An Introduction to Operating Systems

CHAN Tai-Man

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

Supervised by

Prof. PAN Peter

June 2003

Abstract of thesis entitled:
An Introduction to Operating Systems
Submitted by CHAN Tai-Man
for the degree of Master of Philosophy
at The Chinese University of Hong Kong in June 2003

This is the abstract in no more than 350 words.

Acknowledgement

I would like to thank my supervisor...

This work is dedicated to...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Most existing LD algorithms are *polling-based*. It means that a node attempting to identify a task transfer partner will send a query message to a selected target node to ask for its consent [1, 2, 3, 9, 10, 14, 19]. The two negotiating nodes involved may share load information of each other by explicitly declaring their load states in the polling and reply messages. Alternatively, load information can be deduced from the semantics of messages exchanged. As pollings are one-to-one in nature, a severe weakness of polling-based LD algorithms is that load information exchanged during a polling session is confined to the two negotiating nodes only. Consequently, to spread out load information among the constituent nodes, a lot of polling activities need to be conducted.

In most polling-based LD algorithms, there is a *polling limit* to control the number of polling trials that a node can make during each attempt to locate a transfer partner. Polling limits are usually set to $f \cdot N$, where N is the number of nodes in the

DCS and f is a fractional constant [2, 10]. Another weakness of polling-based LD algorithms relates to the use of polling limits. When the DCS grows large (in terms of N), more and more polling trials are required before a node can successfully pair up a suitable task transfer partner. Accompanied with the larger number of pollings is a higher amount of network bandwidth consumption and CPU overhead. If these algorithm execution costs are not controlled carefully, the performance penalty imposed may break even the performance gain obtained via load distribution. In the worse situation, the performance of the system running the LD algorithm may be worse than the performance of a system without load distribution. We say that polling-based algorithms are *not scalable*.

□ **End of chapter.**

Chapter 2

Background Study

Casavant and Kuhl [1] classified load distribution algorithms into static algorithms and dynamic algorithms. In *static algorithms*, task assignment decisions are made *a priori* during compilation time and are remain unchanged during run-time. In contrast, *dynamic algorithms* attempt to use current system workload information for run-time assignment of tasks to the appropriate nodes.

2.1 Suboptimal

It is now commonly agreed that despite the higher run-time complexity, dynamic algorithms can potentially provide better performance than static algorithms. An important category of dynamic algorithms under Casavant and Kuhl's taxonomy is the *dynamic suboptimal heuristic-based algorithms*. The majority of existing dynamic LD algorithms fall into this category [2, 4, 10, 13, 14, 15, 16, 19]. Due to the intrinsic unavailability of accurate and timely global state information in a distributed

system, targeting for suboptimal performance using heuristic-based algorithms is quite reasonable.

2.1.1 Components of Algorithms

The major components of a dynamic LD algorithm are: (1) the *location policy*, referring to the strategy used to search for a task transfer partner; (2) the *information policy*, referring to the way load information is disseminated among processing nodes; and (3) the *transfer policy*, referring to the strategy used to determine whether task transfer activities should be initiated by a node, either as a task sender or a task receiver.

2.2 Location Policy

Most existing location policies are *polling-based*. A polling session can be started by a heavily loaded node in an attempt to locate a lightly loaded task receiver, or *vice versa*. The polling session in the former case is said to be *sender-initiated* and the latter to be *receiver-initiated* [3, 19]. These two basic approaches can be combined to form a *symmetrically-initiated* algorithm, where polling sessions can be started by both senders and receivers. A study on dynamic LD algorithms by Eager *et al* [3] showed that neither pure sender-initiated nor pure receiver-initiated location policy performs consistently over the whole range of system workload. In [14], Shivaratri and Krueger proposed an adaptive symmetrically-initiated location policy so that sender-initiated pollings only occur at low system workload,

whereas receiver-initiated pollings are conducted whenever appropriate. This location policy shows performance advantage over a wide range of system workload. An important property of this algorithm is that load information gathered during polling activities are retained so as to provide a heuristic guide for subsequent polling activities.

A *workload index* measures how busy a node is. The most commonly used workload index is the number of application tasks residing in a node [5, 6]. Most existing transfer policies are *threshold-based*, which means that a node qualifies as a task sender if its workload index exceeds certain threshold value [2, 9, 12, 14, 17]. In contrast, a node qualifies as a task receiver if its workload index is below the threshold. When the workload index equals to the threshold, the node is regarded as "normally" loaded and no task transfer activity is needed.

□ **End of chapter.**

Chapter 3

Anti-Task Based Loading Algorithms

Chapter Outline

This chapter is the beginning of Part II, the study of a new category of load distribution algorithms. The ideas behind these new algorithms are two fold. First, the use of batch assignments which have been proved in Part I to be advantageous. Second, the use of anti-tasks and load state vectors. This new category of LD algorithms is totally different from the usual polling-based algorithms, and is first proposed by the writer.
In this chapter, we study the basic version of the new algorithms. Its benefits and its weaknesses are both evaluated. In the next two chapters, we discuss further improvements of the algorithms.
The content of this chapter has been published in

- "*Concurrency: Practice and Experience*," 10(14):1251–1269, 1998.
- "*Proceedings, The ? ICDCS*,"

Most existing LD algorithms are *polling-based*. It means that a node attempting to identify a task transfer partner will send a query message to a selected target node to ask for its consent [1, 2, 3, 9, 10, 14, 19]. The two negotiating nodes involved may share load information of each other by explicitly declaring their load states in the polling and reply messages. Alternatively, load information can be deduced from the semantics of messages exchanged. As pollings are one-to-one in nature, a severe weakness of polling-based LD algorithms is that load information exchanged during a polling session is confined to the two negotiating nodes only. Consequently, to spread out load information among the constituent nodes, a lot of polling activities need to be conducted.

In most polling-based LD algorithms, there is a *polling limit* to control the number of polling trials that a node can make during each attempt to locate a transfer partner. Polling limits are usually set to $f \cdot N$, where N is the number of nodes in the DCS and f is a fractional constant [2, 10]. Another weakness of polling-based LD algorithms relates to the use of polling limits. When the DCS grows large (in terms of N), more and more polling trials are required before a node can successfully pair up a suitable task transfer partner. Accompanied with the larger number of pollings is a higher amount of network bandwidth consumption and CPU overhead. If these algorithm execution costs are not controlled carefully, the performance penalty imposed may break even the performance gain obtained via load

distribution. In the worse situation, the performance of the system running the LD algorithm may be worse than the performance of a system without load distribution. We say that polling-based algorithms are *not scalable*.

In this chapter, we propose a new LD algorithm which is based on *anti-tasks* and *load state vectors*. This new algorithm avoids the weaknesses of polling-based LD algorithms discussed above. Anti-tasks are composite agents which travel around a DCS to facilitate the pairing up of task senders and receivers, as well as the collection and dissemination of load information. Time-stamped load information of processing nodes is stored in load state vectors which, when used together with anti-tasks, encourage mutual sharing of load information among processing nodes. Anti-tasks, which make use of load state vectors to decide their traveling paths, are spontaneously directed towards processing nodes having high *transient workload*, thus allowing their surplus workload to be relocated quickly to lightly loaded nodes.

Using simulations, we evaluate the performance of our new algorithm by comparing its performance with a number of well-known polling-based LD algorithms. We found that our algorithm provides significant reduction of mean task response time over a large range of system sizes from 25 to 70 processing nodes. The cost for achieving this performance gain in terms of CPU overhead and channel bandwidth consumption is generally comparable to the other algorithms we studied. All these nice properties of our algorithm can be attributed to the fact that

anti-tasks spontaneously travel towards those nodes with high transient workload. The direct consequence is that the available spare processing capacity (as represented by the anti-tasks) is immediately available to the busy nodes so that their surplus workload can be offloaded soonest possible.

3.1 The New Algorithm

A distributed system containing N autonomous processing nodes $P_1, P_2, \ldots, P_N$ is assumed. The system has no shared memory support and message passing is the only means of communication between the nodes. The communication network is fault free and strongly connected, and messages sent across the network are received in the order sent.

We further assume that each node P_i maintains its own local clock C_i. The system runs a distributed clock synchronization algorithm which ensures that the largest difference between the readings of any two clocks C_i and C_j, $1 \leq i, j \leq N$, is not greater than a maximum clock skew ξ. Readers are referred to [7] and [18] for discussions on clock synchronization algorithms.

3.1.1 Load State Monitoring

We define three different *load states* to represent how busy a node is. The load state of P_i is stored in a variable LS_i. The criteria for setting the value of LS_i are as follows:

$$LS_i = \begin{cases} \textbf{HEAVY} & \text{if } Q_i \geq T_{up} \\ \textbf{NORMAL} & \text{if } T_{up} > Q_i \geq T_{low} \\ \textbf{LIGHT} & \text{Otherwise} \end{cases} \tag{3.1}$$

where Q_i is the total number of application tasks residing on P_i; T_{up} is the *upper threshold*; and T_{low} is the *lower threshold*. T_{up} and T_{low} are algorithm design parameters which define the boundary for the **NORMAL** state. There are two kinds of events that can potentially change the value of Q_i, namely, task arrivals and task completions. Note that depending on the current value of Q_i, not all task arrival/completion events will induce a change on LS_i. This avoids a node from switching between the three load states frequently and thus avoids unnecessary load distribution activities. P_i runs a *load state monitor* m_i to monitor Q_i and update LS_i accordingly if necessary.

A node in the **HEAVY** state is regarded as overly busy and is eligible to transfer its task(s) away to other nodes in the **LIGHT** state, if there is any. In other words, a node in the **HEAVY** state is a potential *task sender*, while a node in the **LIGHT** state is a potential *task receiver*. The principle challenge of a LD algorithm is to pair up task senders and receivers using the minimum execution cost.

Each node P_i, $1 \leq i \leq N$, maintains a local view of the load states of all other nodes in the system by using a load state vector, denoted by LV_i.

Definition 3.1

For each P_i, $1 \leq i \leq N$, the load state vector LV_i is a N-dimensional vector where each vector component $LV_i[j]$, $1 \leq j \leq N$, is an ordered pair (CV, LS). $LV_i[j].LS$ records the load state of P_j and $LV_i[j].CV$ records the value of clock C_j at which the load state was announced by P_j.
□

Vector component $LV_i[j]$ stores the latest load state of P_j that P_i has received. It represents the local view of P_i on P_j's "current" load state. Thus, load state vector LV_i represents P_i's local view of the load states of the entire DCS. We use the ith component of P_i's load state vector to record P_i's own load state. That is,

$$\left.\begin{array}{rcl} LV_i[i].CV & \equiv & C_i(*) \\ LV_i[i].LS & \equiv & LS_i \end{array}\right\} \text{for all } \ 1 \leq i \leq N \qquad (3.2)$$

where $C_i(*)$ denotes the reading of C_i at which m_i last recorded a load state change.

3.1.2 Anti-Tasks

We now describe the role and the life cycle of an anti-task. An anti-task A created by a lightly loaded node P_c is designated with a certain amount of processing capacity, denoted by $A.capacity$. It represents the amount of workload that P_c is will-

ing to accept from other busy nodes. [1] P_c is called the *creator node* of A. After its creation, the anti-task travels around the nodes in the system. The order in which the nodes are visited depends on a data structure attached to the anti-task called a *trajectory*, whose formal definition is given below:

Definition 3.2

The trajectory of an anti-task A, denoted by $A.trajectory$, is a N-dimensional vector where each vector component $A.trajectory[i]$, $1 \leq i \leq N$, is an ordered 3-tuple (CV, LS, VF). Tuple element $A.trajectory[i].LS$ denotes the load state of P_i and $A.trajectory[i].CV$ denotes the value of clock C_i at which the load state was announced by P_i. Tuple element $A.trajectory[i].VF$ is a boolean variable which records whether P_i has already been visited by A and is thus called the visit flag.
□

A trajectory is similar to a load state vector, except the use of visit flags. The role of visit flags will be discussed in the next sub-section. In brief, a trajectory assumes two functionalities: (1) to identify appropriate destinations for an anti-task to travel to; and (2) to store load state information collected as an anti-task travels. For fulfilling these functions, a trajectory is dynamically updated as the anti-task travels in the system.

[1] The amount of workload is measured in number of tasks. Other measurement units (such as CPU time) is possible with a corresponding modification of the load state definitions (Section 3.1.1) and appropriate load thresholds T_{up} and T_{low}. The use of different load state measurement schemes does not affect the core part of our algorithm.

When the anti-task A reaches a node P_v, load state information carried by its trajectory (recorded in $A.trajectory[\,].LS$ and $A.trajectory[\,].CV$) is compared with P_v's load state vector, LV_v, for mutual update. By doing so, both A and P_v can acquire more recent load state information. After the mutual update, if A finds that its creator node P_c is no longer lightly loaded (i.e. $A.trajectory[c].LS \neq$ **LIGHT**), A completes its travel by returning to P_c. Otherwise, it tries to negotiate with P_v for any possible task transfer activity.

If P_v is in the **HEAVY** state, one or more tasks will be relocated from P_v to P_c. The number of tasks to be relocated depends on both the anti-task's capacity, and the amount of surplus workload that P_v has. After the task relocation, $A.capacity$ is decremented by the number of tasks relocated. This records the fact that P_c now has higher workload and thus has fewer spare capacity. The next action of the anti-task depends on the resulting value of $A.capacity$. If $A.capacity$ reduces to zero, the anti-task returns to P_c, where load state information captured by $A.trajectory$ is used to update LV_c. Otherwise, the anti-task continues its travel to the next appropriate destination node (if any), the selection of which depends on the updated $A.trajectory$ and will be discussed in the next sub-section.

On the other hand, if P_v is not **HEAVY**, no task relocation can happen between P_v and P_c. The anti-task either travels to the next destination or returns to the creator node.

From the above discussion, it is clear that an anti-task is a composite agent having three components: (1) $A.creator$,

which records the *id* of its creator node; (2) $A.capacity$; and (3) $A.trajectory$.

3.1.3 Anti-Task Handling Procedures

The load distribution algorithm is composed of three procedures called CREATION, NODE_VISIT, and TERMINATION. CREATION is invoked when a lightly loaded node creates an anti-task; NODE_VISIT is invoked when a node is being visited by an anti-task; and TERMINATION is invoked when an anti-task returns to its creator node.

Procedure CREATION

Figure 3.1 (Page 16) shows the procedure CREATION. After anti-task A is created, A is attached with $A.creator$ and $A.capacity$, as stated in lines 1 and 2, respectively. $A.capacity$ is initialized to $T_{up} - T_{low}$ for the following reason. According to the load state definitions given in Section 3.1.1, the workload of creator node P_c (i.e. Q_c) must be smaller than T_{low} for the load state of P_c (i.e. LS_c) to be **LIGHT**. Therefore, we can approximate the additional amount of workload that P_c can receive without turning LS_c into the **HEAVY** state as $T_{up} - T_{low}$. This approximation relies on the assumption that there is no significant elevation of P_c's workload due to local task arrivals before the anti-task finishes its travel and returns to P_c. Should this assumption be violated, a mechanism must be provided to enable A to cease its travel and return to P_c, thus stopping any more

remote tasks to P_c. This mechanism is discussed later when we describe procedure NODE_VISIT.

The trajectory is initialized in lines 3–7. where corresponding components of LV_c are copied to $A.trajectory$. In line 6, visit flags are all initialized to FALSE to record the fact that no remote node has been visited by A so far.

Lines 8 and 9 try to identify a node in the **HEAVY** state. If there exists more than one suitable trajectory entry, the one with the largest CV is selected. In other words, we select the node *most recently* showing the **HEAVY** state. If no trajectory entry shows the **HEAVY** state, the algorithm tries to identify the node in the **NORMAL** state *least recently* as shown in lines 13 and 14, hoping that the node has turned to the **HEAVY** state. Similarly, if no trajectory entry shows the **NORMAL** state, lines 18 and 19 identify the node in the **LIGHT** state *least recently*.

At last, the anti-task is forwarded to the selected destination node in line 20.

Procedure NODE_VISIT

Figure 3.2 (Page 19 shows the procedure NODE_VISIT executed by a node P_v when it is visited by anti-task A. In lines 1–9, P_v mutually updates LV_v and $A.trajectory$ by pairwise comparisons of corresponding components. The key is to compare the clock values $LV_v[i].CV$ and $A.trajectory[i].CV$ in line 2. The one with the larger clock value (i.e. more recent) is used to update the other one. By such mutual update, P_v obtains the

```
PROCEDURE {\sc Creation} \\
%
// $P_c$ = the node creating the anti-task $A$ \\[2mm]
%
BEGIN \\
1 \> $A.creator \longleftarrow P_c$  \\
2 \> $A.capacity \longleftarrow T_{up} - T_{low}$  \\
3 \> FOR $i = 1$ to $N$ DO \\
4 \>\> $A.trajectory[i].CV \longleftarrow LV_c[i].CV$  \\
5 \>\> $A.trajectory[i].LS \longleftarrow LV_c[i].LS$  \\
6 \>\> $A.trajectory[i].VF \longleftarrow \mbox{\bf FALSE}$  \\
7 \> ENDDO  \\[2mm]
%
  \> // select first destination for anti-task $A$ to visit \\
8 \> FOR ALL $j$, such that $A.trajectory[j].LS =$ HEAVY \\
9 \>\> Identify $k$, such that
$A.trajectory[k].CV = \max \{ A.trajectory[j].CV \}$ \\
10 \> IF success \\
11 \>\> GOTO line 20 \\
12 \> ENDIF \\[2mm]
%
13 \> FOR ALL $j$, such that $A.trajectory[j].LS =$ NORMAL \\
14 \>\> Identify $k$, such that
$A.trajectory[k].CV = \min \{ A.trajectory[j].CV \}$ \\
15 \> IF success \\
16 \>\> GOTO line 20 \\
17 \> ENDIF \\[2mm]
%
18 \> FOR ALL $j$, such that $A.trajectory[j].LS =$ LIGHT \\
19 \>\> Identify $k$, such that
$A.trajectory[k].CV = \min \{ A.trajectory[j].CV \}$ \\[2mm]
%
20 \> Forward $A$ to $P_k$  \\
END
```

Figure 3.1: Procedure CREATION — A lightly loaded node p_c creates an anti-task A.

latest load state information collected by the anti-task from its previous destinations. The anti-task can also bring itself up-to-date, as P_v may contain more recent load state information obtained from other anti-tasks visited P_v earlier. We call any alteration to the trajectory which involves a change of tuple element $A.trajectory[\,].LS$ a *mutation*. Mutation helps to quickly direct the anti-task towards those nodes showing heavy transient workload (i.e. in the **HEAVY** state), or away from those nodes recently recorded as **LIGHT** or **NORMAL**.

Lines 10–13 are used to determine whether A should return to its creator node by examining $A.trajectory[A.creator].LS$, which may have been updated with P_v's load state vector in lines 3–4.

In lines 14–18, P_v tries to offload some surplus workload to the creator node of A. It can do so only if it is in the **HEAVY** state. The number of tasks that P_v can offload is limited by the remaining capacity, $A.capacity$. After each task transfer, $A.capacity$ is decremented in line 16. P_v's load state is also adjusted accordingly by its load state monitor m_v in line 17.

Lines 19–20 update P_v's entry in $A.trajectory$ since the load state of P_v may be modified due to task relocations. In line 21, the visit flag $A.trajectory[v].VF$ is set to TRUE to reveal the fact that node P_v has already been visited by the anti-task. This effectively prevents the anti-task from coming back to P_v again, unless it finds that P_v has turned to the **HEAVY** state again later (refer to lines 26–27).

In line 22, $A.capacity$ is examined. Should it reduces to zero,

all the spare capacity promised by the creator node has been consumed. The anti-task therefore ceases its travel and returns to the creator node as shown in lines 23–24.

Conversely, if $A.capacity$ remains greater than zero, the anti-task continues its travel. Lines 26–37 identify the next destination node. The steps here are similar to those in procedure CREATION (lines 8–19) except the additional checks on the visit flag and $P_k \neq A.creator$. If an appropriate destination node cannot be found, the anti-task has no choice but returns to its creator as shown in line 39. This happens when all the nodes have been visited by the anti-task and no **HEAVY** node can be found. At last, the anti-task is forwarded to its next destination in line 41.

Procedure TERMINATION

Figure 3.3 (Page 24) shows the procedure TERMINATION executed by the creator node P_c when its anti-task A returns. The anti-task returns to P_c either because (1) no more appropriate destination node can be selected; (2) the anti-task has been notified that P_c has turned to **HEAVY** or **NORMAL**; or (3) the anti-task's capacity has reduced to zero. Regardless of the reason for A's return, the only thing that P_c needs to do is to update its load state vector using load state information stored in A's trajectory.

```
PROCEDURE {\sc Node\_Visit} \\
%
// $P_v$ = the node being visited by anti-task $A$ \\[1.5mm]
%
BEGIN \\
\> // Mutual update of load state information \\
1 \> FOR $i=1$ to $N$ DO \\
2 \>  \> IF $LV_v[i].CV > A.trajectory[i].CV$ \\
3 \>\>\> $A.trajectory[i].CV \longleftarrow LV_v[i].CV$  \\
4 \>\>\> $A.trajectory[i].LS \longleftarrow LV_v[i].LS$  \\
5 \>\> ELSE \\
6 \>\>\> $LV_v[i].CV \longleftarrow A.trajectory[i].CV$  \\
7 \>\>\> $LV_v[i].LS \longleftarrow A.trajectory[i].LS$  \\
8 \>\> ENDIF  \\
9 \> ENDDO  \\[1.5mm]
%
\> // Is creator node still lightly loaded ? \\
10 \> IF $A.trajectory[A.creator].LS \neq$ LIGHT \\
11 \>\> Forward $A$ to $A.creator$   \\
12 \>\> STOP  \\
13 \> ENDIF  \\[1.5mm]
%
\> // Offload any surplus workload  \\
14 \> WHILE $LV_v[v].LS =$ HEAVY and $A.capacity > 0$ DO \\
15 \>\> Transfer a task from $P_v$ to $A.creator$  \\
16 \>\> $A.capacity \longleftarrow A.capacity - 1$  \\
17 \>  \> Update $LV_v[v].LS$ and $LV_v[v].CV$ by $m_v$
   \hspace{5mm}
   // not a responsibility of the load distribution algorithm \\
18 \> ENDDO  \\[1.5mm]
%
\> // Update anti-task's record on $P_v$'s new load state \\
19 \> $A.trajectory[v].CV \longleftarrow LV_v[v].CV$  \\
20 \> $A.trajectory[v].LS \longleftarrow LV_v[v].LS$  \\
21 \> $A.trajectory[v].VF \longleftarrow$ TRUE  \\[1.5mm]
END
```

Figure 3.2: Procedure NODE_VISIT — A node P_v is visited by an anti-task A.

3.2 Simulation Experiments

In this chapter, simulations are used to evaluate the performance of our anti-task algorithm. We have derived a comprehensive simulation model. In particular, the costs for running LD algorithms, the intra-node scheduling of tasks, and the consumption of network bandwidth have all been carefully modeled. Besides, we have collected and analyzed various simulation statistics which are listed in Table 3.1 (Page 24). We believe that our simulation model is sophisticated enough for comprehensive performance comparisons between our anti-task algorithm and a number of well known LD algorithms we have also studied.

3.2.1 Simulation Model

Each processing node consists of three task queues: the **Local Queue**, the **Remote Queue**, and the **Preemption Queue**. When a task arrives locally, it goes to the end of the Local Queue where it waits for the CPU in the First-Come-First-Serve (FCFS) basis. Tasks in the Local Queue are candidates for remote assignment when the node becomes busy (i.e. in the **HEAVY** state).

Tasks transferred from other nodes are appended to the receiver's Remote Queue. The Remote Queue separates remotely transferred tasks from those locally arrived tasks in the Local Queue so that *task reassignment* is avoided. This prevents a task from continuous shifting among the nodes in the system, in which case unnecessary CPU and communication overhead

would incur and the response time of the shifting task may become exceedingly high. The Preemption Queue allows "overhead tasks" created by LD algorithms to wait for the CPU. As overhead tasks are usually small and time critical, they always preempt the application task in execution (if there is one). The preempted task can resume its execution only when all pending overhead tasks in the Preemption Queue are completed, including those generated during the course of its suspension.

We assume a token-ring network having a bandwidth of 10 Mbits/sec. We select the token-ring network for the sake of simplifying the simulation program. In fact, the use of different network models does not normally have any observable effect on the comparative simulation results, unless the network bandwidth is severely limited.

3.2.2 Algorithm Costs and Simulation Parameters

Many studies on dynamic LD algorithms made an overly simplified assumption that the execution and communication overhead due to load distribution activities are negligible. Kremien and Kramer pointed out, however, that this assumption does not reflect runtime situations realistically [6]. We believe that modeling both CPU overhead and network bandwidth consumptions due to load distribution activities at the appropriate level of complexity is essential to a correct assessment of LD algorithms.

Anti-Task Creation/Processing CPU Overhead

The amount of CPU overhead incurred when a processing node creates, receives, or forwards an anti-task is modeled as follows:

$$\text{Anti-task processing CPU overhead} = \delta \cdot N \tag{3.3}$$

where δ is a constant and N is the number of nodes in the system. The larger the number of nodes in the system and thus the longer the trajectory of an anti-task, the larger the processing overhead needed.

Anti-Task Traveling Delays

The communication delay involved in forwarding an anti-task from one processing node to another depends on the message length, which is modeled as follows.

$$\text{Anti-task message length} = 21 + 5N \quad \text{bytes.} \tag{3.4}$$

For each trajectory entry, we assume that 4 bytes are needed for the tuple element CV (clock value), half byte for LS (load state), and another half byte for VF (visit flag). Thus, 5 bytes are needed for each of the N trajectory entries, giving a total of $5N$ bytes for the entire trajectory. We assume that both $A.creator$ and $A.capacity$ require half byte, giving a total of 1 byte. We also assume that 20 additional bytes are needed by the headers and checksums for the entire anti-task. Thus, additional 21 bytes are needed in each anti-task.

Polling Cost

This applies on polling-based algorithms only. Sending or receiving a polling message incurs a CPU overhead of 3 millisecond. All polling messages are assumed to have a fixed size of 20 bytes.

Task Transfer CPU Overhead

The amount of CPU overhead imposed when a task j is transferred remotely depends on the length(in Kbytes) of the task transfer message generated. This is referred to as the *code length* of task j. The CPU overhead is thus modeled as follows:

$$\text{Task transfer CPU overhead for task } j = \Delta \cdot l_j \tag{3.5}$$

where Δ is the amount of CPU overhead imposed per Kbyte of task code transferred; and l_j is the code length of task j. In our simulations, a typical value of Δ is 3ms per Kbyte. Thus, for a task having a code length of 15 Kbytes, both the sender and the receiver of the task are imposed with a CPU overhead of 45 ms. To characterize the fact that some tasks may have a longer code length than others, an independent exponential distribution is used for task code lengths within each processing node. The mean of this exponential distribution in node P_i is denoted by $\bar{l}_i$. Both Δ and $\bar{l}$ are simulation parameters.

The values of the simulation parameters used are summarized in Table 3.1 (Page 24).

```
PROCEDURE {\sc Termination} \\
%
// $P_c$ = the node created the anti-task $A$, i.e. $P_{A.creator}$ \\[2mm]
%
BEGIN \\
        \> // Update $P_c$'s load state vector \\
1       \> FOR $i=1$ to $N$ DO \\
2       \>      \> IF $A.trajectory[i].CV > LV_c[i].CV$ \\
3       \>      \>      \> $LV_c[i].CV \longleftarrow A.trajectory[i].CV$  \\
4       \>      \>      \> $LV_c[i].LS \longleftarrow A.trajectory[i].LS$  \\
5       \>      \> ENDIF  \\
6       \> ENDDO  \\[2mm]
%
7 \> Destroy $A$  \\
END
```

Figure 3.3: Procedure TERMINATION — An anti-task returns to its creator node.

Table 3.1: Simulation parameters.

Parameter	*Value*
Token ring bandwidth	10 Mbits per sec
Lower threshold, T_{low}	5
Upper threshold, T_{up}	15
Anti-task processing CPU overhead factor, δ	0.1 ms
Task transfer overhead factor, Δ	3 ms per Kbyte
Mean task code length, $\bar{l}_i$	10 Kbytes, $\forall 1 \leq i \leq N$

3.2.3 Simulation Results and Analysis

This section presents our first set of simulation experiments which serves two main purposes: (1) to study the traveling pattern of anti-tasks; and (2) to study if anti-tasks can identify **HEAVY** nodes efficiently. In these experiments, a small system ($N = 10$ nodes) is studied.

Our anti-task algorithm is labeled as ANTI. In addition to algorithm ANTI, we evaluate three polling-based algorithms — S.EAGER, R.EAGER, and SYM.EAGER. S.EAGER is purely *sender-initiated*, meaning that only a heavily loaded node can start a polling session in an attempt to identify a task receiver. R.EAGER is purely *receiver-initiated*, meaning that only a lightly loaded node can start a polling session. In both algorithms, polling targets are identified on a random basis. Thus, no load state information is used to identify a polling target, nor need to be collected. These two algorithms are classical and they have been extensively studied by Eager *et al* in [3]. The third algorithm, SYM.EAGER, is a combination of the above two and is thus *symmetrically-initiated*. The simplicity of these three algorithms' random polling target selections allows properties of polling-based algorithms to be revealed clearly. In the next sub-section, we will compare the ANTI algorithm with a more sophisticated adaptive polling-based LD algorithm, which makes use of load state information in identifying polling targets.

Figure 3.3 (Page 24) presents the performance of the ANTI algorithm. It shows for each node a time trace on the in-

stantaneous task queue length (QL — sum of local queue and remote queue), and the number of anti-tasks visiting a node per unit time, referred to as the *anti-task density* (AD). Correspondingly, Figure 3.3 (Page 24) presents the performance of the SYM.EAGER algorithm. It shows for each node a time trace on the instantaneous task queue length and the number of pollings received by a node per unit time, referred to as the *polling density* (PD). The workload pattern subject to the system is identical in both cases. The time traces for S.EAGER and R.EAGER are not shown because their individual properties can be more or less reflected by the SYM.EAGER algorithm. Besides, SYM.EAGER provides the best performance among the three polling-based algorithms (see Table 3.2, Page 31).

Anti-Task Traveling Patterns

Figure 3.3 shows that anti-tasks travel *spontaneously* towards those nodes with high transient workload. For example, during time 30–85 node 1 becomes heavily loaded because of rapid local task arrivals. Such workload elevation is reflected in the increase in node 1's task queue length (QL). During this period, *all* the anti-tasks in the system travel only to node 1 but not to any of the others, which are all lightly or normally loaded. Similar situations occur at time 150 (at node 2), time 400 (at nodes 1 and 4), time 800 (at node 2), and time 1000 (at node 8). Note that such anti-task traveling pattern occurs regardless the magnitude of the transient workload elevation of the busy node. A direct consequence of this traveling pattern is that all the avail-

able spare processing capacity (represented by the anti-tasks) are *immediately* available to the busy node. This is in contrast to polling-based algorithms where receiver nodes are identified one-by-one — another receiver will be searched for only after the amount of spare capacity of the current receiver has found to be inadequate. Therefore, with the ANTI algorithm, surplus workload of the busy node can be offloaded soonest possible.

Let us now examine the case when multiple busy nodes exist simultaneously. At time 400, both nodes 1 and 4 are subject to transient elevation of workload. We found that anti-tasks are split between these two busy nodes. It happens that slightly more anti-tasks travel to node 4 initially, making the congestions at node 4 to be resolved more quickly than at node 1. Once the congestion at node 4 has been resolved at time 470, the anti-tasks all rush to node 1. This traveling pattern is illustrated in Figure 3.3 as complementary changes in anti-task density between node 1 and node 4 during time 400-550. Note that the anti-tasks do not travel to any other (non-**HEAVY**) nodes during this period.

On the other hand, we found that when there is no busy nodes, the anti-tasks distribute evenly among the nodes in the system in an attempt to search for any busy nodes and to disseminate load state information in doing so. All the anti-task traveling patterns discussed above clearly reveal algorithm ANTI's high adaptability towards changes in system workload.

Polling Patterns

Figure 3.3 shows the time traces on instantaneous queue length and polling density when SYM.EAGER is applied. The system is subject to the same workload pattern as in ANTI. We can see that regardless of the existence of busy nodes, polling messages are always evenly distributed among the nodes in the system. This is because SYM.EAGER does not intrinsically propagate workload information so that each individual node has to search for its polling targets separately. The probability for a node to be polled by a busy node is identical among all the nodes (including those busy nodes), thus the polling density is evenly distributed. Such polling pattern causes two immediate performance penalties: (1) sender-receiver pairings are not efficient and thus surplus workload in busy nodes cannot be resolved quickly; and (2) large amount of pollings are conducted and the associated overhead are incurred to the system.

Performance Comparisons

We can see by comparing Figures 3.3 and 3.3 that algorithm ANTI reduces task congestions at busy nodes more quickly than algorithm SYM.EAGER does. For example, in Figure 3.3, we can see that the transient workload elevation occurring in node 2 at time 800 cannot be resolved until time 1120, in contrast to time 950 when ANTI is used. That is, SYM.EAGER needs more than 2 times the time needed by ANTI. During the period when node 2 is heavily loaded, receiver-initiated pollings addressed to

nodes other than node 2 are all wasted because only node 2 is a potential sender during this period. This implies that (1) CPU overhead and communication channel bandwidth are wasted for the fruitless pollings; and (2) spare processing capacity in those lightly loaded nodes cannot be utilized to process those surplus workload at node 2.

Table 3.2 shows some simulation statistics averaged over the entire simulation period. We can see that ANTI provides the lowest mean task response time (29.9) among the four algorithms, while the second lowest is provided by SYM.EAGER (49.3). In other words, ANTI provides a performance advantage of 39% over SYM.EAGER. ANTI's performance advantage is obviously due to its ability to react to a node's transient workload elevation quickly so that congested tasks at the busy node can be efficiently transferred to other lightly loaded nodes and get processed earlier. This is reflected in ANTI's highest percentage of tasks remotely executed and lowest mean task queue length.

Intuitively, algorithm ANTI should consume a larger CPU overhead than the other three polling-based algorithms because each anti-task may need to travel a number of nodes before it can return to its creator. However, from Table 3.2, we can see that the amount of CPU overhead spent on negotiation activities (anti-task travelings in the case of ANTI and pollings in the other three algorithms) are comparable. This can be explained by the following two factors which suppress the amount of anti-task travels: (1) the arrival of an anti-task to a node delays the node's necessity to create a new anti-task on its own; and (2)

the arrival of a single anti-task to a busy node may induce multiple task relocations (when its capacity is greater than one), thus reducing the number of anti-tasks needed totally. This is in contrast to polling-based algorithms where each successful sender-receiver negotiation induces only one task to be transferred usually. Among the three polling-based algorithms, SYM.EAGER spent the largest amount of CPU overhead on negotiation activities as it adopts both sender-initiated and receiver-initiated pollings.

From Table 3.2, we also find that ANTI spends the largest amount of CPU overhead on task transfer activities. To provide a fair comparison, we divide "*CPU overhead spent on task transfer*" by "*percentage of tasks remotely executed*" to give "*CPU overhead per % task remotely executed*." We can see that this amount is comparable in all four algorithms.

Furthermore, among the three polling-based algorithms, SYM.EAGER consumes the largest amount of channel bandwidth as it employs both sender-initiated and receiver-initiated pollings. ANTI consumes slightly more channel bandwidth than SYM.EAGER. The extra bandwidth consumption can be attributed to its larger amount of remote executions.

As the three Eager's algorithms evaluated above do not use any load state information in the selection of polling targets, they are *non-adaptive* to changes in system workload. Algorithm ANTI's better performance is easily expected as it collects and makes use of load state information. It is essential for us to also compare ANTI with other adaptive polling-based

Table 3.2: Performance comparisons between the four LD algorithms.

Performance Metrics	S.EAGER	R.EAGER	SYM.EAGER	ANTI
Mean task response time	72.3	77.5	49.3	29.9
Mean task queue length	59.7	49.9	29.1	17.4
Percentage of tasks remotely executed	23.7	36.0	42.9	47.6
Net CPU utilization	52.9	57.0	59.4	59.5
CPU overhead spent on negotiation (%)	0.04	0.06	0.10	0.04
CPU overhead spent on task transfer (%)	0.74	1.21	1.50	1.67
Total CPU overhead (%)	0.78	1.27	1.60	1.71
CPU overhead per % task remotely executed (%)	0.033	0.035	0.037	0.036
Communication bandwidth utilization (%)	1.0	1.6	2.0	2.2

algorithms which make use of load state information. We select a symmetrically-initiated algorithm proposed by Shivaratri and Krueger in [14] for this purpose. This algorithm, labeled as SK, has proven to provide good system performance at the expense of a relatively small amount of pollings [10, 14]. In this set of simulation experiments, we also focus to study the *scalability* of the ANTI algorithm by employing different system sizes, from $N = 10$ nodes to $N = 70$ nodes.

Table 3.3 (Page 32) shows the performance comparison between the ANTI and the SK algorithms. We define the *improvement factor* as the percentage improvement of mean task response time, RT, given by ANTI over that of SK, that is,

$$\text{Improvement Factor} = \frac{RT(SK) - RT(Anti)}{RT(SK)} \text{x } 100\%$$

A positive improvement factor denotes a performance improve-

ment while a negative value implies a performance degradation.

Table 3.3: Performance comparisons between ANTI and SK across different system sizes.

Number of nodes, N		*10*	*20*	*30*	*40*	*50*	*60*	*70*
Mean task response time	SK	28.7	20.6	22.6	22.6	25.6	30.1	30.8
	ANTI	29.1	20.9	21.9	19.8	23.8	28.3	31.4
	Imp. Fac. (%)	-2.0	-1.7	+3.0	+12.6	+7.2	+5.8	-2.1
CPU overhead on negotiations (1%)	SK	0.04	0.07	0.08	0.12	0.10	0.09	0.06
	ANTI	0.41	1.71	2.95	5.20	5.21	4.68	3.02
Channel utilization (%)	SK	2.2	4.5	8.7	10.0	15.8	18.5	21.7
	ANTI	2.3	5.3	10.6	14.1	20.9	23.8	25.1

From Table 3.3, we found that ANTI gives a positive improvement factor for system sizes 30 to 60 nodes. By interpolation, we can expect ANTI to perform better than SK when the system contains 24 to 68 nodes. The trend is that the performance of ANTI relative to SK continues to improve as the system size increases. The best performance is achieved when the system contains around 40 nodes, where more than 12% improvement can be obtained. Afterwards, the improvement diminishes. When the system is larger than 68 nodes, a performance degradation occurs. The trend is explained as follows.

When the system size is small, it is easy for a node to identify an appropriate transfer partner by using pollings. Thus, the use of polling-based algorithm is already good enough. The higher CPU overhead incurred by ANTI cannot be justified, thus

ANTI's slightly poorer performance is resulted. As the system size increases, load state information cannot be propagated efficiently by using pollings. It becomes difficult for a node to locate an appropriate transfer partner when the SK algorithm is used. Consequently, congested tasks in busy nodes are slow to be relocated. In contrast, ANTI allows the large number of nodes to share their load state information so that anti-tasks are quickly forwarded to those busy nodes. Task congestions can therefore be resolved quickly. However, when the system size grows further, the trajectory of an anti-task becomes longer, meaning that (1) larger CPU overhead is needed for processing the trajectory; and (2) anti-tasks spend more time to travel in the system and uses more channel bandwidth. This is reflected by ANTI's increasing channel bandwidth utilization as system size increases. The higher CPU overhead consumed by the anti-tasks relative to the polling activities in SK overwhelms the performance gain, leading to ANTI's poorer performance.

□ **End of chapter.**

Chapter 4

Conclusion

A load distribution algorithm based on anti-tasks and load state vectors is proposed. Anti-tasks are composite agents which travel around the distributed system to facilitate the pairing up of task senders and receivers as well as the collection and dissemination of load information. Time-stamped load information is stored in load state vectors which when used with anti-tasks, encourages mutual sharing of load information among processing nodes. The most important property of the algorithm is that anti-tasks are spontaneously directed towards processing nodes with high transient workload, thus allowing their surplus workload to be relocated quickly.

We found from simulation experiments that our algorithm provides significant reduction of mean task response time over a large range of system sizes. The cost for achieving this performance gain in terms of CPU overhead and channel bandwidth consumption is generally comparable to the other algorithms we studied.

On the other hand, we are currently working on an improvement of the anti-task algorithm by employing some *proxy nodes* in the distributed system to store load state information. Each proxy node serves as a mediator to collect and distribute load state information for a subset of nodes. By visiting a proxy node, an anti-task does not need to visit all the nodes in the system in order to acquire load information of the whole system. By then, the CPU overhead and communication cost can be limited.

□ **End of chapter.**

Appendix A

What is an Operating System?

What is an Operating System?

An *operating system* is a program that acts as an *intermediary* between a user of a computer and the computer hardware. The purpose of an operating system is to provide the environment in which a user can execute programs. The primary goal of an operating system is thus to make the computer system ***convenient*** to use. A secondary goal is to use the computer hardware in an ***efficient*** manner.

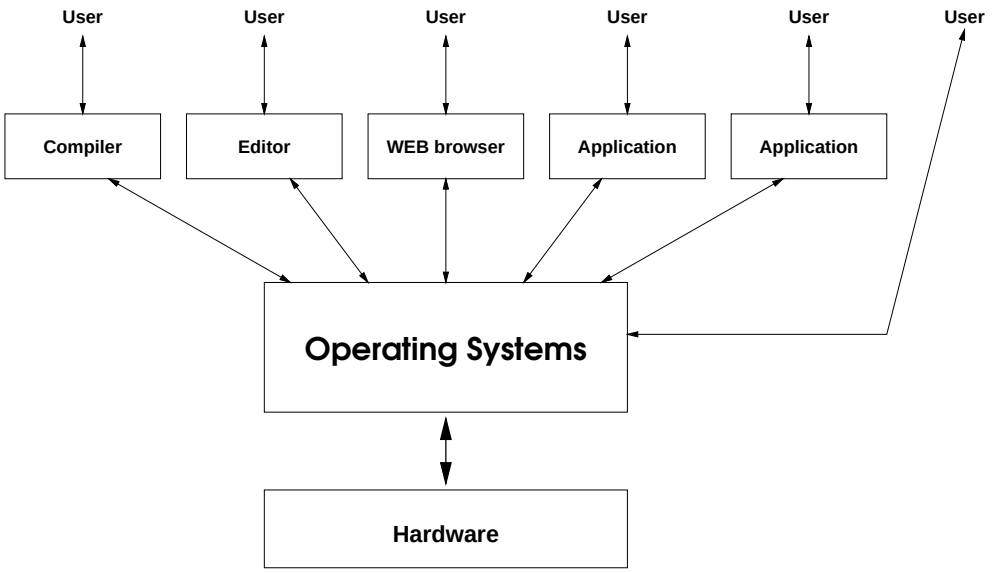

Figure A.1: Abstract view of the components of a computer system.

- A computer system can be divided roughly into four components: the hardware, the operating system, the application programs, and the users.
  - The hardware — CPU, memory, I/O devices, etc. — provides the basic computing resources.
  - The applications — compilers, database systems, games, business programs — defines the ways in which the hardware resources are used to solve the computing problems for the users.
- Examples
  - Mainframe computers — IBM OS/360, ...
  - Minicomputers — Unix (HP-UX, Ultrix, AIX), OS/400, HP MPV, VAX OS, AOS, ...

- Personal computers — CP/M, MS-DOS, MS-Windows, Mac OS, Linux, BeOS, ...
- Palm tops — Windows-CE, Palm OS, ...

Roles of an Operating System

The fundamental goal of computer systems is to execute user programs and to make solving user problems easier. Since bare hardware alone is not particularly easy to use, application programs are developed. These various programs require *certain common operations*, such as those controlling the I/O devices. The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system [8] [11].

- An operating system is similar to a *government* — The operating system controls and coordinates the use of the hardware among the various application programs for the various users.
- An operating system can be viewed as a *resource allocator* — The operating system acts as the manager of the resources and allocates them to specific programs and users as necessary.
- An operating system is a *control program* — The operating system controls the execution of user programs to prevent errors and improper use of the computer.
- In general, there is no adequate definition of an operating system.

□ **End of chapter.**

Bibliography

[1] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng.*, 14(2):141–154, Feb. 1988.

[2] D. L. Eager and E. D. Lazowska. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Softw. Eng.*, 12(5), May 1986.

[3] D. L. Eager and E. D. Lazowska. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation*, 6:53–68, 1986.

[4] M. D. Feng and C. K. Yuen. Dynamic load balancing on a distributed system. In *Proc. 6th IEEE Symposium on Parallel and Distributed Processing*, pages 318–325, Dallas, Texas, US, Oct. 1994.

[5] D. Hatch, D. Finkel, and K. G. Nock. Load indices for load sharing in heterogeneous distributed computing systems. In *Proc. 1990 UKSC Conf. Computer Simulation*, 1990.

[6] O. Kremien and J. Kramer. Methodical analysis of adaptive load sharing algorithms. *IEEE Trans. Parallel and Distribut. Syst.*, 3(6):747–760, Nov. 1992.

[7] S.-T. Levi and A. K. Agrawala. *Real Time System Design*, chapter 2. McGraw-Hill, 1990.

[8] C. Lu and S. L. Hsieh. Facilitating load distribution based on microkernel technology. In *Proceedings of International Conference On Signal Processing Applications & Technology*, volume 1, pages 776–780, Oct. 1996.

[9] C. Lu and S.-M. Lau. A performance study on load balancing algorithms with process migration. In *Proc. IEEE TENCON 1994*, pages 357–364, Aug. 1994.

[10] C. Lu and S.-M. Lau. An adaptive algorithm for resolving processor thrashing in load distribution. *Concurrency: Practice and Experience*, 7(7):653–70, Oct. 1995.

[11] Q. Lu and A. S. L. Hsieh. Dsf: a load distribution facility supporting multiple algorithms on mach. To appear in *Proceedings of the International Conference on Parallel and Distributed Processing Techniniques and Applications*, PDPTA'97, 1997.

[12] R. Mirchandaney, D. Towsley, and J. Stankovic. Adaptive load sharing in heterogeneous systems. *J. Parallel and Distributed Computing*, 9(4):331–346, Aug. 1990.

[13] L. M. Ni, C. W. Xu, and T. B. Gendreau. Drafting algorithm - a dynamic process migration protocol for distributed systems. In *Proc. 5th Int. Conf. Distributed Computing Systems*, pages 539–546. IEEE, 1985.

[14] N. G. Shivaratri and P. Krueger. Two adaptive location policies for global scheduling algorithms. In *Proc. 10th Int. Conf. Distributed Computing Systems*, pages 502–509, May 1990.

[15] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *IEEE Comput.*, 25(12):33–44, Dec. 1992.

[16] N. G. Shivaratri and M. Singhal. A load index and a transfer policy for global scheduling tasks with deadlines. *Concurrency: Practice and Experience*, 7(7):671–688, Oct. 1995.

[17] J. A. Stankovic and I. S. Sidhu. An adaptive bidding algorithm for processes, clusters, and distributed groups. In *Proc. 4th Int. Conf. Distributed Computing Systems*, pages 13–18, May 1984.

[18] A. Tanenbaum. *Distributed Operating Systems*, chapter 3. Prentice-Hall, 1995.

[19] Y. T. Wang and R. J. T. Morris. Load sharing in distributed systems. *IEEE Trans. Comput.*, 34(3), Mar. 1985.