A Performance Study on Dynamic Load Balancing Algorithms

BY

SAU-MING LAU

JANUARY 1995

REVISED ON JUNE 1995

SUPERVISED BY

DR. CHIN LU

A THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF PHILOSOPHY

DIVISION OF COMPUTER SCIENCE

THE CHINESE UNIVERSITY OF HONG HONG

To my parents.

To my Jacqueline.

Abstract

This thesis presents a performance study on dynamic load balancing algorithms, which strive to use the current (or near current) system load information to balance the workload among the processing nodes in a distributed system. Design issues of dynamic load balancing algorithms are also discussed. The major result of this study is a new concept of workload distribution called the *batch assignment*. The possibility of adopting both task assignment and task migration as the workload distribution mechanisms in dynamic load balancing algorithms is also studied.

We found that an adaptive symmetrically-initiated polling-based location policy exhibits the *filtering effect*, which avoids inappropriate processing nodes to be selected for task transfer negotiations. This provides stability to the system at low and high system loads. However, the filtering effect causes an adverse effect called the *processor thrashing*, which results in reduced workload distribution and wasted processing capacity in the system. To put an adaptive symmetrically-initiated polling-based location policy into practical use therefore requires processor thrashing to be resolved.

Another design issue of dynamic load balancing algorithms is the ability to adapt to different task arrival patterns. In particular, the ability to resolve congestions in systems subjected to bursty task arrival patterns efficiently is desired. Most existing load balancing algorithms are in lack of such ability.

In an attempt to solve these two issues, the batch assignment approach is proposed. Batch assignment is based on the tight integration of three components:

1. A batch transfer policy, which allows a number of tasks to be transferred as a single batch from a sender to a receiver. It can smooth out workload imbalance with

significantly less negotiation sessions, and thus CPU and communication overheads are reduced to a minimum. It is the primary vehicle for resolving congestions in systems with bursty task arrivals. Central to the batch transfer policy are three heuristic-based "*Batch Size Determination Rules*" which govern the decision on the optimal batch size.

2. The "*Guarantee and Reservation Protocol*," which together with the batch transfer policy, obtains the mutual agreement between a sender and a receiver on the optimal batch size. The central idea of the GR Protocol is two fold: (1) A sender node has to declare the number of tasks that it guarantees to send to a receiver; and (2) A receiver employs a "quota" scheme for reserving processing capacity for task batches from senders. The GR Protocol can avoid a receiver from being flooded due to incoming task batches. It is the primary vehicle in resolving processor thrashing.

3. An adaptive symmetrically-initiated location policy based on the approach proposed by Shivaratri and Krueger in [SK90].

On the other hand, we show that although task migration in general costs more than task assignment, it can be used as an alternative workload distribution mechanism to augment task assignment for providing extra performance improvement. This is in contrast to the common belief that task assignment should be the sole workload distribution mechanism in dynamic load balancing.

Lastly, we show how a heterogeneous system can be modeled with a set of 3-tuples $(M_i, Throughput_i, \psi_i)$, and how task type compositions imposed on the system can be modeled with a set of 4-tuples $(J_i, w_{1,i}, l_i, \lambda_i)$. We explain why the measurement of workload of a processing node cannot simply be based on the number of tasks residing in the node. Instead, we defined the *node weight* as a basis for workload measurement. We also show that a task selection scheme should cater for any difference in processing throughputs between a sender node and a receiver node. This is important in making the most efficient use of a sender-receiver negotiation session. In particular, we model batch composition as a *Subset-Sum Problem* and a *greedy* solution has been proposed.

Acknowledgement

I would like to express my deepest gratitude to my research advisor, Dr. Chin LU. Her advices and constant encouragement contributed a great deal in this research work. I would also like to thank Dr. Chi-Hung CHI and Dr. Man-Hon WONG for their efforts in making comments on this thesis.

I am also grateful to all the friends I met in the Department of Computer Science, CUHK. Their encouragement, companions, and most importantly, their patience to my poor temper, made my life in CUHK delightful. Special thanks are expressed to Chi-Lok CHAN, Wing-Chung CHAN, and Kei-Fu MAK, who are always willing to accompany me during my difficult time; and to Karmen CHUI and Kelvin CHAK, who gave me valuable comments on Chapters 1 to 4.

Finally, I must dedicate this work to my family members, especially my parents, and to my girl friend Jacqueline. Their understanding, encouragement, support and patience make this thesis possible. I really do not know how can I return all the things I owe them.

Sau-Ming LAU.

January 1995.

Thesis revised on June 1995.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In a distributed computing system (DCS) where a set of processing nodes are connected by a local area network, some nodes tend to have higher task arrival rates than others [TL89] [ML87]. It is desirable for such workload imbalance to be smoothed out so as to make use of the computing capacity of idle or lightly loaded nodes. Thus the CPU utilization and total system throughput can be maximized while the average task response time can be minimized. *Load balancing algorithms* try to accomplish this objective by distributing tasks among processing nodes so that the workload on each node is approximately the same.

The two most commonly used workload distribution mechanisms are task assignment and task migration. *Task assignment* refers to the initial placement of tasks to processing nodes. *Task migration* is the dynamic relocation of an executing task to another processing node. With task migration, the task being migrated is suspended. The *execution state* of the task is then captured and transferred to a remote node, where the task resumes its execution. Typically, the execution state of a task consists of a virtual memory image, a process control block, I/O buffers, messages, file pointers, timers, and so on [Smi88] [SKS92]. The constituent state of a task varies widely with different operating systems. In general, task migration costs more than task assignment in terms of both CPU and communication overheads [Smi88].

Load balancing algorithms can be divided into static and dynamic. With *static* load balancing algorithms, there is *a priori* assignment of processes to processing nodes. The

current state of the distributed system is not taken into consideration [Bok79] [Bok87] [ST85] [Lo88]. With *dynamic* load balancing algorithms, the current (or near current) system load information is taken into account to decide where in the network a task should be processed. Many dynamic algorithms have been discussed [EL86a] [EL86b] [NXG85] [SS84] [Zho88]. This thesis focuses on dynamic load balancing algorithms only.

Besides, load balancing algorithms can be either *adaptive* or *non-adaptive*. An adaptive algorithm is one which changes its decision making policies dynamically according to the previous and current behavior of the distributed system, whereas the policies in a non-adaptive algorithm are fixed, regardless the current or past system behavior.

One of the objectives of this study is to measure the performance of different dynamic load balancing algorithms. As there is a diversity of different load balancing algorithms, which have very different system models, assumptions, and design objectives, it is rather difficult to have a fair comparison between the relative merits among these algorithms. Therefore, an important step in this research work is to develop a system model which serves as a common framework, so that different load balancing algorithms can be compared objectively. Moreover, the performance and efficiency metrics for assessing load balancing algorithms have to be selected and designed carefully.

The second objective of this study is to design dynamic load balancing algorithms which are practical to be implemented in terms of both performance and efficiency. Due to communication delays, a complete and up-to-date picture of workload states of a distributed system may never be obtained. Thus an important design issue of dynamic load balancing algorithms is to study how the local view of the system workload states can be maintained at a node. In general, there are two major methods of doing this: one which assumes the existence of a local load table which stores the load states of all the nodes in the distributed system; and one which relies on polling for gathering workload state information.

Among polling algorithms, adaptive symmetrically-initiated algorithms are the most promising [SK90] [LL94]. In such algorithms, both senders and receivers can initiate the search for transfer partners. However, this kind of algorithms exhibit a phenomenon

called *processor thrashing*, which means that a number of processing nodes poll for (or even transfer tasks to) the same potential transfer partner node simultaneously. Processor thrashing is found to have adverse effects on the system performance. In particular, certain amount of processing capacity is wasted because of the limited degree of workload distribution in the system. This issue has not been extensively studied in literatures however, for example, numerical measurement of the degree of processor thrashing has not been proposed. To resolve processor thrashing is another important design issue of dynamic load balancing algorithms.

Most existing load balancing algorithms rely on the assumption that the workload arrival pattern to the processing nodes in the distributed system is rather stable. Such algorithms cannot provide satisfactory performance when the system is injected with bursty task arrival patterns. This is because such algorithms cannot resolve congestions occurring in the processing nodes exhibiting bursty task arrivals efficiently. The usability of such algorithms is therefore limited to systems with stable workload pattern. The ability to adapt to both stable and bursty task arrival patterns is another design objective of our dynamic load balancing algorithms.

Currently, most load balancing literatures focus either on the interprocessor protocols for the identification of appropriate transfer partners, or on the measurement of processor workload. It is assumed that only a single task can be transferred during each sender-receiver negotiation session, either by task assignment or by task migration. To transfer a certain amount of workload, multiple sender-receiver negotiation sessions are required. We refer to this approach as the *single task assignment/migration*. In contrast, we develop a new task transfer approach, namely the *batch assignment*. Batch assignment is an adaptive mechanism which allows a number of tasks to be transferred as a single batch from a sender to a receiver for each single sender-receiver negotiation session. The *batch size* (the number of tasks contained in a task batch) is determined dynamically according to the relative busyness between the sender and the receiver. Batch transfer can avoid the unnecessary CPU and communication overheads injected by multiple sender-receiver negotiation sessions when a number of tasks have to be transferred between a pair of

processing nodes. It is proved that in terms of both task response time and system predictability, the batch assignment approach performs significantly better than algorithms which allow only single task transfer. To summarize, the different workload distribution approaches that we study in this thesis are: (1) single task assignment, (2) single task migration, (3) batch assignment, and (4) combinations of these.

The design and study of the batch assignment approach represents one line of our research work. Another line of this research is the study of the possibility of adopting both task assignment and task migration as the workload distribution mechanism in a load balancing algorithm. Most of the existing load balancing algorithms assume that task assignment is the sole workload distribution mechanism because it costs much less than task migration. However, we are interested in investigating situations where task migration can be used as an alternative workload transfer mechanism to augment task assignment for providing extra performance improvement. This is particularly important when a sender can find no appropriate "fresh" task for remote assignment, in which case the heavily loaded sender has no way to share the spare processing capacity of potential receivers if only task assignment is allowed.

We are thus now having two alternative approaches for improving the performance of load balancing algorithms. One is the batch assignment approach and the other is the adoption of task migration. As task migration is scarcely supported in today's distributed operating systems, we are also interested in knowing the relative merits between these two approaches.

Lastly, we will apply our batch assignment algorithm to heterogeneous systems in which processing nodes of the distributed system may have different processing throughputs and may be *functionally incompatible* to each other. We will show how we can measure the relative workload of the heterogeneous processing nodes and how we can modify the batch assignment approach to adapt to the heterogeneity.

The rest of this thesis is organized as follows. Chapter 2 discusses how we perceive load balancing as a resource allocation problem. The major components of a dynamic load balancing algorithm are then described. Classification schemes of dynamic load

balancing algorithms are also discussed. In Chapter 3, we present the system model, which serves as a common framework assumed by all the load balancing algorithms that we study. Performance and efficiency metrics for assessing load balancing algorithms are also discussed. In Chapter 4, we present a performance study on two different categories of load information dissemination strategies: one which assumes the existence of local load tables and one which does not. In Chapter 4, we also introduce the preliminary version of the batch assignment approach. In Chapter 5, the batch assignment approach is modified and combined with a new sender-receiver negotiation protocol called the *GR Protocol*. The resulting new batch assignment approach is applied to solve the problem of processor thrashing. In Chapter 6, the new batch assignment approach is applied to systems with bursty task arrival patterns. A *bursty workload pattern characterization model* is also presented in this chapter. Chapters 7 and 8 represent another line of our research. In Chapter 7, we present a preliminary study on the possibility of adopting both task assignment and task migration as the workload distribution mechanism in a load balancing algorithm. This approach is compared with the batch assignment approach in Chapter 8. In Chapter 9, we will show how we can modify the batch assignment algorithm for application to heterogeneous systems. Chapter 10 is the conclusion.

The organization of this thesis is shown in Figure 1.1. The upper grey bar signifies that all the algorithms we study in chapters 5-9 are adaptive, symmetrically-initiated, and are based on polling. The lower grey bar signifies that chapters 5, 6, 8 and 9 are based on the batch assignment approach and the GR protocol.

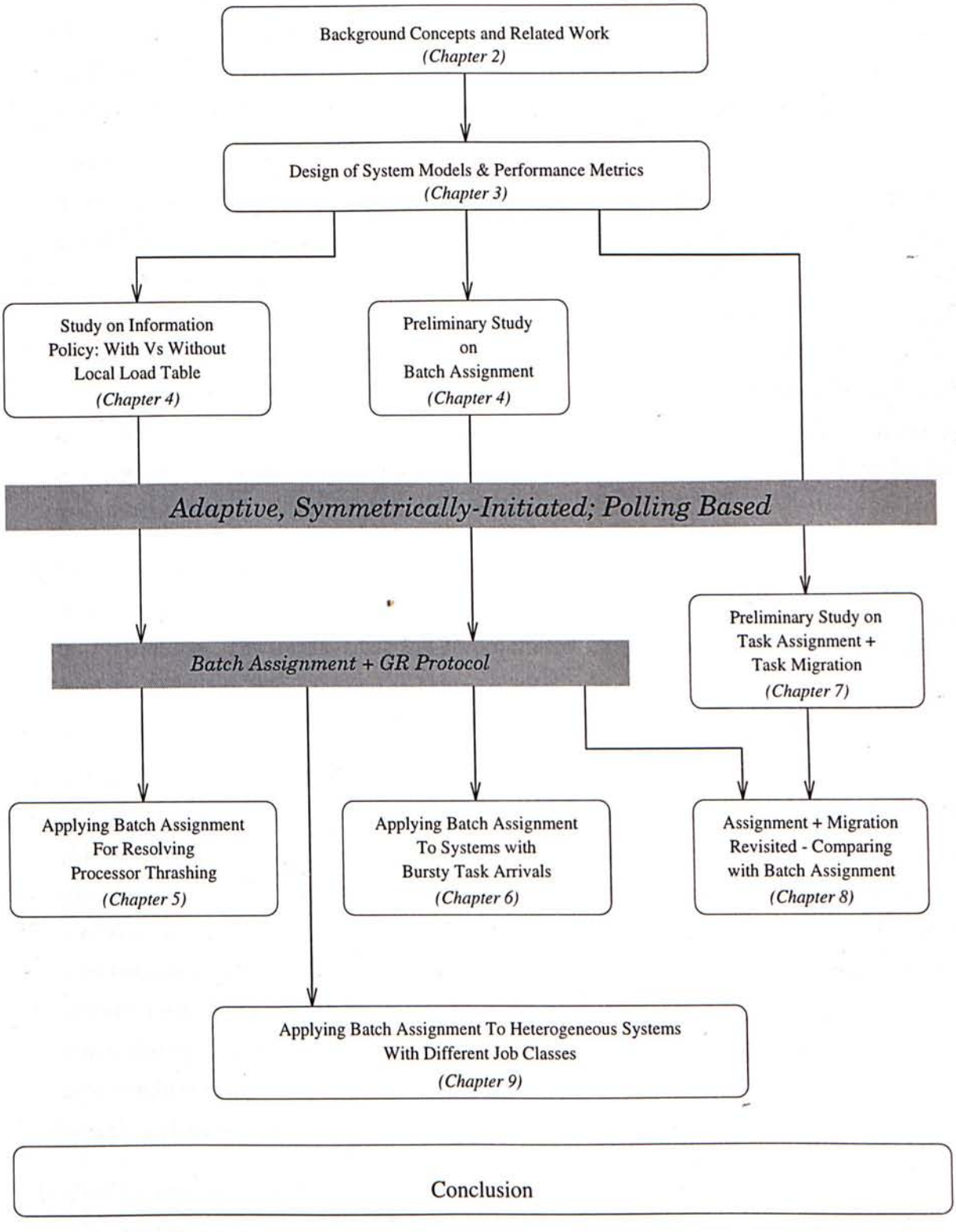


Figure 1.1: Logical flow of the research work and the outline of the thesis.

Publications

1. Chin Lu and Sau-Ming Lau.
   "A Performance Study on Load Balancing Algorithms with Process Migration".
   In *Proceedings, IEEE TENCON 1994*, pages 357–364, Singapore, August 1994.

   Abstract –
   In this paper, we present a performance study on three different load balancing algorithms. The first algorithm employs only task assignment, whereas the other two allow both task assignment and migration. We conclude that although task migration usually costs more than task assignment, under some situations it can augment task assignment to provide extra performance improvement. This is because task migration provides an alternate mechanism for distributing workload in a distributed system. The performance improvement by using this approach is especially significant when a heavily-loaded node has no appropriate tasks for assignment.

2. Chin Lu and Sau-Ming Lau.
   "An Adaptive Load Distribution Algorithm for Systems with Bursty Task Arrivals".
   In *Proceedings, Thirteenth IASTED International Conference for Applied Informatics*, Austria, February 1995.

   Abstract –
   Usually heuristic-based load balancing algorithms cannot provide satisfactory performance with bursty task arrivals because they assume stable arrival patterns. In this paper, we present an adaptive load balancing algorithm, which employs the new *Batch Transfer Approach*. This approach allows a number of tasks to be transferred as a single batch, coupled with a protocol to obtain mutually agreed batch size between a sender and a receiver. Simulations show that: (1) In terms of the system mean task response time, our algorithm provides significant improvement when the system is not saturated. (2) Our algorithm can always improve the system predictability. (3) Our algorithm ensures a stable range of both mean queue length and mean task response time.

3. Chin Lu and Sau-Ming Lau.
   "An Adaptive Algorithm for Resolving Processor Thrashing in Load Distribution".
   Concurrency: Practice and Experience, 7(7), October 1995. Special issue on dynamic resource management in distributed systems; Accepted for publication.

Abstract –

Processor thrashing in load distribution refers to the situation when a number of nodes try to negotiate with the same target node simultaneously. The performance of dynamic load balancing algorithms can be degraded because processor thrashing can lead to receiver node overdrafting, thus causing congestions at a receiver node and reduction of workload distribution. In this paper, we present an adaptive algorithm for resolving processor thrashing in load distribution. The algorithm is based on the integration of three components: (1) a batch task assignment policy, which allows a number of tasks to be transferred as a single batch from a sender to a receiver, (2) a negotiation protocol to obtain a mutual agreement between a sender and a receiver on the batch size, and (3) an adaptive symmetrically-initiated location policy to select a potential transfer partner. Simulations reveal that our algorithm provides a significant performance improvement at high system load because the algorithm can avoid processor thrashing so that CPU capacity is more fully utilized.

Chapter 2

Basic Concepts and Related Work

Resource allocation is one of the major issues of resource management in operating systems. In a DCS, this complex problem is further complicated by the physical distribution of resources, communication delays, redundancy of resources, possibility of partial failures of resources, and by the lack of accurate global state information [Gos91, pp. 439]. Goscinski defined a *resource* as a reusable, relatively stable hardware or software component of a computer system that is useful to system users or their processes. He distinguished two types of resources [Gos91, pp. 440]:

- *Physical resources* — which are the permanent physical components of a computer system, such as processor, main memory, I/O devices, and external memory.
- *Logical resources* — which are collections of information stored within physical resources, such as processes, files, shared programs and data.

A load balancing algorithm can be regarded as a special kind of distributed process scheduling algorithms with the objective to balance the workload between processors. Since processor is an especially important type of physical resource, a load balancing algorithm can therefore be formulated as a *resource management function* [CK88] [Gos91]. On the other hand, a load balancing algorithm itself can be regarded as a logical resource on its own. Thus it can also be referred to as a *resource management resource* [CK88]. The functionality of this management resource can be described in terms of three components: (1) *Consumer(s)*; (2) *Resource(s)*; and (3) *Policy/Policies.* With a dynamic load balancing

algorithm, consumers represent the processes to be scheduled; resources represent the available processors in the DCS; and policies represent the decision strategies used in the dynamic load balancing algorithm. One can observe the behavior of a dynamic load balancing algorithm in terms of how its policies affect the resources and consumers. This relationship between dynamic load balancing algorithms, consumers, and resources is shown in Figure 2.1.

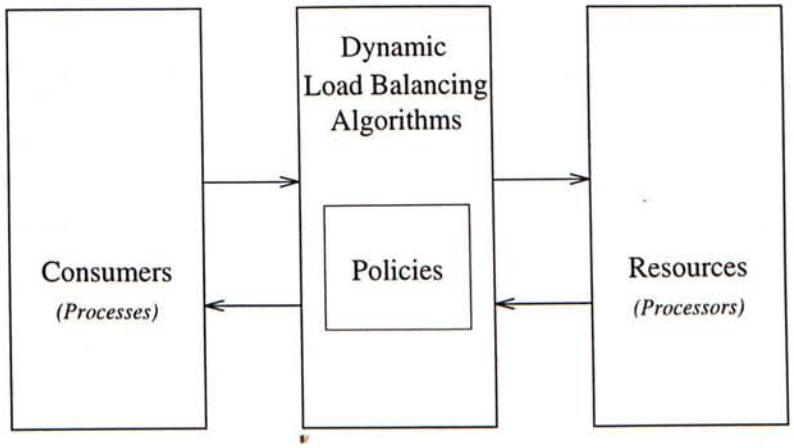


Figure 2.1: A dynamic load balancing algorithm can be regarded as a resource management function, which deals with processes and processors. Adapted from [CK88].

2.1 Components of Dynamic Load Balancing Algorithms

A dynamic load balancing algorithm is composed of three parts:

- **Transfer Policy** —

  A transfer policy has two components: (1) *Algorithm Initiation Scheme*, which determines whether a node should initiate a sender-receiver negotiation session for task distribution; and (2) *Task Selection Scheme*, which selects the task(s) to be transferred from among a set of candidate tasks.

- **Location Policy** —

  which attempts to find an appropriate processing node as a partner for task transfer. This is also called *host selection*. There are three basic types of location policies:

(1) *Sender-initiated*, in which congested nodes search for lightly loaded nodes to which tasks can be transferred; (2) *Receiver-initiated*, in which lightly loaded nodes search for congested nodes from which tasks can be transferred; and (3) *Symmetrically-initiated*, in which both senders and receivers can initiate the search for transfer partner.

- **Information Policy** —
  which determines (1) what information about the load states in the distributed system is needed; and (2) how such information is to be collected or distributed. Information policy is also called *load information dissemination strategy*.

It must be noted that these three components of a load balancing algorithm are not independent of each other. Rather, they interact with each other in a tightly coupled manner [Gos91].

2.2 Classification of Load Balancing Algorithms

The first taxonomy of load balancing algorithms was proposed by Casey [Cas81]. This hierarchical taxonomy reflects research results up to 1980. Since then a large number of additional distinguishing features have been identified. These features allow further differentiation between approaches to workload distribution. One such feature is associated with the type of node that takes the initiative in starting workload distribution. Based on this, Wang and Morris [WM85] presented a dichotomy taxonomy of load balancing algorithms. They classified load balancing algorithms into *source-initiative* and *server-initiative*. [1] Wang and Morris also characterized load balancing algorithms according to the degree of information dependency involved.

Another taxonomy was proposed by Casavant and Kuhl [CK88]. Their taxonomy not only agrees with Casey's classification, but also provides a more detailed and complete look at *distributed scheduling*, in which load balancing is a special case which strives to balance

[1] Source-initiative and server-initiated algorithms are also known as *sender-initiative* and *receiver-initiative* algorithms respectively in some literatures [EL86b] [EL86a].

the workload among the processing nodes in a distributed system. In this section, we look into some details of Casavant and Kuhl's taxonomy. Some useful taxonomies proposed by other researchers are also presented where appropriate.

2.2.1 Casavant and Kuhl's Taxonomy

Casavant and Kuhl's taxonomy is a hybrid of:

- *Hierarchical classification scheme* —
  used as far as possible to reduce the total number of classes; and
- *Flat classification scheme* —
  used when the descriptors of the system may be chosen in any arbitrary order.

2.2.1.1 Hierarchical Classification Scheme

The structure of the hierarchical portion of Casavant and Kuhl's classification is shown in Figure 2.2.

- **Local** *versus* **Global** —
  Local scheduling determines the allocation of a processor among its local processes. It has been described in many literatures [Mil87]. Global scheduling is responsible for the allocation of processes to processors in the distributed system. Therefore, dynamic load balancing algorithms are dealing with global scheduling.
- **Static** *versus* **Dynamic** —
  This choice indicates the time at which scheduling decisions are made. With static load balancing algorithms, there is *a priori* assignment of processes to processors. The current state of the distributed system is not taken into consideration. The principal advantage of static load balancing is its simplicity because system state information need not be maintained. However, it fails to adjust to fluctuations in the system workload pattern. Therefore, static load balancing algorithms are not suitable for systems subjected with bursty task arrival patterns. The design of static

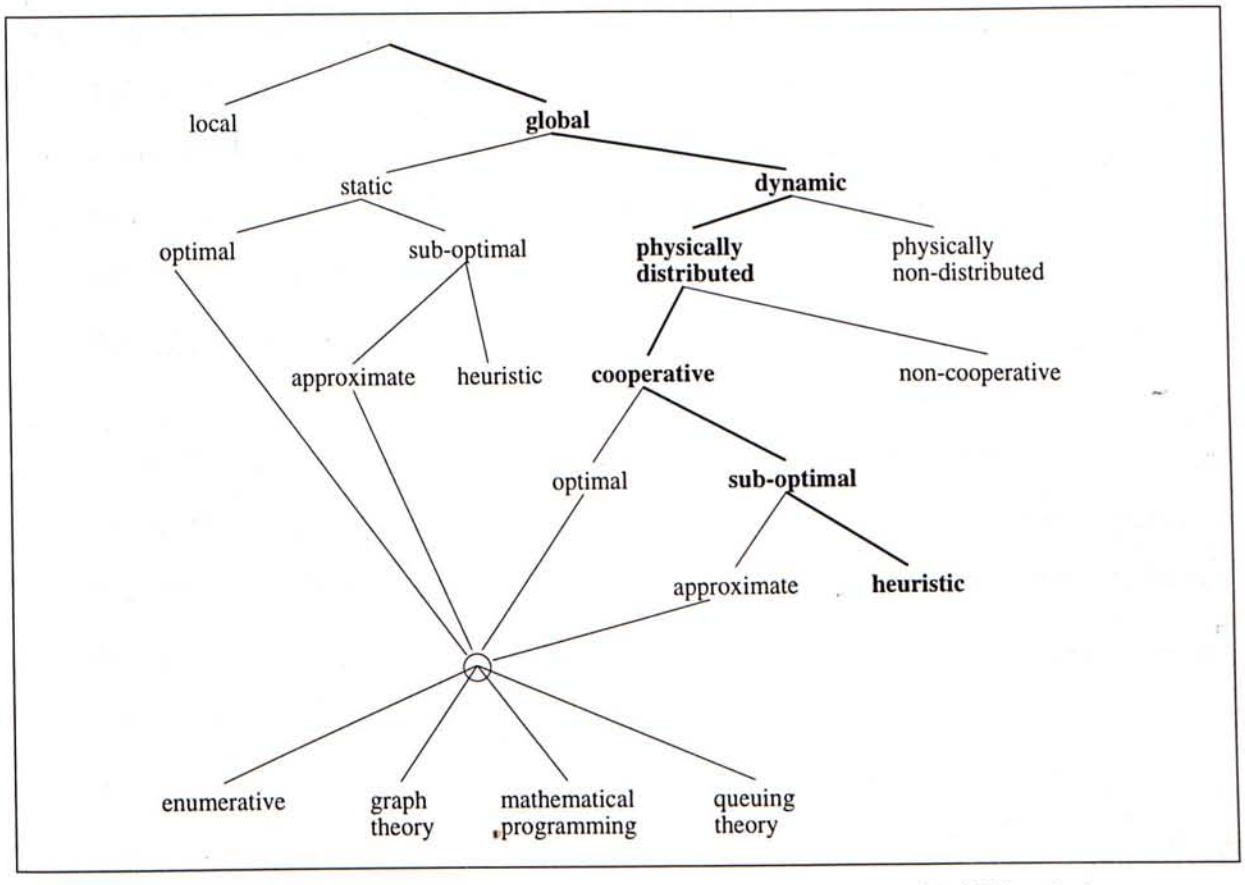


Figure 2.2: Hierarchical portion of Casavant and Kuhl's Taxonomy. Bold-faced classes represent our research interests. Adapted from [CK88].

load balancing algorithms is pioneered by Shahid H. Bokhari [Bok79]. More recent works include [Bok87], [ST85] and [Lo88].

With dynamic load balancing, the more realistic assumption is made that very little *a priori* knowledge is available about the resource needs of a process. Scheduling decisions are made dynamically using the information of current system state. Making a dynamic load balancing decision is much more complicated than finding a static one because gathering and maintaining system state information are needed. However, dynamic load balancing can potentially achieve better performance than static load balancing.

In this research, we focus only on *dynamic load balancing*. The static subtree in the taxonomy is therefore ignored.

- **Physically Distributed** *versus* **Physically Non-Distributed** —

  This issue involves whether the authority and responsibility for making global dynamic load balancing decisions physically reside in a single processor (*physically non-distributed*) or whether the work involved in making the decisions *physically distributed* among the processors.

  The most important feature of the non-distributed approach is its simplicity. However, it suffers from three drawbacks: (1) The functional capacity of any centralized scheduling server is bounded; (2) A centralized server implies a bottleneck to which and from which messages are sent and thus the system state maintained in the server may not be up-to-date; and (3) A centralized server implies a single point of failure.

  In this research, we focus on *global dynamic physically distributed load balancing algorithms*.

  The immediate design issue generated by the distributed nature of this category of algorithms is the relationship between the distributed decision components. This is discussed in the following point.

- **Cooperative** *versus* **Non-Cooperative** —

  The question here is the degree of *autonomy* which each processor has in determining its load balancing decisions. If processors make decisions independently of each other, the load balancing algorithm is said to be *non-cooperative*. In this case, because the processors act as autonomous entities, they are oriented only towards individual goals. As a result, the load balancing decisions may contradict each other and generate performance conflicts.

  If the load balancing algorithm involves cooperation between independent processors, the load balancing algorithm is said to be *cooperative*. The majority of global dynamic physically distributed load balancing algorithms are cooperative in nature.

  In this research, we are primarily focusing on *global dynamic physically distributed cooperative algorithms*.

- **Optimal** *versus* **Sub-Optimal** —

  In the case that all information regarding the state of the system as well as the resource needs of a process are known, an optimal load balancing decision can be made by applying some *criterion functions*. Examples of optimization measures for criterion functions are minimization of total process completion time, and maximization of system throughput. There are four commonly used methods for finding an optimal scheduling decisions: (1) enumerative, (2) graph theory, (3) mathematical programming, and (4) queuing theory.

  If only partial information is available or the load balancing problem is computationally infeasible, suboptimal solutions may be sought for. Obviously, global dynamic optimal load balancing decisions are difficult to achieve because of the lack of an accurate system state picture.

  In this research, we focus on *global dynamic physically distributed cooperative suboptimal algorithms*.

- **Approximate** *versus* **Heuristic** —

  The approximate approach may use the same model for finding a load balancing decision as used in the optimal approach. However, searching for a load balancing "solution" does not cover the whole solution space. Instead, the goal is to find a satisfactory one. This helps to reduce the computation time taken.

  The most distinguishing feature of heuristic load balancing algorithms is that they make use of special parameters which affect the system in indirect ways. Such parameters has an impact on the overall service that users (customers) receive, but cannot be directly related to system performance. It is our intuition that leads us to believe that using such parameters will improve system performance.

  In this research, we focus on *global dynamic physically distributed cooperative suboptimal heuristic algorithms*.

2.2.1.2 Flat Classification Scheme

Casavant and Kuhl's flat classification is used when the descriptors of the system may be chosen in any arbitrary order. Some other useful taxonomies proposed by researchers other than Casavant and Kuhl are also discussed in this section.

- **Adaptive** *Versus* **Non-adaptive** —

  An adaptive load balancing algorithm is one in which the policies and parameters of the algorithm change dynamically according to the previous and current behavior of the system. In other words, previous decisions and their effects on system performance are taken into consideration by the load balancing algorithm. An example adaptive load balancing algorithm is described by Stankovic and Sidhu in [SS84]. This adaptive algorithm evaluates multiple parameters by using a McCulloch-Pitts neuron.

  In contrast to an adaptive load balancing algorithm, a non-adaptive load balancing algorithm is one which does not necessarily modify its basic control mechanism on the basis of the history of system activities. In other words, the policies and parameters of the algorithm are fixed, regardless of the current system load states.

- **Source-Initiative** *Versus* **Server-Initiative** —

  Wang and Morris [WM85] proposed a taxonomy of load balancing algorithms which is based on the type of a node that takes the initiative in the global search for a lightly-loaded or heavily-loaded processing nodes. If the *source* (overloaded) node is responsible for finding a remote location for process execution, the strategy is called *source-initiative* (*sender-initiated*). If a *server* (lightly-loaded) node looks for and requests processes from overloaded processing nodes, the strategy is called *server-initiative* (*receiver-initiated*). Comparative studies of these two types of algorithms had been made by Eager *et al.* [EL86b] [EL86a]. Some algorithms adopt both source-initiative and server-initiative approaches and are classified as *symmetrically-initiated*. An example of symmetrically-initiated algorithms is presented by Shivaratri and Krueger in [SK90].

- **Bidding** —

  The concept behind the bidding approach is the negotiation between processors and the submission of bids for contracts. Each processor is responsible for two roles with respect to the bidding process: *manager* and *contractor*. The manager represents the task in need of a location to execute, whereas the contractor represents a processor which is able to do work for other nodes. The bidding approach works as follows. The manager announces the existence of a task in need of execution by a *task announcement*. It then receives *bids* from the other nodes (contractors). The manager evaluates the bids and awards contracts to the most appropriate node(s) by sending tasks to it. Bidding is a sender-initiated approach.

- **Drafting** —

  The drafting approach is the converse of the bidding approach and is receiver-initiated. A sender-receiver negotiation session starts when a potential receiver announces the availability of its spare processing capacity. Potential sender nodes then send *request messages* to the receiver, who then evaluates the requests and make an offer to one or more of the sender nodes for receiving tasks from them.

- **Classification Based on the Level of Information Dependency** —

  Another taxonomy is based on the level of information dependency that is embodied in a load balancing algorithm. The level of information dependency refers to the degree to which a source node needs to know the states of servers; or a server needs to know the states of sources. Wang and Morris identified seven levels of information dependency in [WM85]. The information levels have been arranged so that the information of a higher level subsumes that of lower levels. One can expect that an increase in the level of information dependency allows the construction of algorithms which provide better performance. This is only true to some extent because at some stage: (1) The increased amount of information exchange implies increases in communication costs and delay time, which may make such information essentially outdated; and (2) Higher computation overhead becomes necessary in

processing such information. Therefore, two important design issues of dynamic load balancing algorithms are the appropriate level of information dependency, and the efficient use of available state information for attaining the best performance improvement.

Chapter 3

System Model and Assumptions

As mentioned in Chapter 2, many dynamic load balancing algorithms have been proposed [EL86a] [EL86b] [NXG85] [SS84] [Zho88]. These algorithms are designed for very different system models with very different assumptions. Some literatures even do not provide adequate description about their system models. It is rather difficult to have a fair comparison between the relative merits among these algorithms. Therefore, we develop a system model which serves as a common framework with which different load balancing algorithms can be compared objectively.

3.1 The System Model and Assumptions

The distributed system model described here applies to most current local area networks where every single processing node is an autonomous machine. The network is fully connected logically. In other words, a processing node is reachable from any other node in the network. We define the node at which a new application task arrives as the *arrival node* of the task, and the node on which a task executes as the *resident node* of that task. In principle, task assignment algorithms bind an arrival task to a resident node. If the resident node is the same as the arrival node, the assignment is said to be *local*. Otherwise, the assignment is said to be *remote*. Similar description can be made for task migration — *local execution* versus *remote execution*.

The analytical model of each node consists of three FIFO queues: the *task queue*, the *service queue*, and the *threshold queue*, as shown in Figure 3.1. The task queue and the

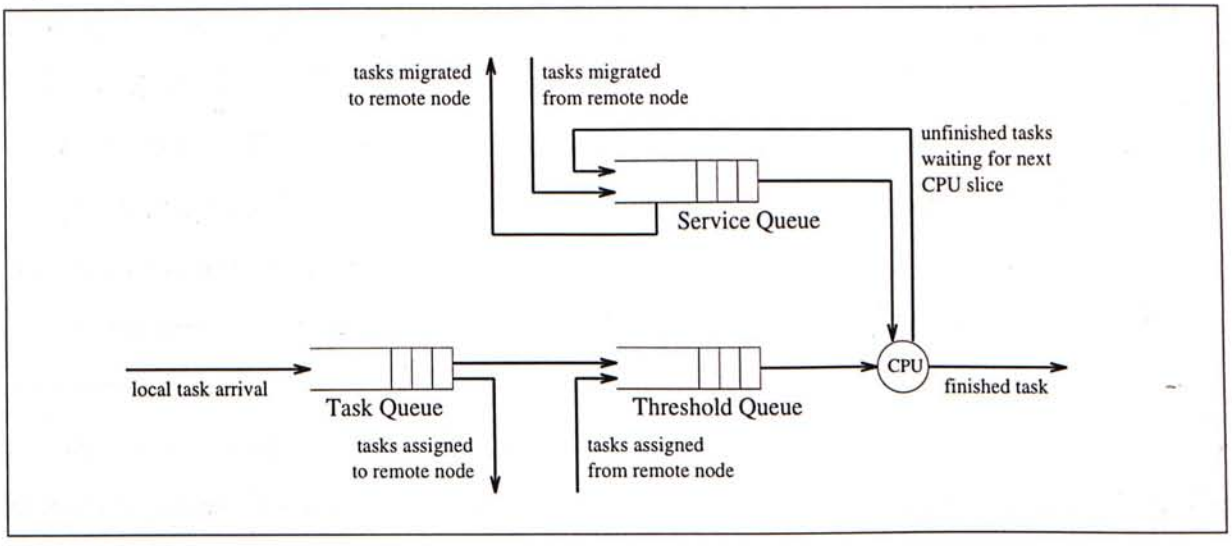


Figure 3.1: Model of a processing node

threshold queue both have infinite capacity. The service queue has a fixed capacity and can hold at most Q_o tasks at any one time.

When local tasks arrive, they go to the task queue to become candidates for task assignment. A task may be assigned either locally by entering the local threshold queue, or remotely by entering the threshold queue on a remote node. Unlike many proposed assignment algorithms which bind a locally arrived task permanently upon its arrival [EL86b] [EL86a] [SK90], we allow a task to stay in the task queue until it is either assigned remotely or assigned locally by going into the local threshold queue. In other words, a task may be considered for remote assignment multiple times. In this way, we have a greater flexibility in controlling the workload of a processing node. Once a task enters the threshold queue, it cannot be reassigned to another node. That is, we do not allow *task reassignment.* This is important in avoiding a task from continuous shifting among the nodes in the system, in which case unnecessary CPU and communication overheads are imposed and the response time of the task may become exceedingly high.

A task in the threshold queue can get the CPU only when another task has completed its execution and thereby frees an empty slot in the service queue, or when the workload of the node is low enough that spare slots in the service queue exist. Tasks in the service queue are processed by the CPU in round robin. A task can hold the CPU for at most a

fixed time slice, T_{CPU}, each time. If a task cannot finish within its time slice, it will go to the end of the service queue, where it waits for its next CPU time slice.

Tasks in the service queue are candidates for migration. A task migration algorithm selects the appropriate task from the service queue and saves its execution states, which are then transferred to a remote node. The migrated task (*migrant*) will have its execution states restored on the destination host, and then be put in the service queue to resume its execution.

The existence of the task queue, the threshold queue and the service queue effectively divides a processing node into two halves — (1) the external part represented by the task queue; and (2) the internal part represented by the threshold queue and the service queue. Tasks residing in the external part are not bound permanently, whereas tasks residing in the internal part will be executed in that particular processing node, at least for one CPU time slice. This allows clear identification of candidate tasks for assignment and migration.

We assume that task arrivals on a node P_i have independent Poisson distribution with mean arrival rate λ_i. Suppose there are N processing nodes in the distributed system, the mean arrival rates of the N nodes (i.e. $\{\lambda_1, \lambda_2, ..., \lambda_N\}$) have a log normal distribution with mean λ_o and standard deviation σ_λ. This distribution characterizes the fact that different nodes may be subjected to different workloads. The standard deviation (σ_λ) of this distribution is a quantitative measure of the load imbalance of the distributed system, and is referred to as the *imbalance factor*. Within a node P_i, the CPU time needed for completing a task (*task service time*) has an exponential distribution with a mean of S_i.

3.2 Survey on Cost Models

Many of existing literatures on study of dynamic load balancing algorithms made an overly simplifying assumptions that the execution and communication overhead of dynamic load balancing algorithms are negligible. As Kremien and Kramer pointed out in [KK92], such

an assumption does not reveal a practical situation. We believe that non-negligible execution and communication overhead should always be assumed in studying the *performance* and *efficiency* of dynamic load balancing algorithms. In this section, we describe two cost models proposed by some famous researchers in this area. We will compare these two models with the one we propose in a later section.

3.2.1 Eager, Lazowska, and Zahorjan's Model

Eager *et al.* studied the effect of information dependency on the performance of dynamic load balancing algorithms in [EL86b]. They also compared the performance of sender and receiver-initiated algorithms in [EL86a]. In their model, task arrivals are in Poisson distribution and task service demands in each processor are independently exponentially distributed. Also, mean task service demand is chosen as one time unit. This allows response times to be reported in units of task service demand. These are usual practice in performance modeling work. In Eager *et al.*'s model, the only cost being considered was the CPU overhead associated with transferring a task to its destination host. The CPU cost associated with polling was neglected. Network communication costs for both task transfer and polling were simply neglected.

- **Task Arrivals**

  Task arrival rate of each processing node is in an independent Poisson Distribution with mean λ.

- **Task Service Demand**

  Task service demand of each processing node is in an independent exponential distribution with the mean equals to one second.

- **CPU Overhead**

  - Only CPU overhead for transferring a task is considered and is taken to be 0.1 second in most cases.
  - Task transfer CPU costs in the range 0.01 to 0.10 are also studied.

- As Eager *et al.* stated in [EL86b]:

  "... the average cost of task transfer C, although non-negligible, can be expected to be quite low relative to the average cost of task processing S; the range of 1-10 percent seems to include the cases of greatest interest."

- **Communication Delay**

  Network communication delay for both task transfer and polling are simply neglected.

3.2.2 Shivaratri, Krueger, and Singhal's Model

Shivaratri *et al.* discussed load distribution in detail in [SK90] and [SKS92]. A rather detailed simulation model has been proposed. The most notable characteristics of their cost model is that CPU and communication overhead for processing and transferring polling messages are assumed to be non-negligible. Similar assumption is made for task transfer. Their simulation model and values of their simulation parameters are summarized below.

- **Task Arrivals**

  Task arrival rate of each processing node is in an independent Poisson Distribution with mean λ.

- **Task Service Demand**

  Task service demand of each processing node is in an independent exponential distribution with the mean equals to one second.

- **CPU Overhead**

  - CPU overhead for send/receive a polling message = 0.003 second
  - CPU overhead for task assignment (single task) = 0.02 second
  - CPU overhead for task migration (single task) = 0.1 second

- **Communication Delay**

  - Network Bandwidth = 10 MBits per second

- Number of Nodes = 40
- Assuming 8 kbytes has to be transferred for each task assignment (single task),

$$\begin{aligned}\text{Communication Delay for Task Assignment} &= \frac{8 * 1024 * 8 \text{ bits}}{10 * 10^6 \text{ bits per second}} \\ &= 0.007 \text{ second}\end{aligned}$$

- Assuming 200 kbytes has to be transferred for each task migration (single task),

$$\begin{aligned}\text{Communication Delay for Task Migration} &= \frac{200 * 1024 * 8 \text{ bits}}{10 * 10^6 \text{ bits per second}} \\ &= 0.16 \text{ second}\end{aligned}$$

3.3 Our Cost Model

As Kremien and Kramer noted in [KK92], modeling CPU overhead and network delay is essential for the correct assessment of both performance and efficiency of dynamic load balancing algorithms. To design a cost model at the *appropriate level of complexity* is of utmost importance.

3.3.1 Design Philosophy

Eager *et al.*'s work described in section 3.2.1 represents a major step in designing cost model for dynamic load balancing algorithms. They emphasized the importance of task assignment CPU overhead, which is expressed as a fraction of task service demand. However, they failed to account for the CPU overhead involved in processing polling activities. As we shall see in chapter 4, pure sender and receiver-initiated load balancing algorithms [1] have substantial effect on the level of CPU overhead, and thus affect task response time significantly. In our cost model, CPU overhead of polling activities and load state enquiry activities are both considered and expressed in terms of percentage of unit CPU time.

Another drawback of Eager *et al.*'s model is that network communication delay is completely neglected. Eager *et al.* illustrated that under reasonable assumptions, network

[1] Or the sender and receiver-initiated components of a hybrid non-adaptive algorithm.

utilization due to dynamic load balancing activities lies in the order of 3 percent for a system with 100 nodes connected with a 10 Mbit network. They concluded that such network bandwidth requirement is insignificant and can therefore be neglected. However, although the network utilization can be neglected, the delay experienced by the messages transferred via the network cannot. A car running in an almost empty freeway does not mean that it can arrive its destination instantly. In fact, communication delay is one of the major characteristics of distributed systems, and is the dominant factor in the design of information policy of dynamic load balancing algorithms. As Philp commented in [Phi90], communication delay has a substantial effect on the accuracy of load state information, which in turn affects the scheduling decisions of load balancing algorithms directly. In our cost model, communication delay for all messages exchanged are taken into account. In modeling the communication delay, current network utilization is not considered however. This is because Eager *et al.* already proved that network utilization due to dynamic load balancing activities can be neglected. Communication delay therefore depends only on the nature of the message to be transmitted.

3.3.2 Polling Query Cost Model

In attempting to locate a transfer partner, a query message may need to be sent to the potential transfer partner to obtain its consent, or to enquire its load state. This operation is called *polling*. We assume that the CPU overhead in processing a polling message is non-negligible and that both the sender and the receiver of the polling message have the same CPU overhead. This CPU overhead is denoted by $CPU_{polling}$.

> **Definition 1** The CPU overhead associated with sending or receiving a polling message is non-negligible and is represented by the parameter $CPU_{polling}$.

Furthermore, the communication delay for transmitting the polling message is assumed to be non-negligible. Since a polling message is inherently short, we assume that only one message injection cost $F_{polling}$ is needed. The communication delay for transmitting a polling message is therefore defined as:

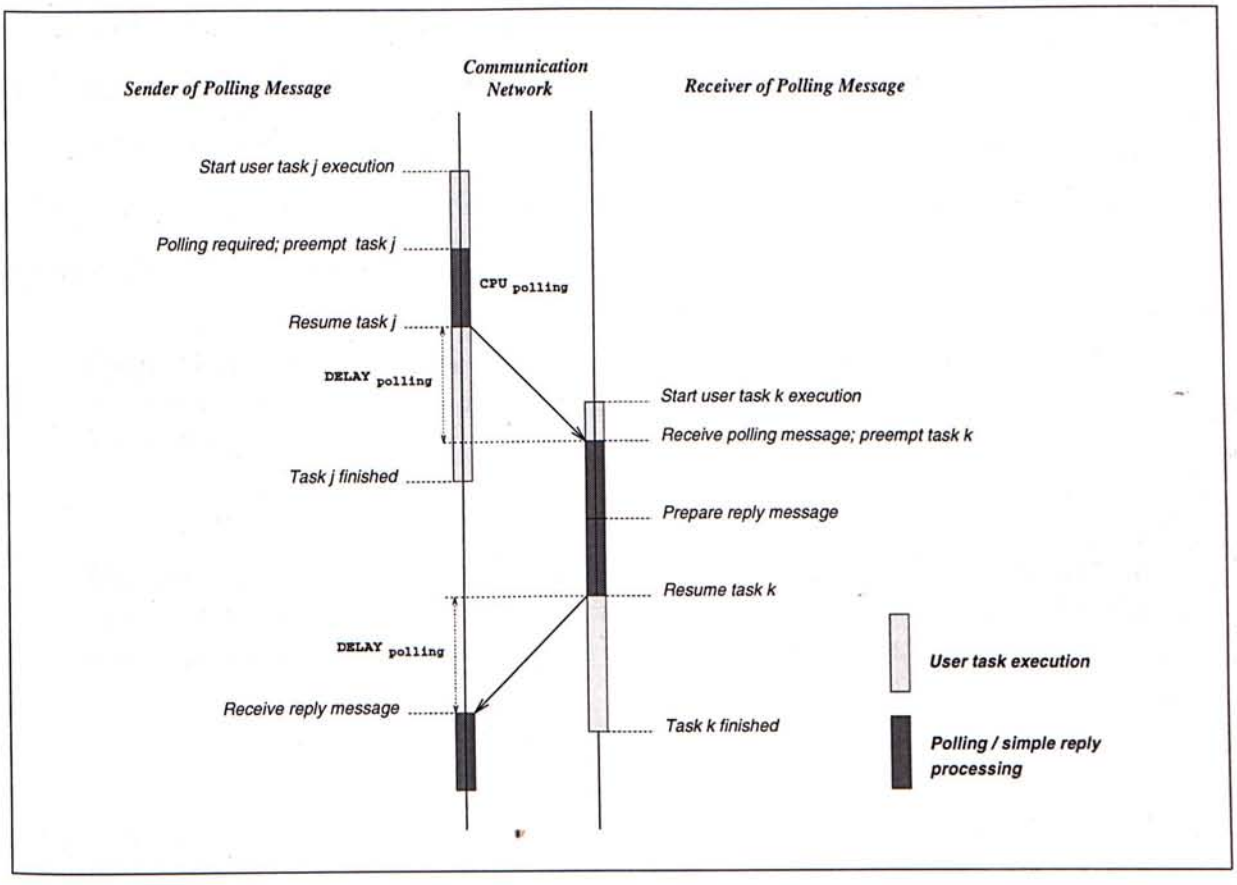


Figure 3.2: Time Sequence Diagram — A scenario showing that a CPU is released after a polling message has been injected into the communication network.

Definition 2 The communication delay experienced by a polling message consists of a single message injection cost $F_{polling}$, plus the propagation delay D, and is represented by $DELAY_{polling}$:

$$DELAY_{polling} = F_{polling} + D$$

Note that once a polling message has been processed by the CPU (with overhead $CPU_{polling}$), the CPU is released from the polling message. The communication delay of the polling message then becomes independent of its sender node. This is illustrated by the time-sequence diagram in Figure 3.2.

3.3.3 Load State Broadcasting Cost Model

Some of the algorithms we study maintain a load table locally in each node. To update a load table, load information is exchanged by broadcasting of load information messages.

Since such a message is also very small as in the case of polling messages, we assume that load state broadcasting shares similar cost model. Note that each of the receivers of a load state broadcasting message has to spend certain CPU overhead ($CPU_{polling}$) in processing the message. Network bandwidth consumption, however, remains the same because there is only one message.

Definition 3 The CPU overhead associated with sending or receiving a load state broadcasting message is non-negligible and is represented by the parameter $CPU_{polling}$.

Definition 4 The communication delay experienced by a load state broadcasting message consists of a single message injection cost $F_{polling}$, plus the propagation delay D, and is denoted by $DELAY_{broadcasting}$:

$$DELAY_{broadcasting} = F_{polling} + D$$

3.3.4 Task Assignment Cost Model

Task assignment involves the transfer of a "fresh" task from one host to another. The CPU overhead involved is significantly higher than that of polling activities. We assume that both the sender and the receiver of the transferred task(s) share the same amount of CPU cost. Besides, some of the algorithms that we studied employ a new approach of task transfer called *batch assignment*. Batch assignment allows a number of tasks to be transferred as a single batch from a sender to a receiver. [2]

Definition 5 The *batch size* of a task batch is the number of tasks contained in the task batch.

Batch size has a crucial effect on the CPU cost in processing (composing and decomposing) a task batch. Obviously, the larger the batch size, the higher the CPU cost. The task transfer CPU cost is defined as follows.

[2]Most load balancing literatures focus on sender-receiver negotiation protocols and the measurement of workload. It is assumed that only a single task can be transferred for each sender-receiver negotiation session. Batch assignment therefore is a new concept in the study of load balancing algorithms. Detail of batch assignment will be presented in later chapters.

Definition 6 The CPU cost associated with task assignment is non-negligible and is represented by $CPU_{assignment}$, which is defined as follows:

$$CPU_{assignment} = C_{assign} + C_{pack} * \sum_{i=1}^{B_a} l_i$$

where C_{assign} and C_{pack} are constants representing the CPU time for running the assignment algorithm, and the CPU time for composing/decomposing each task message packet, respectively. The value B_a is the number of tasks contained in the task batch. The value l_i represents the number of message packets generated for a task i. This is referred to as the *task code length* of i. Within a node, l_i has an independent exponential distribution with mean l_{assign}. The term $\sum_{i=1}^{B_a} l_i$ therefore represents the total number of message packets generated for the task batch.

The communication delay for transmitting a task batch is also assumed to be non-negligible and is also significantly higher than that of polling activities.

Definition 7 The communication delay experienced by an assignment task batch relates to its batch size and is defined as follows:

$$DELAY_{assignment} = (F_{task} * \sum_{i=1}^{B_a} l_i) + D$$

where F_{task} is the time needed for injecting a single task transfer message packet into the communication channel.

3.3.5 Task Migration Cost Model

Task migration involves the transfer of an executing task from one host to another. The CPU overhead involved is higher than that of task assignment because of the high cost of task image saving and restoration, etc. As in the case of task assignment and polling, we assume that the sender and the receiver of the transferred task(s) have the same amount of CPU cost, which is defined as follows.

Definition 8 The CPU cost associated with task migration is non-negligible and is represented by $CPU_{migration}$, which is defined as follows:

$$CPU_{migration} = C_{migrate} + C_{pack} * \sum_{i=1}^{B_m} l_i'$$

where $C_{migrate}$ represents the CPU cost for running migration algorithms, including migrant selection, task image saving/restoration, etc. Within each node, $C_{migrate}$ has an independent exponential distribution with mean $C_{migrate_o}$. This distribution characterizes the fact that some migrations impose more CPU overhead because of more opened files, more established communication channels, and more allocated memory, etc. The value B_m is the number of migrants in the task batch. The value l_i' represents the number of message packets generated for a migrant i. This is referred to as *task state length* of i. Within a node, l_i' has an independent exponential distribution with mean $l_{migrate}$. The term $\sum_{i=1}^{B_m} l_i'$ therefore represents the total number of message packets generated for the migrants.

The communication delay for transmitting a migration task batch is also assumed to be non-negligible.

Definition 9 The communication delay experienced by a migration task batch relates to its batch size and is defined as follows:

$$DELAY_{migration} = (F_{task} * \sum_{i=1}^{B_m} l_i') + D$$

where F_{task} is the time needed for injecting a single task transfer message packet into the communication channel.

3.3.6 Execution Priority

We assume that executions related to load balancing (polling, assignment, migration, etc.) have a higher priority than normal user tasks. An user task in execution will be suspended when a load balancing algorithm has to be invoked. Executions related to load balancing are themselves scheduled in FIFO discipline. A user task will not be resumed until all pending executions related to load balancing have been finished, including those generated in the course of suspension of the preempted user task. Figure 3.3 depicts a scenario showing the execution priority in a node.

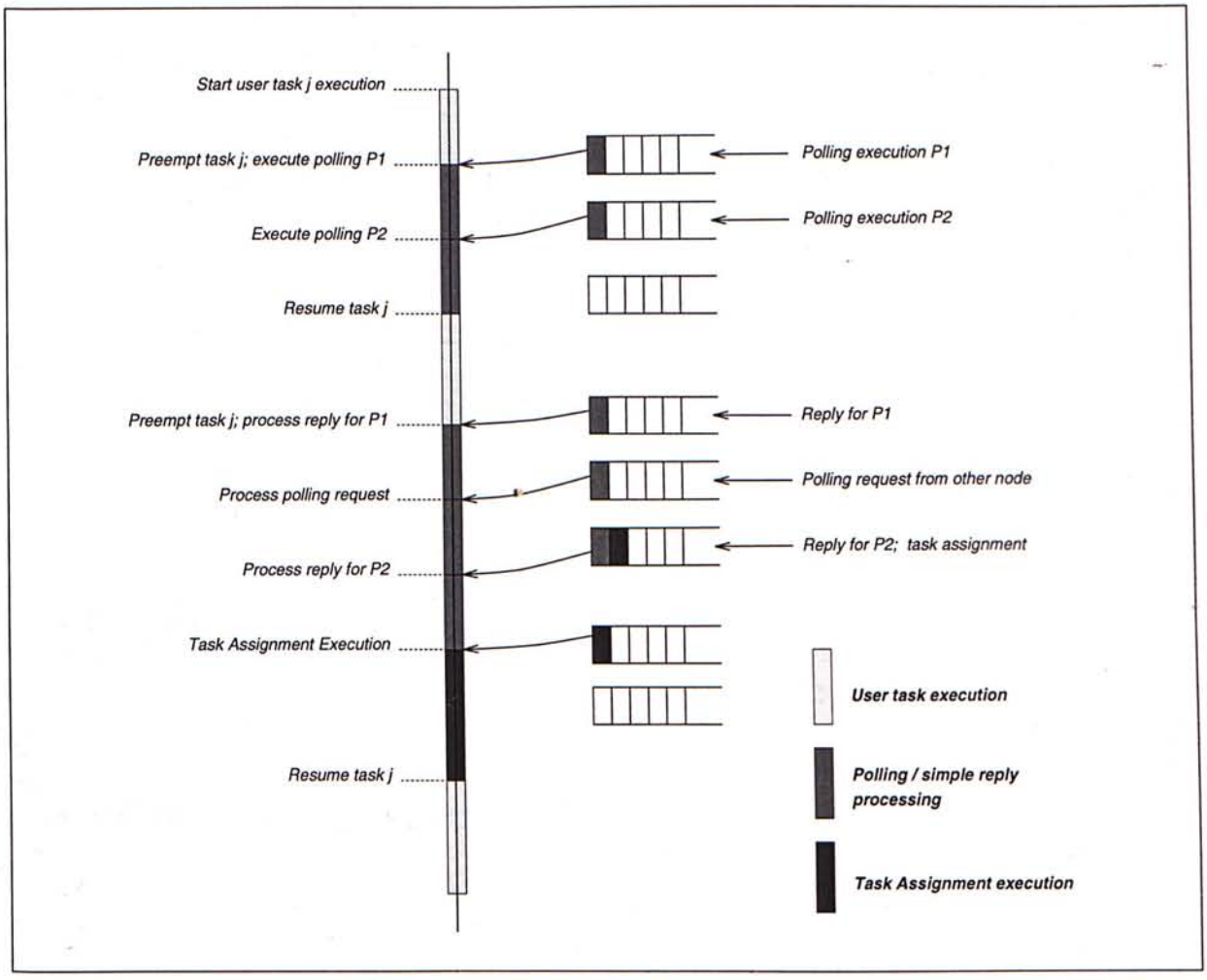


Figure 3.3: Time Sequence Diagram — A scenario illustrating the execution priority within a processing node.

3.3.7 Simulation Parameter Values

Table 3.1 shows the typical parameter values we used for our simulation study. Based on those parameter values, the values of various CPU and communication costs of our cost model are derived below. A comparison between our cost model and those described above in section 3.2 are then given in Table 3.2.

Table 3.1: Typical parameter values used for our simulation study.

Parameter	Value	Parameter	Value
Q_o	30	$CPU_{polling}$	0.005
T_{CPU}	0.2	$F_{polling}$	0.001
N	30	F_{task}	0.005
σ_λ	1	D	0.01
S_o	1	C_{pack}	0.003
		C_{assign}	0.002
		$C_{migrate}$	0.05
		l_{assign}	5
		$l_{migrate}$	7

- Polling CPU overhead (constant):

$$CPU_{polling} = 0.005$$

- Polling communication delay:

$$\begin{aligned} DELAY_{polling} &= F_{polling} + D \\ &= 0.001 + 0.01 \\ &= 0.011 \end{aligned}$$

- Load state broadcasting communication delay:

$$\begin{aligned} DELAY_{broadcasting} &= F_{polling} + D \\ &= 0.001 + 0.01 \\ &= 0.011 \end{aligned}$$

- Task assignment CPU overhead (single task):

$$\begin{aligned} CPU_{assignment} &= C_{assign} + C_{pack} * \sum_{i=1}^{B_a} l_i \\ &= 0.002 + 0.003 * 5 \\ &= 0.017 \end{aligned}$$

- Task assignment communication delay (single task):

$$\begin{aligned} DELAY_{assignment} &= (F_{task} * \sum_{i=1}^{B_a} l_i) + D \\ &= 0.005 * 5 + 0.01 \\ &= 0.035 \end{aligned}$$

- Task migration CPU overhead (single task):

$$\begin{aligned} CPU_{migration} &= C_{migrate} + C_{pack} * \sum_{i=1}^{B_m} l'_i \\ &= 0.05 + 0.003 * 7 \\ &= 0.071 \end{aligned}$$

- Task migration communication delay (single task):

$$\begin{aligned} DELAY_{migration} &= (F_{task} * \sum_{i=1}^{B_m} l'_i) + D \\ &= 0.005 * 7 + 0.01 \\ &= 0.045 \end{aligned}$$

Table 3.2: Comparisons between our cost models and those described in section 3.2.

Parameter	Eager *et al.*	Shivaratri *et al.*	Our Model	
	Value/Remark	Value/Remark	Symbol	Value/Remark
Number of Nodes	20	40	N	30
Task Arrival Rate	Independent Poisson Distribution	Independent Poisson Distribution	mean = λ	Independent Poisson Distribution
Task Service Size	mean = 1 time unit; Independent Exponential Distribution	mean = 1 time unit; Independent Exponential Distribution	mean = S	$S = 1$ time unit; Independent Exponential Distribution
Network Bandwidth	Local area broadcast channel, e.g. Ethernet	10 MBits per second local area network	$\sim$	Local area broadcast channel, e.g. Ethernet
CPU overhead for send/receive a polling message	*neglected*	0.003 time unit	$CPU_{polling}$	0.005 time unit
CPU overhead for task assignment (single task)	0.01 – 0.10	0.02 time unit	$CPU_{assignment}$	0.017 time unit
CPU overhead for task migration (single task)	*only task assignment had been considered*	0.1 time unit	$CPU_{migrate}$	0.071 time unit
Communication delay for polling message	*neglected*	*neglected*	$DELAY_{polling}$	0.011
Communication delay for task assignment (single task)	*neglected*	0.007 time unit	$DELAY_{assignment}$	0.035
Communication delay for task migration (single task)	*neglected*	0.16 time unit	$DELAY_{migrate}$	0.045 time unit

3.4 Performance Metrics

Performance is a quantitative measure of the absolute behavior of an algorithm, usually in terms of mean task response time and total system throughput. *Efficiency* is a relative term to describe the cost and overhead paid to attain a certain level of performance. In analyzing a load balancing algorithm, both performance and efficiency should be studied. This is because an algorithm which provides good performance in the expense of intolerable amount of overhead does not have much practical value. In this section, we describe the performance metrics used in analyzing the behavior of load balancing algorithms.

Some of these metrics are detailed by Kremien and Kramer in [KK92].

1. **Mean Task Response Time** —

   Minimization of mean task response time is the primary performance objective of our load balancing algorithms. In some literatures, an ideal system with zero algorithm execution overhead and zero communication delay is also compared and being regarded as a lower bound. However, this is not a fair comparison since the ideal system can never be achieved. Using such an ideal system as the lower bound causes mis-interpretation about the available room for further improvement. Comparison of performance should always be based on the same system model and assumptions.

2. **Standard Deviation of Task Response Times** —

   This performance metric measures the *fairness of service* because it indicates how much each individual task can expect its response time can differ from a mean value of the system, regardless of its arrival node. In other words, it measures the *predictability* of the system. A successful load balancing algorithm should increase both system throughput and system predictability.

3. **Performance Ratio** —

$$\text{Performance ratio} = \frac{metric(REF) - metric(LB)}{metric(REF)} \tag{3.1}$$

   where $metric()$ is the performance or efficiency metric under study; REF is the reference algorithm; and LB is the load balancing algorithm being studied. The subtraction should be in the order that a positive performance ratio indicates an improvement in system performance. The closer the performance ratio to 1, the better the performance of the LB algorithm with reference to REF. A negative performance ratio indicates a degradation in system performance.

4. **Percentage CPU Overhead** —

   This efficiency metric measures the percentage of total CPU time spent on running

a load balancing algorithm. It measures the level of CPU overhead injected by an load balancing algorithm to attain the corresponding performance improvement.

5. **Net CPU Utilization** —

   This performance metric equals to the observed CPU utilization minus the percentage CPU overhead imposed by running load balancing algorithms. It measures the effectiveness of a load balancing algorithm in making maximal CPU utilization for processing user application tasks.

6. **Remote Execution Percentage** —

   This efficiency metric measures the percentage of tasks executed remotely, either through assignment or migration. Zhou *et al.* found that only a small percentage of remote execution is needed for achieving a significant performance gain [ZF87]. Percentage remote execution has two components: *percentage assignment* and *percentage migration*.

7. **Channel Utilization** —

   This efficiency metric measures the communication overhead injected by a load balancing algorithm, including polling and task transfer messages. A successful load balancing algorithm should not only maximize the total system throughput, but also struggles to reduce the communication overhead injected.

8. **Hit Ratio** —

   The ratio of successful polling decisions to the total number of polls sent out. It measures the quality of scheduling decisions made by a load balancing algorithm. The exact definitions of a *hit* and a *miss* depend on the particular negotiation protocols of an algorithm.

Chapter 4

A Performance Study on Load Information Dissemination Strategies

Recall that a load information dissemination strategy deals with the way load state information of processing nodes are distributed. There are two major categories of load information dissemination strategies:

- In the first category, each node maintains a system load table, which stores the load states of all the nodes in the distributed system. To find a potential sender/receiver, the location policy needs only to examine the local load table only.

- The second category of load information dissemination strategies assumes no load table. Load information must be exchanged on demand by polling. Eager *et al.* studied the effect of complexity of such algorithms on the system performance [EL86b]. They concluded that complex strategies is of little benefit over simple use of state information.

However, the relative merits between these two categories of load information dissemination strategies are not clear. In this chapter, we present a performance study of two sets of dynamic load balancing algorithms which differ in their load information dissemination strategies:

- **AWLT (Algorithms with load table)** —
  Algorithms in this category assume the use of a locally maintained load table which stores the load states of all the processing nodes in the distributed system.

- **AWOLT (Algorithms without load table)** —
  Algorithms in this category do not assume the existence of load tables and load information of other processing nodes must be obtained by polling.

In addition to the study of load information dissemination strategies, we will also introduce the *batch assignment*.

4.1 Algorithm Descriptions

In this section, we present the two sets of algorithms used. Within each category, different location policies are used. Besides, some algorithms adopt the batch assignment approach.

4.1.1 Transfer Policy

Recall that a transfer policy contains two parts: (1) algorithm initiation policy; and (2) task selection scheme. All the algorithms studied in this chapter share the same algorithm initiation policy described below. The task selection scheme is different — some algorithms allow only single task assignment whereas others allow batch assignment.

4.1.1.1 Algorithm Initiation Policy

Task assignments are needed whenever a node is regarded as a potential sender or as a potential receiver. This in turn depends on the load state of a node. It has been shown that precise numerical load measurements do not yield significant performance advantage when compared to simple load index [EL86b] [NXG85]. Therefore it is sufficient to use a 3-level load measurement scheme to describe the busyness of a node. The three load levels are **H-load** (high load), **N-load** (normal load), and **L-load** (light load). A node in H-load is regarded as a potential sender, whereas a node in L-load is a potential receiver. The definitions of these load states are given in Table 4.1. Intuitively, if the total number of

tasks residing in a node P_i, denoted as K_i, is not greater than a designated *lower_threshold*, P_i is regarded as in L-load. If K_i is greater than a designated *upper_threshold*, P_i is regarded as in H-load. Otherwise, the node is normally loaded. For our simulation study presented in this chapter, we define *upper_threshold* as (Q_o+2). Thus, if the service queue is full and at least one task is waiting in either the task queue or the threshold queue, the node is regarded as in H-load. Sender-initiated negotiation starts whenever a local task arrival triggers its arrival node into the H-load state. Receiver-initiated negotiation starts if a task completion makes that node in L-load state.

Let	K_i	=	total number of tasks residing in the node P_i
	Q_o	=	service queue capacity
	lower_threshold	=	simulation parameter
	upper_threshold	=	$(Q_o + 2)$

Table 4.1: The 3-level load measurement scheme used in AWLT and AWOLT algorithms.

Load State	*Criteria*
L-load	$K_i \leq lower\_threshold$
N-load	$lower\_threshold < K_i \leq upper\_threshold$
H-load	$K_i > upper\_threshold$

4.1.1.2 New Task Transfer Approach — Batch Assignment (First Version)

All the existing load balancing algorithms assume the use of single task transfer (either assignment or migration). In this section, we introduce a new concept in the study of load balancing, namely the *batch assignment*. Batch assignment allows a number of tasks to be transferred as a single batch from a sender to a receiver for each single sender-receiver negotiation session. Batch assignment therefore makes a more efficient use of a negotiation session. To transfer a certain amount of workload, batch assignment injects less CPU and communication overhead because less negotiation sessions are required.

The determination of the appropriate batch size is a critical issue in batch assignment. This is governed by three *Batch Size Determination Rules*. The aim of these rules is to

avoid excessive task transfer, which may make the receiver node more heavily loaded than the sender node after the batch assignment. These three rules are stated below.

Let max = maximum number of tasks the receiver is willing to accept

t = number of tasks the sender is willing to send to the receiver

Rule 1: After accepting max tasks, the receiver should not be in H-load, neglecting new arrivals and departures during the negotiation and task transfer operations.

Rule 2: After transferring t tasks, the sender node should not be in L-load.

Rule 3: After transferring t tasks, the expected total number of tasks in the receiver should not be greater than the total number of tasks in the sender, neglecting new arrivals and departures of the receiver.

The values of max and t are determined by two functions: $MaxAssign()$ and $NumAssign()$, respectively. Note that $MaxAssign()$ is used by a receiver, and $NumAssign()$ is used by a sender. These two functions are presented in Figure 4.1. Figure 4.2 shows the invocation of these two functions by a sender and a receiver. For sender-initiated negotiations, a sender node first selects the target potential receiver node by its location policy. A polling message will be sent to the target potential receiver. The potential receiver node, upon receiving the polling message, invokes the function $MaxAssign()$ to determine the value max, the maximum number of tasks the receiver is willing to accept. Note that the function $MaxAssign()$ depends only on the current workload of the receiver and the designated *upper_threshold*. It does not need to know the workload of the sender. The value of max will then be sent to the sender as an acknowledgement (*ack*) message. The sender node, upon receiving this *ack* message, invokes the function $NumAssign()$ to determine the value t, the number of tasks the sender is willing to send to the receiver. Note that the function $NumAssign()$ requires the function parameter max, which is used for estimating the workload of the receiver. Tasks are then selected from the task queue of the sender node to compose the *task batch*, which will then be sent to the receiver. Note that the task selection scheme may select less than t tasks for remote assignment. The

final batch size measured in terms of number of tasks is denoted by b. Receiver-initiated negotiations take a similar way.

4.1.2 Information Policy

The second component of a load balancing algorithm is the information policy, which contains two parts: (1) content of load information, and (2) the information dissemination strategy. For the algorithms we studied in this chapter, load information content is simply the load state of a node as defined in section 4.1.1.1, — either H-load, N-load or L-load. For information dissemination strategy, AWLT algorithms maintain local load tables which store the load states of all other nodes. To update the load tables, a node broadcasts its new load state whenever its load state changes. In AWOLT algorithms, no load table is assumed. Instead, algorithms must poll for load information if remote assignment is being considered.

4.1.3 Location Policy

AWLT and AWOLT each consists of five different algorithms. The algorithms are carefully designed to cover both sender-initiated and receiver-initiated approaches. In order to compare the effect of the existence of a load table, the two sets of algorithms are symmetrical in the sense that each algorithm in one category has a counterpart in another category. Because of this symmetry, we will not repeat the algorithm description for each set.

1. **Single Sender Cyclic Assignment** - $SSCA.with$, $SSCA.without$

   These are sender-initiated single task assignment algorithms. When a sender considers a remote assignment, it negotiates with a target node for getting the consent. The target node is selected as follows. For the AWLT algorithm ($SSCA.with$), nodes are selected in a cyclic manner and the load information maintained in the local load table are referenced. If a node is not in L-load, it is discarded and another node is selected. Thus the load table serves as a filter in target node selection. The

MaxAssign()

Let K_r = Number of tasks currently residing in the receiver
Let K_r' = Number of tasks residing in the receiver after the task transfer

If max tasks are relocated,

$$K_r' \approx K_r + max$$

By **Rule 1**,

$$K_r' \leq upper\_threshold$$

Thus,

$$max \leq upper\_threshold - K_r$$

Taking the largest possible value,

$$max = upper\_threshold - K_r \quad (4.1)$$

For the load state criteria defined in Table 4.1,

$$max = (Q_o + 2) - K_r$$

NumAssign()

Let K_s = Number of tasks residing in the sender when the polling/ack message is received
Let t = Number of tasks sender x is willing to send to the receiver

By **Rule 2**,

$$K_s - t > lower\_threshold$$

$$t < K_s - lower\_threshold \quad (4.2)$$

Using equation (4.1), sender node estimates K_r from max as follows.

$$K_r \approx upper\_threshold - max$$

By **Rule 3**,

$$K_r + t \leq K_s - t$$

Thus,

$$t \leq \frac{K_s + max - upper\_threshold}{2} \quad (4.3)$$

For the load state criteria defined in Table 4.1,

$$t \leq \frac{K_s + max - Q_o - 2}{2}$$

Of course,

$$t \leq max \quad (4.4)$$

t is taken to be the largest integer satisfying the inequalities (4.2) to (4.4).

Figure 4.1: Functions *MaxAssign*() and *NumAssign*() for determining the desired batch size t.

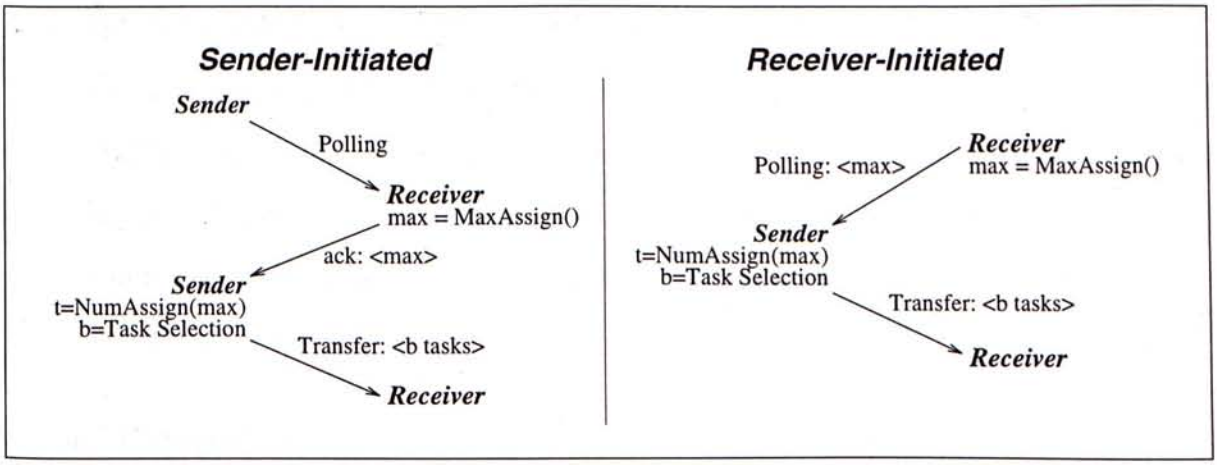


Figure 4.2: The role of $MaxAssign()$ and $NumAssign()$ in determining the batch size b.

location policy stops if either *probe-limit* trials have been made or no L-load node is found in the load table. Similarly, for the AWOLT algorithm ($SSCA.without$), nodes are polled in a cyclic manner. However, no filtering can be made and thus all nodes (except the sender itself) in the system are potential target nodes.

2. **Single Receiver Cyclic Assignment** - $SRCA.with$, $SRCA.without$

   The single receiver cyclic assignment algorithms are similar to the single sender cyclic assignment algorithms ($SSCA$), except that they are receiver-initiated.

3. **Batch Receiver Cyclic Assignment** - $BRCA.with$, $BRCA.without$

   These receiver-initiated algorithms are very similar to $SRCA$ algorithms except that the batch assignment approach as described in section 4.1.1.2 is being employed. Note that these two algorithms are counterparts to $SRCA.with$ and $SRCA.without$. They serve to compare the performance between single task assignment and batch assignment.

4. **Single Symmetrical Cyclic Assignment** — $SXCA.with$, $SXCA.without$

   These symmetrically-initiated algorithms are combinations of the $SSCA$ and the $SRCA$ algorithms. Algorithm $SSCA$ is initiated when a local task arrival triggers its arrival node into the H-load state. Algorithm $SRCA$ is initiated when a task completion makes that node in L-load state.

5. **Batch Symmetrical Cyclic Assignment** — *BXCA.with*, *BXCA.without*

   These symmetrically-initiated algorithms are combinations of the *SSCA* and the *BRCA* algorithms. Algorithm *SSCA* is initiated when a local task arrival triggers its arrival node into the H-load state. Algorithm *BRCA* is initiated when a task completion makes that node in L-load state. *BXCA* algorithms can also be regarded as the batch assignment counterparts of *SXCA* algorithms.

4.1.4 Categorization of the Algorithms

Table 4.2 gives a detailed classification of the algorithms in term of policy types.

Table 4.2: Classification of AWLT and AWOLT algorithms according to policy types

Algorithms		*Sender-Initiated*	*Receiver-Initiated*	*Symmetrically-Initiated*	*Single/Batch Assignment*
AWLT	SSCA.with	√			S
	SRCA.with		√		S
	BRCA.with		√		B
	SXCA.with			√	S
	BXCA.with			√	S/B
AWOLT	SSCA.without	√			S
	SRCA.without		√		S
	BRCA.without		√		B
	SXCA.without			√	S
	BXCA.without			√	S/B

4.2 Simulations and Analysis of Results

According to the algorithms presented in section 4.1, we have run numerous simulations with different parameter values using SimScript II.5. This section describes and analyses the simulation results.

4.2.1 Performance Comparisons

Figure 4.3 and Figure 4.4 present the performance of the AWLT and AWOLT algorithms respectively. The values of the simulation parameters used in these particular sets of simulations are presented in Table 4.3.

Table 4.3: Values of simulation parameters used for studying performance of AWLT and AWOLT algorithms — presented in Figure 4.3 and 4.4.

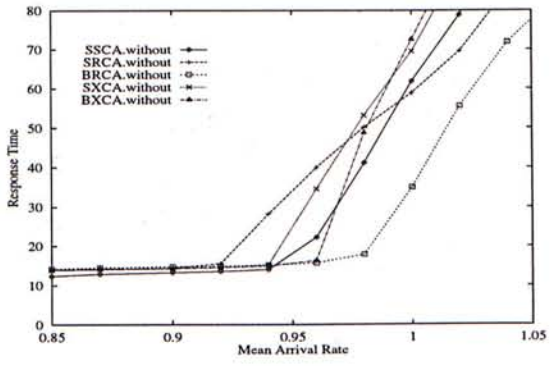
Parameter	*Value*
Q_o	30
T_{CPU}	0.2
N	30
σ_λ	1
S_o	1
lower_threshold	5
upper_threshold	$Q_o + 2$
probe_limit	5

Parameter	*Value*
$CPU_{polling}$	0.005
$F_{polling}$	0.001
F_{task}	0.005
D	0.01
C_{pack}	0.003
C_{assign}	0.002
l_{assign}	5

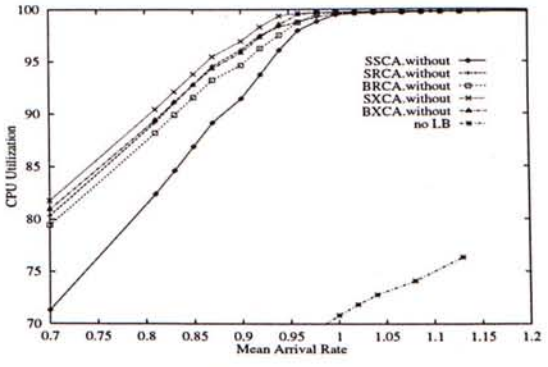
Performance Improvement

Figure 4.3(a) and Figure 4.4(a) depict the average task response time of all the algorithms. Compare with a system running no load balancing algorithm, the average response time improvement is roughly 23 times for both AWLT and AWOLT algorithms. (The data for not using load balancing algorithms are not shown in the figures as they are out of the range in the figures.) Similarly, the CPU utilization is increased for both sets of algorithms as shown in Figure 4.3(b) and Figure 4.4(b). The most significant reason for the increase of CPU utilization is that idle machines and lightly loaded machines are more fully utilized. This also accounts for the improvement in response time. However, the increased CPU utilization is partly contributed by the overhead of running the load balancing algorithms as shown in Figure 4.3(c) and Figure 4.4(c). In any case, the net CPU utilization still outperforms a system without load balancing algorithm, as shown in Figure 4.3(d) and Figure 4.4(d).

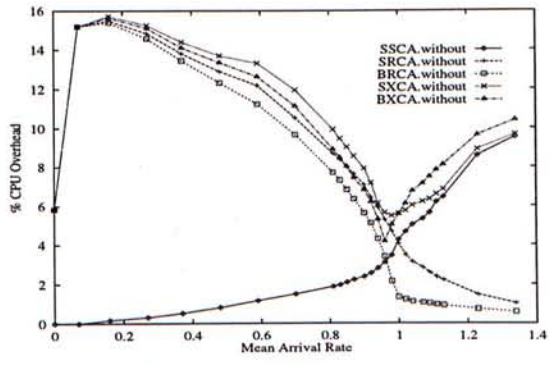
Figure 4.3(e) and Figure 4.4(e) show the standard deviation of response time for the



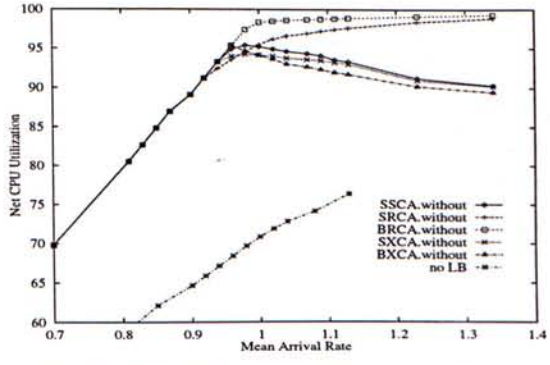
(a) Response Time



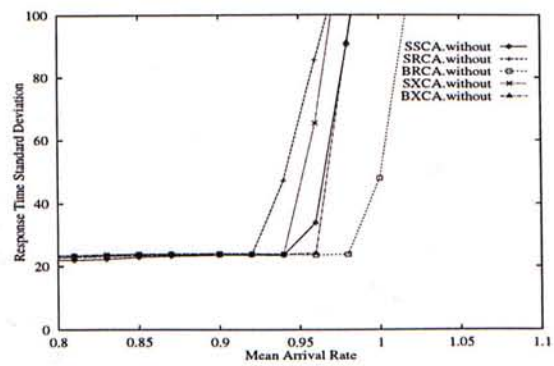
(b) CPU Utilization



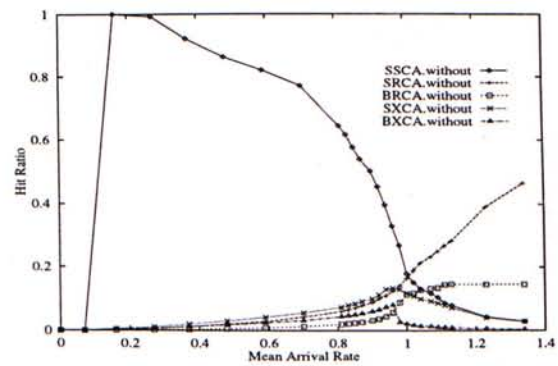
(c) Percentage CPU Overhead



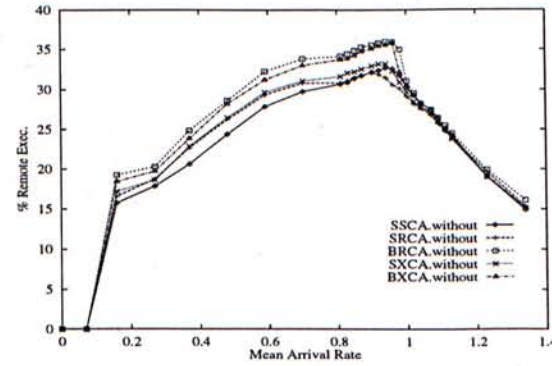
(d) Net CPU Utilization = Measured CPU Utilization - CPU Overhead



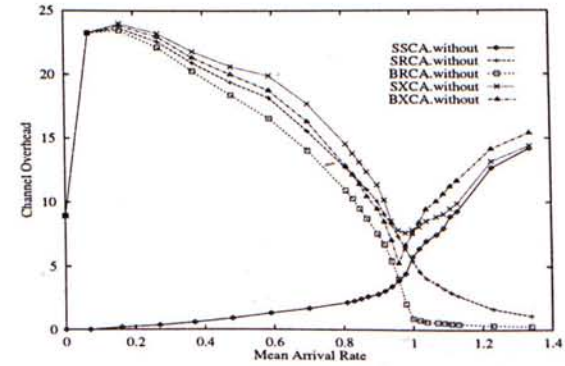
(e) Response Time Standard Deviation



(f) Hit Ratio

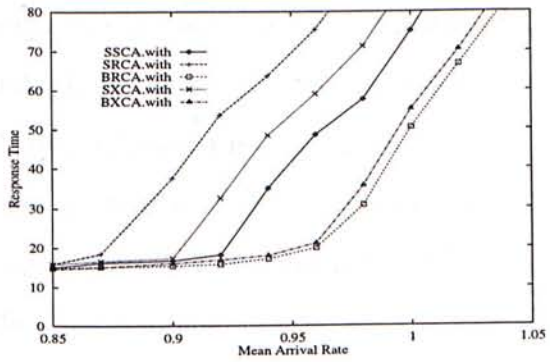


(g) Remote Execution Percentage

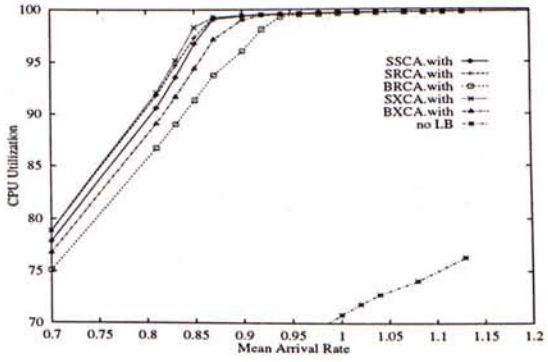


(h) Channel Overhead in Terms of Mean Number of Messages in Channel

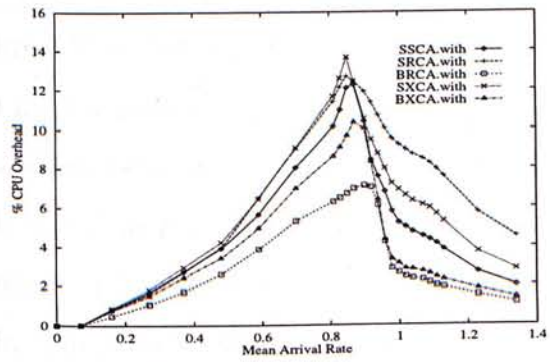
Figure 4.3: Performance of AWOLT algorithms. Values of simulation parameters are given in Table 4.3 on page 44.



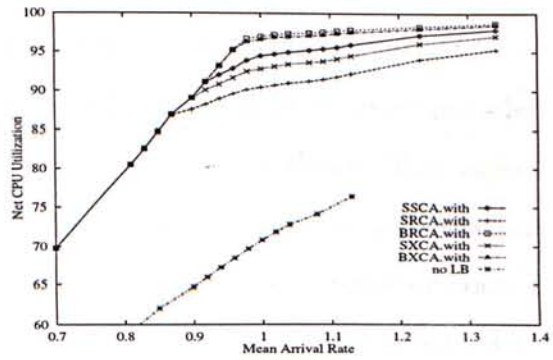
(a) Response Time



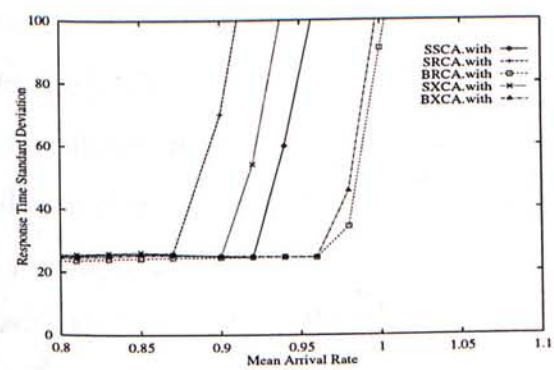
(b) CPU Utilization



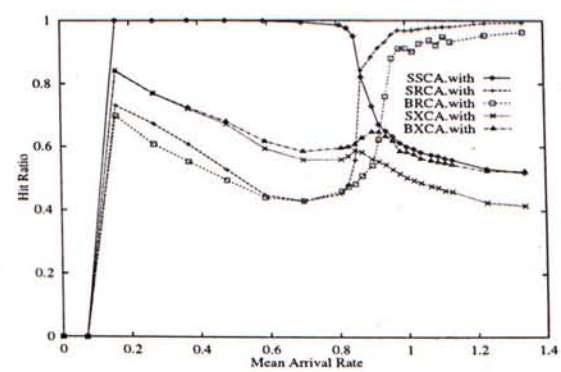
(c) Percentage CPU Overhead



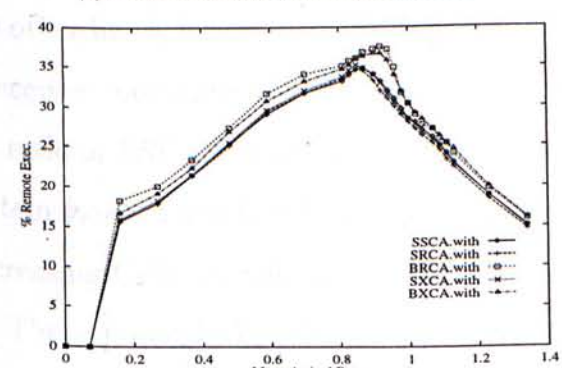
(d) Net CPU Utilization = Measured CPU Utilization - CPU Overhead



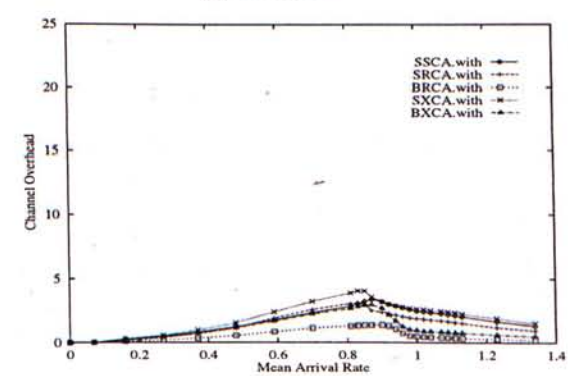
(e) Response Time Standard Deviation



(f) Hit Ratio



(g) Remote Execution Percentage



(h) Channel Overhead in Terms of Mean Number of Messages in Channel

Figure 4.4: Performance of AWLT algorithms. Values of simulation parameters are given in Table 4.3. on page 44

algorithms. The average improvement of this performance metric compared to a system without load balancing algorithm is roughly 17 times for both AWLT and AWOLT algorithms. (Again, the data for not using load balancing algorithms is not shown in the figures as they are out of the range.) This indicates that a system using load balancing algorithms not only provides a faster response time, users can also expect a more predictable response time.

Performance of AWOLT Algorithms

Figure 4.3(a) shows that the pure sender-initiated algorithm (*SSCA.without*) performs the best at system loads up to around 0.93. At higher system loads, the pure receiver-initiated algorithms (*SRCA.without* and *BRCA.without*) outperform *SSCA.without*. This agrees with many published work on load balancing algorithms study and can be explained as follows. At low system load, the probability that a sender node finds a receiver node is high. This is reflected in the high hit ratio of *SSCA.without* as shown in Figure 4.3(f). This also explains why the CPU overhead and channel overhead of *SSCA.without* at low system load are very low as shown in Figure 4.3(c) and (h). At high system load, most nodes are heavily loaded and the probability that a sender node finds a receiver node is low. This results in the decreasing hit ratio for sender-initiated algorithms. A sender node may need to poll many times (but up to *probe-limit*) before it can find a receiver or abandon the polling operation. The CPU and channel overhead of sender-initiated algorithms go up with the system load, thus worsening the already busy sender nodes. Some literatures refer to such pollings with low hit ratio as *indiscriminate probings* (*pollings*) [SK90]. On the other hand, for receiver-initiated algorithms at low system loads, the probability that a receiver node finds a sender node is low. This explains the high CPU overhead and low hit ratio of *SRCA.without* and *BRCA.without* as shown in Figure 4.3(c) and (f). At high system loads, a receiver has no problem in finding a sender node. This accounts for the decreasing CPU and channel overheads of receiver-initiated algorithms.

The symmetrically-initiated cyclic assignment algorithms (**XCA.without*) are essentially non-intelligent combination of the sender- and receiver-initiated components. Thus,

the CPU and channel overhead are more or less the sum of the two components. Their performance is therefore the worst among AWOLT algorithms.

The above results indicate that if polling is being used as the information dissemination strategy, neither pure sender-initiated nor pure receiver-initiated algorithms are satisfactory over the whole range of system load. To remedy such intrinsic weakness of these two kinds of algorithms, an adaptive algorithm which alters its scheduling policy according to current system load can be used [SK90]. Or, we can use the load table approach as in AWLT algorithms.

Comparison Between the Two Categories

The most notable difference between the two categories of algorithms is in the CPU overhead and channel overhead (Figure 4.3(c), (h) and Figure 4.4(c), (h)). For AWOLT algorithms, both CPU and channel overhead change with system load. The trend depends on the nature of the particular algorithm. For pure receiver-initiated algorithms (*SRCA.without* and *BRCA.without*), both the CPU and channel overhead are relatively large at low system load. They are decreasing with increasing system load however. The pure sender-initiated algorithm (*SSCA.without*) has the reverse trend. These trends are a direct result of the indiscriminate pollings discussed earlier. Symmetrically-initiated algorithms have a trend more or less as the addition of their individual sender-initiated and receiver-initiated components.

For AWLT algorithms, both CPU and channel overhead increase with the system load until some point where the system becomes saturated. After the saturation point, they decrease and converge rapidly. This trend reflects that load tables function as a filter to avoid indiscriminate pollings. Few H-load nodes can be found in load tables at low system load and few L-load nodes can be found in load tables at high system load. Inappropriate nodes are avoided from being considered as the targets for getting transfer consent.

Another major difference is in the hit ratio (Figure 4.3(f) and 4.4(f)). It can be seen that the hit ratio of AWLT algorithms is always better than that of AWOLT algorithms. This again reflects the filtering effect of load tables.

Performance of Batch Assignment Algorithms

Figure 4.3(a) & (e) and Figure 4.4(a) & (e) show that the batch assignment algorithms, *BRCA.without* and *BRCA.with*, have the best performance in terms of both task response time and standard deviation of task response time. This is because whenever a receiver node finds a sender node, *BRCA* algorithms can remove a batch of tasks from the sender in a single negotiation session. Load imbalance is smoothed out efficiently. This is reflected in the relatively high remote execution percentage of *BRCA* algorithms (when compared to their single task assignment counterparts, *SRCA* algorithms), as shown in Figure 4.3(g) and Figure 4.4(g). Since less polling cycles are needed, *BRCA* algorithms have relatively low CPU and channel overheads (again, compared to *SRCA.without*), as shown in Figure 4.3(c) and (h). All these mean that *BRCA* algorithms are making the most efficient use of available CPU capacity. This is shown in Figure 4.3(d) and Figure 4.4(d). In summary, batch assignment *BRCA* algorithms can smooth out workload imbalance efficiently through a higher percentage of task transfer with minimum CPU and communication overheads. In terms of both performance and efficiency, batch task assignment is promising.

4.2.2 Effect of Imbalance Factor on AWLT Algorithms

Imbalance factor is a quantitative measure of the degree of workload imbalance. In order to study how the algorithms perform under different degrees of workload imbalance, a set of simulations are conducted with different imbalance factors $\sigma_\lambda = \{0.01, 1.00, and 3.00\}$. Figure 4.5 depicts the distribution of task arrival rates with the three different imbalance factors. It can be seen that with small imbalance factor (0.01), all the nodes in the system have very close mean task arrival rates. As the imbalance factor increases, the distribution of task arrival rates are more dispersed. This means that a few nodes tend to have a much higher task arrival rate than the others. In such case, more task relocations are needed for attaining a workload balance among the nodes. Figure 4.6 shows the effect of imbalance factor on the performance of AWLT algorithms. Corresponding simulation parameters are shown in Table 4.4. We have the following observations.

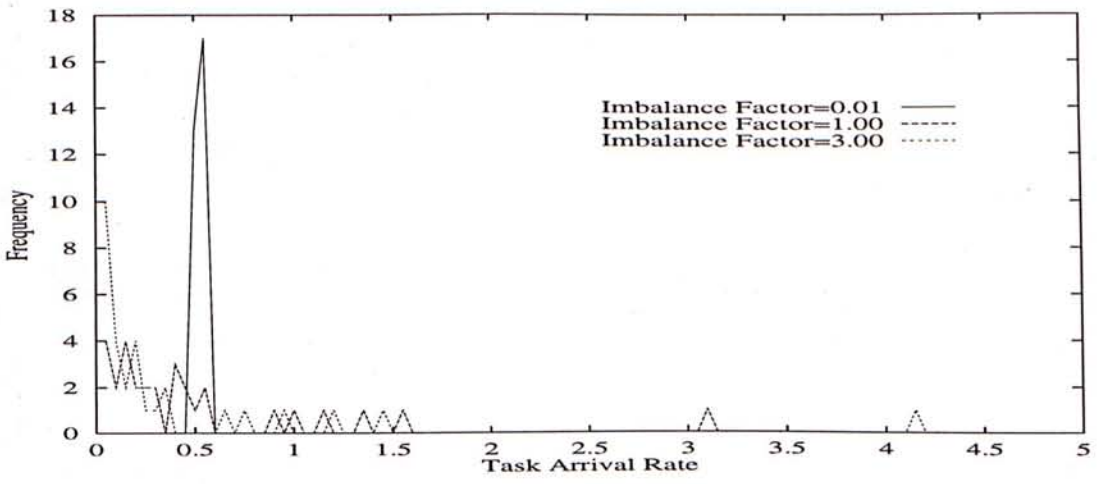


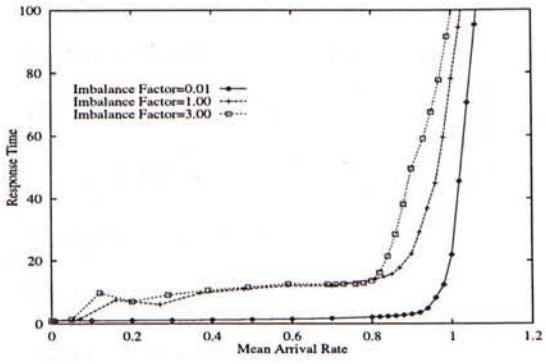
Figure 4.5: Distribution of arrival rates under varying imbalance factors. Log Normal Distribution with mean 0.5. Generated by SimScript II.5 for 30 processing nodes ($N = 30$).

Table 4.4: Values of simulation parameters used in the simulations presented in Figures 4.6.

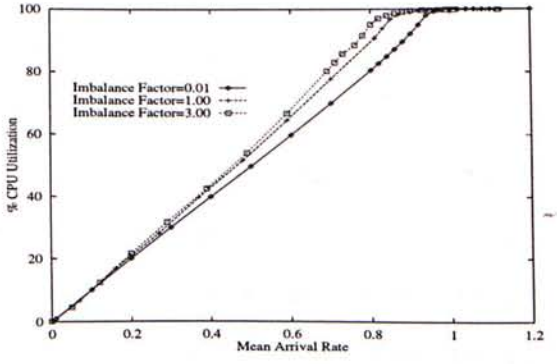
Parameter	*Value*
Q_o	30
T_{CPU}	0.2
N	30
σ_λ	*variable*
S_o	1
lower_threshold	5
upper_threshold	$Q_o + 2$
probe_limit	5

Parameter	*Value*
$CPU_{polling}$	0.005
$F_{polling}$	0.001
F_{task}	0.005
D	0.01
C_{pack}	0.003
C_{assign}	0.002
l_{assign}	5

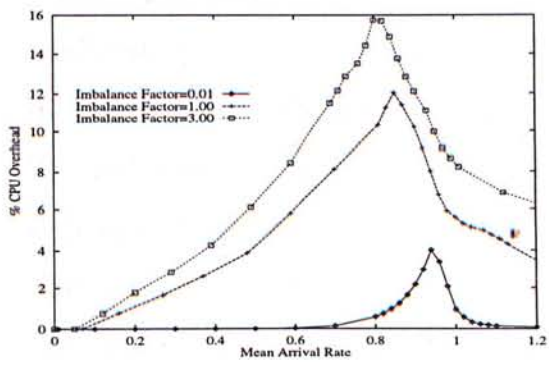
1. Figure 4.6(a) shows that for a small imbalance factor ($\sigma_\lambda = 0.01$), the performance of AWLT algorithms is very close to the theoretical performance of a M/M/1 system, that is, saturation occurs at system load of 1.0. In fact, Figure 4.6(d) shows that the net CPU utilization is very close to 100% when the system load is above 1.0, for $\sigma_\lambda = 0.01$. This can be explained as follows. For small imbalance factor, the task arrival rates of the processing nodes in the system are very close. Most of the nodes therefore have the same load state most of the time. At low system load, a receiver node can hardly find a sender node. Similarly, at high system load, a sender node can hardly find a receiver node. Thus, there are not many task relocations. This is shown in Figure 4.6(e) as a very small percentage remote execution when $\sigma_\lambda = 0.01$. Because of the small number of task relocations, CPU overhead and



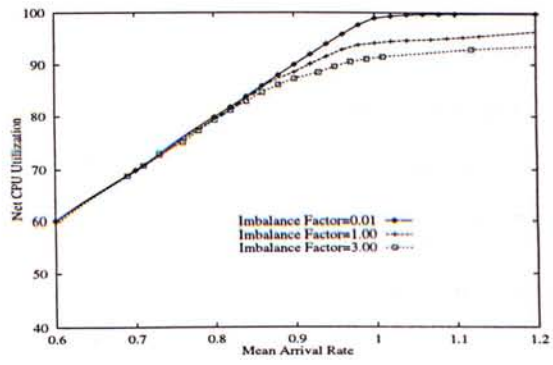
(a) Response Time



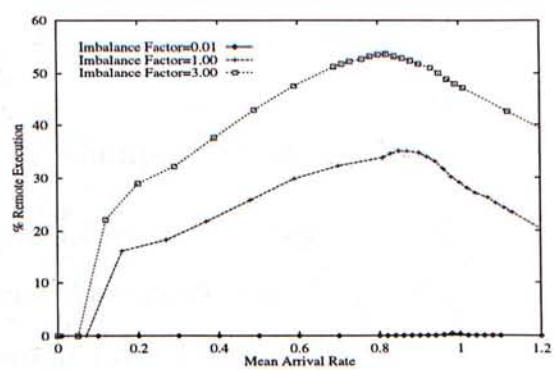
(b) Percentage CPU Utilization



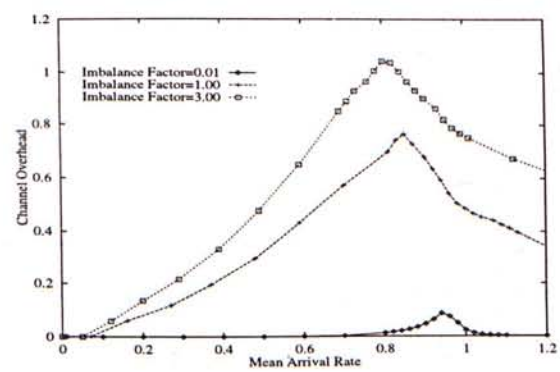
(c) Percentage CPU Overhead



(d) Net CPU Utilization



(e) Percentage Remote Execution



(f) Channel Overhead

Figure 4.6: Effect of imbalance factor on performance of AWLT algorithms. Simulation parameters are shown in Table 4.4 on page 50.

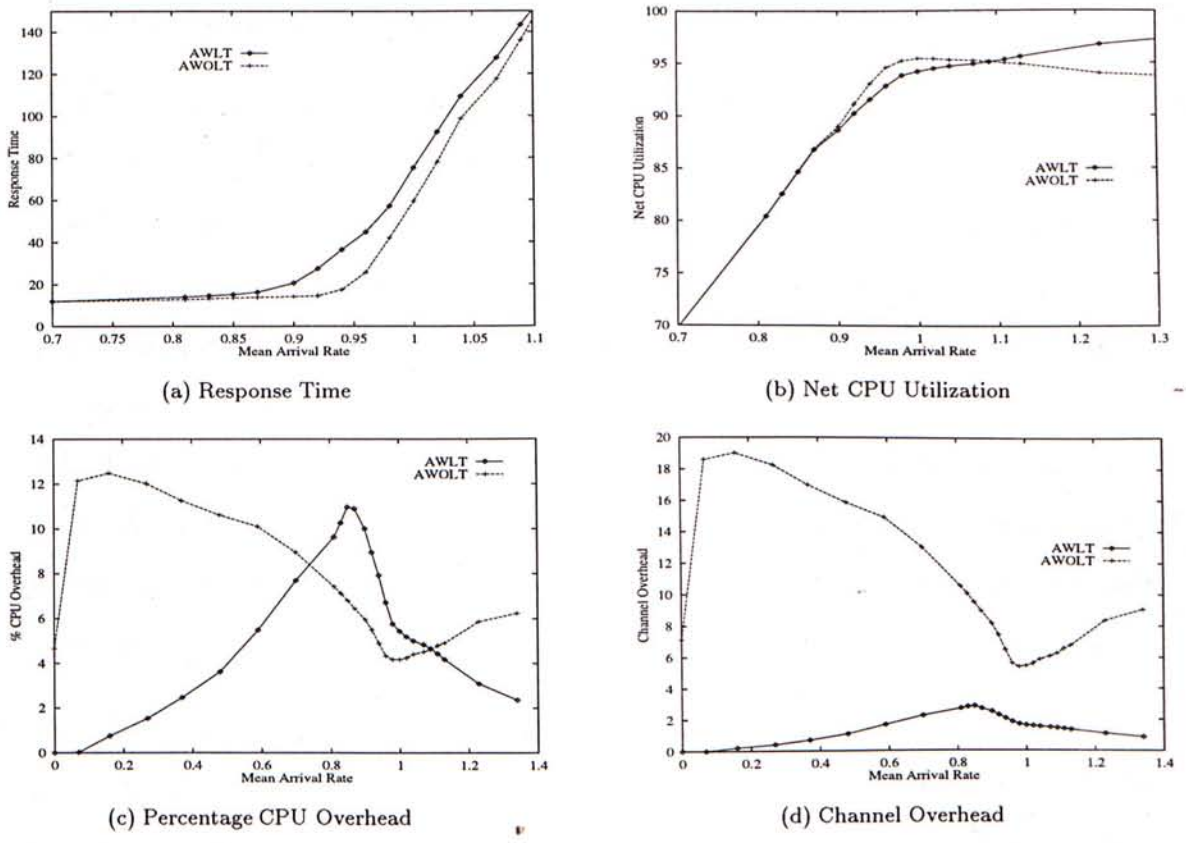
channel utilization associated with task relocations are also small. These are shown in Figure 4.6(c) and Figure 4.6(f), respectively.

2. Figure 4.6(a) shows that as the imbalance factor increases, the performance of AWLT algorithms degrades. This can be explained as follows. For larger imbalance factors, the task arrival rates of the processing nodes are more dispersed. That is, some nodes tend to have significantly larger task arrival rates than others. We can expect that more task relocations are needed in order to maintain the workload balance. Figure 4.6(e) shows that as the imbalance factor increases, the percentage remote executions also increases. Similar trend can be observed for CPU and channel overhead, as shown in Figure 4.6(c) and (f) respectively. Because of the larger CPU overhead, less CPU capacity is available for user task processing. This is shown in Figure 4.6(d) as a decreasing net CPU utilization as imbalance factor increases. The reduced processing capacity accounts for the increasing task response time as imbalance factor increases. Another reason for the increased task response time is that a large portion (around 30% when $\sigma_\lambda = 1.00$ at system load 0.9, and around 55% when $\sigma_\lambda = 3.00$ at system load 0.9) of tasks are executed remotely. These task experienced time delay due to assignment algorithm execution and communication delay via the network.

4.2.3 Comparison of Average Performance

Figure 4.7 shows a comparison of the average performance of AWLT and AWOLT algorithms. The simulation parameters associated with the simulations are identical to those shown in Table 4.3 on page 44. We have the following performance observations.

At low to medium load (up to 0.75), the two categories have identical performance in terms of task response time. At high system load, AWOLT algorithms perform better than AWLT algorithms. This can be attributed to three reasons.



(a) Response Time

(b) Net CPU Utilization

(c) Percentage CPU Overhead

(d) Channel Overhead

Figure 4.7: Average performance of the AWLT and AWOLT algorithms. Simulation parameters are shown in Table 4.3 on page 44. Average performance is taken to be the mean of the performance results of the component algorithms within each category.

1. AWLT — Less CPU capacity is being used for processing user tasks:

   From Figure 4.7(c), it can be seen that for system loads below 0.75, AWLT algorithms have a lower CPU overhead than AWOLT algorithms. In this range of loads, the system has enough spare capacity to accommodate the overhead introduced by running the load balancing algorithms without imposing adverse effect on the mean task response time. The difference in amount of overhead is virtually insignificant. This results in the identical mean task response time.

   Beyond system load 0.75, the system is becoming saturated and the system is less stable than before. In AWLT algorithms, frequent exchanges of load information messages are needed. This results in the high CPU overhead of AWLT algorithms as shown in Figure 4.7(c). This means that in AWLT algorithms, less CPU capacity

is available for processing user tasks, as shown in Figure 4.7(b).

2. AWLT — Out-dated Load Information in Load Tables:
   For both categories of algorithms, a sender or a receiver needs to get the consent from its counterpart before task transfer can take place. Conceptually, in AWOLT algorithms, the polling process is a combination of load state query and transfer negotiation. This means that the polling operation tries to locate a node which is appropriately loaded and to negotiate with that node for the task transfer. In AWLT algorithms, host selection from load tables may not be helpful unless the load information stored is up-to-date. Due to communication delay, this may not be the case however. This problem is particularly serious if the network is heavily congested or the communication delay is high.

3. AWLT — Higher Chance of Processor Thrashing:
   In AWLT algorithms, the filtering effect of load tables limits the number of potential target nodes from which a node can get consent for task transfer. This results in a phenomenon called *processor thrashing*, which refers to the situation where a number of sender nodes try to negotiate with the same potential receiver node simultaneously, or vice versa. Processor thrashing results in the adverse effect of wasted negotiations (low hit ratio) and wasted CPU capacity (high CPU overhead) which otherwise can be utilized for processing user tasks. Since host selection in AWOLT algorithms is more dispersed, the workload may be more evenly distributed.

On the other hand, Figure 4.7(d) shows that AWLT algorithms have much lower channel overhead then AWOLT algorithms. This simply reflects: (1) the indiscriminate pollings of AWOLT algorithms at high and low system loads; and (2) the broadcasting nature of the channel results in efficient distribution of load state information.

4.2.4 Raw Simulation Results

We have run numerous simulations to study the performance of AWOLT and AWLT algorithms. However, due to the limitation of length, we are not able to show all the

simulation results in this chapter. Instead, the simulation results of AWOLT and AWLT algorithms are shown in Appendix C and Appendix D respectively.

4.3 Discussions

For algorithms that make reference to load tables (AWLT algorithms), both CPU and channel overheads exhibit a regular pattern — increases with the system load until system saturation, and then drops and converges rapidly at higher system loads. The pattern can be attributed to the filtering effect of the load table, which limits the number of potential target nodes from which a node can get consent of task transfer. However, the performance of AWLT algorithms is limited by the following three factors:

1. High CPU overhead for load information broadcasting;
2. Out-of-date load information due to communication delay; and
3. Processor thrashing.

The results suggest that maintaining a load table in a broadcasting channel environment does not provide performance advantage over the use of polling. However, if channel utilization is a concern, sacrificing system performance for reducing channel overhead may worth consideration.

On the other hand, neither pure sender-initiated nor pure receiver-initiated polling algorithms are satisfactory over the whole range of system load. To remedy such intrinsic weakness of these two kinds of polling algorithms, an *adaptive* symmetrically-initiated polling-based location policy, which also exhibits the filtering effect can be used. An example of such location policy is proposed by Shivaratri and Krueger in [SK90]. However, because of the filtering effect, processor thrashing can be expected. To put this kind of location policy into practical use, the problem of processor thrashing have to be resolved. This is the major theme of the next chapter.

Chapter 5

Resolving Processor Thrashing with Batch Assignment

In the last chapter, we pointed out that an adaptive symmetrically-initiated location policy suffers from the phenomenon of processor thrashing. Processor thrashing can result in at least two adverse effects:

- The workload in the system may not be as evenly distributed as desired.
- A receiver node may exceed its processing capacity because of congestion at the receiver node due to over-drafting.

In this chapter, we attempt to remedy the problem of processor thrashing by modifying and applying the batch assignment approach proposed in the last chapter. This new batch assignment algorithm is labeled as the *GR.batch* algorithm, which is based on the tight integration of three components:

1. The batch task transfer policy, which allows a number of tasks to be transferred as a single batch from senders to receivers.
2. A sender-receiver negotiation protocol, referred to as the *Guarantee and Reservation Protocol* (or *GR Protocol* for short), which together with the batch transfer policy

The content of this chapter has been accepted for publication by *Concurrency: Practice and Experience*; Special issue on dynamic resource management in distributed systems; October, 1995, volume 7, number 7 [LL95a].

obtains the mutual agreement between a sender and a receiver on the optimal batch size. The central idea of the GR protocol is two fold: (1) A sender node has to declare a *guarantee value* to a potential receiver. This value signifies the number of tasks the sender node guarantees to send to the potential receiver. (2) The potential receiver employs a "quota" scheme for reserving processing capacity for task batches from sender nodes. This GR protocol is the primary vehicle for resolving processor thrashing.

3. An adaptive symmetrically-initiated location policy based on the adaptive location policy proposed by Shivaratri and Krueger in [SK90]. The key feature of the policy is to utilize the information gathered during pollings to keep track of recent workload states of processing nodes. Such workload information serves as a filter to cut off inappropriate polling candidates and thus avoids indiscriminate pollings.

5.1 The *GR.batch* Algorithm

This section describes the three policy components of *GR.batch*. Refer to Figures 5.3-5.7 for the complete *GR.batch* algorithm.

5.1.1 The Guarantee and Reservation Protocol

The GR Protocol is our primary vehicle to reduce processor thrashing. The basic idea of the GR protocol is as follows. When a sender-receiver pair is going to be formed, the sender has to declare the number of tasks it guarantees to send to the receiver. The receiver, based on such a guarantee value, determines the number of tasks it is willing to accept from the sender. The receiver reserves this number as a "quota" for the sender. When making negotiation with other senders, the receiver takes into account such quotas in measuring its load states and in deciding the number of tasks it is willing to accept from them. When tasks are received from the original sender, the corresponding quota is released. Of course, when measuring the workload state, the receiver takes into account these newly arrived remote tasks. The GR protocol requires the use of two attributes for

each processing node:

> *Reservation Value*: of a node P_i, denoted as RES_i, is the total number of tasks that P_i has agreed to accept from other sender nodes.
>
> *Guarantee Value*: of a node P_i, denoted as GUR_i, is the total number of tasks that P_i has guaranteed to transfer to other receiver nodes.

Based on RES_i, GUR_i, and the number of tasks currently residing in a node P_i, we can define the workload of P_i as follows.

> *Effective Load*: of a node P_i, denoted as EL_i, is defined as $EL_i = K_i + RES_i - GUR_i$, where K_i is the number of tasks currently residing in P_i, including those in the task queue, in the threshold queue, and those partially completed tasks residing in the service queue of P_i.

5.1.2 The Location Policy

The location policy of the *GR.batch* algorithm uses the idea of the adaptive location policy proposed by Shivaratri and Krueger [SK90]. This location policy is adaptive in the sense that it uses the receiver-initiated approach at high system load and uses the sender-initiated approach at low system load. Such algorithm initiation strategy is in accordance with the performance study done by Eager *et al.* [EL86a]. The original Shivaratri and Krueger's adaptive location policy is described in detail in Appendix B. The essence of the location policy is as follows.

The key feature of the location policy is to utilize the information gathered during both sender-initiated and receiver-initiated pollings to keep track of the recent load states of other nodes. The load information maintained by a node P_i is stored in three local list structures. The first list, $SList_i$, contains the *id*s of those processing nodes that have identified themselves as potential senders. The second list, $RList_i$, contains the *id*s of those processing nodes that have identified themselves as potential receivers. The third list, $NList_i$, contains the *id*s of those processing nodes that have identified themselves as normally loaded, thus requiring no task transfer. The head of each list always contains the node *id* with the most recent load information. Thus, when a sender-initiated polling session is to be started, the location policy selects the node *id* at the head of its $RList$

as the potential receiver. Conversely, when a receiver-initiated polling session is to be started, the location policy selects the node *id* at the head of its $SList$ as the potential sender. After a negotiation target has been identified, negotiation is carried out by the GR protocol.

The invocation of a polling session on a node is triggered by the node's change of load state. To determine the load state of a node, we use the 3-level load measurement scheme to describe the logical fullness of the queues in a node. Again, the three load levels are **H-load**, **N-load**, and **L-load**. Since the GR Protocol is used, the load state of a node is measured by the effective load of the node. Table 5.1 gives the relationship between the 3-level load measurement scheme and the effective load. Again, sender-initiated polling sessions start whenever a local task arrival triggers its arrival node into the H-load state; while receiver-initiated polling sessions start if a task completion puts that node in L-load state.

Table 5.1: The 3-level load measurement scheme based on effective load, EL_i.

Load State	*Criteria*
L-load	$EL_i \leq lower\_threshold$
N-load	$lower\_threshold < EL_i \leq upper\_threshold$
H-load	$EL_i > upper\_threshold$

5.1.3 Batch Size Determination

The three batch size determination rules are restated below.

Let max = maximum number of tasks a receiver is willing to accept

t = number of tasks a sender is willing to send to the receiver

Rule 1: After accepting max tasks, the receiver should not be in H-load, neglecting new arrivals and departures during the negotiation and task transfer operations.

Rule 2: After transferring t tasks, the sender node should not be in L-load.

Rule 3: After transferring t tasks, the expected total number of tasks in the receiver should not be greater than the total number of tasks in the sender, neglecting new arrivals and departures of the receiver.

Again, the values of max and t are determined by the two functions: $MaxAssign()$ and $NumAssign()$, respectively. However, these functions are modified to cater for the fact that the effective load is being used to measure the workload of a processing node. These two modified functions are illustrated in Figure 5.1.

To determine the size of a task batch, a sender node P_x declares the number of tasks, g, it guarantees to send to receiver P_y in the polling message. Next, the receiver node determines max by calling $MaxAssign()$. The values of max and g are then compared, and reservation r made by receiver P_y is taken to be either max or g, whichever is smaller. Both max and r are sent to the sender P_x as an *ack* message. The function $NumAssign()$ is called by the sender to determine t, which is compared with r to determine the final batch size b. The value of b is taken to be either t or r, whichever is smaller. Note the integration of the GR protocol and the three rules in determining the size of a task batch.

After transferring a task batch to the receiver, the sender estimates the receiver's new effective load with the function $ReceiverNewEL()$. Such estimation is based on max and the batch size b. The function $ReceiverNewEL()$ is illustrated in Figure 5.2. The estimated effective load of the receiver is than used for modifying the list structures of the sender, and thus maintains the load information stored in the sender.

MaxAssign()	*NumAssign*()
Let EL_y = Current effective load of receiver y Let EL_y' = Estimated effective load of receiver y after task transfer If max tasks are relocated, $$EL_y' \approx EL_y + max$$ By **Rule 1**, $$EL_y' \leq upper\_threshold$$ Thus, $$max \leq upper\_threshold - EL_y$$ Taking the largest possible value, $$max = upper\_threshold - EL_y \quad (5.1)$$	Let EL_x = Effective load of sender x when the polling message is received Let t = Number of tasks sender x is willing to send to the receiver. By **Rule 2**, $$EL_x - t > lower\_threshold$$ $$t < EL_x - lower\_threshold \quad (5.2)$$ Using equation (5.1), sender node x estimates EL_y from max as follows. $$EL_y \approx upper\_threshold - max \quad (5.3)$$ By **Rule 3**, $$EL_y + t \leq EL_x - t$$ Thus, $$t \leq \frac{EL_x + max - upper\_threshold}{2} \quad (5.4)$$ Of course, $$t \leq max \quad (5.5)$$ t is taken to be the largest integer satisfying the inequalities (5.2), (5.4), and (5.5).

Figure 5.1: *MaxAssign*() and *NumAssign*() for determining the desired batch size t — based on EL_i.

Let EL_y' = estimated effective load of y
Let EL_y'' = estimated new effective load of y after accepting b tasks

By equation (5.3):

$$EL_y' \approx upper\_threshold - max$$

$$\begin{aligned} EL_y'' &\approx EL_y' + b \\ &= upper\_threshold - max + b \end{aligned} \quad (5.6)$$

Figure 5.2: Function *ReceiverNewEL*() for estimation of receiver's new effective load.

5.1.4 The Complete *GR.batch* Description

Figure 5.3 to 5.5 illustrate the sender-initiated component of the *GR.batch* algorithm. Figure 5.6 to 5.7 illustrate the receiver-initiated component.

Intentionally left blank.

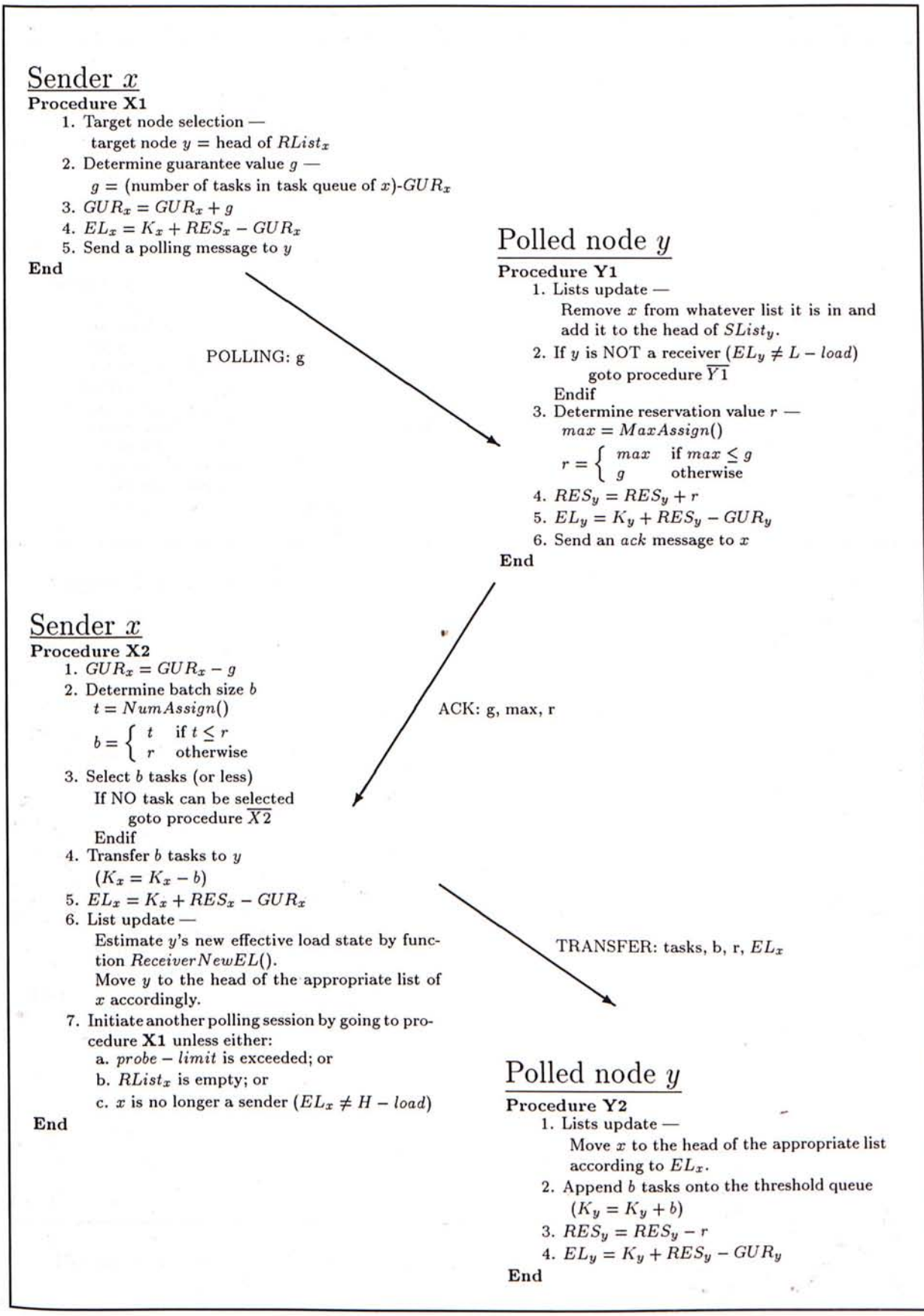


Figure 5.3: Sender-initiated component of the *GR.batch* algorithm

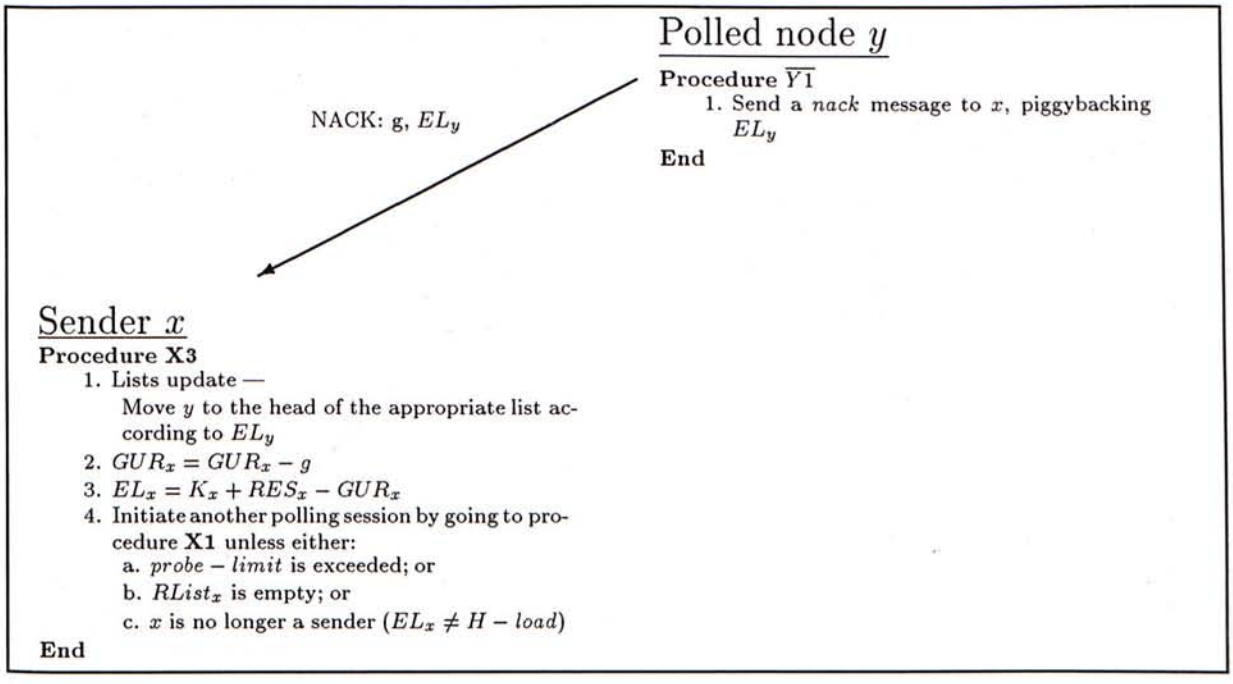


Figure 5.4: Sender-initiated component of the *GR.batch* algorithm — Procedure $\overline{Y1}$

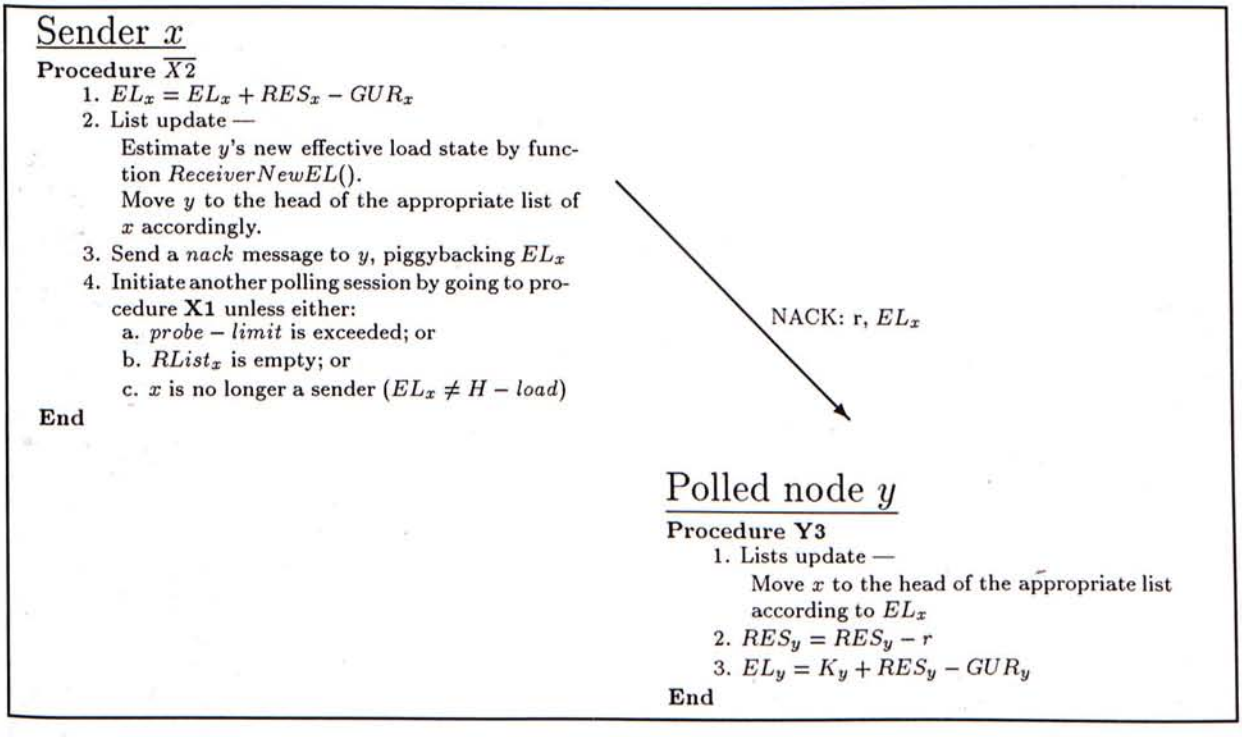


Figure 5.5: Sender-initiated component of the *GR.batch* algorithm — Procedure $\overline{X2}$

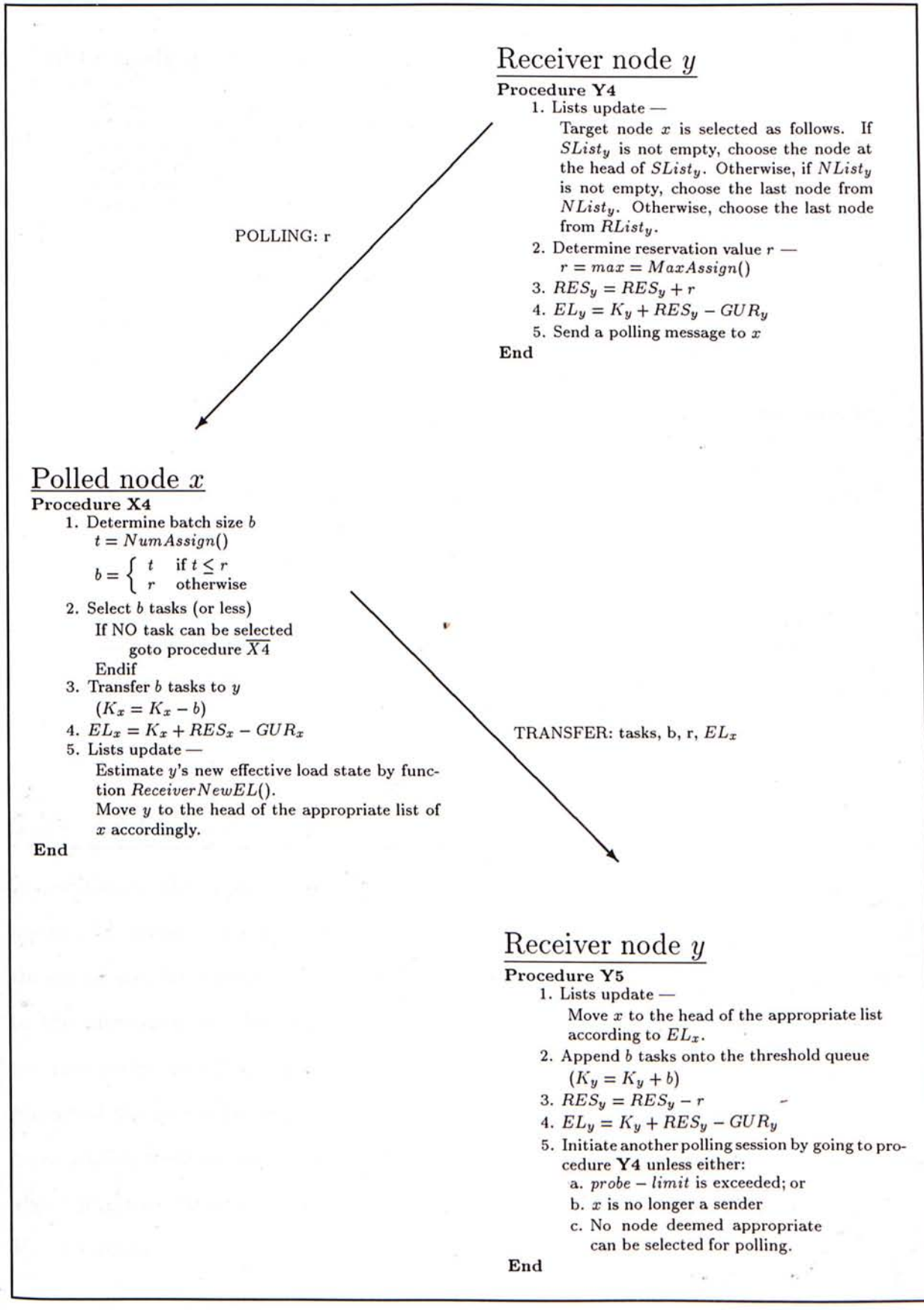


Figure 5.6: Receiver-initiated component of the *GR.batch* algorithm

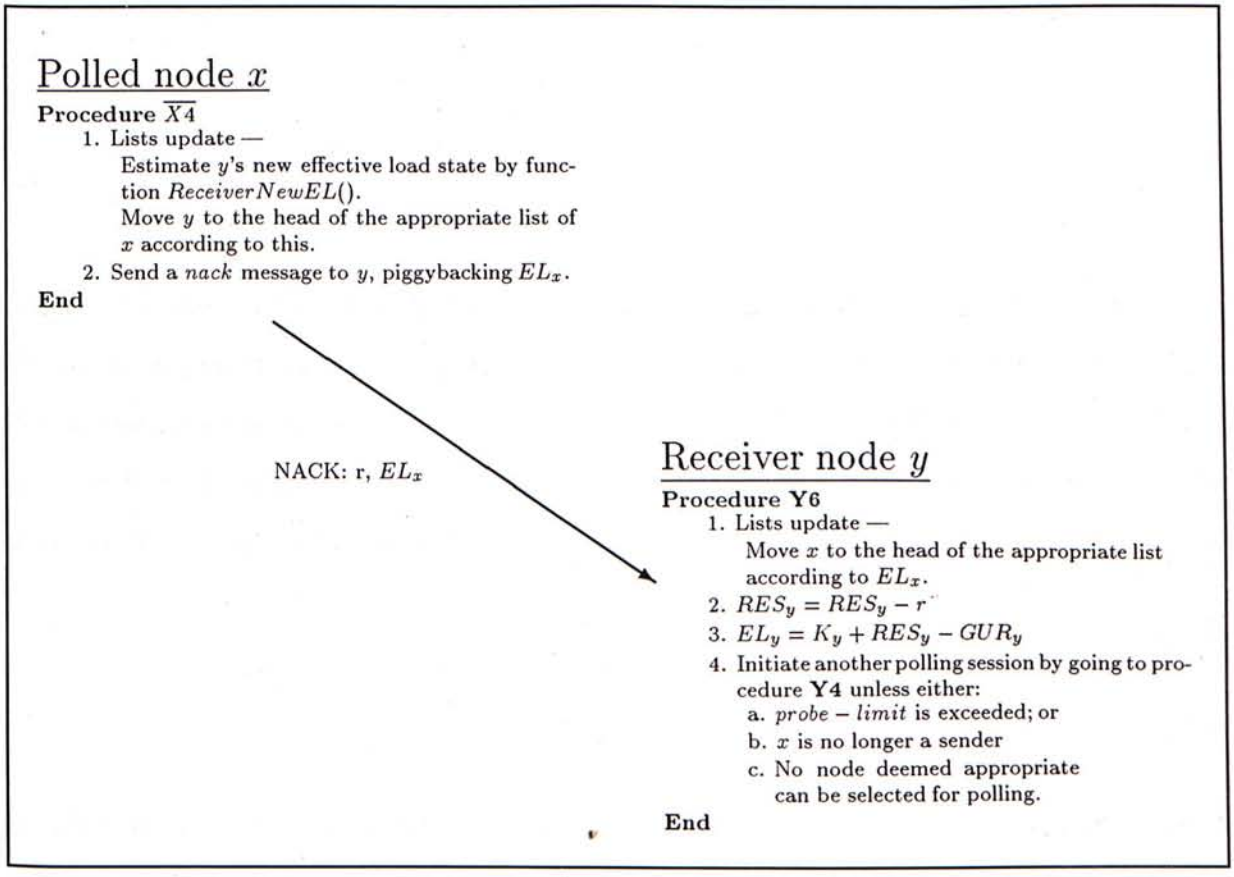


Figure 5.7: Receiver-initiated component of the *GR.batch* algorithm — Procedure $\overline{X4}$

5.2 Additional Performance Metrics

In addition to those performance metrics defined in Chapter 3, we define two additional metrics for measuring the degree of processor thrashing. Two different types of processor thrashing can be identified: *sender thrashing* and *receiver thrashing*. The former refers to the phenomenon when a particular potential sender is being polled by a number of receiver nodes simultaneously. The latter refers to the phenomenon when a particular potential receiver is being polled by a number of sender nodes simultaneously. To have a quantitative measure of processor thrashing as exhibited by an adaptive load balancing algorithm, we define the *sender thrashing coefficient*, V_s, and *receiver thrashing coefficient*, V_r, as follows.

$$V_s = \frac{\sqrt{\frac{\sum_{i=1}^{N}(S_i^{\leftarrow})^2 - N(\overline{S^{\leftarrow}})^2}{N}}}{\overline{S^{\leftarrow}}} = \frac{\sigma_{S^{\leftarrow}}}{\overline{S^{\leftarrow}}} \quad (5.7)$$

where N is the number of nodes in the system; $S_i^{\leftarrow}$ is the total number of times that node P_i has been polled by receiver nodes as if it is a sender during the simulation time; and $\overline{S^{\leftarrow}}$ is the mean of $S_i^{\leftarrow}$ over all the processing nodes in the distributed system. [1] Thus, V_s is in fact the *coefficient of variation* of the variable $S_i^{\leftarrow}$. Similarly, we can define the *receiver thrashing coefficient* as follows.

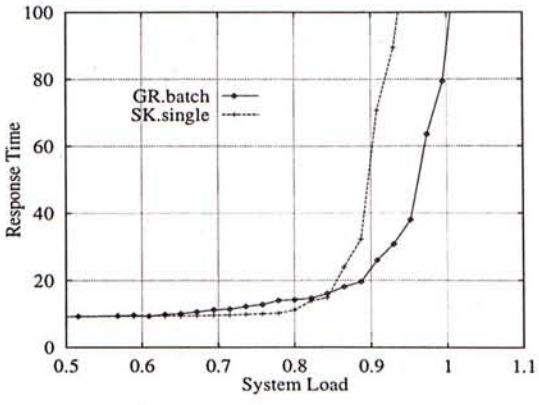
$$V_r = \frac{\sqrt{\frac{\sum_{i=1}^{N}(R_i^{\leftarrow})^2 - N(\overline{R^{\leftarrow}})^2}{N}}}{\overline{R^{\leftarrow}}} = \frac{\sigma_{R^{\leftarrow}}}{\overline{R^{\leftarrow}}} \quad (5.8)$$

Intuitively, V_s and V_r measure the degree of dispersion of receiver-initiated and sender-initiated polling activities respectively. A small value of V_s (V_r) signifies that most nodes receive more or less the same amount of receiver-initiated (sender-initiated) pollings. This in turn implies a larger degree of dispersion of receiver-initiated (sender-initiated) polling activities and thus a lower degree of sender (receiver) thrashing.

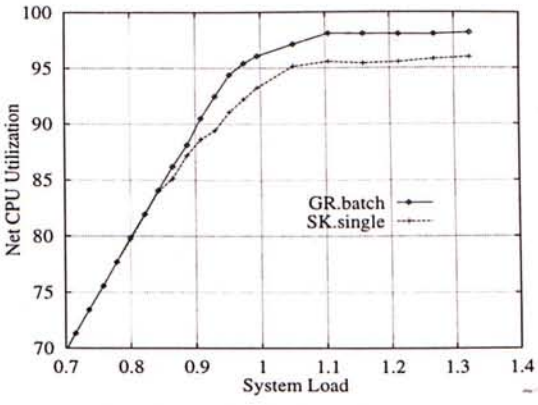
5.3 Simulations and Analysis of Results

In the simulations, an algorithm that uses the original Shivaratri and Krueger's location policy with single task assignment is used as a reference for comparing with the *GR.batch* algorithm. We label this algorithm as *SK.single*. The properties of these two algorithms are summarized in Table 5.3. The performance of the algorithms are compared by simulations. Figure 5.8 shows graphically the performance of the two algorithms. Table 5.2 gives the values of the simulation parameters used for the simulations. We have the following observations and analysis on their performance.

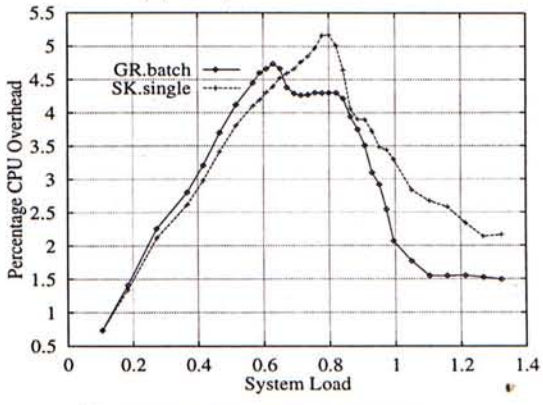
1 $\overline{S^{\leftarrow}} = \frac{\sum_{i=1}^{N} S_i^{\leftarrow}}{N}$



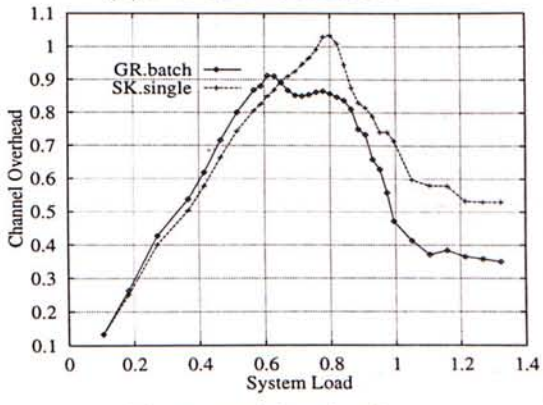
(a) Response Time



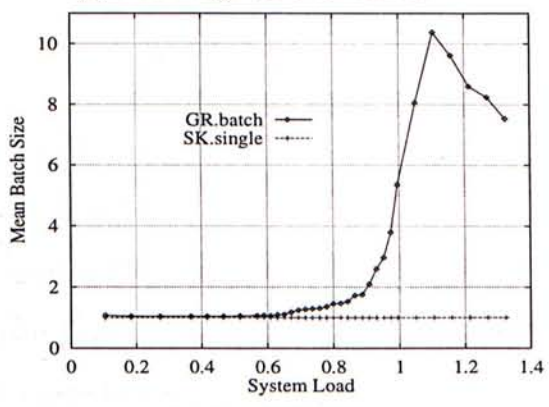
(b) Net CPU Utilization



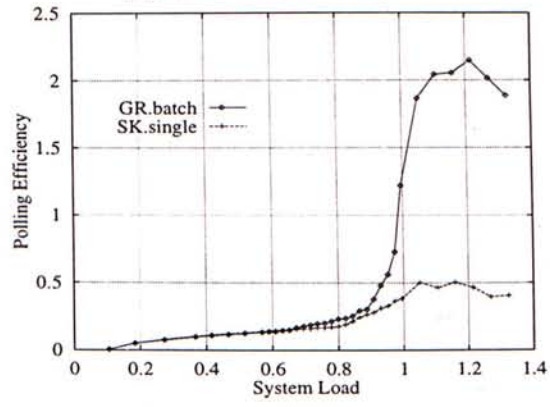
(c) Percentage CPU Overhead



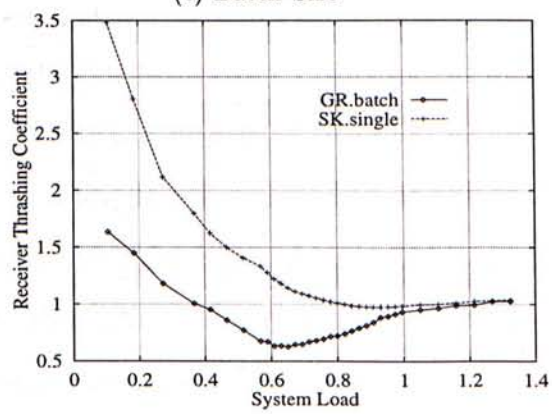
(d) Channel Overhead



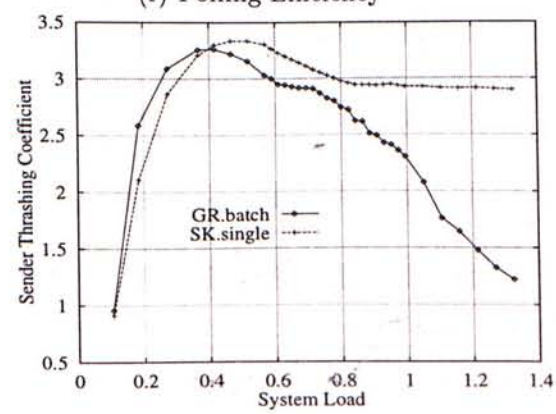
(e) Batch Size



(f) Polling Efficiency



(g) Receiver Thrashing Coefficient



(h) Sender Thrashing Coefficient

Figure 5.8: Performance comparisons between *GR.batch* and *SK.single*. Simulation parameters shown in Table 3.1.

Table 5.2: Values of simulation parameters used in the simulations for studying *GR.batch* and *SK.single*. Simulation results are presented in Figure 5.8 on page 68.

Parameter	Value	Parameter	Value
N	30	$CPU_{polling}$	0.005
Q_o	30	$F_{polling}$	0.001
T_{CPU}	0.1	F_{task}	0.005
σ_λ	3	D	0.005
S_o	1	C_{pack}	0.003
lower_threshold	5	C_{assign}	0.002
upper_threshold	20	l_o	5
probe_limit	5		

Table 5.3: Summary of properties of *GR.batch* and *SK.single*

Algorithm	*Location Policy*	*Load Measurement*	*Transfer Mode*	*GR Protocol*
GR.batch	Shivaratri and Krueger's Location Policy modified	Based on effective load, EL_i. Refer to table 5.1 on page 59.	A batch of tasks for each transfer session	Used
SK.single	Shivaratri and Krueger's Location Policy unmodified	Based on actual number of tasks residing in a node, K_i. Refer to table 4.1 on page 38.	Single task for each transfer session	Not used

Observation One — *SK.single* **and** *GR.batch* **have identical performance at low system load.**

Figure 5.8(a), shows that *SK.single* and *GR.batch* have identical performance in terms of mean task response time up to system load 0.63. As we can see from Figure 5.8(e), the average batch size of *GR.batch* at low system load is very close to 1. This means that *GR.batch* at low system load is roughly identical to single task assignment. However, the use of the GR Protocol in *GR.batch* produces more message exchanges. This explains the higher CPU and channel overhead as shown in Figure 5.8(c) and (d) respectively. Since the system at low system load has enough spare capacity to handle the extra overhead, the GR Protocol does not have adverse effect on the system performance. In fact, Figure 5.8(b) shows that at low system load, the two algorithms provide identical net CPU utilization, meaning that the extra overhead imposed by the GR Protocol consumes only those CPU

capacity which would be otherwise unused.

Observation Two — *GR.batch* **provides a much higher performance saturation point than that of** *SK.single*.

Figure 5.8(a) shows that beyond system load 0.8, *SK.single* becomes saturated and its performance degrades exponentially. Algorithm *GR.batch*, however, still provides a good performance and it becomes saturated at a significantly higher system load. As Figure 5.8(b) shows, *GR.batch* has a much higher net CPU utilization than *SK.single* at higher system load. The difference in net CPU utilization means that with *SK.single*, the CPU capacity of some lightly loaded nodes is not fully used. In other words, system workload in the case of *SK.single* is not as evenly distributed as in the case of *GR.batch*. With *SK.single*, those lightly loaded receivers receives only one task at a time. Therefore, the load imbalance is smoothed out slower than when *GR.batch* is used. Another reason for the uneven workload distribution in the case of *SK.single* is the phenomenon of processor thrashing. With *SK.single*, there is a possibility that tasks from different senders are sent to the same receiver, letting the spare CPU capacity of other potential receivers unused. Since *GR.batch* can utilize the system capacity more fully than that of the *SK.single*, the better average response time is easy to understand.

Observation Three — **At high system load,** *GR.batch* **has lower channel and CPU overhead.**

Figure 5.8(d) shows the channel overhead of the two algorithms. At low system load, the channel overhead of *GR.batch* is slightly higher than that of *SK.single*. At high system load, the reverse occurs. Similar pattern is exhibited by the level of CPU overhead, as shown in Figure 5.8(c). The higher channel and CPU overhead of *SK.single* at high system load can be explained by two reasons: (1) With *SK.single*, a larger portion of polling sessions fail to locate a transfer partner because of the effect of processor thrashing. Except those where *probe_limit* has been reached, each of the failed polling sessions causes another new polling session to be initiated, in an attempt to search for another transfer partner. Thus, a lot more polling sessions are injected. (2) When compared to *GR.batch*,

algorithm *SK.single* requires more polling sessions for transferring the same amount of workload between processing nodes. This is reflected in its significantly lower polling efficiency at high system load, as shown in Figure 5.8(f).

Observation Four — ***GR.batch* exhibits a lower level of receiver thrashing throughout the whole range of system load.**

Figure 5.8(g) shows the variation of receiver thrashing coefficient. It can be seen that as system load increases, *SK.single* exhibits a decreasing receiver thrashing until system load 0.8, beyond which the level of receiver thrashing remains relatively stable at $V_r \approx 1$. *GR.batch* also exhibits a decreasing receiver thrashing at low system load until the trough at around 0.62 is reached. After the trough, the level of receiver thrashing increases gradually until it becomes stable at system load 1.0. Note that *GR.batch* always exhibits a lower receiver thrashing before system saturation. Note also that at high system load, the two algorithms exhibit virtually identical level of receiver thrashing.

At low system load, most processing nodes are lightly loaded potential receivers. There are only a few sender nodes. With *SK.single*, the probability that a sender node being bound by a particular receiver node is high. This is because the receiver has enough spare capacity to serve the sender. Therefore, at low system load, there are only a few actual receiver nodes to which sender-initiated pollings are targeted. This explains the high receiver thrashing at low system load. With *GR.batch*, a sender node is slightly more likely to search for another receiver after transferring a batch of tasks to its current transfer partner. This is because the task batch may be large enough to use up all the spare capacity of the receiver node, in which case, function $ReceiverNewEL()$ avoids that receiver node to be polled again immediately. Sender-initiated pollings are more distributed in the case of *GR.batch* and thus the lower receiver thrashing when compared to *SK.single*. As discussed before, the mean batch size of *GR.batch* at low system load is very close to 1. This implies that the probability that the particular receiver becomes saturated is low. This explains why receiver thrashing of *GR.batch* at low system load is still relatively high, though much lower than that of *SK.single*.

The decreasing receiver thrashing as exhibited by the two algorithms can be attributed to two reasons: (1) As system load increases, the number of sender nodes in the system increases. More receiver nodes are needed to serve the sender nodes. By its definition, receiver thrashing obviously decreases. (2) As system load increases, the spare capacity of those potential receivers diminishes, while the amount of surplus tasks in senders increases. The probability that a receiver becomes saturated and thus a sender node has to search for another transfer partner grows. Sender-initiated pollings are more distributed and hence the decreasing receiver thrashing.

With *SK.single*, the system becomes saturated beyond system load 0.8. The adaptive location policy empties the *RLists* and thus sender-initiated pollings are avoided. Further increase in system load worsens the already congested sender nodes. However, the amount of sender-initiated pollings is not affected. This explains the stable receiver thrashing after system saturation at 0.8. With *GR.batch*, receiver thrashing decreases with increasing system load until it reaches a minimum at 0.62. The increasing receiver thrashing beyond this point can be explained by the reduced number of potential receivers as system load increases. As in the case of *SK.single*, after the system becomes saturated, sender-initiated pollings are avoided and thus the stable receiver thrashing.

Observation Five — *GR.batch* **and** *SK.single* **exhibit a growing sender thrashing at low system load.**

Figure 5.8(h) shows the variation of sender thrashing coefficient. At low system load (below 0.4 in *GR.batch* and below 0.5 in *SK.single*), both algorithms exhibit a growing sender thrashing as system load increases. At low system load, the amount of surplus tasks in sender nodes is small. The probability that a sender node becomes normally loaded after transferring a task(s) to its receiver is high. Since the new load state (real load in *SK.single*; effective load in *GR.batch*) of the sender node is piggybacked on the task transfer messages, the receiver is likely to move the sender node from its *SList* to the head of its *NList*, thus avoiding the node from being polled again immediately. Instead, another node will be selected as the target. Receiver-initiated pollings are therefore

highly distributed at low system load, thus the low sender thrashing at low system load. As system load increases, the probability that a sender node remains in H-load state after task transfer grows. A receiver is therefore more likely to be bound to a particular sender. Receiver-initiated pollings become less distributed, thus the growing sender thrashing as system load increases.

Observation Six — *GR.batch* **exhibits a fast decreasing sender thrashing at medium to high system load.**

Beyond system load 0.8, *SK.single* exhibits a relatively high sender thrashing ($V_s \approx 2.9$), which decreases very gradually. This can be explained as follows. With *SK.single*, the system becomes saturated beyond system load 0.8. The amount of surplus tasks in a sender is large enough to bind a particular receiver. The number of receivers in the system becomes small. Receiver-initiated pollings are therefore targeted to only a few sender nodes, thus the high sender thrashing. Further increase in system load worsens the already congested sender nodes, without significant effect on the dispersiveness of receiver-initiated pollings. This explains the stable sender thrashing at high system load, in the case of *SK.single*.

Beyond system load 0.4, sender thrashing of *GR.batch* is always lower than that of *SK.single*. This can be explained by the fact that with *GR.batch*, a batch of tasks are removed from a sender during each successful polling session. The probability that the sender becomes normally loaded afterwards is therefore higher than that of *SK.single*, which allows only single task transfer. As explained before, the piggybacked new load state of the sender node allows the receiver to select another node for polling, thus the lower sender thrashing when compared to *SK.single*.

5.4 Discussions

Simulations reveal that the *GR.batch* algorithm provides significant performance improvement at high system loads because the CPU capacity is more fully utilized. This can be attributed to the reduced processor thrashing exhibited by the algorithm. The algorithm

also exhibits performance advantage for a highly imbalanced system as batch assignment can smooth out workload imbalance quickly when compared to the single task assignment approach. Less polling sessions are needed for transferring the same number of tasks and therefore channel overhead is reduced.

Chapter 6

Applying Batch Assignment to Systems with Bursty Task Arrival Patterns

Most existing heuristic-based load distribution algorithms rely on the assumption that the workload arrival pattern in a distributed system is rather stable. Such algorithms cannot provide satisfactory performance when the system is injected with bursty workload patterns. This is because congestions in those processing nodes subjected with bursty task arrivals cannot be resolved efficiently. This results in at least four adverse effects:

1. The total system throughput is limited because tasks in the bursty processing nodes are not redistributed efficiently, resulting in wasted processing capacity in those potential receiver nodes.

2. The mean task response time of the bursty processing nodes will be exceedingly high because tasks have to wait for a very long time before they are either processed locally or being assigned to a remote node.

3. The standard deviation of task response time of the whole system will be undesirably large. This is a direct result of (2) above. This also implies that the *predictability* and *fairness* of the system are poor.

The content of this chapter has been published in *Proceedings, Thirteenth IASTED International Conference on Applied Informatics*, February, 1995, Austria [LL95b].

4. To resolve congestions in those bursty nodes, many negotiation sessions are necessary between the bursty nodes and the potential receivers. This injects extra overhead to both the bursty nodes and the communication channel, and thus further worsens the poor situation of the bursty nodes.

The usability of such algorithms is therefore limited to systems with stable workload pattern. In this chapter, we attempt to apply the *GR.batch* algorithm proposed in the last chapter to systems subjected with bursty task arrival patterns.

6.1 Bursty Workload Pattern Characterization Model

We characterize the bursty workload pattern of a system by using a 4-tuple, $(\tau, \alpha, \beta, \gamma)$, where τ is the number of nodes exhibiting bursty task arrivals. The other three components are defined below.

Burst Frequency, α,: is defined as the reciprocal of the *inter-burst period*, which is the mean time between successive burst arrivals.

Burst Amplitude, β,: is the number of tasks arrived locally to a bursty processing node per unit time during a task arrival burst.

Burst Duration, γ,: is the duration of a task arrival burst.

This 4-tuple characterization of bursty workload pattern is depicted in Figure 6.1.

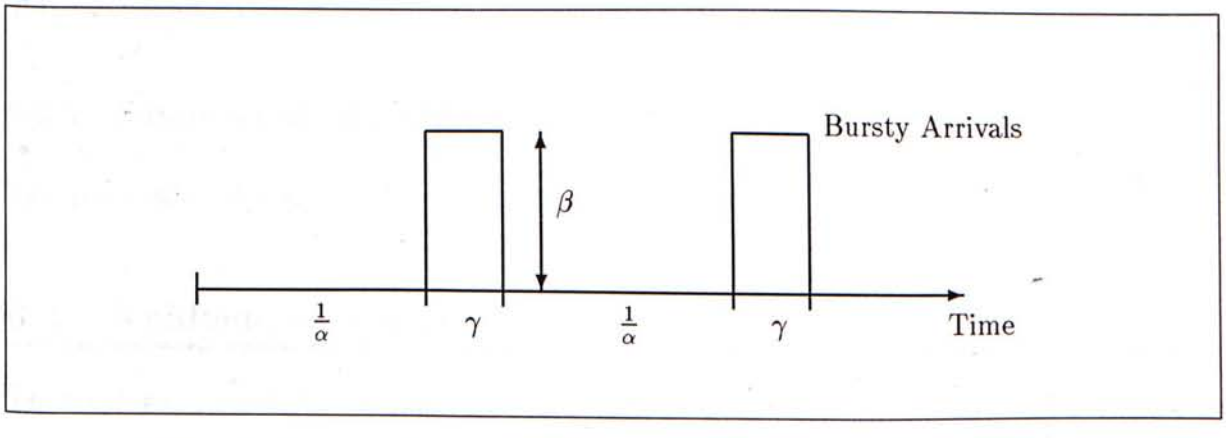


Figure 6.1: Characterization of bursty workload pattern by 4-tuple $(\tau, \alpha, \beta, \gamma)$.

6.2 Algorithm Descriptions

The algorithms that we compare are the *GR.batch* and *SK.single* as described in the previous chapter. For completeness, we briefly mention the features of these two algorithms below.

6.2.1 The *GR.batch* Algorithm

In *GR.batch*, task assignment is the sole workload distribution mechanism. The algorithm employs the batch transfer approach and the GR protocol for obtaining mutual agreement on batch size. Tightly coupled with the GR protocol is an adaptive location policy which selects target nodes for polling and which maintains load information of other nodes locally. The complete *GR.batch* algorithm has been described in section 5.1.4 on page 62.

6.2.2 The *SK.single* Algorithm

The second algorithm, which is used as a reference for comparison, is the *SK.single* algorithm. It allows only single task transfer, meaning that only one task can be transferred during each sender-receiver negotiation session. Therefore, the GR protocol, and the node attributes GUR_i and RES_i, are no longer necessary. Measurement of load state of node P_i is based on the actual number of tasks residing in it. The location policy and negotiation protocol are the same as the symmetrically adaptive location policy proposed by Shivaratri and Krueger in [SK90].

6.2.3 Summary of Algorithm Properties

The properties of the two algorithms are summarized in Table 6.1.

6.3 Analysis of Simulation Results

The performance of the two algorithms are studied by simulations. Table 6.2 shows the values of the simulation parameters used. Figures 6.2 to 6.4 present the performance comparisons of the two algorithms.

Table 6.1: Summary of properties of *GR.batch* and *SK.single*

Algorithm	*Location Policy*	*Load Measurement*	*Transfer Mode*	*GR Protocol*
GR.batch	Shivaratri and Krueger's Location Policy modified	Based on effective load, EL_i. Refer to table 5.1 on page 59.	A batch of tasks for each transfer session	Used
SK.single	Shivaratri and Krueger's Location Policy unmodified	Based on actual number of tasks residing in a node, K_i. Refer to table 4.1 on page 38.	Single task for each transfer session	Not used

Table 6.2: Values of simulation parameters used in the simulations presented in Figure 6.2 to 6.4.

Parameter	*Value*
Q_o	30
T_{CPU}	0.2
N	30
σ_λ	2.0
S_o	1
lower_threshold	5
upper_threshold	$2/3 * Q_o = 20$
probe_limit	5

Parameter	*Value*
$CPU_{polling}$	0.005
$F_{polling}$	0.001
F_{task}	0.005
D	0.01
C_{pack}	0.003
C_{assign}	0.002
l_{assign}	5

Parameter	*Value*
α	0.01
β	*variable*
γ	1
τ	*variable*

6.3.1 Performance Comparison

1. Figure 6.2 shows the mean task response times of the two algorithms with different burst amplitudes. When the system is lightly loaded, *GR.batch* provides performance advantage over *SK.single* at burst amplitudes 300 and 500. It has poorer performance than *SK.single* when the bursty amplitude is 100 however. These show that *GR.batch* provides performance improvement for a lightly loaded system with high enough burst amplitude. This can be attributed to the *GR.batch*'s ability in smoothing out the congestions at those processing nodes subjected with bursty task arrivals. When the burst amplitude is small, congestions are not serious and the extra overhead introduced by the batch assignment approach and the GR protocol cannot be justified by the gain in batch transfers.

   When the system is moderately loaded, *GR.batch* however does not have any performance advantage over *SK.single*. For small burst amplitude ($\beta = 100$), the performance of the two algorithms are very close. For larger burst amplitudes, the system is essentially saturated. There is no point in applying the batch assignment approach because at most time the batch size is close to one. The extra overhead of the batch assignment approach and the GR protocol account for the poorer performance of *GR.batch*.

   When the system is heavily loaded, the two algorithms give essentially identical performance result. This is because the system is highly saturated and thus neither *GR.batch* nor *SK.single* can improve the situation.

2. Figure 6.3 shows the task response time standard deviations of the two algorithms with different burst amplitudes. When the system is lightly loaded, *GR.batch* provides performance advantage over *SK.single*, regardless the magnitude of the burst amplitude. This can be explained by the fact that with *GR.batch*, tasks residing in those congested nodes are quickly relocated to other processing nodes and get processed. Their queueing time is therefore significantly reduced. This in turn results in the smaller response time standard deviation because queueing time is a major

component of task response time. We say that the predictability of the system is highly improved.

An important observation from Figure 6.3(a) is that task response time standard deviation can also be improved even though the burst amplitude is relatively small.

Again, when the system becomes saturated, neither *GR.batch* and *SK.single* can improve the situation. This results in the identical performance of the two algorithms at heavily loaded situations.

3. As in the case of response time standard deviation, queue length standard deviation is highly reduced because congestions in the task queue of those bursty nodes are removed. Thus queue length standard deviation has a similar trend as in the case of response time standard deviation. This is shown in Figure 6.4

6.3.2 Time Trace

Figure 6.5 shows the time trace of the mean task response time of a processing node (P_1) subjected with bursty task arrivals. It can be seen that as τ increases, the difference between the performance of the two algorithms grows. When τ equals 8, the response time associated with *SK.single* grows continuously with time, whereas the response time associated with *GR.batch* remains stable. This can be explained as follows. With *SK.single*, congestions in those bursty nodes are not resolved quick enough. The length of the task queue therefore grows without bound as shown in Figure 6.6(c). This results in the continuously increasing response time. With *GR.batch*, congestions in those bursty nodes are resolved efficiently by the batch transfer approach and the task queue length remains stable.

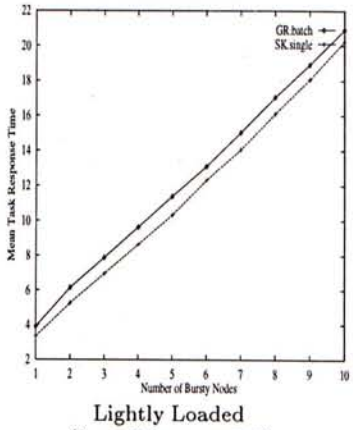
6.4 Discussions

The key findings of the performance characteristics of *GR.batch* when applied to a system subjected with bursty task arrivals are:

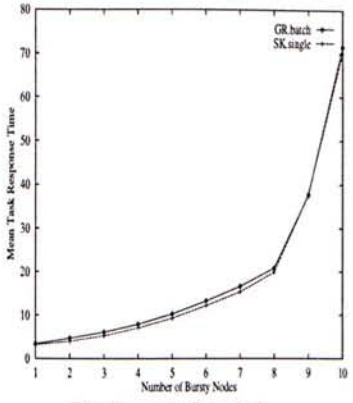
1. In terms of mean task response time, *GR.batch* provides significant improvement for a non-saturated system with large enough burst amplitude.

2. In terms of task response time standard deviation and queue length standard deviation, *GR.batch* shows significant improvement for a non-saturated system, regardless the burst amplitude. In other words, the system predictability and fairness of service are improved.

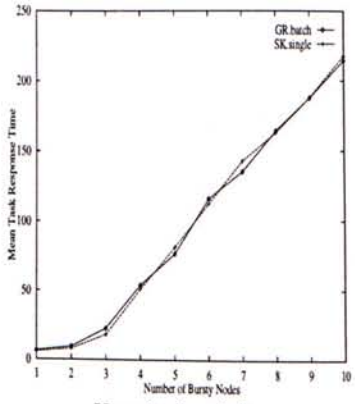
3. *GR.batch* ensures stable task response time and queue length for a system subjected with bursty task arrivals. With single task transfer, these two performance metrics grow continuously with time because congestions are not resolved efficiently.



Lightly Loaded
($\lambda_o = 0.2$, $\sigma_\lambda = 2.0$)

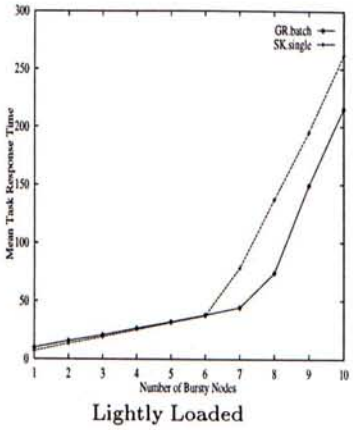


Moderately Loaded
($\lambda_o = 0.7$, $\sigma_\lambda = 2.0$)

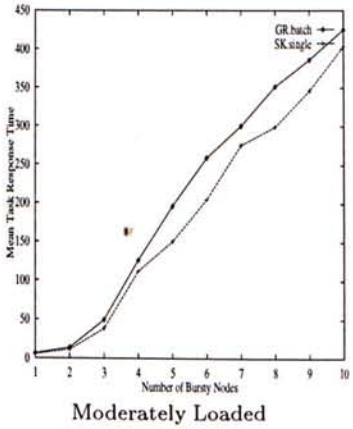


Heavily Loaded
($\lambda_o = 0.9$, $\sigma_\lambda = 2.0$)

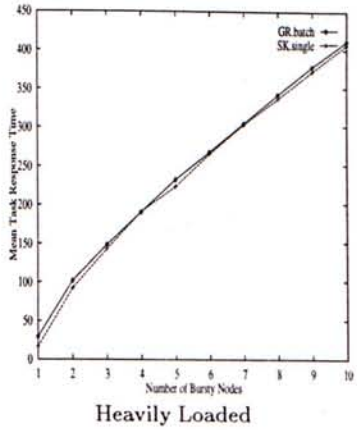
(a) $\beta = 100$



Lightly Loaded

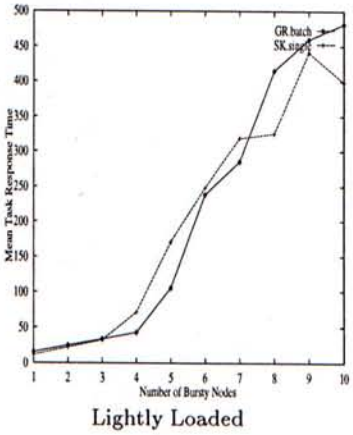


Moderately Loaded

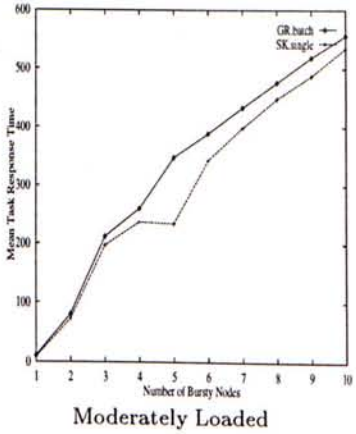


Heavily Loaded

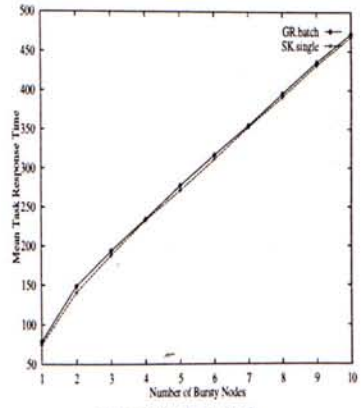
(b) $\beta = 300$



Lightly Loaded



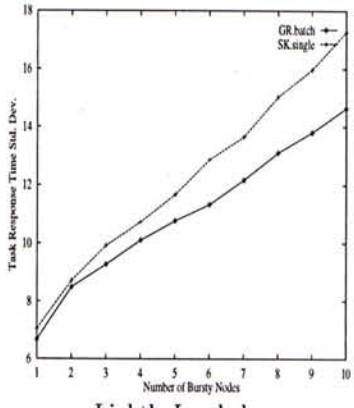
Moderately Loaded



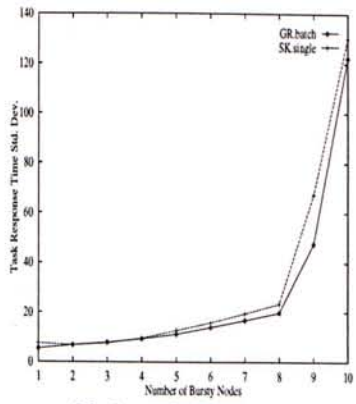
Heavily Loaded

(c) $\beta = 500$

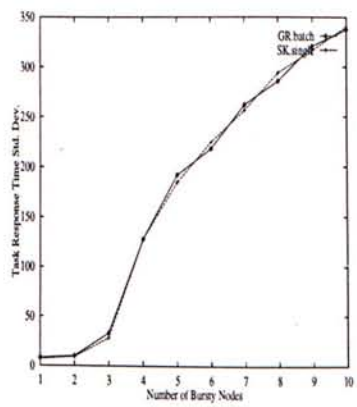
Figure 6.2: Task response time of *GR.batch* and *SK.single* in systems subjected with bursty workload arrivals. Simulation parameters used are shown in Table 6.2 on page 78.



Lightly Loaded
($\lambda_o = 0.2$, $\sigma_\lambda = 2.0$)

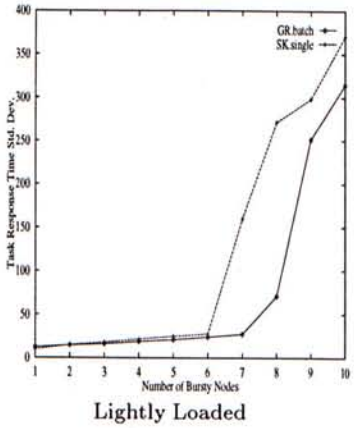


Moderately Loaded
($\lambda_o = 0.7$, $\sigma_\lambda = 2.0$)

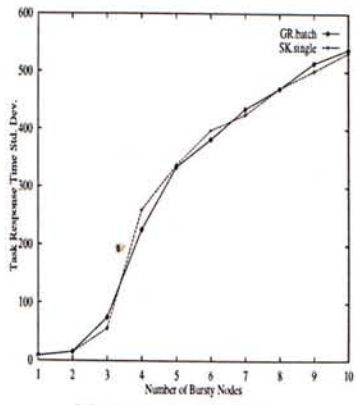


Heavily Loaded
($\lambda_o = 0.9$, $\sigma_\lambda = 2.0$)

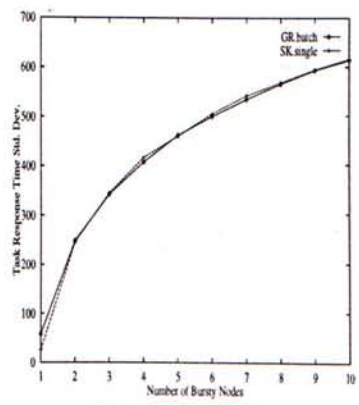
(a) $\beta = 100$



Lightly Loaded

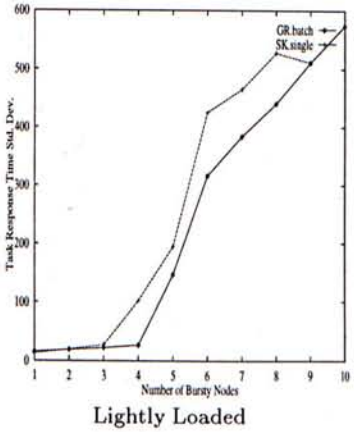


Moderately Loaded

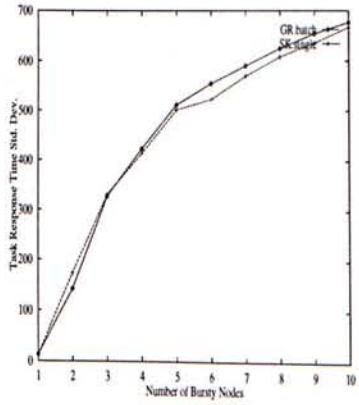


Heavily Loaded

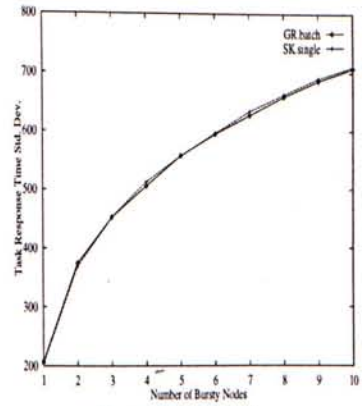
(b) $\beta = 200$



Lightly Loaded



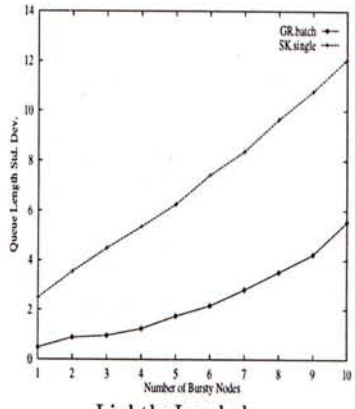
Moderately Loaded



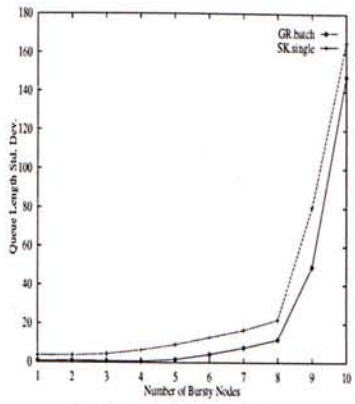
Heavily Loaded

(c) $\beta = 300$

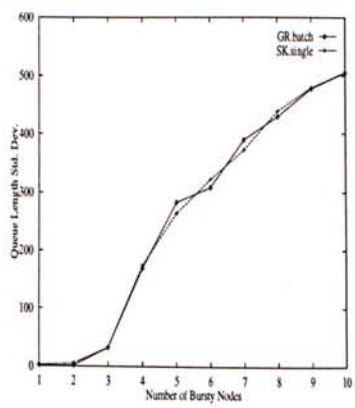
Figure 6.3: Task response time standard deviation of *GR.batch* and *SK.single* in systems subjected with bursty workload arrivals. Simulation parameters used are shown in Table 6.2 on page 78.



Lightly Loaded
($\lambda_o = 0.2$, $\sigma_\lambda = 2.0$)

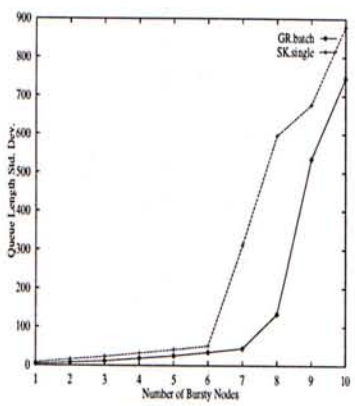


Moderately Loaded
($\lambda_o = 0.7$, $\sigma_\lambda = 2.0$)

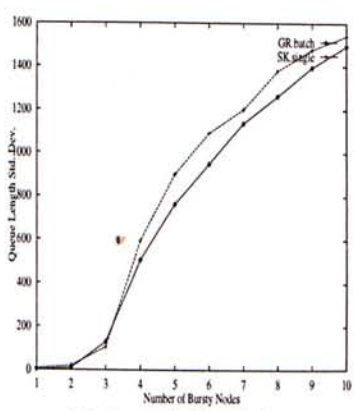


Heavily Loaded
($\lambda_o = 0.9$, $\sigma_\lambda = 2.0$)

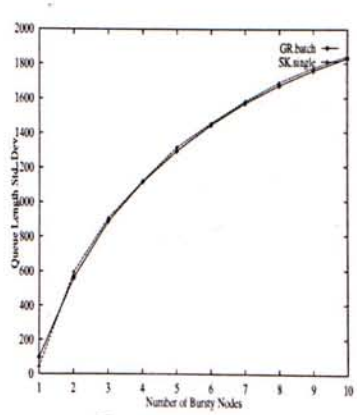
(a) $\beta = 100$



Lightly Loaded

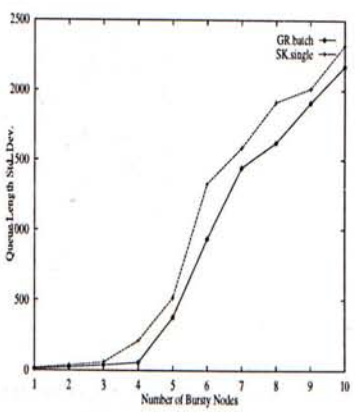


Moderately Loaded

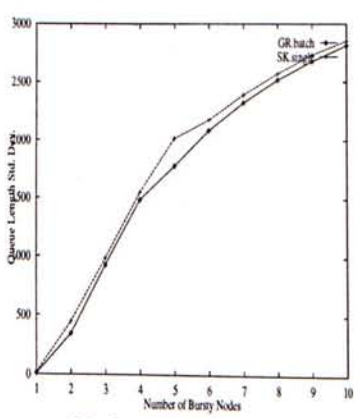


Heavily Loaded

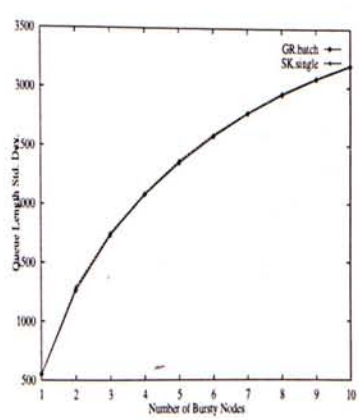
(b) $\beta = 200$



Lightly Loaded



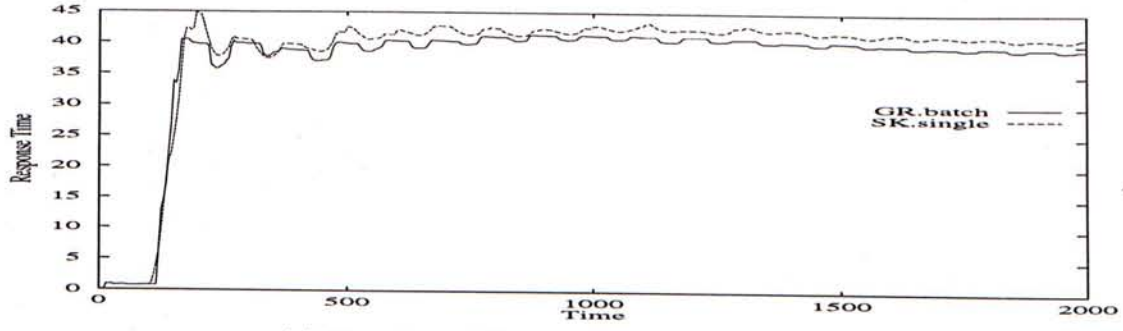
Moderately Loaded



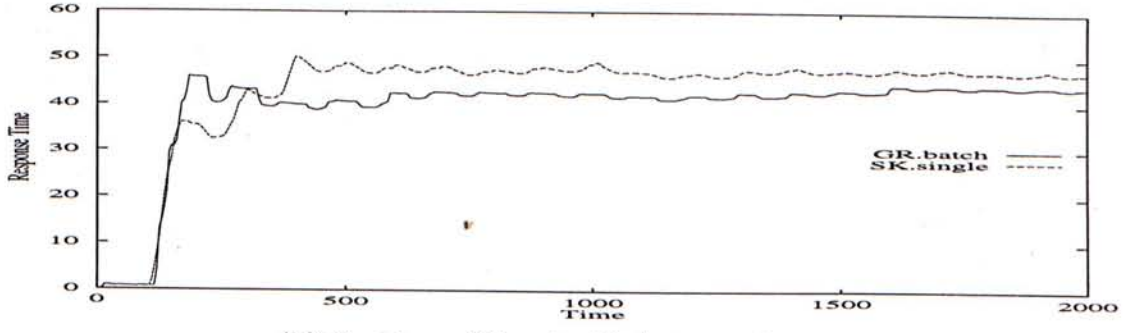
Heavily Loaded

(c) $\beta = 300$

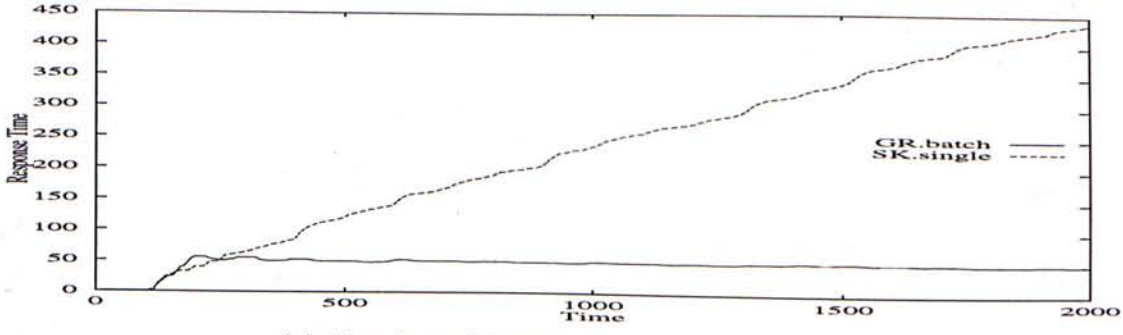
Figure 6.4: Queue length standard deviation of *GR.batch* and *SK.single* in systems subjected with bursty workload arrivals. Simulation parameters used are shown in Table 6.2 on page 78.



(a) Number of Bursty Nodes, $\tau = 5$

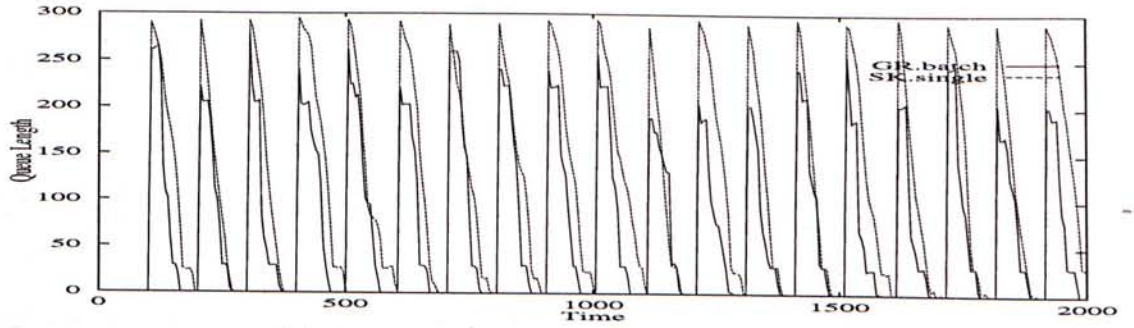


(b) Number of Bursty Nodes, $\tau = 6$

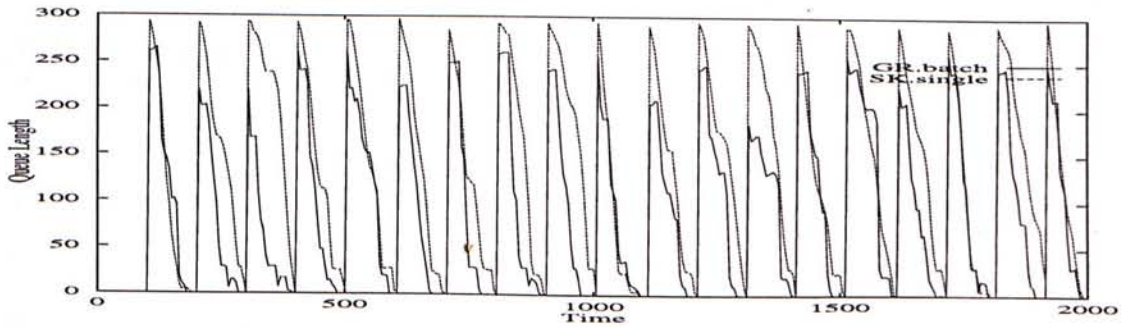


(c) Number of Bursty Nodes, $\tau = 8$

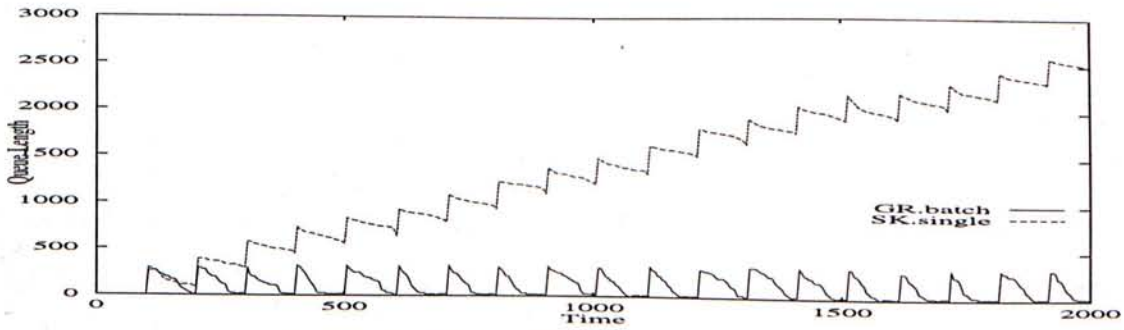
Figure 6.5: Trace of node P_1's mean task response time — *GR.batch* and *SK.single* in systems subjected with bursty workload arrivals. Simulation parameters used are shown in Table 6.2 on page 78, except the following: $\alpha = 0.01$, $\beta = 300$, $\gamma = 1$.



(a) Number of Bursty Nodes, $\tau = 5$



(b) Number of Bursty Nodes, $\tau = 6$



(c) Number of Bursty Nodes, $\tau = 8$

Figure 6.6: Trace of node P_1's queue length — *GR.batch* and *SK.single* in systems subjected with bursty workload arrivals. Simulation parameters used are shown in Table 6.2 on page 78, except the following: $\alpha = 0.01$, $\beta = 300$, $\gamma = 1$.

Chapter 7

A Preliminary Study on Task Assignment Augmented with Migration

The two most commonly used mechanisms in task relocations are task assignment and task migration. Our primary objective in the study presented in this chapter is to investigate situations where migration can augment assignment to provide extra performance improvement. We present a performance study on three different load balancing algorithms. All of these algorithms use the same information and location policy. They differ only in the transfer policy, task selection in particular. The first algorithm (A) employs only task assignment, whereas the second (AM) and the third (AMT) allow both task assignment and migration. The third algorithm differs from the second in that a timer is used for initiating the load balancing algorithm, in addition to the usual event triggering by task arrivals and task completions. Single task transfer is assumed by all these algorithms.

7.1 Algorithm Descriptions

This section presents the information, transfer, and location policies used in the load

The content of this chapter has been published in *Proceedings, IEEE TENCON 1994*, pages 357-364, August, 1994 [LL94].

balancing algorithms. Recall that the three algorithms (A, AM, and AMT) share the same information and location policies.

7.1.1 Information Policy

Again, load information content exchanged between processing nodes is based on the 3-level load measurement scheme. The definitions of load states are given in Table 7.1. Note that *lower_threshold* is defined as $\frac{1}{3}Q_o$, while *upper_threshold* is defined as $\frac{2}{3}Q_o$. Again, sender-initiated negotiation sessions start whenever a local task arrival triggers its arrival node into the H-load state. Receiver-initiated negotiation sessions start if a task completion puts that node in L-load state.

Table 7.1: The 3-level load measurement scheme used in algorithms A, AM, and AMT. K_i is the number of tasks residing in node P_i.

Load State	*Criteria*
L-load	$K_i \leq \frac{1}{3}Q_o$
N-load	$\frac{1}{3}Q_o < K_i \leq \frac{2}{3}Q_o$
H-load	$K_i > \frac{2}{3}Q_o$

7.1.2 Location Policy

Again, the location policy is based on the adaptive symmetrically-initiated location policy proposed by Shivaratri and Krueger [SK90]. Note however that since only single task transfer is allowed, the GR protocol is not necessary.

7.1.3 Transfer Policy

There are two components in a transfer policy: (1) algorithm initiation scheme; and (2) task selection scheme. The algorithm initiation schemes of the three algorithms are identical — task relocation is needed whenever a node is either in H-load or in L-load. Task selection scheme is different among the algorithms however, depending on the allowed

task distribution mechanism (assignment versus migration).

7.1.3.1 Task Selection for Assignment

All the tasks in the task queue of a processing node are candidates for task assignment. These candidate tasks are selected in FIFO discipline by the task selection scheme. That is, we always select the first task in the task queue for remote assignment.

7.1.3.2 Task Selection for Migration

Tasks in the service queue are candidates for migration if the following two criteria are satisfied:

1. The candidate task is locally assigned.

2. The estimated remaining execution time of the candidate task is larger than the estimated overhead (measured in time) if it is migrated. The remaining execution time of the candidate task p, denoted as RET_p, with arrival node P_j and accumulated execution time, AET_p, is estimated as follows.

$$RET_p = S_j - AET_p \tag{7.1}$$

where S_j is the mean task service time of node P_j. The correct selection of a task for migration depends largely on the accuracy of the estimation of RET_p.

7.1.4 The Three Load Balancing Algorithms

- **Algorithm** A —

  This algorithm employs only task assignment but not migration.

- **Algorithm** AM —

  This algorithm allows both task assignment and migration. Task assignment has precedence over migration. That is, migration takes place only if the sender node finds no appropriate task for assignment.

- **Algorithm** *AMT* —

  This algorithm is identical to algorithm *AM* except that a *receiver-timeout mechanism* is used. The mechanism ensures that the receiver-initiated negotiation is invoked whenever a node has been in the L-load state for longer than *receiver-timeout*, which is an algorithm design parameter. This mechanism avoids a potential receiver from remaining idle for a prolonged period without searching for a task to receive.

7.2 Simulations and Analysis of Results

We divided our experiments into two different cases:

1. **Even Task Service Time** —

   All the processing nodes in the distributed system have the same mean task service time requirement, that is $S_i = 1$, $i \in \{1, 2, \cdots, N\}$. In other words, all the tasks throughout the whole system logically belong to the same class. Task arrival rates are however characterized by the log normal distribution $(\lambda_o, \sigma_\lambda)$, as described in section 3.1 on page 19.

2. **Uneven Task Service Time** —

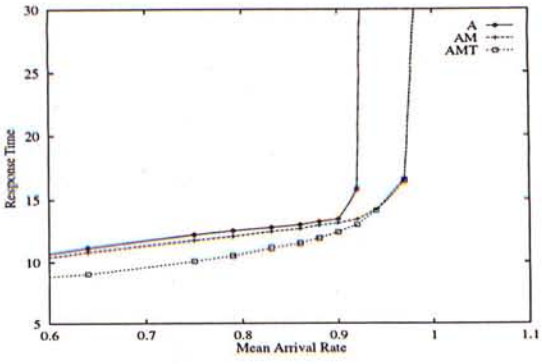
   Processing nodes are subjected to two different classes of tasks, one with longer task service time requirement, and the other with normal task service time requirement. Moreover, task arrival rates between the nodes may be different.

7.2.1 Even Task Service Time

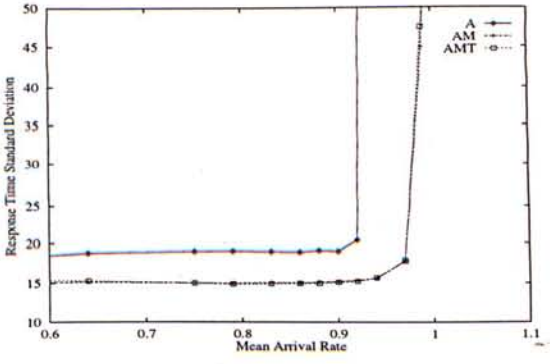
Table 7.2 shows the simulation parameters used in the simulation study. Figure 7.1 shows the comparisons of performance of the three load balancing algorithms.

7.2.1.1 Primary Performance Comparisons

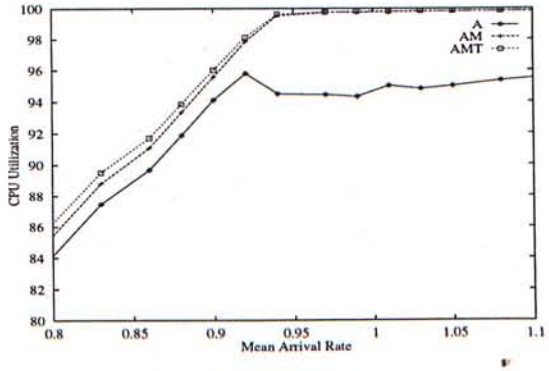
1. Figure 7.1(a) shows that both *AM* and *AMT* perform better than algorithm *A* under the whole range of system load. The difference is more significant when the system load is high (around mean arrival rate 0.9 to 1.0). In fact the system with



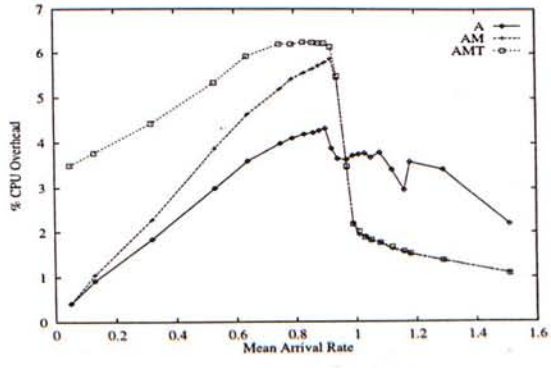
(a) Response Time



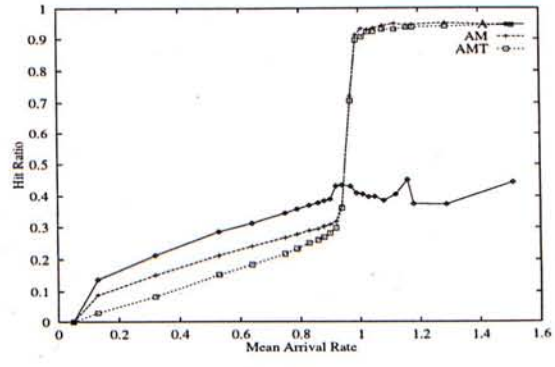
(b) Response Time Standard Deviation



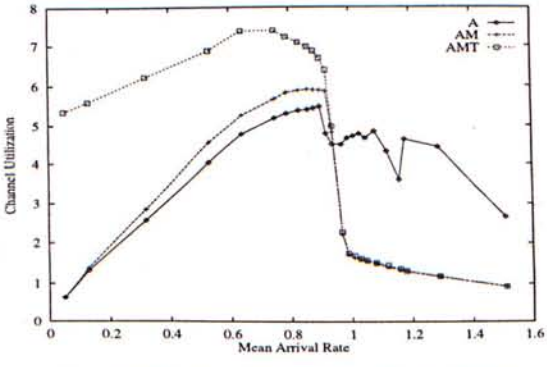
(c) CPU Utilization



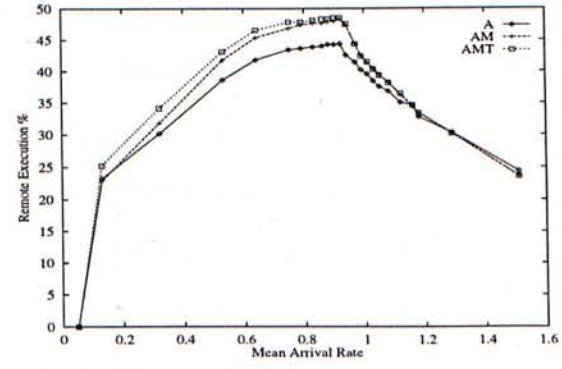
(d) Percentage CPU Overhead



(e) Hit Ratio



(f) Channel Utilization in Terms of Mean Number of Messages in Channel



(g) Remote Execution Percentage

Figure 7.1: Comparison of system performance of algorithms A, AM, and AMT. Simulation parameters used are shown in Table 7.2.

Table 7.2: Values of simulation parameters used in the simulations presented in Figure 4.3 and 4.4.

Parameter	Value
Q_o	30
T_{CPU}	0.2
N	30
σ_λ	2
S_o	1
lower_threshold	$1/3 * Q_o = 10$
upper_threshold	$2/3 * Q_o = 20$
probe_limit	5
receiver-timeout	3

Parameter	Value
$CPU_{polling}$	0.005
$F_{polling}$	0.001
F_{task}	0.005
D	0.01
C_{pack}	0.003
C_{assign}	0.002
$C_{migrate}$	0.05
l_{assign}	5
$l_{migrate}$	7

either AM or AMT becomes saturated at higher system load (≈ 1) when compared with algorithm A (≈ 0.92). The primary reason behind this observation is that the available processing power of the distributed system is more fully utilized with AM and AMT. This is shown in Figure 7.1(c) as higher mean CPU utilizations. This can be explained by the fact that task migrations provide an alternate mechanism for task relocations when a busy node has no appropriate fresh task for assignment.

2. A close examination of Figure 7.1(a) reviews that algorithm AMT performs better than AM at low to medium system load. This difference diminishes when the system load becomes high. This can be explained as follows. At low system load, the majority of nodes are lightly loaded. The time period between two successive task completions in a lightly loaded node may be very long. With algorithm AM, receiver-initiated searching therefore occurs infrequently. A lightly loaded node may remain idle or nearly idle for a long time. The processing power of the node is wasted. With algorithm AMT, such waste of processing power is avoided by the receiver-timeout mechanism. This is shown in Figure 7.1(c) where AMT has a higher CPU utilization, and in Figure 7.1(g) a higher remote execution percentage. At high system load, the receiver-timeout mechanism has no need to trigger extra receiver-initiated task transfers because the probability that the *receiver-timeout*

period expires is small. This accounts for the identical performance of algorithms *AM* and *AMT* at high system load.

3. Figure 7.1(b) shows that both algorithms *AM* and *AMT* have significantly lower response time standard deviation when compared with algorithm *A*. This means that *AM* and *AMT* provides fairer services.

4. Figure 7.1(d) shows a comparison of the percentage CPU overhead between the three algorithms. Algorithms *AM* and *AMT* have higher CPU overhead before the system is saturated at about 0.9. There are two reasons for this. The first reason is that because of task migrations, *AM* and *AMT* have larger number of tasks relocations, which impose non-negligible CPU overhead. This is shown in Figure 7.1(g) as a higher remote execution percentage. Another reason is the smaller hit ratio which means a larger portion of pollings have been failed, Figure 7.1(e). These two reasons also account for the higher channel utilization of *AM* and *AMT* as shown in Figure 7.1(f). When the system is saturated, the CPU overhead imposed by algorithm *A* becomes the highest among the three. This is because algorithm *A* has a lower hit ratio at high system load. More polling sessions are introduced until the probe-limit is reached or a complementary node is found. This is reflected in the higher channel utilization.

5. Figure 7.1(f) shows that all the three algorithms have low channel utilization at low system load. The channel utilizations grow steadily with the system load until a peak where task relocations occur most frequently. After the peak, the channel utilizations steadily drop with increasing system load. This reflects the fact that all of the three algorithms adapt itself to the system load.

   At low system load, most of the nodes are potential receivers and few are senders. While a sender has no problem in locating a receiver, most of the receiver-initiated pollings are failed. This may not have an adverse effect on the system performance however because there is spare processing capacity to cope with the extra overhead.

Also, the failed receiver-initiated pollings have the positive effect of updating the *RLists* of the polled nodes.

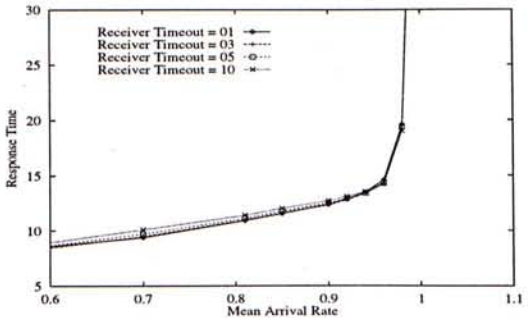
At high system load, most of the nodes are potential senders and few are potential receivers. An initial high rate of failure of sender-initiated pollings results in the removal of entries from *RLists*. Eventually sender-initiated pollings are prevented because there is no entry in *RLists*.

7.2.1.2 Effect of *receiver-timeout*

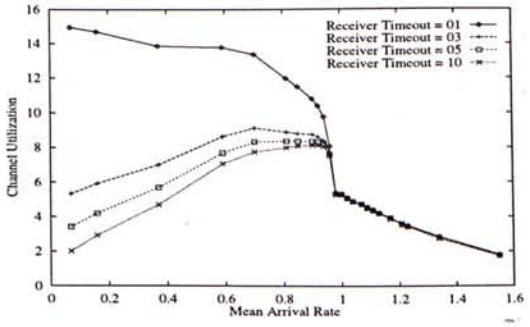
Figure 7.2 shows the effect of *receiver-timeout* on the performance of algorithm *AMT*. It can be seen that a small *receiver-timeout* value does not result in significantly better response time than a large value. In fact, an exceedingly small value (in our case 1) may result in poorer system performance as shown in Figure 7.2(a). The effect of *receiver-timeout* on channel utilization is significant at low system load however, as shown in Figure 7.2(b). This can be explained as follows. At low system load, the probability that a receiver finds a sender successfully is low. A small *receiver-timeout* value results in frequent polling sessions. Most of these pollings fail to locate a sender and cause many unnecessary polling messages to be injected into the network. This also causes extra CPU overhead. The difference diminishs at high system load because the effect of the receiver-timeout mechanism is lost at high system load. From Figure 7.2, the optimal *receiver-timeout* is 3.

7.2.2 Uneven Task Service Time

Simulation results presented in the previous section assume that all processing nodes are subjected with tasks having identical mean service time requirement. However, we are also interested in situations where a few nodes generate "long" tasks while the rest generate "normal" tasks. We therefore run a number of simulations to study the system performance under such situations. To do so, we define six different node types to characterize the workload pattern of a node. The processing node type definitions are



(a) Response Time



(b) Channel Utilization in Terms of Mean Number of Messages in Channel

Figure 7.2: Effect of *receiver-timeout* on performance of *AMT*. Simulation parameters used identical to those shown in Table 7.2 on page 92, except that *receiver-timeout* is now a variable.

given in Table 7.3. Each node type is defined by its mean task service time and its mean task arrival rate. For example, a node which generates long tasks in low rate is denotes as "Long-Low", a node which generates normal tasks in medium rate is denoted as "Normal-Medium", and so on.

Table 7.3: Processing node type definitions for modeling a system with uneven task service time requirements.

Node Type	*Task Service Time*	*Arrival Rate*
Long-Low	10	0.20
Long-Medium	10	0.60
Long-High	10	0.95
Normal-Low	1	0.20
Normal-Medium	1	0.60
Normal-High	1	0.95

Table 7.4 defines different system types by specifying the component node types. In all system types defined, there are five nodes that generate long tasks and 25 nodes that generate normal tasks. We label the system type by specifying the arrival rate of these two kinds of nodes. For example, a system in which the long task nodes have low arrival rate and the normal task nodes have medium arrival rate is labeled as LM; a system in

Table 7.4: System type definitions for modeling a system with uneven task service time.

System Type	*Node Type*	*Number*
LL	Long-Low	5
	Normal-Low	25
LM	Long-Low	5
	Normal-Medium	25
LH	Long-Low	5
	Normal-High	25
ML	Long-Medium	5
	Normal-Low	25
MM	Long-Medium	5
	Normal-Medium	25
MH	Long-Medium	5
	Normal-High	25
HL	Long-High	5
	Normal-Low	25
HM	Long-High	5
	Normal-Medium	25
HH	Long-High	5
	Normal-High	25

which both types of nodes have high arrival rate is labeled as HH; and so on.

Tables 7.6–7.8 show that the performance of algorithm AM (but not AMT) is marginally better than that of algorithm A for a system consisting of a few nodes that generate long task in low arrival rate, whereas other nodes have low to medium load. Table 7.9 shows that algorithms AM and AMT have a performance improvement of about 24% over algorithm A. Table 7.10 shows an even larger performance improvement of about 30%. From these, we can conclude that for a system which consists of a few long task nodes at low to medium load (while the other nodes are not heavily loaded), algorithm AM has performance advantage over algorithm A. This can be explained as follows. In system MM for example, the probability that the five Long-Low nodes become heavily loaded is high. This is because it takes a relatively long time to finish a long task. These heavily loaded nodes do not have appropriate candidates for assignment because their assignment queue may be empty in due course. With algorithm A, potential receivers have no way to share their surplus workload. With algorithms AM or AMT, the workload can be shared by migrating the executing tasks from the Long-Low nodes to the potential receivers. This

Table 7.5: Values of simulation parameters used in the simulations for studying uneven task service time systems.

Parameter	Value
Q_o	30
T_{CPU}	0.2
N	30
	(with different service time and arrival rate combinations)
σ_λ	2
S_o	*variable*
lower_threshold	$1/3 * Q_o = 10$
upper_threshold	$2/3 * Q_o = 20$
probe_limit	5
receivertimeout	3

Parameter	Value
$CPU_{polling}$	0.005
$F_{polling}$	0.001
F_{task}	0.005
D	0.01
C_{pack}	0.003
C_{assign}	0.002
$C_{migrate}$	0.05
l_{assign}	5
$l_{migrate}$	7

accounts for the higher CPU utilization and remote execution percentage. As the workload of normal task nodes increases, the probability that they become a potential receiver diminishes. By the time a receiver-initiated polling arrives at a long task node, the node may have accumulated enough workload that tasks are waiting in its task queue. In such case, assignment takes place rather than migration. This explains why when the normal tasks nodes have high arrival rates, algorithms AM or AMT do not perform better than algorithm A.

Table 7.6: LL system type performance of algorithms A, AM, and AMT. Simulation parameters shown in Table 7.5.

LB Alg.	Response Time	% CPU Util.	% CPU Overhead	% Remote Exec.	% Assignment	% Migration	Hit Ratio	Channel Util.
A	10.77	34.59	1.531	0.599	0.599	~	0.003	2.339
AM	10.64	34.59	1.540	0.609	0.348	0. 260	0.002	2.345
AMT	12.89	36.49	3.467	1.877	1.255	0.622	0.004	5.275

Table 7.7: LM system type performance of algorithms A, AM, and AMT. Simulation parameters shown in Table 7.5.

LB Alg.	Response Time	% CPU Util.	% CPU Overhead	% Remote Exec.	% Assignment	% Migration	Hit Ratio	Channel Util.
A	6.01	69.90	3.586	0.696	0.696	~	0.004	5.476
AM	5.98	69.93	3.619	0.716	0.503	0.213	0.003	5.511
AMT	6.31	70.84	4.552	0.923	0.668	0.256	0.004	6.929

Table 7.8: LH system type performance of algorithms A, AM, and AMT. Simulation parameters shown in Table 7.5.

LB Alg.	Response Time	% CPU Util.	% CPU Overhead	% Remote Exec.	% Assignment	% Migration	Hit Ratio	Channel Util.
A	10.58	99.51	4.101	1.126	1.126	~	0.009	6.246
AM	10.61	99.54	4.143	1.258	0.953	0.304	0.008	6.270
AMT	11.27	99.64	4.234	1.312	0.957	0.355	0.008	6.402

Table 7.9: ML system type performance of algorithms A, AM, and AMT. Simulation parameters shown in Table 7.5.

LB Alg.	Response Time	% CPU Util.	% CPU Overhead	% Remote Exec.	% Assignment	% Migration	Hit Ratio	Channel Util.
A	23.33	68.20	2.183	26.533	26.533	~	0.130	3.130
AM	17.72	68.78	2.638	27.971	17.386	10.584	0.099	3.362
AMT	17.33	70.26	4.104	28.668	20.460	8.208	0.068	5.702

Table 7.10: MM system type performance of algorithms A, AM, and AMT. Simulation parameters shown in Table 7.5.

LB Alg.	Response Time	% CPU Util.	% CPU Overhead	% Remote Exec.	% Assignment	% Migration	Hit Ratio	Channel Util.
A	40.81	98.39	2.300	10.713	10.713	~	0.104	3.329
AM	28.25	99.42	0.955	11.312	7.041	4.271	0.278	0.847
AMT	28.76	99.41	0.985	11.450	6.961	4.489	0.275	0.861

7.3 Discussions

We found that the algorithms which employ both task assignment and migration perform significantly better than the one which only allows task assignment. We can conclude that although task migration usually costs more than task assignment, under some situations, it can augment task assignment to provide extra performance improvement. This is because task migration provides an alternate mechanism for workload distribution in a distributed system. The performance improvement by using this approach is especially significant when a heavily-loaded node finds no appropriate tasks for assignment. In contrast to the common belief, task assignment augmented with task migration is a promising approach to dynamic load balancing.

Chapter 8

Assignment Augmented with Migration Revisited — Comparing with Batch Assignment

In Chapter 7, we presented a preliminary study on combining task assignment and migration. We found that algorithms which employ both task assignment and migration as the transfer mechanism perform significantly better than algorithms which allow only assignment. However, task migration is costly and not widely supported in today's distributed operating systems. On the other hand, in Chapter 5, we introduced the batch assignment algorithm *GR.batch*. We found that batch assignment provides impressive performance advantage and is promising to be a practical load distribution algorithm. In this chapter, we attempt to compare the performance of these two approaches.

8.1 Algorithm Descriptions

The first algorithm that we study in this chapter is the *GR.batch* algorithm described in section 5.1. For the sake of convenience, we rename it as $GR.BATCH.A$ to signify that this algorithm employs the GR protocol and the batch assignment approach, and that it allows only task assignment. The second algorithm is the AM algorithm described in the last chapter. Similarly, AM is renamed as $SK.SINGLE.AM$ to signify that the algorithm adopts the original Shivaratri and Krueger's location policy, uses the single task

transfer approach, and allows both assignment and migration as the transfer mechanism.

8.1.1 The *GR.BATCH.A* Algorithm

In *GR.BATCH.A*, task assignment is the sole workload distribution mechanism. It employs the batch transfer approach and the GR protocol for obtaining mutual agreement on batch size. Tightly coupled with the GR protocol is an adaptive location policy which selects target nodes for polling and which maintains load information of other nodes locally. Measurement of load state is based on the effective load of a node, denoted as EL_i. The complete *GR.BATCH.A* algorithm has been described in section 5.1.4 on page 62.

8.1.2 The *SK.SINGLE.AM* Algorithm

Unlike *GR.BATCH.A*, algorithm *SK.SINGLE.AM* employs both task assignment and task migration as its workload distribution mechanisms. However, it allows only single task transfer. Therefore, the GR protocol, and the node attributes GUR_i and RES_i, are no longer necessary. Measurement of load state is based on the actual number of tasks residing on a node, denoted as K_i. The location policy and negotiation protocol are the same as the symmetrically-initiated adaptive location policy as described by Shivaratri and Krueger in [SK90].

8.1.3 Summary of Algorithm Properties

In the simulations, an algorithm which uses the original Shivaratri and Krueger's location policy, and which allows only single task assignment is used as a reference for comparing with the *GR.BATCH.A* and *SK.SINGLE.AM* algorithms. We label this algorithm as *SK.SINGLE.A*. The properties of these three algorithms are summarized in Table 8.1.

8.2 Simulations and Analysis of Results

The performance of the algorithms were studied by simulations. Table 8.2 shows the

Table 8.1: Summary of properties of $SK.SINGLE.A$, $SK.SINGLE.AM$ and $GR.BATCH.A$.

Algorithm	*Location Policy*	*Load Measurement*	*Workload Distribution Mechanism*	*Transfer Mode*	*GR Protocol*
SK.SINGLE.A	Shivaratri and Krueger's Location Policy unmodified	Based on actual number of tasks residing in a node, K_i.	Assignment only	Single task for each transfer session	Not used
SK.SINGLE.AM	Shivaratri and Krueger's Location Policy unmodified	Based on actual number of tasks residing in a node, K_i.	Assignment plus Migration	Single task for each transfer session	Not used
GR.BATCH.A	Shivaratri and Krueger's Location Policy modified	Based on effective load, EL_i.	Assignment only	A batch of tasks for each transfer session	Used

simulations parameters used. Figure 8.1 shows the performance of the two algorithms. We have the following observations and analysis on their performance.

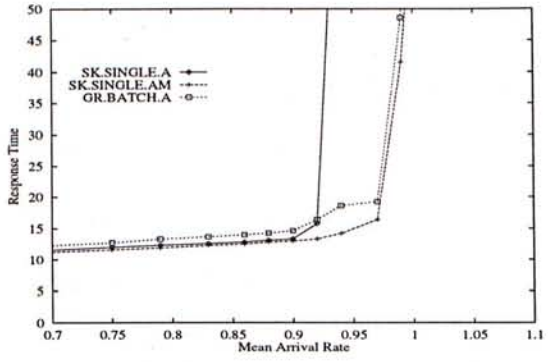
Table 8.2: Values of simulation parameters used in the simulations presented in Figure 8.1.

Parameter	Value
Q_o	30
T_{CPU}	0.2
N	30
σ_λ	3
S_o	1
lower_threshold	$1/3 * Q_o = 10$
upper_threshold	$2/3 * Q_o = 20$
probe_limit	5

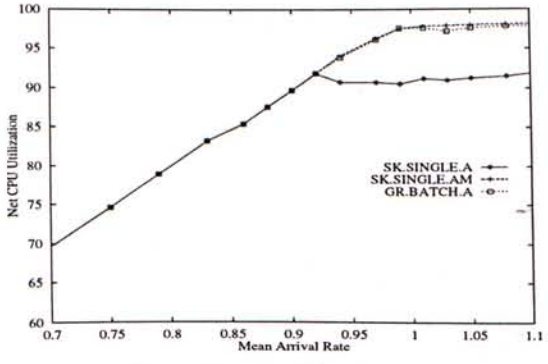
Parameter	Value
$CPU_{polling}$	0.005
$F_{polling}$	0.001
F_{task}	0.005
D	0.01
C_{pack}	0.003
C_{assign}	0.002
$C_{migrate}$	0.05
l_{assign}	5
$l_{migrate}$	7

8.2.1 Performance Comparisons

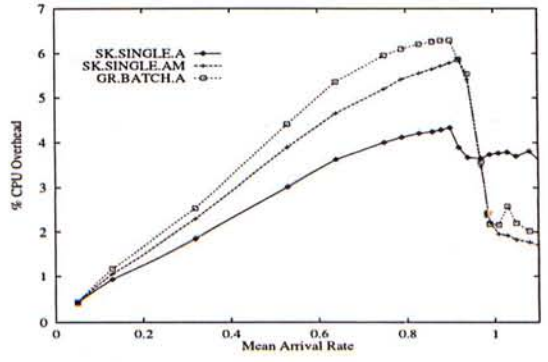
1. From Figure 8.1(a), it can be seen that $SK.SINGLE.AM$ and $GR.BATCH.A$ have comparable performance throughout the whole range of system load. These two algorithms become saturated at system load 0.97. The algorithm $SK.SINGLE.A$ has comparable performance with the other two algorithms up to system load 0.92, after which it becomes saturated.



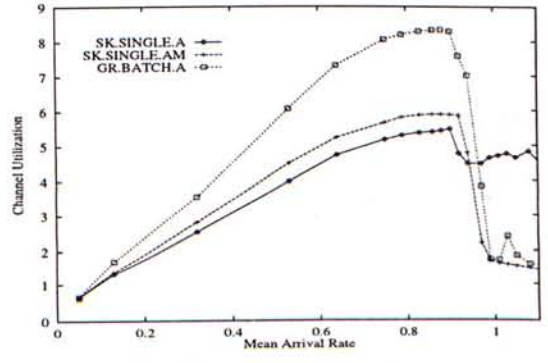
(a) Response Time



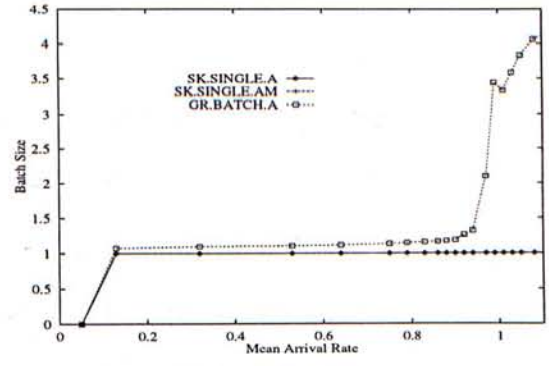
(b) Net CPU Utilization



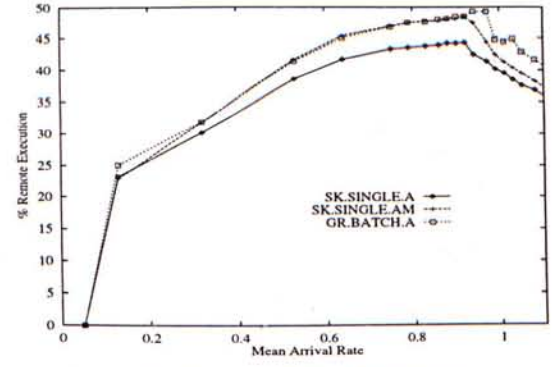
(c) Percentage CPU Overhead



(d) Channel Overhead



(e) Batch Size



(f) Remote Execution Percentage

Figure 8.1: Performance of *GR.BATCH.A*, *SK.SINGLE.AM*, and *SK.SINGLE.A*. Simulation parameters used shown in Table 8.2.

At high system load, both *GR.BATCH.A* and *SK.SINGLE.AM* have a much higher net CPU utilization than *SK.SINGLE.A* (97% versus 92%), as shown in Figure 8.1(b). This means that the system capacity can be more fully utilized with *GR.BATCH.A* and *SK.SINGLE.AM*. This is the primary reason for the performance advantage of these two algorithms over *SK.SINGLE.A* at high system load.

In the case of *SK.SINGLE.A*, since around 8% of the system capacity is wasted, we know that the capacity of some lightly loaded nodes is not used. In other words, the workload in the system is not evenly distributed. This can be attributed to three reasons:

- A sender, after pairing with a potential receiver, may find no appropriate task in its task queue for assignment. Since assignment is the only workload distribution mechanism available in *SK.SINGLE.A*, the sender node has no way to transfer its surplus tasks to the receiver. This is reflected in its smaller remote execution percentage as shown in Figure 8.1(f).
- With the original Shivaratri and Krueger's location policy, a receiver can easily be bound to a particular sender. If so happens that the sender hardly finds an appropriate task for remote assignment to the receiver, the spare capacity of the potential receiver will be wasted. Again, this is reflected in its smaller remote execution percentage as shown in Figure 8.1(f).
- A number of senders may poll a receiver simultaneously. The receiver may become flooded with incoming remote tasks, resulting in a significant degradation of performance. This phenomenon is known as *processor thrashing*, and is further worsened by the fact that a receiver can easily be bound to a particular sender.

2. Figure 8.1(a) shows that for system load below the saturation point of *SK.SINGLE.A*, the performance of *GR.BATCH.A* is slightly poorer than that of *SK.SINGLE.A*

or *SK.SINGLE.AM*. This can be explained as follows. Figure 8.1(e) shows that within this range of system load, the average batch size of *GR.BATCH.A* is very close to 1. This implies that the batch transfer approach does not provide additional performance advantage. However, the GR protocol imposes additional CPU overhead. This is shown in Figure 8.1(c). This extra overhead is accounted for the slightly poorer performance of *GR.BATCH.A* at low system load.

3. Figure 8.1(d) shows the CPU overhead of the algorithms. It can be seen that at low system load, the CPU overhead of *GR.BATCH.A* and *SK.SINGLE.AM* is higher than that of *SK.SINGLE.A*. At high system load, the reverse occurs.

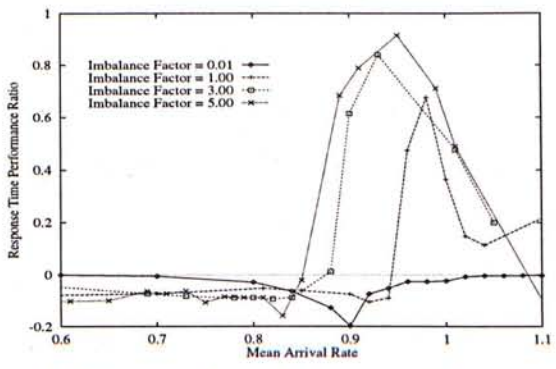
   At low system load, the extra CPU overhead of *GR.BATCH.A* is due to the additional messages generated by the GR protocol. The higher CPU overhead of *SK.SINGLE.AM* simply reflects the higher costs involved in task migration, as compared to task assignment. The higher CPU overhead of *SK.SINGLE.A* at high system load is due to the fact that some pollings are not successful in locating a transfer partner because of processor thrashing. This causes even more polling messages to be created, and thus higher CPU overhead.

   Figure 8.1(d) shows that channel overhead exhibits a similar pattern. This can be explained similarly.

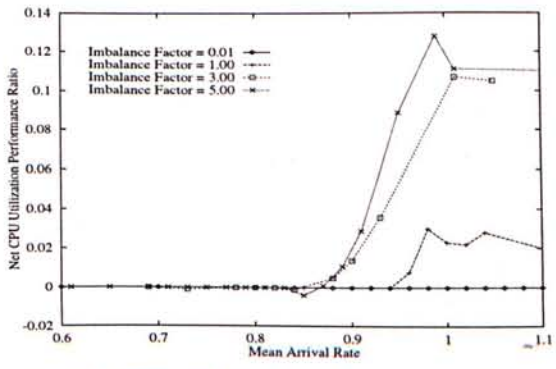
8.2.2 Effect of Workload Imbalance

Figure 8.2 shows the effect of system imbalance on the performance of *GR.BATCH.A*. We have the following observations and analysis on their performance.

1. Figure 8.2(a) shows that under all the imbalance factors studied, *GR.BATCH.A* has a poorer performance than *SK.SINGLE.A* at low system loads. This can be explained as follows. In this range of system load, *GR.BATCH.A* does not provide extra CPU utilization as is shown in Figure 8.2(b). When taking the extra overhead introduced by the GR protocol into consideration, it is clear that the net



(a) Response Time Performance Ratio



(b) Net CPU Utilization Performance Ratio

Figure 8.2: Effect of system imbalance on performance of $GR.BATCH.A$. Reference algorithm is $SK.SINGLE.A$. Simulation parameters used identical to those in Table 8.2, except that imbalance factor is a variable.

CPU capacity available for processing user tasks is reduced. This accounts for the poorer task response time exhibited by $GR.BATCH.A$ at low system loads.

2. Figure 8.2(a) shows that with imbalance factor 0.01, $GR.BATCH.A$ has a poorer performance than $SK.SINGLE.A$ throughout the whole range of system load. The workload homogeneity implies that single task transfer is enough and the additional overhead associated with $GR.BATCH.A$ is not compensated for.

   As the imbalance factor increases, the bell-shaped area with which $GR.BATCH.A$ exhibits positive response time performance ratio increases. This reveals the ability of $GR.BATCH.A$ in handling a highly imbalanced system.

Since we found a similar performance characteristics with the algorithm $SK.SINGLE.AM$, we do not show its performance results here.

8.3 Discussions

We found that both $GR.BATCH.A$ and $SK.SINGLE.AM$ have significant performance advantage over the use of a single task assignment algorithm $SK.SINGLE.A$. This shows that (1) task migration can be used as an alternative workload distribution mechanism for augmenting task assignment to provide extra performance improvement; and

(2) the batch assignment approach can boost up the utilization of the system capacity by resolving the problem of processor thrashing. Moreover, these two algorithms have comparable performance over the whole range of system load, with *SK.SINGLE.AM* performs slightly better in terms of both task response time and algorithm overhead. The choice among these two algorithms largely depends on whether task migration is supported. As task migration is scarcely supported in today's distributed operating systems, we conclude that the batch assignment approach is a simple and practical approach to load balancing.

Chapter 9

Applying Batch Transfer to Heterogeneous Systems with Many Task Classes

All of the previous studies assume that the distributed system is *homogeneous*, meaning that all processing nodes are functionally equivalent. In other words, tasks arrived at any node can be executed in any processing nodes in the system, not only in the arrival node. In this chapter, we attempt to apply the batch assignment approach to *heterogeneous systems*. Our model of heterogeneous systems has two major characteristics:

- Processing nodes have different processing throughputs.
- Tasks are divided into different classes. Each class is identified by its service time demand and its task code length.

The algorithms that we study in this chapter are variations of $GR.batch$ and $SK.single$, which are described in Chapter 5. (Refer to Table 5.3 on page 69.) These algorithms have been modified for the characteristics of our heterogeneous systems. In particular, task selection schemes are modified to cater for the difference in processing throughputs between transfer pairs. This is necessary because the processing time required by a task z when being executed in node P_i may be different from that when z is executed in node P_j. The algorithms that we study are divided into two categories:

- The first category consists of variations of the *SK.single* algorithm.

- The second category consists of a single algorithm which is a variation of the *GR.batch* algorithm. Task selection for composing a task batch is modeled as a *Subset-Sum Problem* (SSP) [MT90]. The objective of the composition scheme is to make maximal use of the allowed batch size b by selecting a subset from among the candidate tasks in the task queue. *Greedy approach* is used to find the approximate solutions for the SSP.

9.1 Heterogeneous System Model

In our homogeneous system model as defined in section 3.1 on page 19, all processing nodes are assumed to be identical. Implicitly defined in the homogeneous model is that the *service rate* of the nodes is one task per unit time. [1] For our purpose, we identify two different types of *heterogeneous systems*.

- In the first type, the processing nodes in the system are functionally identical, meaning that tasks arriving at any node can be executed in any other nodes in the system. In other words, the nodes are "*binary compatible*" to each other. Different processing nodes may have different processing throughputs however. That is, *processing throughput heterogeneity* is being focused.

- The second type of heterogeneous systems is more restricted. Nodes in the system are not binary compatible to each other and thus tasks arriving at a node can be assigned remotely to only a subset of other nodes in the system — those which are functionally equivalent to the arrival node. That is, both *processing throughput heterogeneity* and *functional heterogeneity* are being focused.

For the sake of clarity, we will only study the first type of heterogeneous systems.

[1] Recall that within a node P_i, the task service time of those locally arrived tasks has an exponential distribution with mean S_i. For an "even task service time system," S_i is a constant with the value S_o. Therefore, the service rate of the nodes in the homogeneous system is S_o tasks per unit time. In the simulations, S_o equals to 1.

9.1.1 Processing Node Specification

A heterogeneous system is characterized by the composition of node types. We can define a node type M_i by a 3-tuple $(M_i, Throughput_i, \psi_i)$, where $Throughput_i$ is the processing throughput, and ψ_i is the number of nodes of type M_i in the heterogeneous system. Suppose that there are m different node types. The heterogeneous system can then be represented as a set of 3-tuples:

$$\{(M_1, Throughput_1, \psi_1), (M_2, Throughput_2, \psi_2), \ldots, (M_m, Throughput_m, \psi_m)\} \quad (9.1)$$

We denote the set $\{1, 2, \ldots, m\}$ as M. The total number of processing nodes in the system is given by:

$$Total\ Number\ of\ Nodes = \sum_{i \in M} \psi_i \quad (9.2)$$

Furthermore, it is important to cater for the difference in processing throughputs between the transfer partners during sender-receiver negotiations. We therefore define the *Relative Processing Throughput* as follows.

> *Relative Processing Throughput*: of node type x with respect to node type y, denoted as r_{xy}, is defined as the ratio of the processing throughput of node type x to that of node type y. That is,
>
> $$r_{xy} = \frac{Throughput_x}{Throughput_y} \quad (9.3)$$

Based on the relative processing throughput, we can define the *Relative Processing Throughput Matrix* as follows.

> *Relative Processing Throughput Matrix*: $R = [r_{ij}]$ is a m by m matrix, where r_{ij} is the relative processing throughput of node type i with respect to node type j.

In subsequent discussion, we will need to refer to the relative processing throughput between two processing nodes. We will use the same notation as for the relative processing throughput between node types. That is, if there are two *processing nodes* with ids P_i and P_j, without ambiguous, r_{ij} refers to the relative processing throughput of the node type of P_i with respect to the node type of P_j.

9.1.2 Task Type Specification

Tasks in our heterogeneous model are divided into different categories. Each category is characterized by its service time requirement (i.e. the CPU time needed for completing its execution) and its task code length (i.e. the number of messages generated if the task is assigned remotely). Furthermore, each task type has its individual arrival rate to a processing node. Since processing nodes may have different processing throughputs, the service time requirement should be "calibrated" according to a particular node type. Throughout the study, node type M_1 is always used as the reference for calibration purpose. Thus, if task type J_1 has a service time requirement of 10, a task of type J_1 will take 10 units of time to be completed when being executed in node type M_1. We can define a task type J_i by a 4-tuple $(J_i, w_{1,i}, l_i, \lambda_i)$, where $w_{1,i}$ is the service time requirement with respect to node type M_1; l_i is the task code length; and λ_i is the arrival rate of task type J_i to a processing node. Suppose there are n different task types in the system. The task type composition of the system can then be represented as a set of 4-tuples:

$$\{(J_1, w_{1,1}, l_1, \lambda_1), (J_2, w_{1,2}, l_2, \lambda_2), \ldots, (J_n, w_{1,n}, l_n, \lambda_n)\} \tag{9.4}$$

We denote the set $\{1, 2, \ldots, n\}$ as J. The total task arrival rate at a node when calibrated with node type M_1 is given by:

$$\textit{Total Task Arrival Rate at a node} = \sum_{j \in J} \lambda_j \cdot w_{1,j} \tag{9.5}$$

The total task arrival rate in the system when calibrated with node type M_1 is given by:

$$\textit{Total Task Arrival Rate in the system} = \sum_{i \in M} \psi_i \cdot \sum_{j \in J} \lambda_j \cdot w_{1,j} \tag{9.6}$$

The total service rate of the system when calibrated with node type M_1 is given by:

$$\textit{Total Service Rate in the system} = \sum_{i \in M} \psi_i \cdot r_{i1} \tag{9.7}$$

Saturation of the system occurs if

$$\sum_{i \in M} \psi_i \cdot \sum_{j \in J} \lambda_j \cdot w_{1,j} > \sum_{i \in M} \psi_i \cdot r_{i1} \tag{9.8}$$

9.1.3 Workload State Measurement

Our previous measure of workload of a node is based on the number of tasks residing in the node. The validity of such a scheme relies on the fact that there is only one type of tasks. In the heterogeneous system model, tasks are divided into different classes. Some of the tasks may have a very long task service time, while the others may have a short one. Counting only the number of tasks residing in a node for determining the node's load state is inadequate for a heterogeneous system. Instead, the "weight" of the tasks should also be taken into consideration. Recall that we denote the total number of tasks residing in a node P_i, including those in the task queue, in the threshold queue, and in the service queue, by K_i. In order to measure the workload of a node, we define a new processing node attribute called the *node weight* as follows.

> *Node Weight*: of a node P_i, denoted by W_i, is defined as the sum of the remaining service time requirements of the tasks residing in P_i, measured with reference to P_i. That is,
>
> $$W_i = r_{1i} \cdot \sum_{j \in K_i} \hat{w}_{1,j} \tag{9.9}$$
>
> where $\hat{w}_{1,j}$ is the *remaining service time requirement* of task j with respect to node type M_1. For a task residing in the task queue or in the threshold queue, $\hat{w}_{1,j}$ equals to $w_{1,j}$ since the task has never been executed. For a task in the service queue, $\hat{w}_{1,j}$ equals to $w_{1,j}$ minus the *accumulated processing time* received by the task so far.

Intuitively, processing node P_i maintains the variable W_i. When a task arrive, either locally or remotely, the weight of the task with reference to node P_i is added to the variable W_i. Conversely, when a task is assigned remotely, or when the task has completed its execution in P_i, the weight of the task is deducted from W_i. Based on the node weight of a node, we can define the weighted effective load as follows.

> *Weighted Effective Load*: of a node P_i, denoted as WEL_i, is defined as the node weight of P_i plus the reservation value and minus the guarantee value of P_i. That is,
>
> $$WEL_i = W_i + RES_i - GUR_i \tag{9.10}$$
>
> where RES_i and GUR_i are the reservation value and the guarantee value of node P_i respectively.

ILLUSTRATION

Suppose the processing nodes in a heterogeneous system is defined by the following set of 3-tuples:

$$\{(M_1, 5, 1), (M_2, 30, 2), (M_3, 50, 1)\}$$

That is, there is one node of type M_1, which has a processing throughput of 5 unit; 2 nodes of type M_2, which has a processing throughput of 30 units; and 1 node of type M_3, which has a processing throughput of 50 units. Suppose there are four types of tasks as defined by the following set of 4-tuples:

$$\{(J_1, 1, 10, 0.5), (J_2, 5, 10, 0.2), (J_3, 15, 50, 0.2), (J_4, 100, 30, 0.01), \}$$

Consider node P_3 of type M_2. Let the set of tasks residing in node P_3 as:

$$K_3 = \{J_1, J_1, J_2, J_3, J_4\}.$$

K_3 can be rewritten in terms of service time requirements of tasks as:

$$K_3 = \{1, 1, 5, 15, 100\}.$$

For simplicity, assume that RES_3 and GUR_3 are zero. The weighted effective load of node P_3 can be calculated as:

$$\begin{aligned} WEL_3 &= r_{12} \cdot \sum_{j \in K_3} w_{1,j} + RES_3 - GUR_3 \\ &= \frac{Throughput_1}{Throughput_2} \cdot (1 + 1 + 5 + 15 + 100) + 0 - 0 \\ &= \frac{5}{30} \cdot 122 \\ &= 20.3 \end{aligned}$$

The intuitive meaning of the value 20.3 is as follows. If those tasks residing in node P_3 were executed in a node of type M_1, which is used as a reference for calibration purpose, the total service time required for completing them is 122 time unit. Since node P_3, which is of type M_2, is six times faster than a node of type M_1, P_3 only requires 1/6 of 122 time units for completing the tasks. Therefore, as long as node P_3 is concerned, the tasks apparently take 20.3 time units for completion.

Figure 9.1: Example illustrating the intuitive meaning of Weighted Effective Load, WEL_i.

Figure 9.1 provides an example illustrating the intuitive meaning of the definition of WEL_i. Based on the weighted effective load, we can define the 3-level load measurement scheme as shown in Table 9.1.

9.1.4 Task Selection Candidates

A task selection scheme (part of the transfer policy) is responsible for selecting the task(s) to be sent to the transfer partner. All the tasks residing in a task queue are candidates for task assignment. In principle, task selections are made in a FIFO order: (1) For single

Table 9.1: 3-level load measurement scheme based on weighted effective load, WEL_i.

Load State	*Criteria*
L-load	$WEL_i \leq lower\_threshold$
N-load	$lower\_threshold < WEL_i \leq upper\_threshold$
H-load	$WEL_i > upper\_threshold$

task assignment, the first task in the task queue is selected. (2) For batch task assignment, tasks are selected one by one, starting from the first task in the task queue, until the size of the task batch is fulfilled. However, in some of the algorithms that we study in this chapter, tasks may be selected in arbitrary order from the task queue. To derive the "best" selection decision requires all the tasks in the task queue to be examined. This may be prohibitively inefficient and may inject unnecessary extra overhead to the already busy sender nodes. Therefore, we have to limit the number of tasks that are eligible for consideration during task selection. This can be done as follows.

We denote the set of tasks residing in the task queue of node P_i by Z_i:

$$Z_i = \{1, 2, \ldots, k_i\} \tag{9.11}$$

where k_i is the total number of tasks residing in the task queue of node P_i.

Let η = The maximum number of tasks that can be considered by the selection scheme; algorithm design parameter.

$Z_{i,\eta}$ = The set of candidate tasks that are eligible for consideration by the selection scheme.

$k_{i,\eta}$ = The size of $Z_{i,\eta}$.

The value of $k_{i,\eta}$ is determined as follows:

$$k_{i,\eta} = \begin{cases} \eta & \text{if } \eta \leq k_i \\ k_i & \text{otherwise} \end{cases} \tag{9.12}$$

Therefore,

$$Z_{i,\eta} = \{1, 2, \ldots, k_{i,\eta}\} \subseteq Z_i \tag{9.13}$$

Tasks in the set $Z_{i,\eta}$ are subjected to the task selection scheme for consideration.

For Single Task Assignment:

Let j = The task selected for assignment.

$$j \in Z_{i,\eta} \tag{9.14}$$

For Batch Task Assignment:

Let z_i = The set of tasks selected by the selection scheme, i.e. the task batch.

$$z_i = \{j : 1 \leq j \leq k_{i,\eta}\} \subseteq Z_{i,\eta} \subseteq Z_i \tag{9.15}$$

The conceptual model of task selection is depicted in Figure 9.2.

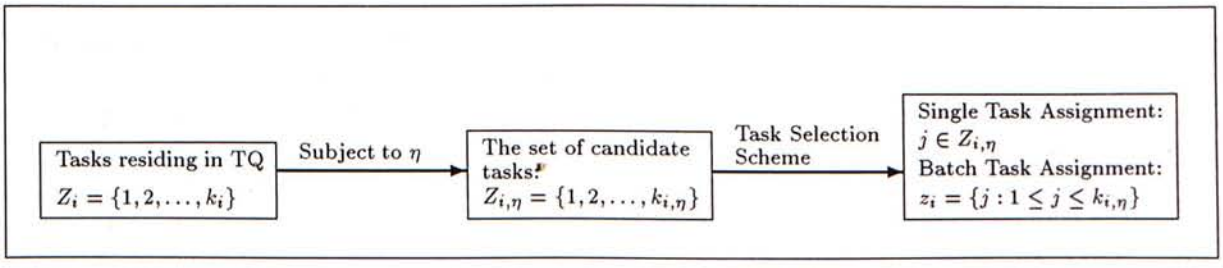


Figure 9.2: Conceptual model of task selection in task assignment algorithms.

9.2 Algorithm Descriptions

In this section, we describe the three categories of algorithms that we study in this chapter.

9.2.1 First Category — The *SK.single* Variations

This category contains two variations of the *SK.single* algorithm, which has been described in Section 5.3 on page 67. In brief, these algorithms employ the original Shivaratri and Krueger's location policy, allow only single task assignment, and use the actual number of task residing in a node for measuring the load state. Note that these algorithms fail to adapt their workload measurement schemes to cater for the difference in service

time requirements of different task classes, nor the difference in processing throughput between transfer partners.

Algorithm 1 — *SK.Single.First*

This algorithm is identical to the *SK.single*, which is renamed for maintaining the consistency in naming scheme. The qualifier "First" signifies that the selection scheme always selects the first task in the task queue for remote assignment, regardless the type of the task and the type of the receiver node.

Algorithm 2 — *SK.Single.BestFit*

This algorithm is similar to *SK.Single.First*, except that the selection scheme selects the "Best Fit" task in the task queue for remote assignment. The selection scheme is stated as follows.

Let j = the task selected for remote assignment
P_x = the sender node
P_y = the potential receiver node

$$\text{Minimize} \quad |w_{1,j} \cdot r_{1y} - 1|, \qquad j \in Z_{i,\eta} \tag{9.16}$$

Intuitively, the first η tasks in the task queue of the sender node P_x are examined. The task whose service time requirement with respect to the receiver node P_y is closest to 1 is selected. The heuristic we used here is: All we know about the sender-receiver negotiation is that the receiver agreed to accept one task, or more accurately, one unit of service time requirement with respect to the receiver's processing throughput. Selecting a task with service time requirement less than 1 unit may waste the spare processing capacity of the receiver node, and thus making the negotiation session less efficient. Selecting a task with service time requirement greater than 1 unit may overload the receiver. Thus, the best "guess" here is one unit of service time requirement with respect to the receiver node.

9.2.2 Second Category — The *GR.batch* Variation Modeled with SSP

This category contains a single algorithm labeled as *GR.SSP.Greedy*, which is a variation of the *GR.batch* algorithm. The *GR.batch* algorithm is modified to adapt to the system heterogeneity. In particular, the determination of batch size has to cater for any difference in processing throughputs between the sender and the receiver. For example, when a sender P_x guarantees to send g tasks to a receiver P_y, this actually means $(g \cdot r_{xy})$ unit of CPU capacity to P_y. Similarly, when a sender determines the desired batch size, the reservation value received from the receiver should be calibrated according to the processing throughput of the sender itself.

Besides, as different task classes have different service time requirements, it is no longer appropriate to measure batch size in terms of the number of tasks in a task batch. Instead, batch size should be measured in terms of the total service time requirements of all the tasks in the task batch. These all arise the problem of task batch composition. In *GR.SSP.Greedy*, task selection for composing a task batch is modeled as a *Subset-Sum Problem* (SSP) [MT90]. The objective of the task batch composition scheme is to make the maximal use of the allowed batch size b by selecting a subset of tasks from among the candidate tasks in the task queue. A *greedy* approach is used to solve the SSP and thus the algorithm is labeled as *GR.SSP.Greedy*.

SSP Task Batch Composition Scheme:

With the original *GR.batch* algorithm, we select b tasks in the assignment queue of the sender node in a FIFO manner. The value of b is derived partly with the reservation r, i.e. the number of tasks the receiver has reserved for the sender. (Refer to section 5.1.3 on page 60.) When applied to a heterogeneous system, the value r is first adjusted according to the relative processing throughputs between the transfer partner, before it is used for deriving the value of b. After the batch size b has been determined, we have to select tasks to be transferred to the receiver node. Since different classes of tasks have different service time requirements, we have to select the appropriate set of tasks so as to make the maximal use of the batch size b. In this way, we are making the most efficient use of a

negotiation session. To achieve this, we formulate the task selection scheme for composing a task batch as a subset-sum problem as follows.

Given a set of tasks $Z_{i,\eta} = \{1, 2, \ldots, k_{i,\eta}\}$ and a maximum batch size b, with

$$w_{i,j} = \text{service time requirement of task } j \text{ with respect to node } P_i, \quad j \in Z_{i,\eta}, \tag{9.17}$$

select a subset of tasks $z_i \subseteq Z_{i,\eta}$ whose total service time requirement with respect to node P_i is closet to, without exceeding, b, i.e.

$$\text{Maximize} \qquad \hat{b} = \sum_{j \in Z_{i,\eta}} w_{i,j} \cdot x_j \tag{9.18}$$

$$\text{Subject to} \qquad \sum_{j \in Z_{i,\eta}} w_{i,j} \cdot x_j \leq b, \tag{9.19}$$

$$x_j = 0 \text{ or } 1, \qquad j \in Z_{i,\eta} = \{1, 2, \ldots, k_{i,\eta}\}, \tag{9.20}$$

where

$$x_j = \begin{cases} 1 & \text{if task } j \text{ is selected} \\ 0 & \text{otherwise} \end{cases} \tag{9.21}$$

The most immediate approach to the *heuristic* solution of the above formulation of SSP is the *greedy approach* [MT90], which consists of examining the candidate tasks, $Z_{i,\eta}$, in a FIFO order and inserting each task into the task batch z_i if it fits. To guarantee a worst-case performance of 1/2, the task with the largest service time requirement will be considered as a possible alternative solution. The task selection procedure is shown below.

```
Procedure Selection.SSP.Greedy
Input: b, Z_{i,η}, (w_{i,j})
Output: b̂, z_i, (x_j)
BEGIN
    b̂ := 0;
    z_i := φ;
    b̄ := b;
    j* := 1;
    FOR j := 1 TO k_{i,η} DO
        IF w_{i,j} > b̄
            x_j := 0;
        ELSE
            x_j := 1;
            z_i := z_i ∪ j;
            b̄ := b̄ − w_{i,j};
        ENDIF
        IF w_{i,j} > w_{i,j*}
            j* := j;
        ENDIF
    ENDDO
    b̂ := b − b̄ ;
    IF w_{i,j*} > b̂
        FOR j := 1 to k_{i,η} DO
            x_j := 0;
        ENDDO
        x_{j*} := 1;
        b̂ := w_{i,j*};
        z_i := {j} ;
    ENDIF
END
```

Figures 9.3– 9.7 depict the complete *GR.SSP.Greedy* algorithm. The basic skeleton of the algorithm is similar to the *GR.batch* algorithm. Note that the superscripts of g^x, max^y, and r^y signify that the value of g is calibrated with respect to node P_x, whereas the value of max is calibrated with respect to node P_y, and so on. Furthermore, the functions $MaxAssign()$ and $NumAssign()$ are modified to cater for the system heterogeneity. These two modified functions are depicted in Figure 9.8. The function $ReceiverNewEL()$, which is used by a sender to estimate the receiver's new effective load in the original *GR.batch* algorithm, is modified and renamed as $ReceiverNewWEL()$. This new function is depicted in Figure 9.9.

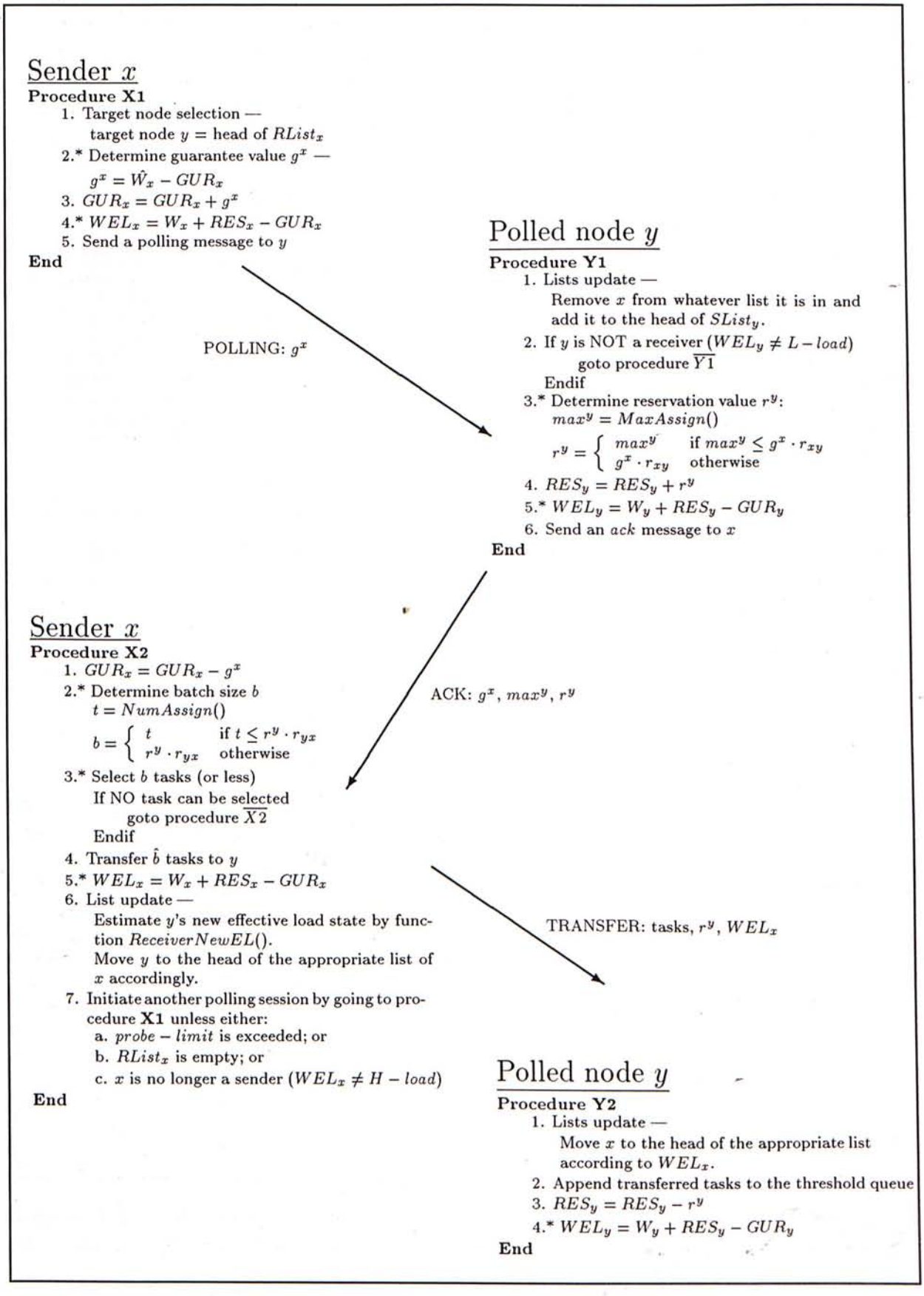


Figure 9.3: Sender-initiated component of the *GR.SSP.Greedy* algorithm. Steps marked with * represent major modifications for adapting to the system heterogeneity.

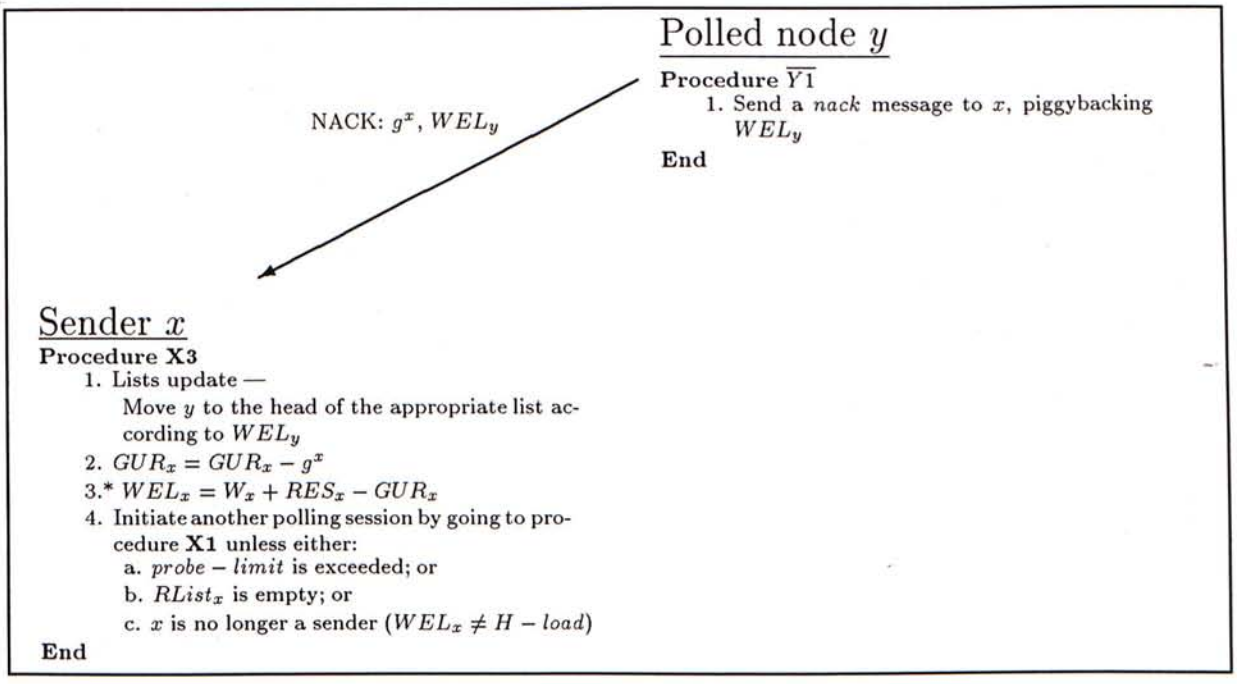


Figure 9.4: Sender-initiated component of the *GR.SSP.Greedy* algorithm — Procedure $\overline{Y1}$. Steps marked with * represent major modifications for adapting to the system heterogeneity.

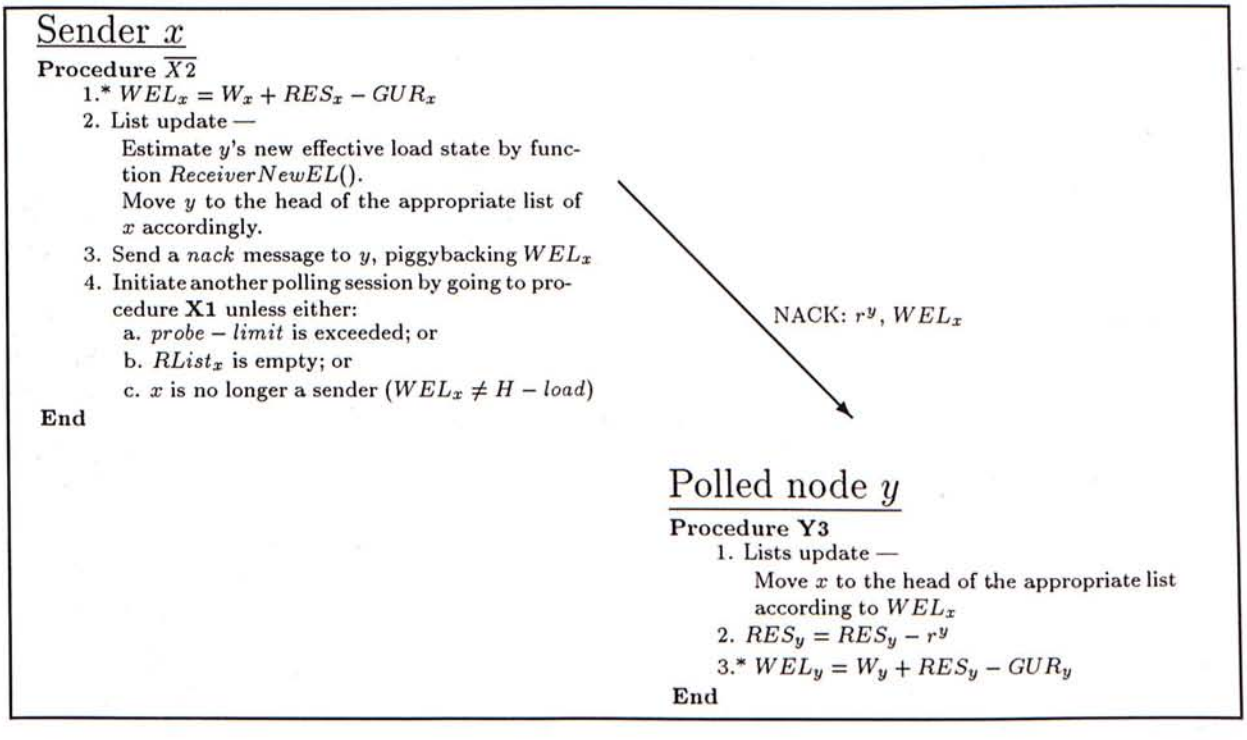


Figure 9.5: Sender-initiated component of the *GR.SSP.Greedy* algorithm — Procedure $\overline{X2}$. Steps marked with * represent major modifications for adapting to the system heterogeneity.

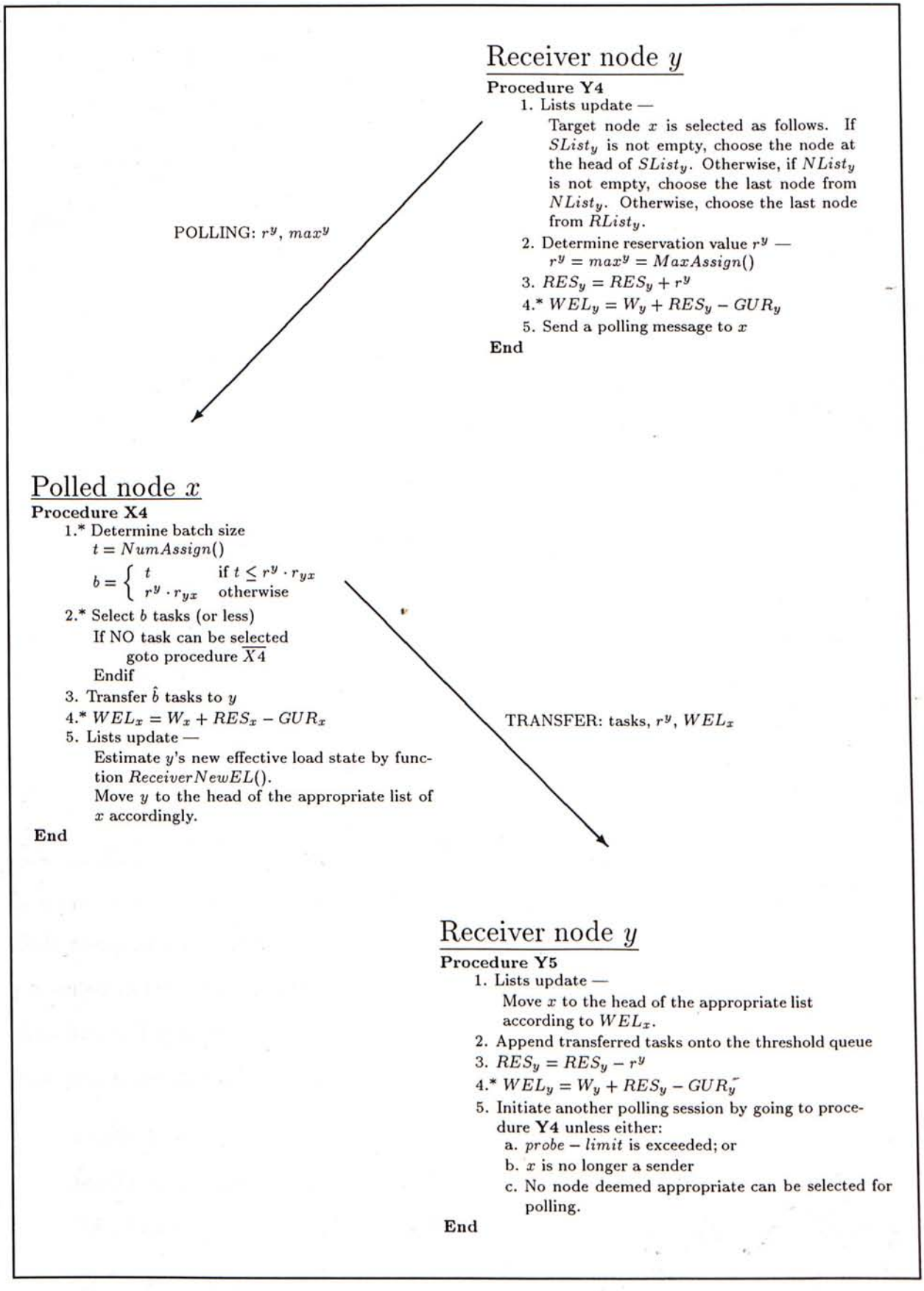


Figure 9.6: Receiver-initiated component of the *GR.SSP.Greedy* algorithm. Steps marked with * represent major modifications for adapting to the system heterogeneity.

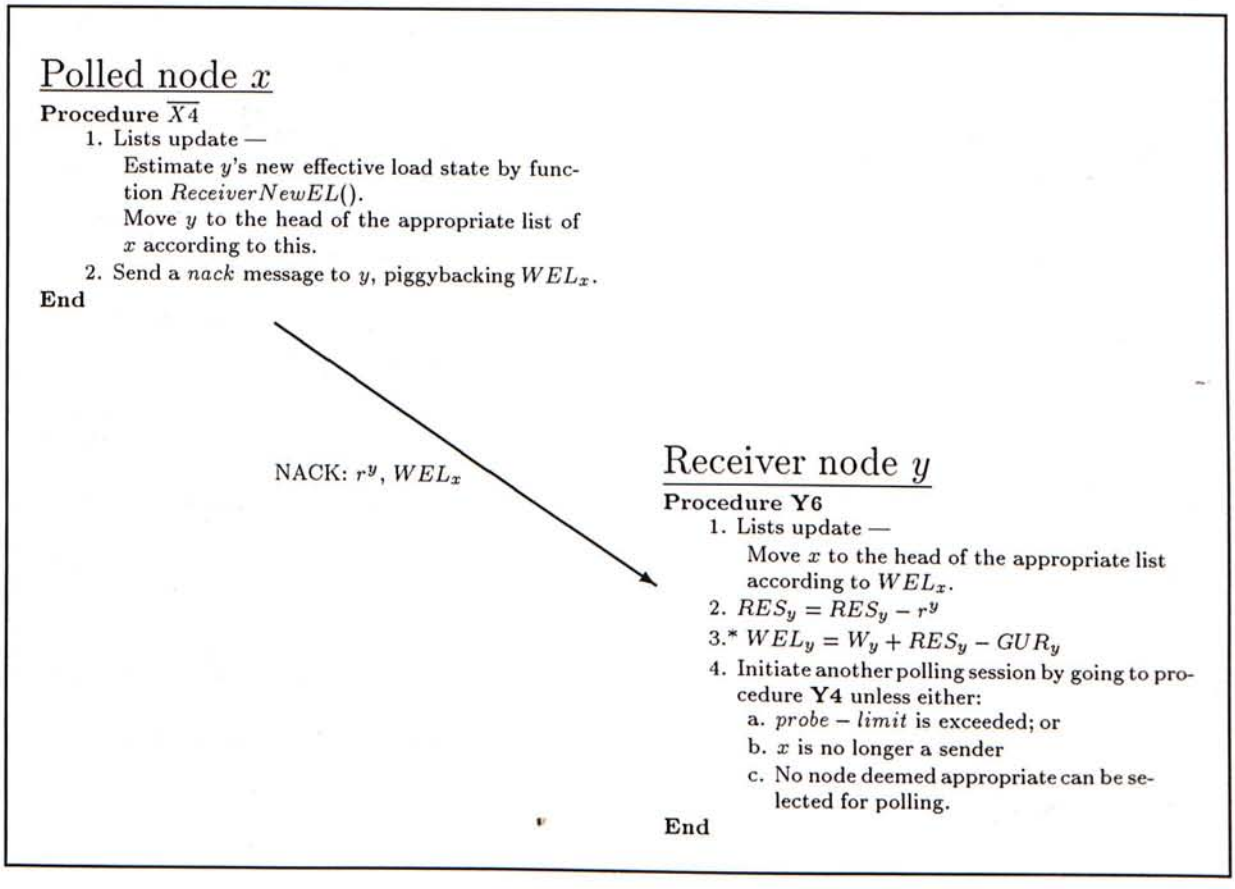


Figure 9.7: Receiver-initiated component of the *GR.SSP.Greedy* algorithm — Procedure $\overline{X4}$. Steps marked with * represent major modifications for adapting to the system heterogeneity.

9.3 Analysis of Simulation Results

It must be noted the performance of the algorithms largely depends on the processing node compositions and the task classes imposed on the system. The simulation results presented in this section therefore serve only to provide some cues to the usability of the algorithms. Figure 9.10 presents a comparison of the performance of the three algorithms. Simulation parameters used are given in Table 9.2.

1. Figure 9.10(a) shows the performance of the three algorithms under a system with homogeneous processing nodes having a single task class. It can be seen that *SK.Single.First* and *SK.Single.BestFit* have identical performance. As there is only one single task class, the task selection scheme of both algorithms therefore

MaxAssign()	*NumAssign()*
Let WEL_y = Current weighted effective load of receiver y Let WEL_y' = Estimated weighted effective load of receiver y after task transfer	Let WEL_x = Weighted effective load of sender x when the polling message is received Let t^x = Weight of tasks sender x is willing to send to the receiver.
If max^y tasks are relocated, $$WEL_y' \approx WEL_y + max^y$$	By **Rule 2**, $$WEL_x - t^x > lower\_threshold$$ $$t^x < WEL_x - lower\_threshold \quad (9.23)$$
By **Rule 1**, $$WEL_y' \leq upper\_threshold$$	Using equation (9.22), sender node x estimates EL_y from max as follows. $$WEL_y \approx upper\_threshold - max^y \quad (9.24)$$
Thus, $$max^y \leq upper\_threshold - WEL_y$$	By **Rule 3**, $$WEL_y + t^x \cdot r_{xy} \leq WEL_x - t^x$$
Taking the largest possible value, $$max^y = upper\_threshold - WEL_y \quad (9.22)$$	Thus, $$t^x \leq \frac{WEL_x + max^y - upper\_threshold}{1 + r_{xy}} \quad (9.25)$$ Of course, $$t^x \leq max^y \cdot r_{yx} \quad (9.26)$$ t^x is taken to be the largest integer satisfying the inequalities (9.23), (9.25), and (9.26).

Figure 9.8: $MaxAssign()$ and $NumAssign()$ for determining t^x.

Let WEL_y' = estimated weighted effective load of y
Let WEL_y'' = estimated new weighted effective load of y after accepting $\hat{b}$ tasks

By equation (9.24):

$$WEL_y' \approx upper\_threshold - max^y$$

$$\begin{aligned} WEL_y'' &\approx WEL_y' + \hat{b} \cdot r_{xy} \\ &= upper\_threshold - max^y + \hat{b} \cdot r_{xy} \end{aligned} \quad (9.27)$$

Figure 9.9: Function $ReceiverNewWEL()$ for estimation of receiver's new effective load based on max^y and $\hat{b}$.

always selects the first task in the task queue for remote assignment. The two algorithms are essentially identical.

At low ($\lambda_1 = 0.3$) and medium ($\lambda_1 = 0.6$) system loads, *GR.SSP.Greedy* has comparable performance to the other two algorithms. This is because the system in both cases has large enough spare processing capacity and the difference between the algorithms is insignificant.

At high system loads ($\lambda_1 = 0.9$), *GR.SSP.Greedy* performs slightly poorer than the other two algorithms. This can be attributed to the extra overhead of the batch transfer approach and the GR protocol. Besides, as the system is homogeneous, the benefit of *GR.SSP.Greedy* in considering both processing nodes types and task classes in its batch composition scheme is not exhibited.

2. Figure 9.10(b) shows the performance of the three algorithms under a system with heterogeneous processing nodes and multiple task classes.

   In the first task composition, there is a task type with exceedingly large service time requirement (500) and low arrival rate (0.1). Among the three algorithms, *GR.SSP.Greedy* performs the best in terms of task response time (12.91); whereas *SK.Single.BestFit* performs the worst with a task response time of 20.06. Similar result is found for the third task composition (response time 29.97 Vs 50.48).

Table 9.2: Values of simulation parameters used in the simulations for studying the performance of *SK.Single.First*, *SK.Single.BestFit*, and *GR.SSP.Greedy*.

Parameter	Value
Q_o	30
T_{CPU}	0.2
lower_threshold	$1/3 * Q_o = 10$
upper_threshold	$2/3 * Q_o = 20$
probe_limit	5

Parameter	Value
$CPU_{polling}$	0.005
$F_{polling}$	0.001
F_{task}	0.005
D	0.01
C_{pack}	0.003
C_{assign}	0.002

Task Composition	*Algorithms*	*Response Time*	*CPU Util.*	*% CPU Overhead*	*Hit Ratio*	*% Remote Execution*	*Channel Overhead*
$\{(J_1, 1, 5, 0.3)\}$	*SK.Single.First*	1.42	31.91	2.611	0	0	3.995
	SK.Single.BestFit	1.42	31.91	2.611	0	0	3.995
	GR.SSP.Greedy	1.42	31.91	2.610	0	0	3.993
$\{(J_1, 1, 5, 0.6)\}$	*SK.Single.First*	2.65	63.64	4.669	0	0	7.144
	SK.Single.BestFit	2.65	63.64	4.669	0	0	7.144
	GR.SSP.Greedy	2.63	63.57	4.668	0	0	7.143
$\{(J_1, 1, 5, 0.9)\}$	*SK.Single.First*	13.16	93.05	4.925	0.007	1.048	7.507
	SK.Single.BestFit	13.16	93.05	4.925	0.007	1.048	7.507
	GR.SSP.Greedy	13.56	92.73	4.965	0.006	0.669	7.604

(a) Homogeneous Processors Composition: $\{(M_1, 1, 30)\}$
Subjected to single task type at different arrival rates.

Task Composition	*Algorithms*	*Response Time*	*CPU Util.*	*% CPU Overhead*	*Hit Ratio*	*% Remote Execution*	*Channel Overhead*
$\{(J_1, 1, 5, 0.3),$	*SK.Single.First*	17.67	96.10	1.025	0.220	30.305	1.376
$(J_2, 0.1, 5, 0.3),$	*SK.Single.BestFit*	20.06	97.15	0.924	0.305	36.226	1.162
$(J_3, 500, 5, 0.1)\}$	*GR.SSP.Greedy*	12.91	92.97	3.014	0.347	5.269	4.619
$\{(J_1, 1, 5, 0.3),$	*SK.Single.First*	405.26	99.50	0.029	0.190	0.602	0.420
$(J_2, 0.1, 5, 0.3),$	*SK.Single.BestFit*	405.26	99.50	0.029	0.190	0.602	0.420
$(J_3, 500, 5, 0.3)\}$	*GR.SSP.Greedy*	409.40	99.50	0.320	0.050	0.574	0.221
$\{(J_1, 1, 5, 0.3),$	*SK.Single.First*	50.48	99.37	0.081	0.335	4.462	0.103
$(J_2, 0.1, 5, 0.3),$	*SK.Single.BestFit*	54.84	99.34	0.096	0.383	6.294	0.116
$(J_3, 1000, 5, 0.1)\}$	*GR.SSP.Greedy*	29.97	99.26	1.557	0.036	1.038	2.383

(b) Heterogeneous Processor Composition: $\{(M_1, 1, 30),\ (M_2, 100, 5),\ (M_3, 0.01, 5)\}$
Subjected to different task compositions.

Figure 9.10: Performance of *SK.Single.First*, *SK.Single.BestFit* and *GR.SSP.Greedy*

Chapter 10

Conclusions and Future Work

This thesis presented a comparative study on the performance of different dynamic load balancing algorithms. Issues regarding the design of dynamic load balancing algorithms are also discussed. Dynamic load balancing algorithms strive to use the current (or near current) system load information to balance the workload among the processing nodes in a distributed system by distributing tasks among the processing nodes. The potential benefits that may be achieved by load balancing algorithms include the minimization of task response time, and the maximization of CPU utilization and total system throughput. The major work that we have done are summarized below:

1. The design of a system model which serves as a common framework with which different dynamic load balancing algorithms can be compared objectively.

2. The study of load information dissemination strategies.

3. The development of the new task transfer approach, namely the *batch assignment*.

4. The application of the batch assignment approach in resolving processor thrashing.

5. The application of the batch assignment approach in resolving congestions in systems subjected with bursty task arrival patterns.

6. The study of the possibility of combining task assignment and task migration for pursuing extra performance improvement.

7. The application of batch assignment to heterogeneous distributed systems with many task classes.

The Study of Load Information Dissemination Strategies

In Chapter 4, we studied two different load information dissemination strategies: one with the presence of locally maintained load tables, and one which relies on polling for gathering load information of other processing nodes. We found that the presence of a load table avoids inappropriate processing nodes to be selected by a location policy and thus maintains system stability at low and high system loads. We refer to this as the *filtering effect*. Adaptive symmetrically-initiated polling-based location policy, such as the one proposed by Shivaratri and Krueger in [SK90], also exhibits the filtering effect and *indiscriminate pollings* are avoided. However, the filtering effect imposes an adverse effect called *processor thrashing* to the system. Processor thrashing means that a number of nodes poll for the same processing node simultaneously. It results in reduced workload distribution and thus reduced CPU utilization. To put an adaptive symmetrically-initiated polling-based location policy into practical use therefore requires processor thrashing to be resolved.

The Batch Assignment Approach

The batch assignment approach allows a number of tasks to be transferred as a single batch from a sender to a receiver with only a single sender-receiver negotiation session. It can therefore smooth out workload imbalance in an efficient manner. Since significantly less negotiation sessions are required for distributing the same amount of workload, batch assignment is more efficient in terms of both CPU and communication overheads. Central to the batch assignment approach are three *Batch Size Determination Rules*, which avoid a task batch receiver from being flooded by an incoming task batch. We found that the batch assignment approach provides promising performance results. Also, the CPU and communication overheads injected by using this approach are relatively small, when

compared to the traditional single task transfer approach.

We have also developed the *Guarantee and Reservation Protocol* which attempts to obtain the mutual agreement between a sender and a receiver on the optimal batch size. The central idea of the GR Protocol is two fold: (1) A sender node has to declare the number of tasks that it guarantees to send to a receiver; and (2) A receiver employs a "quota" scheme for reserving processing capacity for task batches from senders. It is the primary vehicle in resolving processor thrashing. This has been shown in Chapter 5.

In Chapter 6, the batch assignment approach has been applied to systems subjected with bursty task arrival patterns. Since algorithms using the traditional single task transfer approach cannot resolve congestions in such systems, the performance exhibited by them are not satisfactory and the system predictability is very poor. In contrast, the batch transfer approach can resolve congestions efficiently because significantly less polling sessions are needed for detracting the workload of those congested nodes.

Assignment Augmented With Migration

In Chapter 7, we successfully showed that although task migration in general costs more than task assignment, it can be used to augment task assignment for achieving extra performance improvement. This is in contrast to the common belief that task assignment should be the sole workload distribution mechanism in dynamic load balancing. This approach has been compared with the batch assignment approach in Chapter 8, where we found that they have comparable performance.

Batch Assignment in Heterogeneous Systems

In Chapter 9, we showed how heterogeneous systems can be modeled with a set of 3-tuples $(M_i, Speed_i, \psi_i)$, and how task type compositions can be modeled with a set of 4-tuples $(J_i, w_{1,i}, l_i, \lambda_i)$. In addition, we explained why the measurement of workload of a processing node cannot be based on the number of tasks residing in the node. Instead, we defined the *node weight* as a basis for workload measurement.

We also modified and applied the batch assignment approach to such heterogeneous systems. We showed that the task selection scheme should cater for the difference in processing speeds between a sender node and a receiver node. This is important in making the most efficient use of a sender-receiver negotiation session. In particular, we modeled batch composition as a *Subset-Sum Problem* and a *greedy* solution has been proposed.

Future Work

- Although we have run numerous simulations (of which a very small portion is shown in this thesis), the performance of the batch assignment approach should be studied more rigorously with a diverse set of simulation parameters. This is important in identifying situations where batch assignment is or is not appropriate.

- Since both batch transfer and task assignment augmented with migration are promising approaches for dynamic load balancing, it may be possible to combine the two for further improving system performance. This will mean a new task selection scheme, which should employ task assignment as long as possible to avoid unnecessary extra overhead due to task migration. In other words, task migration should be used restrictly. In addition, message structures which allow a task batch to consists of both type of transfer mechanisms have to be developed.

- The validity of our findings may be further proved by measurements. This will involve the implementation of an actual system which supports dynamic workload distribution. In general, we may have two different approaches of doing this: (1) Implementation in the operating systems level; and (2) Implementation on top of cluster programming toolsets such as PVM [CG90] and p4 [BL92].

Bibliography

[BL92] R. Butler and E. Lusk. *"User's Guide to the p4 Programming System"*. Argonne National Laboratory, 1992. Technical Report ANL-92/17.

[Bok79] Shahid H. Bokhari. "Dual Processor Scheduling with Dynamic Reassignment". *IEEE Transactions on Software Engineering*, SE-5(4):326–334, July 1979.

[Bok87] Shahid H. Bokhari. *"Assignment Problems in Parallel and Distributed Computing"*. Kluwer Academic Publishers, 1987.

[Cas81] L. M. Casey. "Decentralized Scheduling". *The Australian Computer Journal*, 13(2):58–63, 1981.

[CG90] N. Carriero and D. Gelernter. *"How to Write Parallel Programs: A First Course"*. Cambridge: MIT Press, 1990.

[CK88] Thmoas L. Casavant and Jon G. Kuhl. "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems". *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[EL86a] D. L. Eager and E. D. Lazowska. "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing". *Performance Evaluation*, 6:53–68, 1986.

[EL86b] D. L. Eager and E. D. Lazowska. "Adaptive Load Sharing in Homogeneous Distributed Systems". *IEEE Transactions on Software Engineering*, SE-12(5), May 1986.

[Gos91] A. Goscinski. *"Distributed Operating Systems — The Logical Design"*. Addison-Wesley, 1991.

[KK92] O. Kremien and J. Kramer. "Methodical Analysis of Adaptive Load Sharing Algorithms". *IEEE Transactions on Parallel and Distributed Systems*, 3(6):747–760, November 1992.

[LL94] Chin Lu and Sau-Ming Lau. "A Performance Study on Load Balancing Algorithms with Process Migration". In *Proceedings, IEEE TENCON 1994*, pages 357–364, Singapore, August 1994.

[LL95a] Chin Lu and Sau-Ming Lau. "An Adaptive Algorithm for Resolving Processor Thrashing in Load Distribution". *Concurrency: Practice and Experience*, 7(7), October 1995. Special issue on dynamic resource management in distributed systems; Accepted for publication.

[LL95b] Chin Lu and Sau-Ming Lau. "An Adaptive Load Distribution Algorithm for Systems with Bursty Task Arrivals". In *Proceedings, Thirteenth IASTED International Conference for Applied Informatics*, Austria, February 1995.

[Lo88] Virginia Mary Lo. "Heuristic Algorithms for Task Assignment in Distributed Systems". *IEEE Transactions on Computers*, 37(11):1384–1397, November 1988.

[Mil87] Milan Milenkovic. *"Operating Systems — Concepts and Design"*. McGraw-Hill, 1987.

[ML87] M. W. Mutka and M. Livny. "Scheduling Remote Processing Capacity in a Workstations-Processor Bank Computing System". In *Proceedings, the 7th International Conference on Distributed Computing Systems*, pages 2–9, 1987.

[MT90] Silvano Martello and Paolo Toth. *"Knapsack Problems — Algorithms and Computer Implementations"*. John Wiley & Sons, 1990.

[NXG85] L. M. Ni, C. W. Xu, and T. B. Gendreau. "Drafting Algorithm - A Dynamic Process Migration Protocol for Distributed Systems". In *Proceedings, the 5th International Conference on Distributed Computing Systems*, pages 539–546. IEEE, 1985.

[Phi90] Ian R. Philp. "Dynamic Load Balancing in Distributed Systems". In *Proceedings, IEEE 1990 SouthEast Conference*, pages 304–307, 1990.

[SK90] N. G. Shivaratri and P. Krueger. "Two Adaptive Location Policies for Global Scheduling Algorithms". In *Proceedings, The 10th International Conference on Distributed Computing Systems*, pages 502–509, May 1990.

[SKS92] N. G. Shivaratri, P. Krueger, and M. Singhal. "Load Distributing for Locally Distributed Systems". *IEEE Computer*, pages 33–44, December 1992.

[Smi88] Jonathan M. Smith. "A Survey of Process Migration Mechanisms". *Operating Systems Review*, 22(3):28–40, July 1988.

[SS84] John A. Stankovic and Inderjit S. Sidhu. "An Adaptive Bidding Algorithm for Processes, Clusters, and Distributed Groups". In *Proceedings, The 4th International Conference on Distributed Computing Systems*, pages 13–18, May 1984.

[ST85] Chien-Chung Shen and Wen-Hsiang Tsai. "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion". *IEEE Transactions on Computers*, c-34(3):197–203, March 1985.

[TL89] Marvin M. Theimer and Keith A. Lantz. "Finding Idle Machines in a Workstation-Based Distributed Systems". *IEEE Transactions on Software Engineering*, 15(11), November 1989.

[WM85] Y. T. Wang and Robert J. T. Morris. "Load Sharing in Distributed Systems". *IEEE Transactions on Computers*, c-34(3), March 1985.

[ZF87] S. Zhou and D. Ferrari. "A Measurement Study of Load Balancing Performance". In *Proceedings, The 7th International Conference of Distributed Computing Systems*, pages 490–497. IEEE, 1987.

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

Appendix A

System Model Notations and Definitions

Appendix A.1 Processing Node Model

Notation	*Meaning/Description*
Q_o	Capacity of service queue
T_{CPU}	CPU Time Slice
N	Number of processing node in the DCS
P_i	A processing node with id i
λ_i	Mean task arrival rate of node P_i
λ_o	Mean task arrival rate of N nodes
σ_λ	Standard deviation of task arrival rates of N nodes; also called imbalance factor
S_i	Mean task service time of node P_i

Appendix A.2 Cost Models

Notation	*Meaning/Description*
$CPU_{polling}$	CPU overhead associated with sending or receiving a polling message
$F_{polling}$	Time needed for injecting a polling message into the communication channel
$DELAY_{polling}$	Communication delay experienced by a polling message
$DELAY_{broadcasting}$	Communication delay experienced by a load state broadcasting message
CPU_{assign}	CPU cost associated with task assignment
C_{assign}	CPU cost in running assignment algorithm
C_{pack}	CPU cost for composing/decomposing each task message packet
l_i	Number of message packets generated for an assignment task i
l_{assign}	Mean of l_i in a processing node
B_a	Batch size in an assignment batch
$CPU_{migration}$	CPU cost associated with task migration
$C_{migrate}$	CPU cost in running migration algorithm
$C_{migrate_o}$	Mean of $C_{migrate}$
$l_i^{'}$	Number of message packets generated for a migration task i
$l_{migrate}$	Mean of $l_i^{'}$ in a processing node
B_m	Batch size in a migration batch
D	Propagation delay in communication channel
F_{task}	Time needed for injecting a single task message packet into the communication channel

Definition 1 The CPU overhead associated with sending or receiving a polling message is non-negligible and is represented by the parameter $CPU_{polling}$.

Definition 2 The communication delay experienced by a polling message consists of a single message injection cost $F_{polling}$, plus the propagation delay D, and is represented by $DELAY_{polling}$:

$$DELAY_{polling} = F_{polling} + D$$

Definition 3 The CPU overhead associated with sending or receiving a load state broadcasting message is non-negligible and is represented by the parameter $CPU_{polling}$.

Definition 4 The communication delay experienced by a load state broadcasting message consists of a single message injection cost $F_{polling}$, plus the propagation delay D, and is denoted by $DELAY_{broadcasting}$:

$$DELAY_{broadcasting} = F_{polling} + D$$

Definition 5 The *batch size* of a task batch is the number of tasks contained in the task batch.

Definition 6 The CPU cost associated with task assignment is non-negligible and is represented by $CPU_{assignment}$, which is defined as follows:

$$CPU_{assignment} = C_{assign} + C_{pack} * \sum_{i=1}^{B_a} l_i$$

where C_{assign} and C_{pack} are constants representing the CPU time for running the assignment algorithm, and the CPU time for composing/decomposing each task message packet, respectively. The value B_a is the number of tasks contained in the task batch. The value l_i represents the number of message packets generated for a task i. This is referred to as the *task code length* of i. Within a node, l_i has an independent exponential distribution with mean l_{assign}. The term $\sum_{i=1}^{B_a} l_i$ therefore represents the total number of message packets generated for the task batch.

Definition 7 The communication delay experienced by an assignment task batch relates to its batch size and is defined as follows:

$$DELAY_{assignment} = (F_{task} * \sum_{i=1}^{B_a} l_i) + D$$

where F_{task} is the time needed for injecting a single task transfer message packet into the communication channel.

Definition 8 The CPU cost associated with task migration is non-negligible and is represented by $CPU_{migration}$, which is defined as follows:

$$CPU_{migration} = C_{migrate} + C_{pack} * \sum_{i=1}^{B_m} l_i'$$

where $C_{migrate}$ represents the CPU cost for running migration algorithms, including migrant selection, task image saving/restoration, etc. Within each node, $C_{migrate}$ has an independent exponential distribution with mean $C_{migrate_o}$. This distribution characterizes the fact that some migrations impose more CPU overhead because of more opened files, more established communication channels, and more allocated memory, etc. The value B_m is the number of migrants in the task batch. The value l_i' represents the number of message packets generated for a migrant i. This is referred to as *task state length* of i. Within a node, l_i' has an independent exponential distribution with mean $l_{migrate}$. The term $\sum_{i=1}^{B_m} l_i'$ therefore represents the total number of message packets generated for the migrants.

Definition 9 The communication delay experienced by a migration task batch relates to its batch size and is defined as follows:

$$DELAY_{migration} = (F_{task} * \sum_{i=1}^{B_m} l_i') + D$$

where F_{task} is the time needed for injecting a single task transfer message packet into the communication channel.

Appendix A.3 Load Measurement

Reservation Value: of a node P_i, denoted as RES_i, is the total number of tasks that P_i has agreed to accept from other sender nodes.

Guarantee Value: of a node P_i, denoted as GUR_i, is the total number of tasks that P_i has guaranteed to transfer to other receiver nodes.

Effective Load: of a node P_i, denoted as EL_i, is defined as $EL_i = K_i + RES_i - GUR_i$, where K_i is the number of tasks currently residing in P_i, including those in the task queue, in the threshold queue, and those partially completed tasks residing in the service queue of P_i.

Appendix A.4 Batch Size Determination Rules

Rule 1: After accepting max tasks, the receiver should not be in H-load, neglecting new arrivals and departures during the negotiation and task transfer operations.

Rule 2: After transferring t tasks, the sender node should not be in L-load.

Rule 3: After transferring t tasks, the expected total number of tasks in the receiver should not be greater than the total number of tasks in the sender, neglecting new arrivals and departures of the receiver.

Appendix A.5 Bursty Arrivals Modeling

Burst Frequency, α_i: is defined as the reciprocal of the *inter-burst period*, which is the mean time between successive burst arrivals.

Burst Amplitude, β_i: is the number of tasks arrived locally to a bursty processing node per unit time during a task arrival burst.

Burst Duration, γ_i: is the duration of a task arrival burst.

Appendix A.6 Heterogeneous Systems Modeling

Relative Processing Throughput: of node type x with respect to node type y, denoted as r_{xy}, is defined as the ratio of the processing throughput of node type x to that of node type y. That is,

$$r_{xy} = \frac{Throughput_x}{Throughput_y}$$

Relative Processing Throughput Matrix: $R = [r_{ij}]$ is a m by m matrix, where r_{ij} is the relative processing throughput of node type i with respect to node type j.

Node Weight: of a node P_i, denoted by W_i, is defined as the sum of the remaining service time requirements of the tasks residing in P_i, measured with reference to P_i. That is,

$$W_i = r_{1i} \cdot \sum_{j \in K_i} \hat{w}_{1,j}$$

where $\hat{w}_{1,j}$ is the *remaining service time requirement* of task j with respect to node type M_1. For a task residing in the task queue or in the threshold queue, $\hat{w}_{1,j}$ equals to $w_{1,j}$ since the task has never been executed. For a task in the service queue, $\hat{w}_{1,j}$ equals to $w_{1,j}$ minus the *accumulated processing time* received by the task so far.

Weighted Effective Load: of a node P_i, denoted as WEL_i, is defined as the node weight of P_i plus the reservation value and minus the guarantee value of P_i. That is,

$$WEL_i = W_i + RES_i - GUR_i$$

where RES_i and GUR_i are the reservation value and the guarantee value of node P_i respectively.

Appendix B

Shivaratri and Krueger's Location Policy

Shivaratri and Krueger's symmetrically-initiated location policy is shown on the next page. Note that it has been rephased to adapt to our system model. However, the essence is no difference from [SK90].

Data Structures

A node i has three ordered lists - $RList_i$, $SList_i$ and $NList_i$. $RList_i$ contains the ids of nodes that have identified themselves to i as potential receivers. $SList_i$ contains the ids of nodes that have identified themselves to i as potential senders. $NList_i$ contains the ids of nodes that identified themselves to i as normally loaded. The information maintained by these lists may not be up-to-date. However, the heads of the lists always contain the most recent information received. The lists are maintained by the location policy and this is discussed later.

Initialization. Initially all nodes assume that every other node is idle and is therefore a receiver. For node i, $RList_i = i{+}1,\ i{+}2,\ ....,\ n,\ 1,\ ...,\ i{-}1$; $SList_i = null$; $NList_i = null$; where n is the number of nodes in the DCS. This ordering for $RList_i$ helps dispersing initial negotiation activity among the nodes.

Sender-Initiated Negotiation

Sender node s does the following.

(1) If $RList_s$ is empty, stop.
Else, probe the node, say j, at the head of $RList_s$, to determine if j is a receiver.

(2) If j identifies itself as a receiver, return the id j to the transfer policy and stop.

[The transfer policy selects the appropriate task for assignment / migration and transfers that task to node j. Assignment always has higher precedence over migration.]

[This is a hit.]

(3) If j is not a receiver, move it to the head of either $SList_s$ or $NList_s$, depending on the reply from j. Start another probing by going to step 1 unless if either s has probed *probe-limit* nodes without success, if $RList_s$ is empty, or if s is no longer a potential sender. *probe-limit* is an algorithm design parameter.

[This is a miss.]

Probed node j does the following.

(1) On receipt of the probing message from sender node s, remove s from whatever list it is in and add it to the head of $SList_j$.

(2) Send a reply message to s indicating the current load state of j, which is determined by the transfer policy.

Receiver-Initiated Negotiation

Receiver node r does the following.

(1) Probe a selected node, j, to determine if it is a sender. j is selected as follows. If $SList_r$ is not empty, j is the first entry in $SList_r$. Otherwise, if $NList_r$ is not empty, j is the last entry in $NList_r$. Otherwise, j is the last entry in $RList_r$.
The contents of the lists may be changed during the current negotiation session because of other negotiations executing in parallel. Nodes that join $SList_r$ in this way may be probed in the current session. Nodes that join $NList_r$ or $RList_r$ in this way are not considered for probing. See [SK90] for details.

(2) If j responds that it is a sender, accept the task relocated from j. Move j to the head of the appropriate list depending on the new load state of j, which is piggybacked on the reply message from j. Then stop.

[We deliberately use the work "relocate" to reveal the fact that the task may be assigned remotely or migrated from node j.]

[This is a hit.]

(3) If j is not a sender, move it to the head of either $RList_r$ or $NList_r$, depending on the reply from j. Start another probing by going to step 1 unless if either r has probed *probe-limit* nodes without success, if all the nodes that might be considered for probing have been probed (see step 1), or if r is no longer a potential receiver.

[This is a miss.]

Probed node j does the following.

(1) If j identifies itself as a sender, and an appropriate task for assignment / migration can be found, transfer the task to r. Piggybacking j's new load state.

(2) If j is not a sender or no appropriate task for assignment / migration can be found, send a reply message to r indicating the current load state of j. Remove r from whatever list it is in and add it to the head of $RList_j$.

$\supseteq$ *END* $\subseteq$