-be-late, and late [7]. A task that can finish by the deadline is on-time. A task that can finish before the deadline only if it is transferred to another node is said to be in to-be-late state. A task known not to finish by its deadline is in late state.

4.2.1 A Local Scheduling Policy

We propose to use “Least-Laxity First” as the local scheduling policy. By scheduling tasks with small laxities before the tasks with large laxities, a scheduler increases the probability of finding nodes that can satisfy the deadlines of tasks with higher
laxities before those tasks become late. Hence least-laxity first policy increases the probability that a higher number of tasks (both with small and high laxities) meet their deadlines. On the other hand, scheduling policies such as “Least Processing Time First” used by Chang and Livny [7] and “First Come First Served” used by Kurose and Chipalkatti [22] do not take advantage of the knowledge about laxity.

On-time tasks have priority over to-be-late and late tasks. A newly arrived task cannot preempt an executing task unless the executing task is already late. We assume that late tasks are not discarded, but are eventually served to completion. Global scheduling activities at a site such as message handling and task transfers have preemptive priority over task execution.

4.3 Free-Time: A Load Index

If a global scheduler has to take preventive measures to avoid deadline misses, its transfer policy component should be able to lookahead and alert the scheduler that a node is a sender even before a task actually becomes to-be-late at that node. This cannot be done effectively by just using the queue length as the load index at a node for the reasons already mentioned.

An ideal index of load at a node would accurately indicate the node’s ability to satisfy deadlines. Maintaining an ideal load index would be expensive since it has to look at all the possible ways of scheduling tasks in a queue. In this paper, we propose to use free-time in the local schedule at a node as the load index. Free-time is composed of idle time in a schedule and indicates a node’s ability to satisfy an incoming task’s deadline. While a large free-time in the schedule indicates that an
incoming task can receive service soon (hence, can meet the deadline), a small free-time in the schedule indicates that an incoming task can receive service not so soon (hence, may miss the deadline). Free-time in a schedule has been used before by Ramamritham, Stankovic, and Zhao [33] in deciding the surplus resources available at a node. We will point out the differences in how it is computed and used in our approach and the previous approach in Section 4.6.

We next give the algorithm for computing free-time, and to make the exposition clear, we give two examples and explain how free-time relates to the waiting time of a task. Input to this algorithm is the local schedule at a node. In this schedule, tasks are ordered in the increasing order of laxity since our local scheduling policy is least-laxity first. Since scanning the entire length of a schedule to calculate free-time can be expensive, we scan only a part of the local schedule which is defined by the window. The window defined below encompasses the current and the near future activities of the local schedule. Note that, tasks which are already late are not considered in computing free-time. Hence a node having only late tasks has free-time equal to the window length.

Window of a schedule := (current time) to (current time + window size).

Window-size := window-factor * (mean task service time + mean task laxity),

where window-factor is a parameter of the algorithm.

Free-time in a node's schedule is computed as follows.

new-current-time := current-time;
free-time := window-size;
wf-time := current-time + window-size; (* Window finish time *)
for all on-time and to-be-late tasks in the schedule do
(* Task service time = residual processing time required by the task *)
begin
if new-current-time + task service time \leq \text{task finish time} \text{ then}
  begin
  if (\text{task finish time} < \text{wf-time}) \text{ or } (\text{free-time} = \text{window-size}) \text{ then}
    begin
    \text{free-time} := \text{free-time} - \text{task service time};
    \text{new-current-time} := \text{new-current-time} + \text{task service time};
    end
  else
    begin (*outside the window, exit the loop*)
    \text{start-time} := \text{task finish time} - \text{task service time};
    if \text{start-time} < \text{wf-time} \text{ then}
      \text{free-time} := \text{free-time} - (\text{wf-time} - \text{start-time});
      (* ignore tasks outside the window *)
      \text{exit loop};
    end;
  end
  else
    begin (* task is in to-be-late state *)
    \text{free-time} := 0;
    \text{exit loop};
    end;
end

Examples: Figure 12 gives two examples of the calculation of the free-time wherein we assume task X is executing and task Y is waiting next in the line. Rectangles in Figure 12 refer to the residual service time of tasks X and Y.

Note that in Figure 12(a), task Y’s deadline is outside the window. If the window size is approximately equal to mean task service time (S) plus mean task laxity (X), then it implies that Y has a large laxity. In this case, we calculate free-time in the schedule as the sum of unutilized time slots of CPU available until the latest possible starting time for task Y. Free-time calculated this way indicates a node’s ability to serve a task that needs to be served immediately.
On the other hand, when a task's deadline is within the window (task Y in Figure 12(b)) and window-size is approximately equal to \((S + X)\), it implies that the task does not have a large laxity. In that case, it is important that the task receives service at the earliest so that it can complete before its deadline. Hence, in this case, we calculate the free-time in the schedule as the unutilized time slots of CPU available after the earliest completion time possible for task Y, as the node is really not in a position to serve a new task before serving Y.

With this background, we next discuss the significance of the free-time as a load index.

**Significance of Free-time:** To see why free-time is a good indicator of a node's ability to satisfy the deadline of a task, consider the following. Assuming a window factor of 1, a small free-time in the window implies one of the following scenarios:
a) A task has just begun executing. Hence a new arrival will have to wait considerably before receiving service (delay \(\approx\) mean task service time, is considered to be large).

b) A task is already waiting. Hence a new-arrival/waiting-task has to wait for the waiting-task/new-arrival to finish before receiving service. (Note: Least-laxity first policy can place a new arrival before the waiting tasks.)

A large free-time implies one of the following scenarios: a) The executing task is just about to complete or the queue is empty. Hence a new arrival can expect to receive service soon. b) Executing task is just about to complete and the next waiting task has a large laxity (laxity \(\approx\) mean task service time, is considered large). Hence a new arrival having a smaller laxity than that of the waiting task can expect to receive service soon. The task with a larger laxity is delayed favoring the task with a smaller laxity because with a larger laxity the chances of finding a node that can satisfy a task’s deadline is higher.

At this point, we would like to stress that the computation of free-time is simple and straightforward. The for loop in the free-time algorithm rarely executes beyond two or three iterations because the window size is approximately equal to \((S + X)\). Thus, the computation of free-time imposes very little overhead on the system.

### 4.4 A Transfer Policy

In this section, we present a new transfer policy that can effectively take advantage of the knowledge available through free-time load index.

Our transfer policy identifies a node as a sender of a task whenever the free-time in the node’s schedule is \(< LTFT\) (lower threshold for free-time, which is a parameter
of the algorithm). A small free-time indicates that the node is going to be busy for an extended period and hence can delay the start of the execution of new arrivals. A delay in the execution means potential deadline misses in the future. On arrival of a task, a node can become a sender under two conditions. 1) The newly arrived task's deadline cannot be met locally. 2) The newly arrived task's deadline can be met locally, however, the free-time in the local schedule is < LTFT. The second case triggers anticipatory task transfer anticipating a deadline miss (i.e., it is looking ahead) should another task arrive in the near future before the current task finishes. The window to calculate the free-time begins at the current time.

Our transfer policy identifies a node as a receiver of a to-be-late task when the node can satisfy the deadline constraint of the task. For anticipatory task transfers, a node is identified as a receiver of an on-time task when it can satisfy the deadline constraint of the task and the free-time in the node's schedule is ≥ UTFT (upper threshold for free-time, which is a parameter of the algorithm). The window to calculate free-time begins at the estimated arrival time of the task. The estimated arrival time takes into consideration the transmission delays as well as the CPU overhead to send and receive messages. A large free-time indicates that the node will be able to begin the execution of a task immediately (if the node is idle) or without much delay. When a node replies that it is a receiver, a flag "task-in-transit" is set. A node whose task-in-transit flag is set is identified not to be a receiver for any probes received. The flag is reset on receipt of the remote task. (Note that a timeout can be employed to reset the flag in case the remote task fails to arrive.)
Unlike the transfer policies of Chang and Livny [7], our policy does not require transfer of partially executed tasks. All task transfers are only due to placement of new tasks. Note that transferring a task that has already begun execution is much more expensive than transferring a newly arrived task because the task state, which must accompany it to its new node, becomes large and is much more difficult to obtain after execution has started [9]. For some systems, gathering the state of an executing task may be extremely difficult. As a result, a global scheduler may be considerably simplified if it transfers only newly-arrived tasks.

4.5 Performance Study

Previously, Chang and Livny compared the performance of sender-initiated and receiver-initiated global scheduling algorithms for tasks with deadline constraints [7]. These algorithms used “least processing time first” as the local scheduling policy. A mechanism to transfer partially executed tasks was assumed to be available. The task service time and laxity were assumed to be exponentially distributed. However, in our simulation study, unless noted otherwise, we have assumed that task service time and laxity follow Normal distribution. This is because, real-time tasks in applications such as process control [10, 24, 34], automated manufacturing [3], air traffic control [14], periodic jobs [25], program trading [1], etc., tend to perform the same computations repeatedly for different input parameters. Under such conditions, their service time and laxity requirements are more accurately modeled by Normal distribution rather than by Exponential distribution. (Note: Exponential distribution would model a large number of tasks with small service time and small laxity, interspersed
with few tasks having - large service demand and large or small laxity – or – small service demand and large or small laxity.) We use miss ratio as the main performance metric which is the percentage of tasks not meeting their deadlines [7]. All simulation results presented have less than 10% error at the 90% confidence interval, while most of the results have less than 5% error.

We first describe the algorithms that are studied in this section. We then present the results of our simulation study. The global scheduling algorithms, typically, have four policy components [37]. We describe the algorithms in terms of these policy components.

4.5.1 Global Scheduling Algorithms

1. SEND-LP: SEND-LP is the sender-initiated algorithm proposed by Chang and Livny [7].

Location Policy: Nodes are probed randomly until either a receiver node is found or 'probelimit' probes have been completed without a receiver being found ('probelimit' is a parameter of the algorithm).

Transfer Policy: A node is identified as a sender under any of the following conditions: 1) One or more tasks in the queue is/are in to-be-late or late state. 2) There are more than UT (upper threshold) tasks in the queue.

A node is identified as a receiver of a remote task if any of the following conditions is true: 1) There are less than LT (lower threshold) tasks in
the queue. 2) The remote task's deadline can be satisfied. UT and LT are parameters of the algorithm.

**Initiation Policy:** Search is initiated only when a task originating at a node causes it to be in the sender state.

**Selection Policy:** Selection of a task to be transferred is done in the following order: 1) A to-be-late task with the earliest deadline. 2) Select any late task. 3) Select an on-time task with the longest service-time such that transfer delay will not cause it to miss its deadline.

**Local scheduling policy:** Least processing time first.

2. **SEND-LL:** SEND-LL is the same as SEND-LP except for the following: (1) The local scheduling policy is least-laxity first (discussed in Section 4.2.1). (2) Task chosen for transfer is the newly originated task which caused the node to become a sender.

3. **RECV-LP:** RECV-LP is the receiver-initiated algorithm proposed in [7].

**Location policy:** In the first phase, nodes are probed randomly until either a sender (node having a to-be-late or a late task) is found or probelimit probes have been completed without a sender being found. If the first phase fails, then in the second phase, nodes are randomly probed until either a sender (node having an on-time task which has sufficient laxity such that transfer delays will not cause it to miss its deadline) is found or probelimit probes have completed without a sender being found.
**Transfer policy:** A node is identified as a receiver when the local queue length is less than LT. Conditions for identifying a node as a sender are already mentioned in the location policy above.

**Initiation policy:** Search is initiated when the completion of a task leaves a node in the receiver state. Also when both the phases of a search to locate a sender fail, the search is reinitiated after a *wake-up-delay* which is a parameter of the algorithm.

**Selection Policy:** A task that caused the node to become a sender as discussed in the location policy.

**Local Scheduling Policy:** Least processing time first.

4. **RECV-LL:** RECV-LL is same as RECV-LP except that we substitute the local scheduling policy by least-laxity first.

5. **SEND-FT:** SEND-FT is same as SEND-LL except that it makes use of our transfer policy described in Section 4.4.

6. **RAND-LL:** RAND-LL is the random algorithm. We study this algorithm for the following reason. In [33], Ramamritham, Stankovic, and Zhao studied the Random algorithm along with Flexible, Bidding, and Focussed-addressing algorithms. In their study, their best performing adaptive algorithm, Flexible, which made use of the knowledge about the surplus capacity available in the system, performed only slightly better (less than 5% improvement at moderate system loads, and even less at low and high system loads) than the Random
algorithm. This strongly motivated us to compare the performance of SEND-FT with the Random algorithm (which is simple yet performs quite well) rather than with the much more complex algorithms like Flexible. The Random algorithm is described next.

**Location Policy:** A node is randomly selected as a receiver without checking whether the selected node can satisfy the task's deadline.

**Transfer Policy:** A node is identified as a sender when the newly arrived task's deadline cannot be met locally.

**Initiation Policy:** Search is initiated only when a task originating at a node causes it to be in the sender state.

**Selection Policy:** Task chosen for transfer is the newly originated task which caused the node to become a sender.

**Local Scheduling Policy:** The local scheduling policy is least-laxity first (discussed in Section 4.2.1).

### 4.5.2 Performance Results

**Performance model:** In the simulation study, we made the following assumptions:
- task interarrival time is Exponentially distributed; the system contains 15 nodes;
- the CPU overhead incurred in either sending or receiving a probe message is 0.003 timeunit; LT = 1; UT = 2; probelimit = 5, wake-up-delay = 1 timeunit; mean service time = 1 timeunit; unless noted otherwise, service time and laxity are Normally distributed with a variance of 25% of their respective means with the values of random
variables bounded by the variance around the mean; partially executed task size is
Exponentially distributed with a mean of 20 packets; the packet size is 4096 bits; a
probe is assumed to need no more than one packet; an Ethernet network having a
bandwidth of 10 Mbits/second is assumed. Message transit and task transit delays
are calculated using these numbers.

In the following discussion, we use the notations: $S = \text{mean service time}$; $X$
$= \text{mean laxity (percentage of } S) ; W = \text{window size, } U = \text{utilization/system load}.$
LTFT (lower threshold for free-time) and UTFT (upper threshold for free-time) are
expressed as a percentage of $S$. Unless noted otherwise, all nodes are assumed to have
the same task arrival rate (i.e., homogeneous workload).

To facilitate the reading of the graphs, legend's list follows the order of curves in
the graphs in the decreasing order of miss ratio.

Main Comparisons

In this section, we compare the performance of all the 6 algorithms described in
Section 4.5.1. Figure 13 plots mean miss ratio against offered system load for these
algorithms at a small laxity (0.2-S). (Note that Laxity = 2-S was used in [7].) The
improvement in performance due to the “least-laxity first” local scheduling policy
only can be seen by comparing SEND-LL/RECV-LL versus SEND-LP/RECV-LP.
The improvement in performance due to the combined effects of the least-laxity first
local scheduling policy and the new transfer policy can be seen by comparing SEND-
FT vs. SEND-LP. The improvement in performance due to the new transfer policy
only can be seen by comparing SEND-FT vs. SEND-LL.

**Observation 1:** *Relative to the other algorithms, SEND-FT reduces the number of deadline misses by several hundred percent up to a system load of 0.65 and significantly beyond it.* The new transfer policy makes better use of the idle processing capacity by doing anticipatory transfers and by not waiting for a task to become to-be-late or a node's queue length to exceed UT. Note that Livny and Melman [26] have shown that the probability that at least one task waits for service at a node and at least one node is idle is almost ONE for system loads between 0.50 and 0.85. Therefore, senders
are able to find receivers in SEND-FT resulting in successful anticipatory transfers. Anticipatory transfers result in tasks being served earlier than if they had stayed at their originating node. By being served earlier: (1) The delays due to transferring a task, overhead incurred by the CPU due to probing and replying to probes are less likely to cause a task to miss its deadline. (2) Tasks finish earlier increasing free-time at a node, thereby, increasing a node's ability to meet the deadlines of future tasks.

In the absence of anticipatory transfers (in SEND-LL and SEND-LP if the queue-length $\leq UT$, and in RAND-LL), on-time tasks are not transferred, thereby, delaying the beginning of their execution. However, if a new arrival's deadline cannot be satisfied, overhead due to the task transfer activity will delay both the executing task and the on-time tasks that are waiting, thereby, increasing the probability of them missing the deadline. (Note that task transfer activity has priority over task execution and imposes CPU processing overhead.) These delays cause delayed completion of the tasks and hence delay the beginning of the execution of the waiting tasks and of the future arrivals. These delays have a cumulative effect of increasing the probability of tasks missing deadlines in future higher and higher until there is a pause in the task arrivals due to statistical fluctuations. (Henceforth, lesser/higher wait before the start of the execution can be associated with lower/higher miss ratio.) From this point onwards, we refer to the above phenomenon as the cumulative phenomenon.

At high loads, SEND-LL improves performance compared to RAND-LL since at such loads, few nodes have large free-time to satisfy deadlines of remote tasks. Under such circumstances, many task transfers performed by RAND-LL are useless and
impose additional overhead on the system as few nodes can satisfy the deadlines of transferred tasks.

At high loads, SEND-FT improves the performance moderately compared to SEND-LL since at such loads, few nodes are likely to be receivers. Nevertheless, SEND-FT performs better than the other algorithms because of tasks being served earlier whenever the attempts for anticipatory transfers are successful.

We conclude that anticipatory transfers (preventive measures) are quite effective in minimizing the number of deadline misses.

**Observation 2:** \textit{RECV-LP and SEND-LP both perform poorly compared to the algorithms using least-laxity first policy:} In “least processing time first” local scheduling policy [7], any new task originating at a node having service demand less than the current executing task (which may be on-time) will result in the preemption of the executing task. When tasks have small laxities, a preemption will generally result in a preempted on-time task becoming to-be-late. Note that transferring partially executed tasks requires transferring the task’s state, hence, delaying every task at the node much more than in the case of new task transfers. Hence, the effect of the cumulative phenomenon is much more severe. Even if a transfer attempt is successful, a new task originating at the remote node with a lesser service time will cause the preemption of the transferred task again and as a consequence the preempted task misses its deadline with a very high probability.

The better performance of SEND-LL, RECV-LL, and RAND-LL clearly shows the superiority of the “least-laxity first” local scheduling policy of Section 4.2.1 over the
“least processing time first” policy. The improved performance is due to: 1) Absence of deadline misses due to the preemption of on-time tasks. 2) Waiting tasks have higher laxities than the executing task; The higher the laxity of a task, the higher the probability of finding a node that can satisfy the deadline of that task. Thus, least laxity first policy increases the chances of both executing and waiting tasks meeting their deadlines, thereby, decreasing the miss ratio.

Observation 3: RECV-LP performs better than SEND-LP at high system loads: In SEND-LP, the probing activity increases with system load. Also, the rate of preemptions of on-time task increases with an increase in the task arrival rate when “least processing time first” local scheduling is used (see Observation 2). However, at high system loads, there are few receivers and therefore, the to-be-late tasks (which include the preempted tasks that were previously on-time) are most likely to miss their deadlines. Thus, SEND-LP wastes a significant CPU capacity through futile probes for searching receivers at high loads. This wastage of computing power further increases the chances of preemptions of on-time tasks and to-be-late tasks missing deadlines. On the other hand, in RECV-LP, there is less overhead due to probing activity since there are plenty of senders at high loads. Thus, most of the processing power is spent to serve tasks in RECV-LP. Therefore there is less likelihood of newly originating tasks having smaller service time demand than the currently executing task, and thus, fewer task preemptions of on-time tasks. Fewer preemptions of on-time tasks and less overhead due to probing activity results in fewer deadline misses in RECV-LP.
SEND-LP performs better than RECV-LP at low to moderate loads. This is because task transfer activity is started in accordance to the needs of tasks arriving at the senders in SEND-LP as opposed to senders waiting for receivers to find them through random probing in RECV-LP. (This explanation also holds for the better performance of SEND-LL compared to RECV-LL at low to moderate loads.)

Observation 4: In contrast to observation 3, SEND-LL performs better than RECV-LL at high system loads: At high loads, there are few receivers since queue length of a node is rarely < LT. Hence in RECV-LL, the probing activity is generally inactive and hence task transfers seldom occur. When tasks have small laxities and task transfers are infrequent, to-be-late tasks are nothing but late tasks. Thus, a node in RECV-LL will find itself executing a late task often when a to-be-late task may be waiting elsewhere.

In SEND-LL, senders are actively probing at high loads. When these probes are received by a node having only late tasks, the task transfers succeed, thus, reducing the miss ratio. (Note that, unlike in SEND-LP, preemptions of on-time tasks are absent in SEND-LL.)

We study only SEND-LL, SEND-FT, RAND-LL, and RECV-LL in the subsequent sections, since these algorithms perform consistently better compared to RECV-LP and SEND-LP algorithms.
Sensitivity to Laxity

Figure 14 plots mean miss ratio against laxity. This figure shows that even when laxity is high, anticipatory transfers help in reducing the miss ratio. The main reason for the lower miss ratio is the better use of idle capacity in the system due to early start of task execution.

The miss ratio under both RECV-LL and SEND-LL decrease with increasing laxity. The crossover at laxity of 0.4 between RAND-LL and RECV-LL and the poorer performance of RAND-LL compared to SEND-LL is due to the useless task transfers that occur in RAND-LL.
Figure 15 plots the number of anticipatory transfers in SEND-FT against the offered system load (curve VARY-SL) and laxity (curve VARY-LX). For each experiment, the number of anticipatory transfers are obtained for a total of 15,000 tasks execution.

The number of anticipatory transfers increases with an increase in the system load for the following reason: As the system load increases, the number of instances of nodes becoming senders increases, and hence the number of anticipatory transfer attempts. Between the system loads of 0.5 and 0.8, the probability that at least one task waits for service and at least one node is idle is almost one [26]. Hence, there is
a rise in the number of successful anticipatory transfers at such loads. At high loads, there are few receivers, thus, fewer anticipatory transfers.

As laxity increases, the number of instances wherein a node can satisfy the new arrival's deadline increases, and hence the number of anticipatory transfer attempts. With higher number of on-time tasks, more anticipatory transfers succeed since senders can find receivers (because system load of 0.7 is moderate). However, at high laxity, a schedule has even more free-time, thereby, decreasing the number of instances wherein a new task originating at a node causes the node to become a sender. Hence, the number of anticipatory transfers decreases as a node is capable of meeting most task's deadlines locally. In Figure 15, we see the curve VARY-LX flattening at high laxities.

The MRATIO curve in Figure 15 plots miss ratio under SEND-FT with respect to laxity. For MRATIO curve, Y-axis denotes miss ratio and X-axis denotes laxity. Note that the curves MRATIO and VARY-LX are mirror images of each other. As the number of anticipatory transfers increase, the miss ratio decreases. Also observe that when the curve VARY-LX flattens for a laxity > 0.7, so does the curve, MRATIO. This clearly shows that an increase in the anticipatory transfers is responsible for reducing the miss ratio. Reduction in miss ratio over a wide range of laxity because of anticipatory transfers can be seen by comparing the curves SEND-FT and SEND-LL in Figure 14.
Sensitivity to Upper Threshold for Free-Time (UTFT)

Figure 16 plots the mean miss ratio against UTFT for three different values of laxities while keeping the offered system load fixed at 0.7. (Recall that UTFT is expressed as a percentage of service time (S).) Clearly, the new transfer policy is not very sensitive to UTFT. With UTFT’s value becoming smaller, a node is optimistic about satisfying a deadline, and will accept an anticipatory transfer even when the node is going to be busy serving the task executing currently. With UTFT’s value growing larger, a node being pessimistic will not accept anticipatory transfers even when accepting anticipatory transfer would not have worsened the local situation and would have
reduced the waiting time of a remote task. These are the reasons why the miss ratio is higher at either extremes of X-axis in figure 16. Note, however, that the concave shape of the curves indicate that the value of the mean service time (S) is a good heuristic for UTFT.

Sensitivity to Lower Threshold for Free-Time (LTFT)

Figure 17 plots mean miss ratio against LTFT for 3 different values of laxities while keeping system load fixed at 0.7. (Recall that LTFT is expressed as a percentage of service time (S).) The figure shows that the mean laxity (X) serves as a good heuristic for LTFT. However, the new transfer policy is not too sensitive to variations in LTFT.
Given $W \simeq (\text{mean service time} + \text{mean laxity})$, free-time $\leq$ mean laxity implies that an on-time task is executing currently. Under such circumstances, a new task arrival triggers an anticipatory transfer, which will utilize idle capacity available elsewhere and reduce the waiting time of tasks. Keeping $\text{LTFT} \geq$ mean laxity subsumes the anticipatory transfers that occur when free-time $\leq$ mean laxity. On the other hand free-time much larger mean laxity in general implies, that the node is free (i.e., no on-time tasks in the schedule). Thus, increasing LTFT beyond mean laxity does not result in further reductions in the miss ratio due to the absence of anticipatory transfer attempts when free-time $> \text{mean laxity}$.

$\text{LTFT} < \text{mean laxity}$ discourages anticipatory transfer attempts even when transfers would have resulted in the reduction of task waiting times by utilizing the idle capacity available elsewhere in the system. Delay in receiving service increases the miss ratio.

The above effects are more pronounced for a small laxity (see curve for $X=0.3S$).

**Sensitivity to Window Size**

Figure 18 plots mean miss ratio against window factor (WF) which varies the window size (note: $W = WF(S+X)$). The figure shows that a window size close to $(S+X)$ provides sufficient information to compute free-time (node’s ability to satisfy deadlines of a new arrival) over a wide range of laxities.

The effect of increasing the window size is to increase the value of free-time artificially, and hence, to reduce the number of instances of a node being a sender
(for anticipatory transfers). Reduction in anticipatory transfers result in less efficient usage of idle capacity available in the system, thus, resulting in a higher miss ratio.

Decreasing the window size artificially decreases the free-time in a schedule which in turn increases the number of anticipatory transfer attempts. However, the number of acceptance of anticipatory transfers decreases due to the artificial reduction in the free-time. The reduction in the number of anticipatory transfers and the overhead due to the probing activity results in an increase in the waiting time of tasks at senders causing the cumulative phenomenon, and hence a higher miss ratio.

The curve for $X = 0.3$, shoots up for a window factor < 0.85 because window size < UTFT, and hence free-time can never be >UTFT which resulted in task transfers
only when nodes are idle. (Note: SEND-FT accepts task transfers whenever a node's queuelength = 0 irrespective of the value of the free-time.) The effect of task transfers only when queuelength = 0 at the destination node is less number of anticipatory transfers, and hence higher waiting times for tasks before receiving service. As a result, the miss ratio jumps to higher values for WF<0.85.

Sensitivity to Heterogeneity

In this section, we examine the effect of heterogeneous task arrival rates on the performance of SEND-FT, SEND-LL, and RECV-LL. In Figure 19, load-generating nodes originate all of the system load (uniformly distributed among these nodes), with none originating at the remaining nodes. In the following discussion, lower degree of heterogeneity means fewer non load-generating nodes are present; higher degree of heterogeneity means higher number of non load-generating nodes are present. RECV-LL performance becomes worse at a lower degree of heterogeneity than any other algorithm. This is because receiver-initiated random probing is quite likely to fail in finding a sender when only a few number of nodes are generating load. In addition, the search for senders is not started in accordance to the need of tasks at senders. This results in both fewer and delayed task transfers causing significant delays before a task receives service, and hence a higher miss ratio.

SEND-LL performs better with increasing heterogeneity due to the following reason: Higher the degree of heterogeneity, higher the number of receivers (i.e., non load-generating nodes). Hence, few senders (i.e., load-generating nodes) have little
difficulty in finding receivers.

In contrast to SEND-LL, SEND-FT's performance grows worse for moderate heterogeneity due to task transfers occurring from one load-generating node to another. Since load-generating nodes are subjected to a higher task arrival rates relative to homogeneous workload case, these transfers worsen the situation further. These task transfers occur because of the following reason: Once a load-generating node transfers all the to-be-late and on-time tasks in its queue, free-time in its schedule grows bigger as the executing task (if present) receives more service, eventually becoming greater than UTFT. Should a load-generating node receive a probe when free-time > UTFT,
it will accept a remote task.

At higher degree of heterogeneity, SEND-FT performs better than it does under the homogeneous workload case (when the number of load generating nodes = 15) for the same system load due to the following reason: (1) Earlier initiation of task transfers (due to anticipatory task transfers). (2) Load-generating nodes are nearly always senders because of the very high task arrival rate at high heterogeneity. Hence, receivers are generally non load-generating nodes. (3) With large number of non load-generating nodes, senders generally succeed in finding receivers. The combined effect of the above three facts is to reduce the wait time of tasks and hence a lower miss ratio.

RAND-LL's performance grows worse with moderate increase in heterogeneity, simply because of the useless task transfers that occur among load-generating nodes. However, at higher degrees of heterogeneity, there are few load-generating nodes, and thus, fewer task transfers among load-generating nodes. In addition, task transfers to randomly selected nodes balances the workload among all the nodes in the system, thus, decreasing the miss ratio at higher degrees of heterogeneity.

“Exponentially” Distributed Task Service Time and Laxity

To check the robustness of the proposed transfer policy, we study the performance of SEND-FT when tasks have service time and laxity Exponentially distributed.

Figure 20 plots the mean miss ratio against the offered system load when service time and laxity are exponentially distributed. SEND-FT performs better than
SEND-LL and RAND-LL only moderately for the following reason: When Exponential distribution is used for laxity, most of the tasks have laxity smaller than the mean value (note assumed laxity = 0.2 S, which is small) and few tasks with large laxity. When tasks with very small laxities originate at busy nodes, their state is most likely to-be-late and seldom on-time. Hence the number of instances wherein a new arrival causes the node to become a sender (for anticipatory transfers) are few. Hence SEND-FT performs moderately better than SEND-LL due to few anticipatory transfers that occur when tasks with large laxities originate at busy nodes. Note, however, that in real-time systems, system’s performance can seriously degrade even
under very few deadline misses. Therefore, even moderate reduction in the number of deadline misses may be significant [33].

Note that SEND-FT uses a small LTFT (= 0.2S = mean laxity X) for the following reason: When service time and laxity are Exponentially distributed, free-time in a schedule is usually large as most tasks have service time and laxity smaller than the mean values. Under such conditions, only those anticipatory transfers caused by tasks with large service time help the situation by reducing the waiting time of tasks arriving in the near future. Large LTFT initiates anticipatory transfers even when free-time is large. But these transfers hurt the receivers as receivers become unable to meet the deadlines of tasks originating at them as most of the tasks have laxity much smaller than the mean laxity.

Performance of RECV-LL is the worst because the task transfer activity is not started in accordance to the needs of tasks arriving at senders.

4.6 Related Work

Previously Ramamritham, et al., studied [33] four distributed algorithms, namely, focused addressing, bidding, flexible, and random to schedule tasks with deadline and resource requirements. Only bidding algorithm is relevant to our work as it makes use of a concept similar to the free-time. We next give an outline of the bidding algorithm and then point out the difference between the free-time used in [33] and in our scheme.

The scheduling actions are triggered when a task originates at a node and its deadline cannot be met locally (i.e., no preventive measures are taken). Each node
periodically calculates the node surplus and sends it to a subset of the nodes in the system. The node surplus is a vector where each entry gives the total amount of time a resource was not used during a past time window \([\(t - \omega l\), \(t\)]\) where \(t\) is the current time and \(\omega l\) is the window length. The steps taken by a node \(i\) when it cannot meet the deadline of task \(T\) are as follows:

1. Node \(i\) selects \(k\) nodes with sufficient surplus in the resources needed by \(T\) and sends request-for-bid messages to these nodes.

2. When a node receives the request-for-bid message, it calculates a bid, indicating the likelihood that task \(T\)'s deadline can be met at the node, and sends the bid to node \(i\).

3. Node \(i\) will send task \(T\) to the node which offered the best bid.

The bid in step 2 is calculated as follows:

\[
\text{MaxBid} = \frac{\text{Min}(\text{FreeTime of each resource required by the task})}{\text{Computation time of the task}}
\]

Free-time is the sum of the lengths of the free time slots between the estimated earliest task arrival time on this node and the task's deadline. If the MaxBid computed is greater than the Minimum bid (a parameter of the algorithm), then the actual bid is calculated. To calculate the actual bid, a ‘number of instances of task \(T\)’ less than or equal to MaxBid are inserted into the current schedule and checked to see how many instances can be guaranteed without jeopardizing the previously guaranteed tasks in the schedule. The actual bid is the number of instances of task \(T\) that can be
guaranteed without jeopardizing the previously guaranteed tasks. If the actual bid is greater than the minimum bid, then the bid is sent to the appropriate node. The inserted instances of the remote task are then removed from the schedule (i.e., the resources are not reserved).

Although our transfer policy and the bidding algorithm [33] have used the notion of free-time, the way free-time is calculated in our scheme, its significance, and usage are completely different. The window length used to calculate free-time in [33] is the time interval between the estimated earliest task arrival time at the destination node and the task's deadline. The window length used to calculate free-time in our scheme is the time interval between the estimated earliest task arrival time at the destination node plus window-factor-(mean task service time + mean task laxity). A window length calculated using the mean values of service time and laxity is expected to adapt better to statistical fluctuations in the characteristics of tasks in the system. The calculation of free-time in our scheme depends on whether the deadlines of tasks present in the schedule is within the window or not (see the algorithm and the examples in Section 4.3). On the other hand, the calculation of free-time in [33] is the sum of the unutilized time slots in the window. In our scheme, free-time in a node's schedule is used to decide whether to accept an anticipatory transfer and more importantly to decide whether a node should trigger anticipatory task transfers. Hence, the significant usage of free-time in our algorithm is to indicate a node's ability to satisfy the deadlines of future arrivals and to help in averting potential deadline misses. Note that in our scheme, free-time is not even computed when a node is
polled to satisfy a to-be-late task. While on the other hand, usage of free-time in [33] is largely to avoid the overhead of calculating the actual bid in case max-bid is less than the minimum bid.

4.7 Summary and Conclusions

Previous global scheduling algorithms for real-time systems perform only corrective actions, i.e., task transfer activity is triggered only after a task’s deadline cannot be met or when local load is high. These algorithms perform very poorly when the deadlines are tight (even when the system load is low) due to the occurrence of the cumulative phenomenon.

In this paper, we proposed a load index and a transfer policy. The load index “free-time” gives a more accurate indication of a node’s ability to satisfy the deadlines of future arrivals than just the “queue-length” at a node. The key features of the new transfer policy are: (a) It takes preventive measures by doing anticipatory transfers in addition to corrective measures, thereby, reducing the number of potential deadline misses. (b) Using free-time it adapts much better to the system state by looking-ahead and does not require the transfer of partially executed tasks. (c) It is not very sensitive to minor variations of the free-time thresholds and window size, and in this sense the new transfer policy is robust. We showed that “Least-Laxity First” is a better local scheduling policy to satisfy deadlines when compared to “Least Processing Time First”. A simulation study reveals that synergism resulting from Least-Laxity-First local scheduling policy, free-time, and our transfer policy, reduces the number of deadline misses significantly over a wide range of system parameters.
Random algorithm (RAND-LL) performed quite well compared to the other algorithms. (This, in fact, supports the findings in [33].) However, SEND-FT, which uses the proposed transfer policy and free-time load index, consistently outperformed RAND-LL and SEND-LL, indicating that the anticipatory transfers are quite effective in reducing the number of deadline misses. In addition, it can be expected that under the conditions studied in this paper, SEND-FT will perform better than flexible, bidding, and focussed addressing algorithms [33] as flexible, the best performing algorithm among these algorithms, performed only marginally (< 5%) better than the random algorithm.

In conclusion that the proposed transfer policy and the computation of free-time are simple enough to be practical, yet effective over a wide range of conditions.
5.1 Introduction

In recent years, load distributing schemes have been implemented in many distributed systems [2, 9, 23]. With the increasing realization of the benefits of load-distributing, load-distributing is expected to become common in commercial, industrial, academic, and real-time environments. Load-distributing schemes, developed so far, typically make use of idle computing capacity available by transferring tasks from heavily loaded nodes to idle or lightly loaded nodes.

Distributed systems have been proposed for applications such as control systems, industrial process control, automated manufacturing, air-traffic control, program trading, etc [38]. In these systems, it is important that certain critical tasks complete within certain timelimits. These tasks can be viewed as high priority tasks which must get better service than other type of tasks. Moreover, in a general computing facility, certain users/tasks may receive higher priority than others because of project deadlines [28], importance of the tasks [28], the user is paying higher fees for better service, etc.
In this chapter, we address scheduling tasks with priorities in distributed systems. The objective is to ensure that when a task is scheduled at a node, that task has the highest priority among all the tasks waiting for service in the entire system. This objective is difficult to realize in practice as the state information—tasks with what priority are waiting at which nodes—is not readily available in distributed systems. To realize the above objective in distributed systems, whenever a node is ready to schedule a task, it is necessary to make the up-to-date state information available to it. It should be ensured that the state information is made available with minimum delay and by exchanging minimum number of messages. Otherwise, the overhead due to scheduling may outweigh the benefit of scheduling.

An ideal solution for the above problem is to have a global observer which has the perfect system state information available to it at no cost. The observer will direct the scheduling activity for the entire system. However, the realization of this global observer is impossible in distributed systems.

A practical alternative in distributed systems is to have a central coordinator that maintains the state information. Whenever a node completes a task, it will contact the coordinator. The coordinator using the state information available locally will determine where the highest priority task is waiting and will initiate a task transfer to the node (if necessary). The main advantages of this approach are that (1) it is simple and (2) the state information at the coordinator can easily be updated as tasks get scheduled and arrive at individual nodes. A serious disadvantage of this method, however, is that the central coordinator is a potential bottleneck.
An alternative to the central coordinator is to make the system state information available to all nodes. Then all nodes can make scheduling decisions based on the state information available locally. This approach requires that whenever a task is originated or is scheduled, this event be notified to all the nodes. The main disadvantage of this approach is that of the heavy overhead imposed by the messages exchanged to make the state information available and to update it both when tasks originate and when tasks are scheduled. Note that, if the state information at the nodes is not updated as tasks get scheduled, there will be a significant delay in scheduling tasks due to many failed attempts before finding a task of appropriate priority.

In this chapter, we present a distributed protocol that overcomes the aforementioned problems in scheduling tasks with priority in distributed systems.

5.2 Previous Work

Chang [6] is the only one (to the best of our knowledge) to have proposed a distributed scheduling algorithm for tasks with priority, the outline of which is given next. When a task originates at a node, it is executed locally if the node is available (a node is available if it is idle or is running a task of a lower priority than that of the new task). Otherwise, randomly selected nodes are serially polled until an idle node is found for the task execution or a polllimit (a parameter of the algorithm) polls are made. If no idle node but an available node was found, it is selected for the task execution. (This node may start its own search for an available node for the preempted task.) If the polls fail to find an available node, the task is locally queued.
At the time of a task completion, if a task of absolute highest priority in the system is locally available, it is scheduled. (Chang's model assumes there are only two priority classes, namely, low and high priority. Therefore, a high priority class task has the absolute highest priority in the system.) Otherwise, a polllimit number of randomly selected nodes are polled to find out if they have a task of high priority. If the polls fail to find a high priority task, a local task if present is scheduled, otherwise the node lies idle.

As one can see, the serious limitation of this algorithm is that it requires that there be no more than two priority classes. If more than two priority classes are allowed, a node on completing a task, may have to poll the entire system to locate an absolute highest priority task. Note that polling the entire system imposes heavy overhead and is not scalable. To reduce the overhead, busy nodes limit the search to polllimit number of nodes which however, curtails the ability to pick the best available node for the task execution. We do not study this algorithm further due to its limitations. In addition, this algorithm requires facility for the transfer of partially executed tasks, which is quite complex to build and expensive to operate.

5.3 A Token-based Scheduling Algorithm

The goal of the token-based scheduling algorithm is to capture the benefits of the central coordinator approach (simplicity and ease in maintaining the state information up-to-date) while avoiding its disadvantage (potential bottleneck).

We propose a floating repository (referred to as the token) for the state information, which moves between nodes as necessary. In this scheme, each node forwards
information regarding task arrivals and their priorities to the token. The node holding the token updates the token with the incoming state information. Nodes that are ready to schedule their next task, obtain the token and use the state information available in the token to schedule the highest priority task waiting in the system. The node with the token also updates the token to reflect the new system state after it has scheduled a task, which requires no message exchange. The state information in the token can be stored as a list of tuples (task priority, node id). However, the search for the highest priority task can be made more efficient if a different list for each priority class is used.

Two issues that must be addressed to implement the above scheme efficiently are:

1. How to deposit the information in the token without sending a large number of messages.

2. How to locate and obtain the token without sending a large number of messages.

The above two issues are addressed by using distributed pointers that maintain a path to the location of the token. Each node maintains a pointer ($Next$) whose value is either the id of the node holding the token or the id of the node next in the path towards the token. Initially, the token is at some default node, and the pointer $Next$ at every node is set to that default node. The $Next$ pointer gets modified as the token is requested from various nodes as well as when tasks are requested from nodes holding the token. Under this scheme, the $Next$ pointers impose a logical tree structure on the set of nodes, with the root node holding the token [29]. The topology of the tree changes dynamically according to the activity of the system. As the lightly
loaded nodes are likely to be requesting the token frequently to schedule their next
tasks, under this scheme, it is expected that the token will be shuttling among the
nodes that are lightly loaded. Hence, these nodes, using the path information in
the Next pointers can obtain the token quickly without exchanging a large number
of messages. Moreover, since the lightly loaded nodes frequently request tasks from
heavily loaded nodes, the next pointers at heavily loaded nodes point towards the
lightly loaded nodes. This facilitates updating the state information in the token
(shuttling among the lightly loaded nodes) without exchanging a large number of
messages. The outline of the basic algorithm is given next.

The Basic Algorithm

1) When a task with a priority Task.pty originates at a busy node i, node i sends the
following message containing the priority of the task and the node id to update the
state information in the token:

Send Deposit_Msg (Task.pty, i) to Next_i.

2) A node j on receiving a Deposit_Msg(Task.pty, i) message:

If j has the token

insert (Task.pty, i) into the token.

else

Send Deposit_Msg (Task.pty, i) to Next_j.

3) When a node i completes a task, it performs the following operations:

If i does not have the token

Send Token_Request (i) to Next_i; Wait for the token arrival;

Next_i := i;
Obtain the highest priority task as per the state information in the
token by sending Task_Request(i, Task.pty) message.
Remove the corresponding entry from the token.
4) A node $j$ on receiving a Token_Request ($i$) message:

Next$_j := i$; /* This step short circuits the path towards the node holding
the token. */
If $j$ has the token
    Send the token to $i$;
else
    Send Token_Request ($i$) to Next$_j$;

5) A node $j$ on receiving a Task_Request($i$, Task_pTy) message:

Next$_j := i$; /* This step short circuits the path towards the node holding the token. */
Send a task of priority equal to Task_pTy.

Upon a close inspection of the basic algorithm, we can identify many situations,
wherein delays in scheduling tasks and overheads due to the scheduling protocol can
be reduced. First, upon completion of every task, every node obtains the token before
scheduling the next task irrespective of the priority of the tasks waiting (if any) locally.
Since the percentage of priority tasks relative to the entire work load is normally less,
one can reduce the delay by scheduling a priority task (priority > lowest priority) if
one is waiting locally and not requesting the token in such a case.

The number of instances where a token is requested can further be reduced, thus,
reducing the delay between task completion and task scheduling as follows. A node
that has more than THRESHOLD (a parameter of the algorithm) number of lowest
priority tasks waiting, can schedule one of its local tasks instead of obtaining the
token and trying to find a higher priority task. This action is motivated by the
following fact: Livny and Melman [26] have shown that there is a high probability
of finding at least one idle node while at least one task waits for service elsewhere because of statistical fluctuations in the arrival of tasks to nodes and task service time requirements. In other words, we will let only idle or lightly loaded nodes search for priority tasks as the probability of the existence of an idle node at any given time is quite high in distributed systems. This modification prevents starvation for lowest priority tasks.

Because of the above modification, it may so happen that the token is informed of the arrival of a task of certain priority \( p \) which later gets scheduled locally. If another task of the same priority originates, the token need not be notified of this event, as it already has the information indicating that a task of priority \( p \) exists at the node.

Finally, the delay in scheduling tasks can further be reduced as follows. When a deposit message is on its way to the token, it may reach an intermediate node which is idle and is not requesting the token. In such cases, the intermediate node will request for the transfer of a task with priority greater than or equal to the priority found in the deposit message from the originator of the deposit message.

The improved algorithm given next incorporates all the above improvements. In addition, it provides the details of the actions taken when messages are received when a node is requesting the token, using the token, and is idle. In the following: Deposit\(_i[p]\), initialized to TRUE, is a boolean variable at node \( i \) which indicates whether the information regarding the origination of a task of priority \( p \) should be forwarded to the token; Intransit\(_i\), initialized to FALSE, is a boolean variable at node
i which indicates whether a task is in transit to node i; temp_deposit_queue_i is a queue at node i to hold the Deposit messages received at i when i is requesting the token; temp_token_queue_i is a queue at node i to hold the requests for the token while i is waiting for the token.

In the algorithms described in this chapter, an executing task cannot be preempted by another task of equal or higher priority. We chose this policy to avoid having to transfer partially executed tasks which are very expensive and difficult to transfer [2, 9]. The scheduling actions, such as sending and receiving messages, transferring tasks have preemptive priority over task execution.

The Improved Algorithm

1) When a task with priority Task_p ty originates at a node i:
   
   queue the task locally;
   if (node i is idle) AND (Intransit_i = FALSE) AND (No tasks are waiting)
   
   Execute the task locally
   
   else
   if (node i is busy OR Intransit_i = TRUE) AND (Deposit_i[Task_p ty] = TRUE)
   
   Send Deposit_Msg (Task_p ty, i) to Next;
   
   else
   if (node i is busy OR Intransit_i = TRUE) AND (Deposit_i[Task_p ty] = FALSE)

   Deposit_i[Task_p ty] = TRUE;

2) A node j on receiving a Deposit_Msg(Task_p ty, i) message:
   
   If (j has the token) AND (j is not idle)
   
   insert (Task_p ty, i) into the token
   
   else
   If requesting_token_j = TRUE


insert (Task.pty, i) into temp.deposit_queue;

else
If j is idle /* Whether j has the token or not */

Send Task_Request(j, Task.Pty) to i;
If a Sorry message is received, do nothing
/* This can happen either if i has received a Task_Request (r, p ≤ Task.pty)
or completed a task and started executing the Task.Pty task after
sending the deposit msg. */
else
If a (YES, task’s priority, n) message is received

\text{Intransit}_j := \text{TRUE};
if (task’s priority > Task.Pty) AND (n > 0)
Send Deposit_Msg(Task.pty, i) to Next_j.
/* Received a task with different priority than requested, so forward the original Deposit msg. */
else
Send Deposit_Msg(Task.pty, i) to Next_j.

3) When a node i completes a task, it performs the following operations:

if i does not have the token

if (a task with priority > lowest priority is present locally) OR
(no. of lowest priority tasks waiting locally > THRESHOLD)
Schedule a task with the highest priority among all the
local tasks.
Deposit_i[scheduled task’s priority] := FALSE;
else
requesting_token_i := \text{TRUE};
Send Token_Request (i) to Next_i; /* request token */
Wait for the token arrival;
Next_i := i; /* token has arrived */
requesting_token_i := \text{FALSE};

using_token_i := \text{TRUE};
Remove information from temp.deposit_queue, and insert into the token appropriately while temp.deposit_queue is not empty;
Remove information from temp_token_queue, and insert into token_queue in the token while temp_token_queue is not empty; /* token queue holds pending token requests in the token */

if (i is idle) AND (Intransit; = FALSE) AND (token is not empty)

While token is not empty do

Select a node k with the highest priority task as per the state information in the token and send Task_Request(i, Expected_pty) message to k;
If k replies with a Sorry message
Remove the corresponding (Expected_pty, k) tuple from the token;
If k replies with (YES, task's priority, n) message
Intransit; = TRUE;
if task's priority > Task_Pty
Remove the corresponding (task's priority, k) tuple from the token;

If token_queue is not empty /* request pending for token */

k := head(token_queue); remove k from token_queue; Send the token to k;

using_token; := FALSE;

4) A node j on receiving a Token_Request (i) message:

Nextj = i; /* This step short circuits the path towards the node holding the token. */
if (j has the token) AND (using_token_j = FALSE)
Send the token to i
else
if (j has the token) AND (using_token_j = TRUE)
insert i to token_queue /* token_queue holds pending token requests in the token. */
else
if (requesting_token_j = TRUE)
insert i to temp_token_queue;
else
Send Token_Request (i) to Next_j;
5) A node \( j \) on receiving a Task\_Request\((i, \text{Requested\_pty}) \) message:

- If the Task\_Request msg is from the node holding the token
  \[ \text{Next}_j = i; \quad \text{/* This step short circuits the path towards the node holding the token. */} \]

- If a task \( t \) of priority \( p \) such that \( p > \text{Requested\_pty} \) is available
  \[ \text{/* Send a task with the highest priority among the waiting tasks */} \]
  Send (YES, \( t \)'s priority, No. of Requested\_pty tasks waiting) to \( i \);
  Send task \( t \) to \( i \);

- else

- If a task \( t \) of priority \( p \) such that \( p = \text{Requested\_pty} \) is available
  Send (YES, \( t \)'s priority, 0) to \( i \);
  \( \text{Deposit}_j[p] := \text{TRUE}; \)
  Send task \( t \) to \( i \);

- else
  Send Sorry message to \( i \);
  \( \text{Deposit}_j[\text{Requested\_pty}] := \text{TRUE}; \)

6) On receipt of a task at node \( i \):

- \( \text{Intransit}_i := \text{FALSE}; \)
- Install the task and execute it;

5.4 List-based Algorithm

In this section, we modify the list-based algorithm (proposed in Chapter 3) to schedule priority tasks since it performed well for general systems. The list-based algorithm has two components, namely, transfer policy and location policy.
Transfer policy

The transfer policy identifies a node as a sender when the queue length at the node is greater than UPPERTHRESHOLD (a parameter of the algorithm). The transfer policy identifies a node as a receiver when the queue length at the node is less than LOWERTHRESHOLD (a parameter of the algorithm). The transfer policy identifies a node as OK when the queue length at the node is neither greater than UPPERTHRESHOLD nor less than LOWERTHRESHOLD.

Location Policy

Sender-Initiated Negotiation

When a task originates at a node $i$, if the transfer policy at node $i$ determines that node $i$ is a sender, the location policy at node $i$ does the following to find a suitable receiver:

1. Probe the node at the head of $Rlist_i$, say $j$, to determine whether $j$ is a receiver.

2. If $j$ replies that it is a receiver, return its ID to the transfer policy and STOP.

The transfer policy transfers a waiting task with the highest priority. If more than one tasks have identical priority, then the task which originated first is chosen. If $j$ is not a receiver, remove it from $Rlist_i$ and add it to the head of either $Slist_i$ if $j$ is a sender, or $OKlist_i$ if $j$ is OK.
3. STOP negotiation and return FAILURE to the transfer policy if either (a) \( i \) has probed \( \text{Probelimit} \) nodes (where \( \text{Probelimit} \) is a parameter of the algorithm) without success, (b) \( Rlist_i \) is empty, or (c) \( i \) is no longer a potential sender (due to task completion at the node). Otherwise go to Step 1.

At probed node \( j \):

1. Since this probe has identified node \( i \) as a sender, remove \( i \) from whatever list it is in and add it to the head of \( Slist_j \). (Note: If \( i \) was already in \( Slist_j \), this action simply changes its position in the list thereby reflecting the system state more accurately.)

2. Send a reply message to \( i \) indicating whether \( j \) is a receiver, sender, or OK (determined by the transfer policy at node \( j \)).

**Receiver-Initiated Negotiation**

To find a suitable sender, the location policy at a receiver node \( i \) does the following:

1. Probe a selected node, say \( j \), to determine whether \( j \) is a sender. The selected nodes are chosen in the following order:

   (a) From the head of \( Slist_i \) (most up-to-date information is used first).

   (b) If \( Slist_i \) is empty, then from the tail of \( OKlist_i \) (most out-of-date information is used first in the hope that the state has changed from OK to sender).
(c) If $OKlist_i$ is also empty, from the tail of $Rlist_i$ (again the most out-of-date information is used first).

Note that the contents of the lists may change as negotiation proceeds, because of negotiations initiated by other nodes occurring concurrently with this negotiation session. To avoid probing a node that has indicated that it is not a sender very recently, nodes that join $OKlist_i$ or $Rlist_i$ after a session has begun are not probed in that session.

2. If $j$ replies that it is a sender and that it will remain a sender after transferring a task, add it to the head of $Slist_i$, return its ID to the transfer policy, and STOP. If $j$ replies that it is a sender but will not remain a sender after transferring a task, remove it from whatever list it is in and add it to the head of $OKList_i$, return its ID to the transfer policy, and STOP. If $j$ replies that it is either a receiver or OK, remove it from whatever list it is in and add it to the head of either $Rlist_i$ (if it is a receiver), or $OKList_i$.

3. STOP negotiation and return FAILURE to the transfer policy if either $i$ is no longer a receiver (due to a task arrival), if $i$ has probed $Probelimit$ nodes without success, or if all the nodes that were in $OKlist_i$ and $Rlist_i$ have been probed.

At probed node $j$:

1. If $j$ is not a sender, return a message informing $i$ of $j$'s state (receiver or OK).

Since $i$'s probe has identified $i$ as a receiver, remove $i$ from whatever list it is in
and add it to the head of \( Rlist_j \).

2. If \( j \) is a sender, transfer a task with the highest priority to \( i \). Inform \( i \) of \( j \)'s state after the migration.

5.5 Performance Study

In this section, we study the performance of the two proposed algorithms and compare it with that of the ideal situation where a global observer guides scheduling, using up-to-date state information available to it at no cost. To make the study realistic, however, the overheads incurred to transfer and to receive tasks are included in all the three algorithms. We also provide performance figures for a system which performs no load distributing (denoted by NLD) which gives the worst bound for the performance, while ideal gives the optimistic bound. In keeping up with our non-preemptive local scheduling policy for the other two algorithms, NLD also uses non-preemptive local scheduling policy.

**System Model:** The results we present are from the simulations of a distributed system containing 30 nodes, interconnected by a 10 Megabit/second Ethernet communication network. Task interarrival times and service demands are assumed to be independently Exponentially distributed, and the average task CPU service demand is assumed to be one second. The size of scheduling messages is assumed to be 8 bytes, and the CPU overhead to either send or receive a message is assumed to be 3 milliseconds. A task transfer is assumed to incur a CPU overhead of 18 milliseconds.
Task transfer overhead is assumed to be divided evenly between the sending and the receiving nodes. The amount of information that must be communicated for a task transfer is assumed to be 8 Kbytes. We have studied the performance under two conditions, namely, 5 priority classes and 3 priority classes. For 5 priority classes, we assume that 50%, 15%, 15%, 10%, 10% of the tasks are priority 1, 2, 3, 4, 5 tasks, respectively. For 3 priority classes, we study the performance under two conditions: (1) We assume that 50%, 30%, 20% of the tasks are priority 1, 2, 3 tasks, respectively. (2) We assume that 80%, 12%, 8% of the tasks are priority 1, 2, 3 tasks, respectively. Finally, higher the number of the priority class, higher is the priority. All results we present have less than 5% error at 90% confident level, while most of the results have less than 2% error. 

Tables 1 through 9 compare the average task response time (expressed in units of mean service demand) under homogeneous workload, where all nodes are generating workload at the same rate. Tables 10 through 12 compare the average task response time under heterogeneous workload, at a constant system offered load of 80%. Under heterogeneous workload, only the load-generating nodes generate the entire system load, while the non load-generating nodes generate none. The “Ave” row in all the tables presents the average response time across all the priority classes. The “Right Decisions %” row gives the percentage of times where a task waiting with the highest priority in the entire system was scheduled by the scheduling algorithm. The “Ave Deviation” row gives the average difference between the highest priority task waiting
in the system and the priority of the task scheduled whenever the system makes a wrong decision (i.e., the system schedules a task of a priority less than that of the highest priority task waiting in the system). The "communication cost" row gives the average number of point-to-point messages exchanged in scheduling a task.

The key observation is that for high priority tasks (i.e., class 2 and above), the performance of the token-based (Tk) algorithm differs from that of the ideal (Id) algorithm (which of course is impossible to realize in practice) by at most 10% (see column Difference%) in all cases except when the offered load is 90%. When the offered load is 90% (see tables 3 and 6), the performance of the token-based algorithm differs from that of the ideal algorithm by at most 17%. In addition, the percentage of right decisions under token-based algorithm is over 80% under all the conditions studied except when the offered system load is 90% under homogeneous case. The performance of NLD is much worse than the token-based algorithm across the spectrum.

We also studied the performance of NLD under preemptive priority scheduling policy. While we do not present the performance figures for this case, NLD performed much worse than the token-based algorithm except for the highest priority tasks, where it was only marginally better. This shows that the token-based algorithm is collecting, disseminating, and updating the entire system state efficiently and quite accurately.

The performance of the token based algorithm for high priority tasks is consistently better than the list based algorithm under all the conditions studied, and especially better at high offered system loads. The reason for this better performance is because
the percentage of right decisions under token-based algorithm is always higher than under the list based algorithm. In addition, the average deviation under token-based algorithm is consistently less compared to the list-based algorithm. A key thing to note is that, the communication cost under token-based algorithm is also less than under that of the list-based algorithm except under high offered load of 90%.

5.6 Fault-tolerance

One concern with the token-based algorithm is that of the token loss. Token loss, however, is of no major concern as far the state information in the token. Temporary token loss will result in poor scheduling decisions. Some degree of load distribution will still occur since the idle nodes in the path towards the token will obtain tasks from the originator of deposit messages. Once the token is generated, the NEXT pointer at all nodes needs to be updated to point to the current location of the token. A node waiting for the token can employ a timeout to trigger the detection of token loss and the generation of a new token. Protocols to detect the token loss and regenerate the token can be found in [30].

5.7 Conclusions

Collecting, disseminating, and maintaining system state information is a difficult task and typically imposes substantial overhead in distributed systems. Scheduling priority tasks is one application which requires up-to-date system state. In this chapter,
we proposed a token-based algorithm for scheduling tasks with priority in distributed systems. We also modified the list-based algorithm previously proposed for general systems to schedule priority tasks. We compared the token-based algorithm's performance with an ideal algorithm and found that the new algorithm collects, disseminates, and updates the entire system state efficiently and quite accurately. In addition, a system employing the token-based algorithm is shown to perform better than the list-based algorithm and substantially better than a system that performs no load distributing at all.
Table 1: Homogeneous workload, THRESHOLD = 1, LOWERTHRESHOLD = 1, UPPERTHRESHOLD = 1, Response time at Offered system load of 50%.

<table>
<thead>
<tr>
<th>Priority</th>
<th>NLD</th>
<th>List</th>
<th>Tk</th>
<th>Id</th>
<th>Difference%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.3352</td>
<td>1.0693</td>
<td>1.0573</td>
<td>1.0307</td>
<td>2.58</td>
</tr>
<tr>
<td>2</td>
<td>1.8158</td>
<td>1.0666</td>
<td>1.0567</td>
<td>1.0274</td>
<td>2.85</td>
</tr>
<tr>
<td>3</td>
<td>1.6844</td>
<td>1.0809</td>
<td>1.0594</td>
<td>1.0292</td>
<td>2.93</td>
</tr>
<tr>
<td>4</td>
<td>1.5883</td>
<td>1.0752</td>
<td>1.0466</td>
<td>1.0295</td>
<td>1.66</td>
</tr>
<tr>
<td>5</td>
<td>1.5286</td>
<td>1.0755</td>
<td>1.0652</td>
<td>1.0126</td>
<td>5.19</td>
</tr>
<tr>
<td>Ave</td>
<td>2.0044</td>
<td>1.0785</td>
<td>1.0572</td>
<td>1.028</td>
<td>2.84</td>
</tr>
<tr>
<td>Right Decisions %</td>
<td>93.76</td>
<td>96.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ave Deviation</td>
<td>2.1614</td>
<td>2.1132</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comm Cost (msgs/task)</td>
<td>11.21</td>
<td>8.38</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Homogeneous workload, THRESHOLD = 1, LOWERTHRESHOLD = 1, UPPERTHRESHOLD = 1, Response time at Offered system load of 70%.

<table>
<thead>
<tr>
<th>Priority</th>
<th>NLD</th>
<th>List</th>
<th>Tk</th>
<th>Id</th>
<th>Difference%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.5622</td>
<td>1.1476</td>
<td>1.144</td>
<td>1.0549</td>
<td>8.44</td>
</tr>
<tr>
<td>2</td>
<td>2.4376</td>
<td>1.1528</td>
<td>1.0984</td>
<td>1.0568</td>
<td>3.93</td>
</tr>
<tr>
<td>3</td>
<td>2.078</td>
<td>1.1349</td>
<td>1.0987</td>
<td>1.0341</td>
<td>6.25</td>
</tr>
<tr>
<td>4</td>
<td>1.8907</td>
<td>1.1451</td>
<td>1.1006</td>
<td>1.0424</td>
<td>5.58</td>
</tr>
<tr>
<td>5</td>
<td>1.7223</td>
<td>1.1299</td>
<td>1.082</td>
<td>1.044</td>
<td>3.64</td>
</tr>
<tr>
<td>Ave</td>
<td>3.3217</td>
<td>1.1445</td>
<td>1.1248</td>
<td>1.0497</td>
<td>7.15</td>
</tr>
<tr>
<td>Right Decisions %</td>
<td>75.88</td>
<td>88.59</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ave Deviation</td>
<td>2.3159</td>
<td>2.1288</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comm Cost (msgs/task)</td>
<td>11.59</td>
<td>7.1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 3: Homogeneous workload, THRESHOLD = 1, LOWERTHRESHOLD = 1, UPPERTHRESHOLD = 1, Response time at Offered system load of 90%.

<table>
<thead>
<tr>
<th>Priority</th>
<th>NLD</th>
<th>List</th>
<th>Tk</th>
<th>Id</th>
<th>Difference%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17.1743</td>
<td>2.0159</td>
<td>2.6024</td>
<td>1.8076</td>
<td>43.97</td>
</tr>
<tr>
<td>2</td>
<td>3.3682</td>
<td>1.6833</td>
<td>1.2992</td>
<td>1.1159</td>
<td>16.42</td>
</tr>
<tr>
<td>3</td>
<td>2.6011</td>
<td>1.5801</td>
<td>1.2228</td>
<td>1.0945</td>
<td>11.72</td>
</tr>
<tr>
<td>4</td>
<td>2.2204</td>
<td>1.5085</td>
<td>1.1616</td>
<td>1.0886</td>
<td>6.7</td>
</tr>
<tr>
<td>5</td>
<td>2.0123</td>
<td>1.4768</td>
<td>1.1473</td>
<td>1.0741</td>
<td>6.81</td>
</tr>
<tr>
<td>Ave</td>
<td>9.9058</td>
<td>1.796</td>
<td>1.9104</td>
<td>1.451</td>
<td>31.66</td>
</tr>
</tbody>
</table>

Right Decisions % | 24.33 | 68.5
Ave Deviation     | 2.8375 | 2.2144
Comm Cost (msgs/task) | 5.37 | 6.11

Table 4: Homogeneous workload, THRESHOLD = 1, LOWERTHRESHOLD = 1, UPPERTHRESHOLD = 1, Percentage of Priority Tasks = 50%, Response time at Offered system load of 50%.

<table>
<thead>
<tr>
<th>Priority</th>
<th>NLD</th>
<th>List</th>
<th>Tk</th>
<th>Id</th>
<th>Difference%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.3352</td>
<td>1.0693</td>
<td>1.0625</td>
<td>1.0303</td>
<td>3.12</td>
</tr>
<tr>
<td>2</td>
<td>1.7498</td>
<td>1.0738</td>
<td>1.0555</td>
<td>1.0301</td>
<td>2.46</td>
</tr>
<tr>
<td>3</td>
<td>1.5582</td>
<td>1.0754</td>
<td>1.0638</td>
<td>1.0192</td>
<td>4.37</td>
</tr>
<tr>
<td>Ave</td>
<td>2.0042</td>
<td>1.0719</td>
<td>1.0606</td>
<td>1.028</td>
<td>3.17</td>
</tr>
</tbody>
</table>

Right Decisions % | 96.25 | 98.5
Ave Deviation     | 2.3372 | 1.3202
Comm Cost (msgs/task) | 11.21 | 8.36


Table 5: Homogeneous workload, THRESHOLD = 1, LOWERTHRESHOLD = 1, UPPERTHRESHOLD = 1, Percentage of Priority Tasks = 50%, Response time at Offered system load of 70%.

<table>
<thead>
<tr>
<th>Priority</th>
<th>NLD</th>
<th>List</th>
<th>Tk</th>
<th>Id</th>
<th>Difference%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.5662</td>
<td>1.1479</td>
<td>1.1523</td>
<td>1.0507</td>
<td>9.67</td>
</tr>
<tr>
<td>2</td>
<td>2.2569</td>
<td>1.1462</td>
<td>1.1018</td>
<td>1.034</td>
<td>6.55</td>
</tr>
<tr>
<td>3</td>
<td>1.8064</td>
<td>1.1373</td>
<td>1.0943</td>
<td>1.0525</td>
<td>3.97</td>
</tr>
<tr>
<td>Ave</td>
<td>3.3215</td>
<td>1.1453</td>
<td>1.1255</td>
<td>1.046</td>
<td>7.6</td>
</tr>
<tr>
<td>Right Decisions %</td>
<td>78.27</td>
<td>88.83</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ave Deviation</td>
<td>1.391</td>
<td>1.3159</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comm Cost (msgs/task)</td>
<td>11.59</td>
<td>7.09</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Homogeneous workload, THRESHOLD = 1, LOWERTHRESHOLD = 1, UPPERTHRESHOLD = 1, Percentage of Priority Tasks = 50%, Response time at Offered system load of 90%.

<table>
<thead>
<tr>
<th>Priority</th>
<th>NLD</th>
<th>List</th>
<th>Tk</th>
<th>Id</th>
<th>Difference%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17.4571</td>
<td>1.9951</td>
<td>2.6446</td>
<td>1.6677</td>
<td>58.57</td>
</tr>
<tr>
<td>2</td>
<td>3.0036</td>
<td>1.6369</td>
<td>1.2577</td>
<td>1.1042</td>
<td>13.9</td>
</tr>
<tr>
<td>3</td>
<td>2.1222</td>
<td>1.4977</td>
<td>1.1485</td>
<td>1.0634</td>
<td>8.0</td>
</tr>
<tr>
<td>Ave</td>
<td>10.0541</td>
<td>1.7882</td>
<td>1.9317</td>
<td>1.3779</td>
<td>40.02</td>
</tr>
<tr>
<td>Right Decisions %</td>
<td>31.1</td>
<td>68.19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ave Deviation</td>
<td>1.5621</td>
<td>1.3531</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comm Cost (msgs/task)</td>
<td>5.25</td>
<td>6.02</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 7: Homogeneous workload, THRESHOLD = 1, LOWERTHRESHOLD = 1, UPPERTHRESHOLD = 1, Percentage of Priority Tasks = 20%, Response time at Offered system load of 50%.

<table>
<thead>
<tr>
<th>Priority</th>
<th>NLD</th>
<th>List</th>
<th>Tk</th>
<th>Id</th>
<th>Difference%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.1162</td>
<td>1.071</td>
<td>1.0619</td>
<td>1.0313</td>
<td>2.96</td>
</tr>
<tr>
<td>2</td>
<td>1.5823</td>
<td>1.0749</td>
<td>1.0687</td>
<td>1.0379</td>
<td>2.97</td>
</tr>
<tr>
<td>3</td>
<td>1.5229</td>
<td>1.0768</td>
<td>1.0631</td>
<td>1.03</td>
<td>3.21</td>
</tr>
<tr>
<td>Ave</td>
<td>2.0047</td>
<td>1.0719</td>
<td>1.0628</td>
<td>1.032</td>
<td>2.98</td>
</tr>
</tbody>
</table>

Right Decisions %: 96.79, 98
Ave Deviation: 1.3886, 1.3798
Comm Cost (msgs/task): 11.21, 8.37

Table 8: Homogeneous workload, THRESHOLD = 1, LOWERTHRESHOLD = 1, UPPERTHRESHOLD = 1, Percentage of Priority Tasks = 20%, Response time at Offered system load of 70%.

<table>
<thead>
<tr>
<th>Priority</th>
<th>NLD</th>
<th>List</th>
<th>Tk</th>
<th>Id</th>
<th>Difference%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.7025</td>
<td>1.1463</td>
<td>1.1344</td>
<td>1.0495</td>
<td>8.09</td>
</tr>
<tr>
<td>2</td>
<td>1.8699</td>
<td>1.143</td>
<td>1.099</td>
<td>1.0381</td>
<td>5.86</td>
</tr>
<tr>
<td>3</td>
<td>1.7117</td>
<td>1.129</td>
<td>1.082</td>
<td>1.0436</td>
<td>3.68</td>
</tr>
<tr>
<td>Ave</td>
<td>3.3223</td>
<td>1.1445</td>
<td>1.126</td>
<td>1.0477</td>
<td>7.47</td>
</tr>
</tbody>
</table>

Right Decisions %: 85.36, 94.48
Ave Deviation: 1.4293, 1.384
Comm Cost (msgs/task): 11.59, 7.08
Table 9: Homogeneous workload, THRESHOLD = 1, LOWERTHRESHOLD = 1, UPPERTHRESHOLD = 1, Percentage of Priority Tasks = 20%, Response time at Offered system load of 90%.

<table>
<thead>
<tr>
<th>Priority</th>
<th>NLD</th>
<th>List</th>
<th>Tk</th>
<th>Id</th>
<th>Difference%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12.0683</td>
<td>1.8129</td>
<td>2.0422</td>
<td>1.398</td>
<td>46.08</td>
</tr>
<tr>
<td>2</td>
<td>2.2046</td>
<td>1.4966</td>
<td>1.1721</td>
<td>1.0727</td>
<td>9.26</td>
</tr>
<tr>
<td>3</td>
<td>1.9972</td>
<td>1.4723</td>
<td>1.1397</td>
<td>1.0722</td>
<td>6.29</td>
</tr>
<tr>
<td>Ave</td>
<td>10.079</td>
<td>1.7477</td>
<td>1.8656</td>
<td>1.3335</td>
<td>39.9</td>
</tr>
</tbody>
</table>

Right Decisions % 33.15 84.46
Ave Deviation 1.5916 1.3874
Comm Cost (msgs/task) 5.48 6.05

Table 10: Heterogeneous workload, THRESHOLD = 1, Response time when No. of load generating nodes = 30.

<table>
<thead>
<tr>
<th>Priority</th>
<th>List</th>
<th>Tk</th>
<th>Id</th>
<th>Difference%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.289</td>
<td>1.3748</td>
<td>1.1329</td>
<td>21.35</td>
</tr>
<tr>
<td>2</td>
<td>1.2589</td>
<td>1.1406</td>
<td>1.0684</td>
<td>6.75</td>
</tr>
<tr>
<td>3</td>
<td>1.2441</td>
<td>1.1283</td>
<td>1.0563</td>
<td>6.81</td>
</tr>
<tr>
<td>4</td>
<td>1.2323</td>
<td>1.1305</td>
<td>1.0529</td>
<td>7.37</td>
</tr>
<tr>
<td>5</td>
<td>1.2107</td>
<td>1.1215</td>
<td>1.0488</td>
<td>6.93</td>
</tr>
<tr>
<td>Ave</td>
<td>1.2643</td>
<td>1.2529</td>
<td>1.0953</td>
<td>14.38</td>
</tr>
</tbody>
</table>

Right Decisions % 55.4 82.47
Ave Deviation 2.8375 2.1542
Comm Cost (msgs/task) 10.35 6.96
Table 11: Heterogeneous workload, THRESHOLD = 1, Response time when No. of load generating nodes = 20.

<table>
<thead>
<tr>
<th>Priority</th>
<th>List</th>
<th>Tk</th>
<th>Id</th>
<th>Difference%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.279</td>
<td>1.4201</td>
<td>1.1558</td>
<td>22.86%</td>
</tr>
<tr>
<td>2</td>
<td>1.2314</td>
<td>1.1487</td>
<td>1.0702</td>
<td>7.33%</td>
</tr>
<tr>
<td>3</td>
<td>1.2213</td>
<td>1.1451</td>
<td>1.0677</td>
<td>7.25%</td>
</tr>
<tr>
<td>4</td>
<td>1.2003</td>
<td>1.126</td>
<td>1.0638</td>
<td>5.84%</td>
</tr>
<tr>
<td>5</td>
<td>1.1943</td>
<td>1.1192</td>
<td>1.0502</td>
<td>6.57%</td>
</tr>
<tr>
<td>Ave</td>
<td>1.2469</td>
<td>1.2787</td>
<td>1.11</td>
<td>15.2%</td>
</tr>
<tr>
<td>Right Decisions %</td>
<td>58.56</td>
<td>81.59</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ave Deviation</td>
<td>2.4516</td>
<td>2.1441</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comm Cost (msgs/task)</td>
<td>10.28</td>
<td>7.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 12: Heterogeneous workload, THRESHOLD = 1, Response time when No. of load generating nodes = 10.

<table>
<thead>
<tr>
<th>Priority</th>
<th>List</th>
<th>Tk</th>
<th>Id</th>
<th>Difference%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.262</td>
<td>1.4692</td>
<td>1.1319</td>
<td>29.8%</td>
</tr>
<tr>
<td>2</td>
<td>1.2113</td>
<td>1.16</td>
<td>1.0634</td>
<td>9.08%</td>
</tr>
<tr>
<td>3</td>
<td>1.201</td>
<td>1.1598</td>
<td>1.0658</td>
<td>8.82%</td>
</tr>
<tr>
<td>4</td>
<td>1.1953</td>
<td>1.1354</td>
<td>1.0509</td>
<td>8.04%</td>
</tr>
<tr>
<td>5</td>
<td>1.1795</td>
<td>1.1271</td>
<td>1.0478</td>
<td>7.56%</td>
</tr>
<tr>
<td>Ave</td>
<td>1.231</td>
<td>1.307</td>
<td>1.0952</td>
<td>19.34%</td>
</tr>
<tr>
<td>Right Decisions %</td>
<td>65.0</td>
<td>81.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ave Deviation</td>
<td>2.3804</td>
<td>2.1383</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comm Cost (msgs/task)</td>
<td>9.64</td>
<td>7.15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER VI

Summary and Future Work

Over the past decade, the mode of computing has shifted from mainframes to networks of computers, which are often engineering workstations. Such a network promises higher performance, better reliability, and improved extensibility over mainframe systems. The total computing capacity of such a network can be enormous; however, to realize the performance potential of such a system, a good load distributing scheme is essential.

6.1 Summary

General Systems

Global scheduling algorithms improve the performance of general systems by judiciously redistributing load among the nodes in the system. In this thesis, we have presented two adaptive location policies, namely, symmetrically-initiated adaptive location policy and sender-initiated adaptive location policy for global scheduling algorithms. By increasing the efficiency of negotiations, these policies improve performance but at the same time reduce the threat of system instability relative to non-adaptive policies, yet add little complexity to global scheduling.
A key feature of the proposed policies is that they do not discard the data gathered during probing which is normally discarded by previously proposed policies. In addition, whenever a node receives a probe it implicitly learns the state of the probing node. In other words, no extra messages are required to collect state information. The information gathered is used to focus the probes towards the nodes that are most likely to be in a suitable state instead of probing randomly selected nodes. Moreover, the information gathered is used to cut off probes when they are likely to fail and hurt system performance. Another key feature of the proposed policies is that they are general, and can be used with a broad range of transfer policies.

The symmetrically-initiated adaptive location policy presented maintains stability and out-performs all the non-adaptive policies studied at all system loads under both homogeneous and heterogeneous workloads. An interesting and an important performance feature of this policy is that its performance improves under heterogeneous workloads as the degree of heterogeneity increases. This improvement in performance in effect indicates that the proposed policy adapts well to the changing system operating conditions.

The sender-initiated adaptive location policy performs better than its non-adaptive counterpart under all conditions studied. A limitation of the sender-initiated adaptive policy is that it is not stable under extreme workload heterogeneity. While not matching the performance of the symmetrically-initiated adaptive policy, the sender-initiated policy has the advantage of not requiring transfers of partly-executed tasks.
In conclusion, the proposed adaptive location policies are simple enough to be practical, yet effective over the wide range of workload conditions that are likely to occur in practical systems.

**Real-time Systems**

In real-time systems, tasks have deadlines associated with them and they must complete by their deadlines. Previously it has been shown that the number of tasks that miss their deadlines can be reduced by judiciously transferring tasks among nodes in the system. Algorithms proposed previously for real-time systems perform only corrective actions, i.e., task transfer activity is triggered only when a task's deadline cannot be met or when local load is high. These algorithms perform very poorly when the deadlines are tight (even when the system load is low) due to the occurrence of the cumulative phenomenon.

In this thesis, we proposed a load index and a transfer policy. The load index "free-time" gives a more accurate indication of a node's ability to satisfy the deadlines of future arrivals than just the "queue-length" at a node. The free-time at a node is composed of idle times in the node's schedule. Only a small part of the schedule encompassing the near term activities is used to compute free-time as it indicates a node's ability to serve a task immediately and keeps the overhead to compute free-time small.
The new transfer policy proposed takes preventive measures by doing anticipatory transfers in addition to corrective measures, thereby, reducing the number of potential deadline misses. By using free-time to characterize the node state, it adapts much better to the system state by looking-ahead and does not require the transfer of partially executed tasks. Finally, the transfer policy is not very sensitive to minor variations of the free-time thresholds and window size, and in this sense the new transfer policy is robust. A simulation study reveals that synergism resulting from Least-Laxity-First local scheduling policy, free-time, and our transfer policy, reduces the number of deadline misses significantly over a wide range of system parameters.

The proposed transfer policy and the computation of free-time are simple enough to be practical, yet effective over a wide range of conditions.

Priority Systems

In distributed systems that serve tasks with different priorities, the scheduler's objective is to schedule highest priority task waiting in the system first. This objective is difficult to realize in practice as the state information—tasks with what priority are waiting at which nodes—is not readily available in distributed systems. Scheduling priority tasks requires up-to-date system state. Collecting, disseminating, and maintaining system state information is a difficult task and typically imposes substantial overhead in distributed systems. The overhead incurred by the scheduler can outweigh the benefit of priority scheduling in distributed systems.
In this thesis, we proposed a token-based algorithm for scheduling tasks with priority in distributed systems. The basic idea behind this algorithm is as follows. Each node forwards information regarding task arrivals and their priorities to the token. The token acts as a floating repository. It stores the state information and moves between nodes as necessary. The node holding the token updates the token with the incoming state information. Nodes that are ready to schedule their next task, obtain the token and use the state information available in the token to schedule the highest priority task waiting in the system. The node with the token also updates the token to reflect the new system state after it has scheduled a task, which requires no message exchange. To locate the token efficiently (i.e., without exchanging a large number of messages), we make use of distributed pointers that maintain a path to the location of the token. The pointers are updated as and when the requests for token and requests for tasks are received at nodes. This dynamic updating helps shorten the path to the token as the pointers values reflect the changing system state.

We compared the performance of the token-based algorithm with an ideal (but infeasible) algorithm and found that the new algorithm collects, disseminates, and updates the entire system state efficiently and quite accurately. We also modified the symmetrically-initiated algorithm proposed in Chapter 3 for scheduling priority tasks. The performance study shows that a system employing the token-based algorithm performs better than the symmetrically-initiated algorithm and performs substantially better than a system that performs no load distributing at all.
Finally, all the protocols proposed in this thesis except the symmetrically initiated location policy are designed to use non-preemptive transfers exclusively. As a result, all carry relatively low overhead, both in terms of software development and maintenance time, and in terms of execution overhead.

6.2 Future Research

Over the past decade, a number of schemes and protocols have been proposed for load distributing in distributed systems. In addition, several experimental systems for load distribution have been developed. In spite of these developments, implementing nonpreemptive task transfers (transferring tasks that are yet to begin execution) and preemptive task transfers (transferring partially executed transfers) is still an extremely complex undertaking. Preemptive task transfers involve transferring a task that is partially executed. This operation is generally expensive, since collecting a task’s state (which can be quite large and/or complex) is often difficult. Typically, a task state consists of a virtual memory image, a process control block, unread I/O buffers and messages, file pointers, timers that have been set, etc. Non-preemptive task transfers, on the other hand, involve only the tasks which have not begun execution, and hence do not require transferring the task’s state. In both types of transfers, information about the environment in which the task will execute must be transferred to the remote node. This information may include the user's current working directory, the privileges inherited by the task, etc.
One of the reasons for the difficulty in collecting a task’s state is that the state information is hard to locate. For examples: unread I/O buffers may be buried in the data structure of the file system module of the operating system; unread messages and ports that are set up by a task may be buried in the data structure of the module implementing the IPC mechanism. Therefore a question is whether the organization of process’s state currently used (e.g., UNIX) is a good organization in light of the requirements of load distributing environment.

Future research also involves development of suitable structure for storing process state. An option is to look into the suitability of 64-bit addressing architecture [8, 16, 40] for load distributing environment. An attractive feature of this architecture is that it provides unique memory addresses across the network. Therefore, when a task is transferred to a new host, all its addresses and pointers values are still valid at the new host. This feature may make developing load distributing mechanisms simpler.

Most of the previously proposed load distributing schemes assume LAN based distributed systems. They assume logical connectivity among all the computers and in their current proposed form, they do not take advantage of the underlying topology or the communication protocols of the network. For example, fiber-optic communication system using wavelength-division multiplexing techniques allow multiple channels on a single fiber-optic cable. Taking advantage of the features specific to the underlying communication facility needs to be explored further.
Finally, there is much work to be done in the area of analytical performance modeling of adaptive load distributing protocols. Future research can focus on developing analytical models for the protocols developed in this thesis.
BIBLIOGRAPHY


[38] N. G. Shivarati and M. Singhal. A Transfer Policy for Global Scheduling Algo-
rithms to Schedule Tasks with Deadlines. In Proceedings of the 11th International

ties for the V-System. In Proceedings of the 10th ACM Symposium on Operating

Protection in a 64-bit Address Space. In Proceedings of the Summer USENIX