A stochastic load balancing algorithm for *i-Computing*

(*Short title:* **Load Balancing for i-Computing)**

Yuk-Yin WONG

(The corresponding author)

Computing Services Centre

City University of Hong Kong, Tat Chee Avenue, KLN., Hong Kong

clevin.wong@cityu.edu.hk

(Phone: +852 2628 2680 FAX: +852 2628 2661)

Kwong-Sak LEUNG

Department of Computer Science and Engineering

Chinese University of Hong Kong, Shatin, N.T., Hong Kong

ksleung@cse.cuhk.edu.hk

Kin-Hong LEE

Department of Computer Science and Engineering

Chinese University of Hong Kong, Shatin, N.T., Hong Kong

khlee@cse.cuhk.edu.hk

SUMMARY

This paper presents a stochastic dynamic load balancing algorithm for Internet Computing, which is a new type of distributed computing involving heterogeneous workstations from different organizations on Internet. To realize the practical environment, we assume the system comprised of heterogeneous, untrusted and non-dedicated workstations connected by non-dedicated network. Our algorithm uses the product of the average processing time and the queue length of system jobs as the load index. Dynamic communication delay is included in the execution cost calculation. The transfer policy and the location policy are combined in a stochastic algorithm. State information exchange is done via information feedback and mutual updating. Simulations demonstrate that our algorithm outperforms conventional approaches over a wide range of system parameters. These results are reconfirmed by empirical experiments after we have implemented the algorithms on the DJM global virtual machine.

KEY WORDS: Internet computing; stochastic load balancing; distributed systems; Java computing

1. INTRODUCTION

As a result of the rapid advances in the high performance workstations and high-speed networking technologies, the idea of exploiting coarse-grained concurrency over network of workstations becomes increasingly competitive compared to the expensive parallel machines [1]. The promises of such type of distributed system include inexpensive parallel computing, scalable processing power, and more efficient use of existing resources. In recent years, due to the rapid growth of the Internet infrastructure and the invention of the platform independent language *Java* [2], larger scale of distributed computing that involves workstations from different organizations on the Internet is gaining popularity [3][4][5][6]. We call such type of computing as *Internet Computing*, or *i-Computing* for short. i-Computing enables large-scale resources sharing among different organizations. Thanks to the huge number of workstations on the Internet, i-Computing provides opportunities of forming virtual computers with gigantic processing power. It also offers the potential of selling and buying of surplus computing power over the Internet [7]. For example, the *Distributed Java Machine (DJM)* [4][7] is a global virtual machine targeted for i-Computing. DJM allows heterogeneous workstations from anywhere on the Internet to dynamically form a distributed computing system simply by using a web browser to visit a web page.

In a distributed computing system, it is very common that some nodes have heavier transient workload than the others [8][9]. A *load balancing (LB)* strategy would be extremely useful to improve the system efficiency. The goal of a LB algorithm is to improve the performance of a distributed system by appropriately transferring jobs from heavily-loaded nodes to lightly-loaded nodes.

Many LB algorithms (e.g. [10][11][12][13][14]) are proposed for the traditional distributed systems in an enclosed laboratory environment. However, little work has been done to cater for the following unique characteristics in the practical i-Computing environment: heterogeneous servers, untrusted peers, non-dedicated servers, non-dedicated communication network, and considerable communication delay. The main reason may be due to their intractable nature. In this paper, we propose a novel distributed dynamic LB algorithm specially designed to handle the above characteristics of i-Computing [15]. Our algorithm uses a stochastic approach in selecting nodes for remote job execution. By comparing the experiment results of a number of well-known algorithms, it is observed that our algorithm reduces the average job response time over a wide range of system parameters.

The rest of this paper is organized as follows. Section 2 presents the related work on load balancing and our motivation for a new approach. Section 3 introduces the system model. Section 4 describes the detailed design of the proposed algorithm. In Section 5, the performance of our algorithm is compared with other well-known approaches in a series of simulation and empirical experiments. The practical experience gained from designing and implementing the algorithm is shared in Section 6. Finally, this paper is concluded in Section 7.

2. RELATED WORK AND OUR MOTIVATION

In the past decade, a lot of research has been directed towards the task of effective LB algorithms in a distributed computing system. LB algorithms can be classified into static and dynamic approaches [16]. *Static LB algorithms* assume that *a priori* information about all the characteristics of the jobs, the server nodes and the communication network is provided. LB decisions are made deterministically (e.g. [17]) or probabilistically (e.g. [18]) at compile time and remain constant during runtime. Static LB problems have been treated in the following approaches: graph-theoretical approach [17][19], integer programming [20][21][22], queuing model [23], and heuristics methods [24][25][26]. Static approach is attractive because of its simplicity and the minimized runtime overhead. However, it has two major disadvantages. Firstly, the workload distribution of many applications cannot be predicted before program execution. Secondly, it assumes the characteristics of the computing resource and communication network are all known in advance and remain constant. Such assumption may not apply to a non-dedicated computing environment. Because static approach cannot response to the dynamic runtime environment, it may lead to load imbalance on some nodes and significantly increase the job response time.

In contrast, *dynamic LB algorithms* attempt to use the runtime state information to make more informative decisions in balancing the system load. The optimal solution of dynamic

LB is *NP-complete* in the general case [27]. However, sub-optimal solutions can be obtained by heuristic approaches. Dynamic algorithms are applicable to much wider ranges of applications. Using simple queuing analysis and simulation, Livny & Melman [12] have shown that dynamic algorithms perform better than their static counterparts. Eager et al. [11] have compared dynamic LB algorithms of different complexities. They have shown that simple algorithms using a very small amount of system information could improve the performance comparable to that of the more complex algorithms.

Dynamic LB algorithms can be further classified into centralized approach and distributed approach. In the *centralized approach* (e.g. [28][29][30][31]), only one node in the distributed system acts as the central controller. It has a global view of the load information of all the nodes in the system, and decides how to allocate the jobs to all the nodes in the system. The rest of the nodes act as slaves. They only execute the jobs assigned by the controller. The centralized approach is more beneficial when the communication cost is less significant, e.g. in the shared-memory multiprocessor environment. It has the weakness that the central controller may become (1) bottleneck of the system when the number of nodes increases, and (2) the single point of failure.

In the *distributed approach* (e.g. [10][11][12][32]) all nodes in the distributed system are involved in making the LB decision. It is commonly agreed that distributed algorithms are more scaleable and better fault tolerant. Since the LB decisions are distributed, it is costly to let each node obtain the dynamic state information of the whole system. Most LB algorithms only use partial information stored in the local node to make a sub-optimal solution.

LB algorithms have been extensively reported in the literature, but most of the previous studies (e.g. [11][26][33][34]) only consider the traditional trusted environment with some over-simplified assumptions that are not applicable for the practical i-Computing environment. For example, most of them assume that all the nodes are homogeneous, the inter-node communication delay is negligible, and there is no external load on the computing nodes and the communication network. Some studies (e.g. [35]) also assume the existence of an efficient broadcasting service on the communication network.

The main motivation of our study is to propose a distributed dynamic LB algorithm that can cater for the following unique characteristics in the practical i-Computing environment:

(a) *Heterogeneous servers*

There may be difference in the hardware architecture, computing power and resource capacity among the nodes. Moreover, the nodes may be running different operating systems. For example, it is common to find Sun Sparc workstations running Solaris and Intel Pentium machines running Windows 98/NT on the Internet.

(b) *Untrusted peers*

Since i-Computing involves computers from different management domains, security is a critical issue. Certain information on the host machine should be prohibited from being accessed by the network programs running on it. For example, the security model of *Java applets* prevents programs loaded over the network to retrieve many system parameters of the client workstation.

(c) *Non-dedicated servers*

The workstations are not dedicated. There may be transient background jobs on the workstations.

(d) *Non-dedicated communication network*

The inter-node communication network is non-dedicated too. There is always external traffic on the Internet that not generated by the distributed system.

(e) *Considerable communication delay*

The inter-node communication delay is always one of the most costly and the least reliable factor in i-Computing. We should not ignore the considerable dynamic communication delay on the Internet.

Our target is to devise a general and practical algorithm to cater for the above characteristics. We hope the target algorithm can be used in practical i-Computing systems like the DJM global virtual machine.

3. SYSTEM MODEL

It is assumed that the system consists of a set $\boldsymbol{P}$ of nodes connected by a communication network. $\boldsymbol{P}$ contains n nodes, labeled as $P_1 \ldots P_n$. The nodes may be of different hardware architectures and running different operating systems. $\boldsymbol{P}$ is *fully interconnected*, meaning that there exists at least one communication path between any two nodes. There is no shared-memory among the nodes. The only way of inter-node communication is by message passing. No global clock exists in the distributed system.

For any node $P_i \in \boldsymbol{P}$, there are *system jobs* arriving at P_i. All jobs are assumed to be mutually independent and can be executed in any node. As soon as a job arrives, it must be assigned to exactly one node for processing. When a job is completed, the executing node will return the result to the originating node of the job. We use $\boldsymbol{O}$ to denote the set of all system jobs generated at $\boldsymbol{P}$. $\boldsymbol{O} = \{O_1 \ldots O_m\}$. *(Note: for simplicity, the term "job" implies "system job" unless explicitly specified).*

A non-dedicated system is assumed. Some nodes may have background jobs that are not generated by the system. Due to the security constraints of i-Computing environment, the

LB algorithm cannot have access to the information about the background jobs. Without loss of generality, we assume $\exists b \in \{1..n\}$ s.t. $\forall j \in \{1..b\}$, there are *background jobs* at node P_j in addition to the system jobs. We use $\mathbf{P_B}$ to denote the set of nodes having background jobs. $\mathbf{P_B} = \{P_1 ... P_b\}$. $\mathbf{P_B} \subset \mathbf{P}$.

There is a non-trivial communication delay on the communication network between the nodes. The communication delay is different between different pairs of nodes and is affected by external communication traffics. The underlying network protocol guarantees that messages sent across the network are received in the order sent. There is no efficient broadcasting or multicasting service available.

Definition 1

$\forall O_x \in \mathbf{O}$, we define the following functions ($\mathbf{T}$ denotes the time domain):

(1) $ori(O_x) : \mathbf{O} \to \mathbf{P}$. It denotes the originating node of the job O_x.

(2) $exe(O_x) : \mathbf{O} \to \mathbf{P}$. It denotes the executing node of the job.

(3) $arr(O_x) : \mathbf{O} \to \mathbf{T}$. It denotes the arrival time of job O_x, which is the time when the job is generated at $ori(O_x)$.

(4) $fin(O_x) : \mathbf{O} \to \mathbf{T}$. It denotes the finish time of O_x, which is the time when node $ori(O_x)$ receives O_x's result from node $exe(O_x)$.

(5) $res(O_x) : \mathbf{O} \to \mathbf{T}$. It denotes the response time of O_x. $res(O_x) = fin(O_x) - arr(O_x)$.

The objective of a distributed dynamic load balancing algorithm is to minimize the average response time of all the system jobs.

Definition 2 (Objective)

The objective of the load balancing algorithm is defined by:

$\forall O_x \in \mathbf{O}$, determine $exe(O_x)$ at time $arr(O_x)$ using only the information available on node $ori(O_x)$ s.t. $\left(\sum_{y=1..m} res(O_y) / m \right)$ is minimized.

4. THE PROPOSED ALGORITHM

In the following paragraphs, we discuss the details of the distributed dynamic LB algorithm proposed by us.

4.1 Job migration

Some researchers have considered migration of partly-executed jobs in their LB algorithms [10][36]. However, migrating a partly-executed job is far from trivial in practice. It involves collecting all system states (e.g. virtual memory image, process control blocks, unread I/O buffer, data pointers, timers...) of the job, which is large and complex. Many studies [35][37][38][39][40][41] shown that (1) migrating partly-executed jobs is often difficult in practice, (2) the operation is generally expensive in most systems, and (3) there are no significant benefits of such mechanism over those offered by non-migratory counterparts. Furthermore, owing to the architectural differences between machines, job migration is undesirable in a heterogeneous computing environment. For example, it is very complicate to migrate the execution image of a job from a personal computer running Windows 98 to a Sun workstation running Solaris. Hence, we do not consider migration of partly-executed jobs in this paper. In order to prevent *processor thrashing*, we restrict that once a job is transferred to a particular node, it cannot be reassigned and must proceed to completion by that node.

Dynamic LB algorithms can be classified into sender-initiative algorithms and receiver-initiative algorithms according to their location policies [26][33]. *Sender-initiative algorithms* let the heavily-loaded nodes take the initiative to request the lightly-loaded nodes to receive the jobs, while *receiver-initiative algorithms* let the lightly-loaded nodes invite heavily-loaded nodes to send their jobs. In the practical computer systems that schedule local processes by time-sharing, the decision on supporting job migration has a direct impact on the feasibility of using receiver-initiative algorithms. Due to the reasons discussed in Section 6.2, without job migration, receiver-initiative algorithms always lead to inferior performance. Hence, we only consider sender-initiative algorithms in this study.

4.2 State information

Each node P_i maintains a local view of the state information of all the nodes by using a *state vector* S_i. The state vector helps a server to estimate the loading of other servers at any time without message transfer.

Definition 3 (State Vector)

" P_i, P_j Î ***P****, the state vector S_i is an n-dimensional vector maintained by P_i. Each vector component $S_i[j]$ is an ordered list (PT, QL, TS, CD):*

$S_i[j]$.PT records the moving average of the processing delay (waiting time plus serving time) of the last j jobs in P_j.

$S_i[j]$.QL records the number of system jobs in P_j. The number of background jobs is excluded because this information is usually inaccessible.

$S_i[j]$.TS records the value of P_j's local clock when the value of PT is reported.

$S_i[j]$.CD records the round-trip communication delay of the last message exchanged between P_i and P_j. " i Î {1..n}, $S_i[i]$.CD = 0.

4.3 Load index

An important issue in designing a dynamic LB algorithm is to identify the *load index* that measures the current loading of a server. A good index must be easily measured with minimum overhead, and correlate well with the response time of the job.

Most algorithms in the literature solely use the instantaneous *CPU queue length* (i.e. the number of jobs being served or waiting for service at the sampling instant) as the load index of a node [35][42][43][44]. This approach is based on the "join the shortest queue" intuition.

The CPU queue length may be a good load index if we assume all the nodes of the system are homogeneous and the inter-node communication delay is negligible or constant. However, it is not a reliable load indicator in a heterogeneous environment. It ignores the variations in computing power and machine architecture. Utilization of other resources like memory is disregarded [45]. Most important, this approach assumes the LB scheduler can retrieve the number of background jobs on the server. Because of the security concern, such an assumption is not valid under the i-Computing environment that involving more than one management domain. For example, the Java applets are forbidden to retrieve any information of the background jobs on the executing node.

Owing to the above reasons, we do not use the CPU queue length as the load indicator. Instead, a load index that is equal to the product of the queue length and the average processing time of the system jobs is utilized. The queue length of the system jobs equals to the CPU queue length only when no background job exists.

Definition 4 (Load Index)

" P_i, Î ***P****, the load index of P_i at time t is defined as:*

$L_i(t) = QL_i(t) * PT_i(t)$

where $QL_i(t)$ is the number of system jobs currently waiting or serving on P_i at time t, $PT_i(t)$ is the moving average job processing delay (i.e. waiting time plus serving time) of the last j jobs on P_i at time t.

4.4 Execution cost

Unlike conventional approaches that only consider the load index in calculating the cost of executing a job on a node, we include the dynamic communication delay in the cost calculation. It is because the dynamic and considerable communication delays may have great influence on the performance of a LB algorithm in the practical i-Computing environment. For example, it may be more efficient to send a job to a node with heavier load but small communication delay. Ignoring the communication delay, job transfer between two distant nodes can result in performance degradation during load balancing [46].

Definition 5 (Estimated Execution Cost)

" P_i, P_j Î ***P****, the execution cost of sending a job from P_i to P_j is estimated by P_i as:*

$C_i[j] = max \{ S_i[j].QL * S_i[j].PT + S_i[j].CD, x \}$

where x is a very small positive value utilized to prevent the "zero division" error when we calculate the value of $1/C_i[j]$.

In the above equation, $S_i[j].CD$ measure how long to transfer the request from to the destination node and get back the return value to the source node. The value $(S_i[j].PT * S_i[j].QL)$ estimate the processing delay of the job on the destination node.

4.5 Information policy

A LB algorithm can be divided into three inter-related component policies. Firstly, the *information policy* determines how to maintain the state information among the nodes. It decides what type of information to be collected, when to collect it, and from where the information to be collected [47]. Secondly, the *transfer policy* determines whether an arriving job should be considered for remote processing. Finally, if the transfer policy concludes that a job is eligible for remote processing, then the *location policy* is used to determine, based on the information collected in the information policy, a suitable remote node to transfer the job [11]. The state information to be collected has been presented in Section 4.2, we will focus on the *information exchange strategy* of the information policy here. The transfer policy and location policy will be discussed in the next section.

The traditional strategy on information exchange can be classified into two approaches. In the first approach, the state information is collected by periodic or aperiodic *state change broadcasting* [48][49]. The second approach is based on *demand-driven polling* [11][12][33][34]. When a node wants to send a job for remote execution, it sends a short query message to a predefined number (this number is known as the *probe limit*, L_p) of nodes to search for a suitable destination. Examples of polling strategies include *state probing* [11], *drafting* [50], and *bidding* [51].

We cannot use the first approach owing to the unavailability of efficient broadcasting services. For the decentralized LB algorithms that assume zero communication delay and negligible polling processing overheads, the demand-driven polling approach is widely used. It is because their assumptions imply that polling incurs no processing overheads and no time delays. It means that polling can retrieve perfectly accurate state information of the remote nodes without any cost. However, such assumptions are far from accurate in the practical environment, especially when talking about i-Computing. Polling approach has several problems in practice:

1. Repeated polling wastes the processing time of the polling nodes and polled nodes. This problem becomes significant when the general system load is heavy. When most of the nodes are heavily loaded, they keep on polling each other for the sparse lightly loaded node. In the worst case, polling may cause system instability when all the nodes are heavily loaded. Each node will make L_p void attempts to poll each other for the non-existing lightly loaded node whenever a new job arrives.
2. Repeated polling generates large amount of network traffic. This problem will become more significant if the network bandwidth is limited.
3. As the job needs to wait for the polling result, polling will increase the response time of the waiting job. It is a problem if the communication delay is significant.
4. It is difficult to set a good value for the probe limit L_p. In a medium to heavy loaded system, if the probe limit is small, lightly loaded nodes may not be discovered. If the probe limit is large, then (a) most of the heavily loaded nodes may find the same lightly-loaded nodes and dump their loads to them, (b) the above problems caused by repeated polling will multiply.

We propose to use *information feedback* and *mutual updating* to reduce the processing and communication overheads. For any node P_i $\hat{I}$ $\boldsymbol{P}$, P_i maintains its states information in its state vector element $S_i[i]$. The load index $S_i[i].PT$ is calculated as the moving average of the processing time for the last j system jobs executed on P_i. $S_i[i].QL$ counts the number of system jobs waiting or serving on the S_i. Other elements of the state vector are maintained

by message exchange with other nodes. Figure 1 outlines the procedure when P_i sends a job O_x to P_j for processing.

As show in Figure 1, state information exchange is done by mutual updating. P_i appends the load indexes of itself and $\hat{o}$ (a small integer) random nodes to the job transfer request sent to P_j *(steps 1-2)* by piggybacking. P_j then updates the corresponding load indexes in its state vector by comparing the timestamps *(step 7)*. Similarly, P_j inserts the load indexes of itself and $\hat{o}$ random nodes in the job completion reply *(steps 12-13)* for P_i to update *(step 17)*. The round-trip communication delay between P_i and P_j is measured by the timestamps $T1_i$, $T2_i$, $T1_j$ and $T2_j$ *(step 18)*.

Please note that since there is no global clock in the system, we can only compare the time values relative to the same clock. For example, we can compare the values of the pair ($T1_i$, $T2_i$) because both are relative to the local clock of P_i. The similar case is for ($T1_j$, $T2_j$). Values $S_i[y].TS$ and $S_j[y].TS$ are compared in steps 7 and 17 because both are relative to the local clock of P_y.

In order to ensure the values of the state vectors are up to date, a periodic information exchange procedure is employed. In any node P_i $\hat{I}$ $\boldsymbol{P}$, if the state vector element $S_i[j]$, $i \neq j$, has not been updated for a predefined period y, then the LB scheduler will send an information exchange message to P_j. The periodic information exchange procedure is the same as that shown in Figure 1 except that steps 4 and 8-11 are skipped.

Steps processed in P_i :

1. $Y \neg \{P_i\} \grave{E}\{ \hat{o}$ *random nodes from* $(P \setminus \{P_i,P_j\}) \}$ /* P_i select nodes for info exchange */
2. *"*$P_y \hat{I} Y$ *,* P_i *appends* $(S_i[y].PT, S_i[y].QL, S_i[y].TS)$ *to the job transfer request M*
3. P_i *appends its local time* $T1_i$ *to M*
4. $S_i[j].QL \neg S_i[j].QL + 1$
5. P_i *sends message M to* P_j

Steps processed in P_j:

6. P_j *records local time* $T1_j$ *when M is received.*
7. *"*$P_y \hat{I} Y$*, if* $S_i[y].TS > S_j[y].TS$, /* P_j updates the state vector using P_i' s info */
   then $(S_j[y].PT \neg S_i[y].PT, \ S_j[y].QL \neg S_i[y].QL, \ S_j[y].TS \neg S_i[y].TS)$
8. $S_j[j].QL \neg S_j[j].QL + 1$
9. P_j *put* O_x *into its local CPU queue for processing*
10. O_x *is completed.* P_j *uses* O_x*' s processing time to update the value of* $S_j[j].PT$
11. $S_j[j].QL \neg S_j[j].QL - 1$
12. $Z \neg \{P_j\} \grave{E}\{ \hat{o}$ *random nodes from* $(P \setminus \{P_i,P_j\}) \}$ /* P_j select nodes for info exchange */
13. *"*$P_z \hat{I} Z$ *,* P_j *appends* $(S_j[z].PT, S_j[z].QL, S_j[z].TS)$ *to the job completion reply R*
14. P_j *appends its local time* $T2_j$ *to R*
15. P_j *sends message R to* P_i

Steps processed in P_i:

16. P_i *records its local time* $T2_i$ *when receives the reply R is received*
17. *"*$P_z \hat{I} Z$ *, if* $S_j[z].TS > S_i[z].TS$, /* P_i updates the state vector using P_j' s info */
    then $(S_i[z].PT \neg S_j[z].PT, \ S_i[z].QL \neg S_j[z].QL, \ S_i[z].TS \neg S_j[z].TS)$
18. $S_i[j].CD \neg (T2_i - T1_i) - (T2_j - T1_j)$. /* Calculate the round-trip communication delay */

Figure 1. Job transfer procedure when node P_i transfers a job O_x to a remote node P_j for processing. It includes the procedure of state information exchange between P_i and P_j.

4.6 Transfer policy & location policy

The transfer policy and the location policy are based on some predefined *thresholds* in most studies (e.g. [11][12][26][33][44]). Most of them assume a node becomes eligible to transfer the arriving jobs for remote processing when its load index exceeds a predefined threshold T_{high}. A node is qualified to receive remote jobs when its load index is below this threshold. However, determining the optimal value of the threshold is far from trivial. It depends on the system-wide load information [43]. In general, low thresholds are appropriate for low system loads and low transfer costs, while high thresholds are superior for high system loads and high transfer costs [11]. Mirchandaney et al. [52] reported that when the job transfer delay increases, the optimal threshold various a lot (from 0 to 24). If the communication delay is non-negligible, inappropriate thresholds may lead to performance degradation. For example, Dasgupta et al. [42] reported that using static thresholds might lead to inferior performance under a shared communication network with limited bandwidth.

Our algorithm uses a novel approach by combining the transfer policy and the location policy in a stochastic approach. Instead of using non-adaptive predefined thresholds, the job assignment is done in a probabilistic fashion. The probabilities assigned to the nodes are inversely proportional to their execution costs. Our approach is inspired by the *"proportional betting"* methodology mentioned by Cover & Thomas [53]. We assume that from the viewpoint of a node $P_i \in \boldsymbol{P}$, $\forall P_j \in \boldsymbol{P}$, the reciprocal of the estimated execution cost of P_j (i.e. $1/C_i[j]$) reflects P_j's *probability of winning* (P_j "wins" if it gives the minimum response time for a new job sent from P_i).

When a job O_x arrives at a node P_i, its execution node will be selected randomly according to the following probability function:

Definition 6 (Transfer Policy & Location Policy)

$\forall O_x \in \boldsymbol{O}$, *s.t.* $ori(O_x) = P_i \in \boldsymbol{P}$.

if $(C_i[i] - Min_{k=1..n}\{C_i[k]\}) \leq \varepsilon$ *then* $exe(O_x) = P_i$ ..... (Equation A)

else $\forall P_j \in \boldsymbol{P}$, $Probability(exe(O_x)=P_j) = (1/C_i[j]) / \sum_{k=1..n} \{1/C_i[k]\}$ ... (Equation B)

where ε *is a positive value close to zero.*

Please note that we do not divide Definition 6 by stating that Equation A is the transfer policy and Equation B is the location policy. It is because the originating node of the job, P_i, in also included in Equation B. The probability function in Equation 2 is the core of the algorithm. Equation A serves as an exception case to bias the local node because the local state information is always accurate, while the information of other nodes is only an

estimate. A job will be processed locally if the cost of local execution is close to the minimum cost among all nodes.

We take a stochastic approach for distributed load balancing in the i-Computing environment because of the following reasons:

(1) Owing to the rapidly changing environment, there is a great degree of randomness and unpredictability in the system state [49]. This is caused by (a) arrival and departure of system jobs, (b) arrival and departure of background jobs, and (c) variation in the external traffic on the non-dedicated communication network. Stochastic approach should be more adaptive to fluctuations in background workload and external network traffic.

(2) There is a propagation delay for the load information across the network. The state vector can only serve as a local estimate for the "current" status of the remote nodes.

(3) To balance a distributed system, we should assign jobs to each node in proportional to its dynamic capacity [54]. Hence a "proportional betting" approach should be better than a "join the shortest queue" approach if we cannot always retrieve the most updated values of all the queue lengths.

(4) A stochastic approach can avoid system instability when all nodes transfer the jobs to the node estimated to have the minimum load.

5. EXPERIMENTS

As discussed in Section 4.1, we only consider sender-initiative algorithms. Our algorithm (labeled as *Stoc*) is compared with the following algorithms in the simulation and empirical experiments:

(a) *Local*: All jobs are locally processed by their originating nodes.

(b) *Random*: A job will be executed locally if the number of system jobs of the local node is less than a predefined threshold T_{high}. Otherwise, a node is selected at random to process the arriving job.

(c) *Threshd:* A job will be executed locally if the number of system jobs of the local node is less than a predefined threshold T_{high}. Otherwise, nodes are polled one by one according to a random order. If no node with number of system jobs below T_{high}, the job will be executed locally, otherwise the first node with the number of system jobs below T_{high} will be selected.

(d) *Lowest:* A job will be executed locally if the number of system jobs of the local node is less than a predefined threshold T_{high}. Otherwise a set of L_p random nodes will be polled to compare the number of system jobs. If no node with number of system jobs below

T_{high}, the job will be executed locally; else the target will be the node with the lowest number of jobs.

(e) *NoCost*. It assumes that the LB scheduler can retrieve the current value of the total number of system jobs and background jobs of all the nodes immediately without cost, but the transfer costs of the job are considered. The node with the minimum number of system jobs plus background jobs is selected. This algorithm is used as an *estimate* for a LB algorithm with "prefect" information.

Algorithms (b), (c), and (d) are proposed by Eager, Lazowaka & Zahorjan [33]. We select the above algorithms because they represent a reasonably large collection of approaches.

In order to measure the performance of our algorithm, we define the *improvement factor* as follows:

Definition 7. *The improvement factor L of algorithm X over algorithm Y is given by:*

$$L(X \text{ over } Y) = (F(Y) - F(X)) / F(Y),$$

where F (X) denotes the average response time of all the system jobs using algorithm X. A positive value of L indicates an improvement, while a negative value implies degradation.

5.1 Simulation Experiments

In this section, we first study the performance of algorithms under different system parameters by simulation experiments. The simulation results will be verified by empirical experiments in Section 5.2.

For simulation purpose, we assume the system jobs arriving at P_i according to a Possion process with a mean of λ jobs per time-unit. The service time for a job is exponentially distributed with a mean of X time-units. The arrival rate of the background jobs is exponentially distributed with a mean of λ_B jobs per time-unit, and the average service time is X_B time-units.

The LB policies were tested on a system of 40 nodes with same processing capacity. Each node processes the system jobs and background jobs according to a round-robin scheduling discipline with priority. The system jobs and background jobs have the same priority in using the CPU, while other control functions of the LB algorithm have higher priorities. As shown in Figure 2, the inter-node communication network is modeled as a star network. Each node has a private link connected to the central hub. The central hub assumed to have a much higher bandwidth than the private links. Hence, the communication delay is mainly caused by the private links.

To be more practical, we have considered the cost of collecting and processing the state information, and the dynamic delay caused by the congestion of communication network. Table 1 displays the values of the parameters used in the simulation experiments. Results of the experiments are average from five independent simulation runs using different random seeds. In each run, 2000 system jobs are generated in each of the nodes.

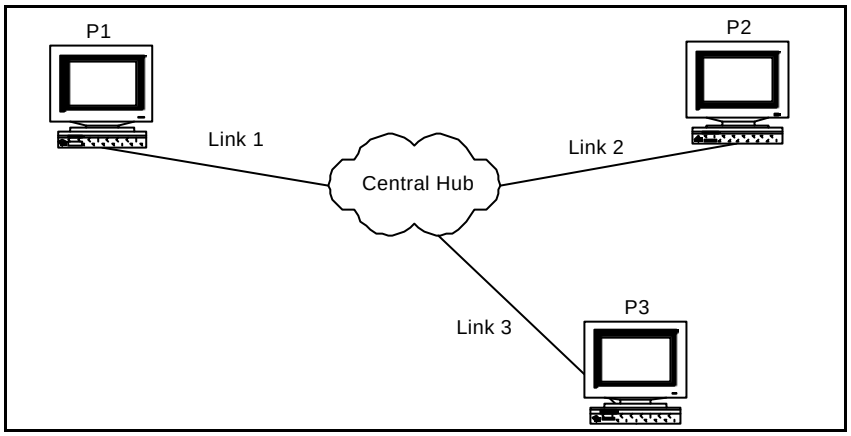

Figure 2. Network configuration in the simulation experiments.

Simulation Parameter	Value
Mean service time of system jobs, X	1.0 tu
Mean inter-arrival time of system jobs, l	2.1 tu
Mean service time of background jobs, X_B	1.0 tu
Mean inter-arrival time of background jobs, l_B	2.1 tu
Round-robin CPU scheduling quantum	0.01 tu
Mean network transmission delay in each private link, c	0.05 tu
Mean network transmission delay in the central hub	0.001 tu
Period for periodic info. exchange, y	10.0 tu
CPU overhead in sending/receiving a message	0.003 tu
CPU overhead in sending/receiving a polling/info-exchange message	0.001 tu
Probe limit, L_p	3
CPU Queue Threshold for local processing, T_{high}	3
No. of random nodes for mutual update, $\hat{o}$	2
Size of the moving average in calculating the load index, j	2

Table 1. Simulation parameters. (tu=time-unit).

5.1.1 Experiment on different system size

In experiment S1, we assume that there are background jobs on half of the nodes *(i.e. b = n/2)* and vary the total number of nodes *n* from 8 to 40. Results shown in Figure 3 and Table 2 illustrate that besides the unrealistic NoCost algorithm, Stoc consistently gives the best performance across all the values of *n*. Stoc gives an average improvement of 15% and 18% over Threshd and Lowest respectively.

5.1.2 Experiment on different number of nodes with background jobs

In experiment S2, we set the total number of nodes *n* to 40 and vary the number of busy nodes *b* from 0 to 40. Results are shown in Figure 4 and Table 3.

The results demonstrate that besides the unrealistic NoCost algorithm, Stoc gives the minimum average response time across all the values of *b*. Stoc has an average improvement factor of 17% over Threshd & Lowest. The improvement factor increases when the number of nodes with background job increases. The reason may be due to the fact Stoc is the only algorithm that takes the background workload into consideration. As discussed in Section 4.3, LB algorithms cannot retrieve the information of the background workload due to security reasons. However, we know that there is a direct relationship between the amount of background workload and the average processing delay of the system jobs. Since Stoc has considered the average processing delay of the system jobs in the cost estimation, it gives more accurate estimations for the loading of the nodes when there are background jobs. Hence, Stoc gives a better performance in load balancing.

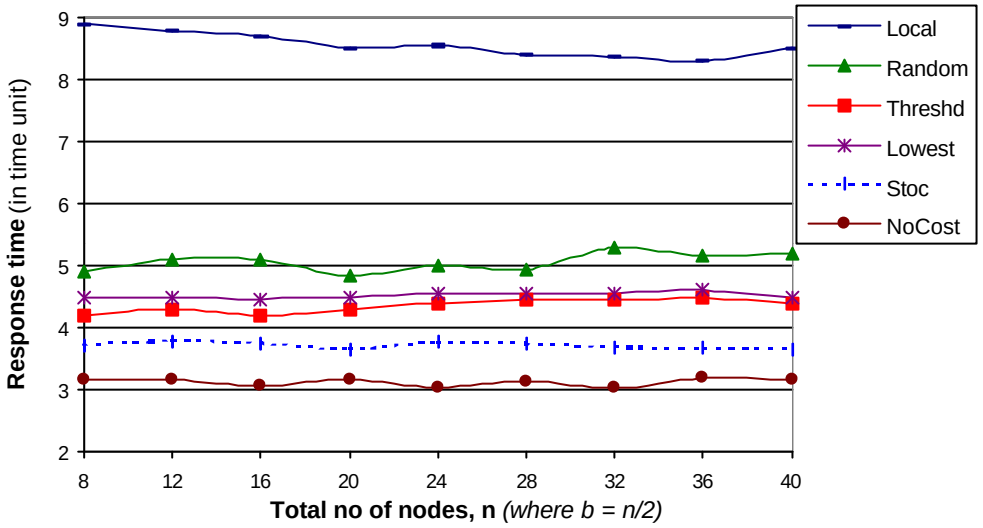

Figure 3. Average response time of all the algorithms in experiment S1.

n =	8	12	16	20	24	28	32	36	40	Mean
Local	58	57	57	57	56	55	56	56	57	57
Random	24	26	26	25	25	24	30	29	30	27
Threshd	11	12	11	15	14	16	17	18	17	15
Lowest	17	16	16	19	17	18	19	20	19	18
NoCost	-18	-20	-22	-16	-24	-19	-22	-16	-16	-19

Table 2. The improvement factor (in %) of Stoc over other algorithms in experiment S1.

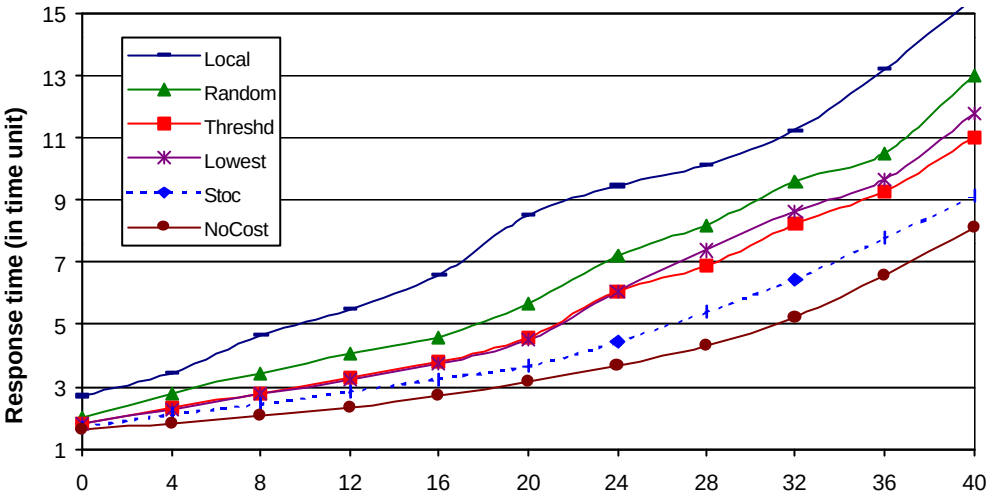

Figure 4. Average response time of all the algorithms in experiment S2.

b=	0	4	8	12	16	20	24	28	32	36	40	Mean
Local	38	39	48	48	51	57	53	47	43	41	41	46
Random	15	24	29	30	29	36	38	34	33	26	30	30
Threshd	8	10	12	14	14	21	27	22	22	16	17	17
Lowest	6	8	12	12	13	19	27	27	25	20	22	17
NoCost	-3	-16	-17	-21	-21	-16	-20	-25	-23	-18	-13	-17

Table 3**.** The improvement factor (in %) of Stoc over other algorithms in experiment S2.

5.1.3 Experiment on different values of communication delay

In experiment S3, we assume that there are 40 nodes and half of them have background jobs. We vary the mean network transmission delay in each link from 0 to 0.3 time-units. Results shown in Figure 5 and Table 4 illustrate the following points:

Firstly, besides the unrealistic NoCost algorithm, algorithm Stoc consistently gives the best performance across all the values of n. Stoc gives an average improvement of 15% and 18% over Threshd and Lowest respectively. The performance of Stoc is especially apparent when the communication delay of the link is high. We suggest this is because Stoc is the only algorithm taking the communication delay into consideration.

Secondly, besides the Local algorithm that has no network traffic, the response time of all algorithms increase when the communication delay increases. But the increasing rate of Stoc is much smaller than that of Threshd and Lowest. We suggest this is because Stoc does not use the polling policy for information collection.

Thirdly, algorithm Lowest is very sensitive to the communication delay of the link. It becomes worse than Threshd when the communication delay is over 0.05 time-unit, and even worse than Random when the delay increases to 0.14 time-unit. When the delay of each link increases to 0.25 time-units, Lowest give the worst performance. We suggest this is caused by the polling policy used by Lowest. In this algorithm, when a job arrives, if the number of system jobs in the local node is higher than the threshold T_{high}, the job must wait until L_p polling processes are finished. This waiting time is significant when the communication delay is high. Also based on polling, Threshd is better than Lowest because the polling process of Threshd can terminate earlier (i.e. less than L_p pollings) if a node of load less than T_{high} is found.

5.1.4 Discussion on the simulation results

In the above simulation experiments, all the nodes are assumed to have the same processing power. The communication network is modeled as a framework of first-in-first-out queues with fixed delay. The variation in communication delay is only caused by the queuing effect. We have not modeled (1) the differences in processing power among the nodes, (2) the differences in communication bandwidths between nodes, and (3) the existence of external traffic on the communication network. We believe that the improvement of Stoc over the other algorithms can be higher if these three factors are also modeled. It is because Stoc is the only algorithm that has considered the heterogeneity of server capacity, the variations in communication bandwidths between different pairs of nodes and the existence of external traffic.

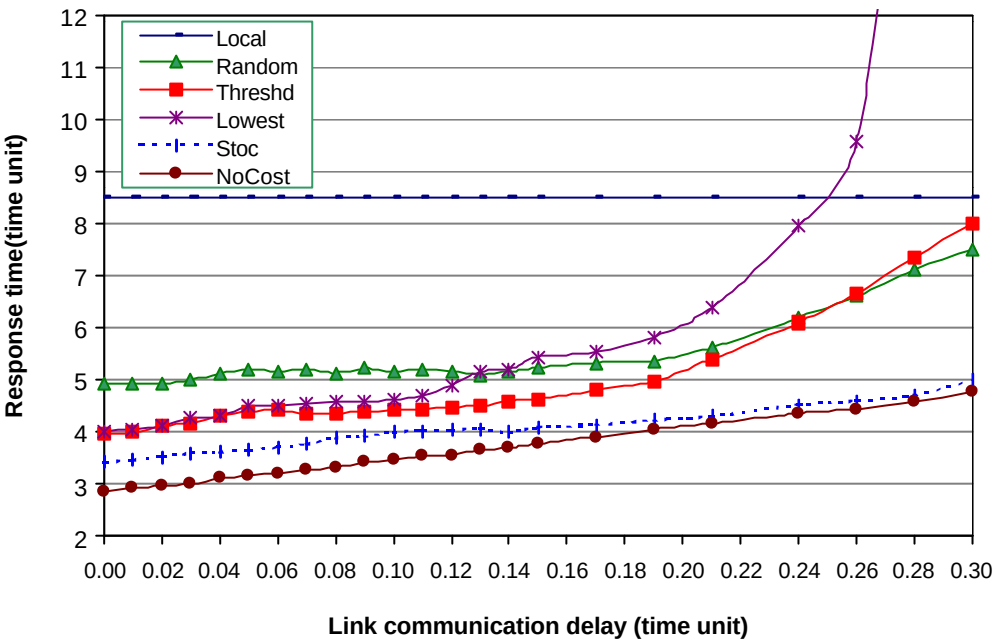

Figure 5. Average response time of all the algorithms in experiment S3.

	0.00	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09	0.10	0.11
Local	60	59	58	58	57	57	56	56	54	54	53	53
Random	30	29	28	28	29	30	28	28	24	25	23	23
Threshd	14	13	14	14	16	17	16	14	11	11	10	10
Lowest	15	15	14	16	16	19	18	17	15	15	13	14
NoCost	-19	-19	-18	-20	-16	-16	-15	-15	-17	-15	-15	-14

	0.12	0.13	0.14	0.15	0.17	0.19	0.21	0.24	0.26	0.28	0.3	Mean
Local	53	52	53	52	51	50	49	47	46	45	41	**53**
Random	22	20	22	22	22	21	24	27	30	34	33	**26**
Threshd	10	10	13	12	14	15	20	26	31	36	38	**17**
Lowest	18	21	23	25	25	27	33	43	52	73	75	**26**
NoCost	-13	-11	-8	-8	-6	-4	-3	-4	-3	-3	-5	**-12**

Table 4. The improvement factor (in %) of Stoc over other algorithms in experiment S3.

5.2 Empirical experiments

In order to reconfirm the simulation results and further validate the effectiveness of our LB algorithm in a real i-Computing environment, we have done some empirical experiments after implementing the LB algorithms in pure Java on the DJM [4][7] system. The codes are written in pure Java using the *Java Developer's Kit (JDK)* version 1.1.

In the experiments, each node will execute 200 jobs. The inter-arrival time of the jobs in each node are uniformly distributed in the range of 2 to 8 seconds. Each job contains *L* loops; each loop involves one double precision floating-point multiplication of two random numbers. The loop number *L* is uniformly distributed in the range of 100,000 to 500,000.

We have done four empirical experiments, labeled E1 to E4, on 16 heterogeneous machines. Table 5 lists the configuration of the machines. Table 6 shows the machines involved in each experiment.

All results are averaged from five runs in a student environment under normal loading at office hours. There are background jobs on the machines created by other users. All the machines are connected by the same campus network. The inter-node network is non-dedicated, there is external traffic generated by other users.

Figure 6 displays the average response time of these four experiments. The improvement factor of the Stoc over other algorithms is presented in Table 7. These empirical results reconfirm that Stoc consistently gives the best performance across different number of nodes. Stoc gives an average improvement of 16% and 21% over Threshd and Lowest respectively.

Machine no	Machine type	Memory Size	Operating System	Network speed
#1, #5	Sun Sparc-4	32M bytes	Solaris 2.5	10 Mbps
#2, #6	Intel Pentium 90	32M bytes	Windows 95	10 Mbps
#3, #7	Intel Pentium 133	64M bytes	Windows 95	10 Mbps
#4, #8-#16	Sun Ultra-5/10	64M bytes	Solaris 2.6	10 Mbps

Table 5. Configuration of the 16 machines in the empirical experiments.

Experiment	No of nodes	Machine involved
E1	4	#1 to #4
E2	8	#1 to #8
E3	12	#1 to #12
E4	16	#1 to #16

Table 6. Machines involved in the four empirical experiments.

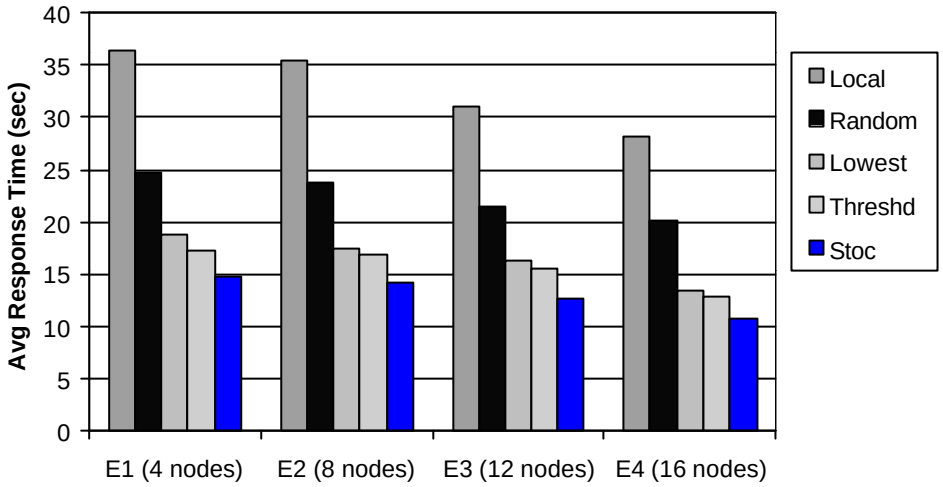

Figure 6. Average response time of the LB algorithms in empirical experiments E1 to E4.

	E1 (4 nodes)	E2 (8 nodes)	E3 (12 nodes)	E4 (16 nodes)	Average
Local	60	60	59	61	**60**
Random	40	41	41	46	**42**
Threshd	15	17	18	16	**16**
Lowest	22	19	22	19	**21**

Table 7. The improvement factor (in %) of Stoc over other algorithms in empirical experiments E1 to E4.

6. EXPERIENCE SHARING

In this section, we will share some experience gained from designing and implementing the proposed load balancing algorithm on the DJM global virtual machine using pure Java. We hope the computing community can benefit from our experience.

6.1 Choosing the load index

Load index is a very important parameter of a LB algorithm. The following are the most common load indicators in conventional LB strategies: the CPU queue length (i.e. total number of system jobs and background jobs), the CPU idle time, the CPU utilization, the free memory size, the context-switching rate, the system call rate, and the interrupt rate. However, there is no system method to retrieve any of the above system parameters in the current version of Java API. Even if such system methods are available in future, we believe that they will still be inaccessible for i-Computing (i.e. in the Java applet mode) because of the security concern. Our algorithm utilizes the processing time and the queue length of the system jobs to indicate the system load because these values can be calculated by the algorithm itself.

Initially we solely use the average processing time of the system jobs as the load index. This index will be updated only if there is a new *job completion event.* Very soon we discover that there might be problems in two extreme situations for any load index of similar nature.

In the first case, if a node has a large load index due to previous jobs, it is possible that no job will be sent to this node[a]. As there is no new job, there will be no job completion event. Hence the load index cannot be updated to reflect the idle situation. We call this the *prolonged idle* problem. In the second case, it is possible that many jobs will be arriving at a lightly-load node P_i at the same time[b]. Under a time-sharing operating system, the CPU is shared by all the jobs in a round-robin manner. The completion time of a job depends on the total number of jobs sharing the same CPU. It will take longer to complete a job and to update the load index of P_i when the job number increases. This may lead to a recursive problem because if the load index cannot be updated, other nodes do not know the busy situation of P_i, they will continue to send jobs to it. We call this the *prolonged congestion* problem.

In order to avoid the above problems, we have modified the load index formula. Our new load index is equal to the product of the queue length and the average processing time

[a] This is not the necessary consequence because we use a stochastic approach.
[b] Again, this is not the necessary consequence because we use a stochastic approach.

of the system jobs (i.e. $L_i = QL_i * PT_i$). In the prolonged idle case, when P_i is idle, although there is no job completion event to update PT_i, value of QL_i will decrease to zero. The load index L_i will reset to zero to reflect the idle situation, thus the prolonged idle problem will not occur. Similarly, in the prolonged congestion case, when P_i suddenly becomes very busy, value of L_i will be rapidly raised by the increasing value of QL_i to reflect the busy situation. Hence the prolonged congestion problem can also be avoided.

6.2 FCFS and time-sharing scheduling

Though most of the current operating systems use the time-sharing discipline for process scheduling, many conventional LB strategies (e.g. [11][31][42][55]) are based on the simple first-come-first-service (FCFS) discipline. It is usually claimed that theoretically there is no significant difference between these two disciplines because they share many similar characteristics according to the queuing theory [56]. One important similarity is that the average response time formula of a M/G/1 queuing system with *processor-sharing* scheduling (processor-sharing is a close approximation of the time-sharing discipline) is exactly the same as that of a M/M/1 queuing system with FCFS scheduling. However, we found that the FCFS and the time-sharing (or processor-sharing) scheduling disciplines do have a number of significant differences when implementing the LB algorithms in practice.

Firstly, under the time-sharing discipline, we should not consider the receiver-initiative LB algorithms if the system does not support migration of partly-executed jobs. It is because under this discipline, the CPU is shared by all the runnable jobs in a round-robin manner. It means that a new job will start to execute near immediately after it is arrived. Hence it is unlikely that a receiver node would probe a sender node just after a new job arrived at the sender but before the job starts to execute. Therefore, without job migration, receiver-initiative algorithms always perform worse than their source-initiative counterparts [35]. In contrast, under the FCFS discipline, newly arrived jobs will be served one by one. If the sender node has more than one job, the receiver node can always find jobs arrived but not executed. Hence the receiver-initiative algorithms are feasible on FCFS systems even if the migration of partly-executed jobs is not supported.

Secondly, the prolonged congestion problem mentioned in Section 6.1 will only occur under the time-sharing discipline. Concerning the second case in Section 6.1, if the system uses a FCFS discipline, then even many jobs arrive at a lightly-load node P_i simultaneously, the load index of P_i can be updated promptly because the jobs are served one by one. Hence, there will be no prolonged congestion.

6.3 Broadcasting and multicasting

As mentioned in Section 4.5, many LB policies in the literature are based on the broadcasting services where a node sends the same message to all nodes on the network. Frequent broadcasting creates performance problems on the whole network. Broadcasting is not feasible for the iComputing environment because multiple networks under different management domains are involved. *Multicasting* is a limited version of broadcasting, where a node sends the same message to multiple number of destination nodes anywhere on the network. The set of destination nodes forms a *multicast group*. Multicasting is still not deployed on the Internet at large. It is achievable only for some controlled environment. For example, when all nodes are within the same organization. In general, random machines on the Internet cannot communicate by multicasting.

Implement multicasting on Java has further difficulties. Although Java has the *MulticastSocket* class, it is not easy to run a multicast code on the applet mode. It is because the Java security model forbids an applet to receive/send messages from/to an arbitrary member of a multicast group. An applet can only sends *unicast* (i.e. point-to-point) datagram packets to and receives multicast/unicast datagram packets from its *originating host* (i.e. the host from where the applet is downloaded).

6.4 Multi-threaded programming on Java

Comparing to other traditional programming languages, writing multi-threaded program on Java seems to be easier because of the system-supported *Thread* object. However, if the codes will be run on different operating systems (like our case), programmer must be very careful owing to the difference in *Java Virtual Machines (JVM)* implementations. Threads of equal priority may get very different treatment on different platforms.

All JVM supports the scheduling policies that (1) if the currently running thread blocks (e.g. wait for I/O completion) or exits, a new runnable thread with the highest priority will be selected to run; (2) when a thread with higher priority becomes runnable, the currently running thread will be preempted. The problem is that the scheduling policy for equal priority threads is unspecified. For JVM on Windows 95/98 and NT implementations, threads of equal priority are scheduled in a round-robin manner using a time-slicing scheme. However, for JVM on Macintosh and most UNIX implementations, thread scheduling is non-timesliced. For threads with the same priority, one thread will run to completion (or until blocked) before another one can run. Therefore, on these platforms, equal priority threads must be "cooperative" among themselves. Other threads may encounter starvation if any one of them is "inconsiderate" in using the CPU.

The difference in scheduling of equal priority threads has great impact on the results of our load balancing algorithm. Extensive programming effort on thread scheduling is needed to ensure the threads are cooperative in using the processor. Usually such type of programming is done by calling the *Thread.sleep()* and *Thread.yield()* methods wisely. How to use the above *sleep()* and *yield()* methods effectively is not trivial. Usually we need to search for the best solution by some trials on different platforms.

For example, consider a Java program that has several threads of the same priority running the following loop:

```
while (! finish) {
    doSomeUsefulWork();
    yield();         //use yield() or sleep(someTime) here?
}
```

The above code runs smoothly on the UNIX and the Macintosh platforms, but it will cause performance problem on the Windows 95/98 and the Windows NT platforms. It is because frequent calling of the *yield()* method will waste most of the CPU time allocated to the JVM on the latter platforms. Therefore we need to switch to the *sleep()* method as a better cross-platform solution. However, this solution will waste some time for sleeping when there is only one runnable thread, thus one may need to search for the optimal "sleeping time" by further trials on different platforms.

6.5 Difficulties in modeling

Although distributed computing is always a hot research topic, and the popularity of Internet technology has increased dramatically in the pass few years, we are surprised that the mentioned five unique characteristics of i-Computing have not been dealt with to any large degree in the literature. We guess this may due to the fact that the intractable nature of these characteristics making them very difficult to model and analyze.

Queuing theory [56][57] is the most common theoretical model to analyze load balancing strategies. However, queuing theory is based on the assumptions of (1) homogeneous servers, (2) dedicated servers, (3) dedicated communication network, and (4) negligible inter-node communication delay. As we have discussed, all of these assumptions are not valid under the practical i-Computing environment.

Among the unique characteristics of i-Computing mentioned, we believe modeling and analysis of the effects caused by the dynamic communication delay on the Internet in the presence of external traffic is most difficult. The communication delays have two impacts on a load balancing strategy: (1) it increases the overhead of remote job processing, and (2) it adversely affects the quality of state information [52]. Despite the active discussion on analyzing and modeling the communication delay of distributed systems on a dedicated

local area network, little work has been done on that of the communication delay on the real Internet environment. Most LB algorithms ignore the communication delay by assuming it is negligible or constant. Up to now, we found no LB algorithm has considered the dynamic communication delay that is non-negligible and affected by external traffic. The external Internet traffic has not been modeled in our simulations because we do not find any discussion on this topic in the literature.

7. CONCLUSIONS

Internet computing is a new type of distributed computing that involves heterogeneous workstations from different organizations on the Internet. Due to the concerns of server heterogeneity, non-dedicated resources, communication overheads and security restrictions, the Internet Computing environment has many characteristics that made it unique from the traditional distributed systems. These characteristics have significant impact on the performance of load balancing. In this paper, we proposed a stochastic distributed dynamic load balancing algorithm to cater for these characteristics.

Owing to the heterogeneity and the security concerns, we do not use the CPU queue length, which includes both the system jobs and background jobs, as the load index. Instead, our algorithm defines the load index as the product of the average processing time and the queue length of system jobs. The non-negligible dynamic communication delay is considered in calculating the execution cost of a node. Inspired by the theory of "proportional betting", we take a novel approach by combining the transfer policy and the location policy using a stochastic function. The probability of a node being selected is inversely proportional to its execution cost. Rather than using the conventional state change broadcasting or demand-driven polling approaches, state information exchange in our algorithm is done via information feedback and mutual updating to reduce the processing and communication overheads.

Through simulation experiments, it is found that our algorithm can give shorter average response time than other well-known approaches over a wide range of system parameters. These results are reconfirmed by the empirical experiments after we have implemented the algorithms on the DJM global virtual machine. The practical experience gained from designing and implementing the proposed load balancing algorithm is shared in this paper.

Owing to the stochastic and dynamic nature of the practical Internet Computing environment, designing an "ideal" load balancing algorithm on it still remains a challenge. We hope our algorithm can serve as an example for continuing work on searching a general and practical solution.

REFERENCES

1 Barak M, R. Buyya R. Cluster Computing: The Commodity Supercomputer. *Software: Practice and Experience 1999;* 26(6), 551-576, 1999.

2 Gosling J, B. Joy B, G. Steele G. *The Java Language Specification*, Addison-Wesley, 1996.

3 Chandy KM, Rifkin A, Sivilotti PAG, Mandelson J, Richardson M, Tanaka W, Weisman L.. A World-Wide Distributed System Using Java and the Internet. In *Proc. of PDC-5'96*, 11-18, 1996.

4 Leung K-S, Lee K-H, Wong Y-Y. DJM: a Global Distributed Virtual Machine on Internet. *Software: Practice and Experience*, 28(12), 1269-1297, Oct 1998.

5 Sarmenta LFG. 1998. Bayaniham: web-based volunteer computing using Java. In *Proc. Worldwide Computing and Its Applications - WWCA'98*, 444-461.

6 Strumpen V. 1995. Coupling hundreds of workstations for parallel molecular sequence analysis. *Software: Practice and Experience*, 25(3), 291-304.

7 Lee K-H, Leung K-S, Wong Y-Y. DJM: a Novel Model for Distributed Computing on Internet & Intranet. In *Proc. IASTED Int. Conf. on Parallel and Distributed Sys.*, 340-345, Jun. 1997.

8 Mutka MW. A comparison of workload models of the capacity available for sharing among privately owned workstations. In *Proc. 24th Annual Hawaii Int. Conf. on System Sci*, 353-362, Jan. 1991.

9 Tandiary F, Kothari SC, Dixit A, E.W. Anderson EW. Batrun: Utilizing Idle Workstations for Large-Scale Computing. *IEEE Parallel & Distributed Technology*, 41-48, Summer 1996.

10 Bryant RM, Finkel RA. A stable distributed scheduling algorithm. In *Proc., 2nd Int. Conf. Distrib. Comput. Syst.*, Apr. 1981.

11 Eager DL, Lazowska ED, Zahorjan J. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Software Engineering*, SE-12, 662-675, May 1986.

12 Livny W, Melman M. Load balcncing in homogeneous broadcast distributed systems. *Proc. ACM Comp Network Perf. Symp.*, 47-55, 1982.

13 Ramamritham K, Stankovic JA. Dynamic task scheduling in distributed hard real-time system. *IEEE Software*, 1(3), 65-75, Jul. 1984.

14 Rotithor HG, Pyo. Decentralized decision making in adaptive task sharing, in *Proc. 2nd IEEE Symp. Parallel and Distributed Processing*, 34-41, Dec. 1990.

15 Wong Y-Y, Lee K-H, Leung K-S. A Stochastic Load Balancing Algorithm for Internet Computing Environment. In *Proc, Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Las Vegas (USA), 2587-2593, Jun 1999.

16 Casavant TL, Kuhl JG. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Software Engineering*, 14(2), 141-154, Feb. 1988.

17 Bokhari SH. Dual processor scheduling with dynamic reassignment, *IEEE Trans. Software Eng.*, SE-5, 4, Jul. 1979.

18 Tantawu AN, Towsley D. Optimal static load balancing in distributed computer systems. J. ACM, 32, 445-465, Apr. 1985.

19 Stone HS. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans Softw Eng*, SE-3, Jan. 1977.

20 Chu WW. Optimal file allocation in a multiple computing system. *IEEE Trans. Comput.*, C-18, 885-889, Oct. 1969.

21 El-Dessouki OI, Huan WH. Distributed enumeration on network computers. *IEEE Trans. Comput*, C-29, 828-825, Sep. 1980.

22 Reed DA, Fujimoto RM. *Multicopmuter Networks, Message-Based Parallel Processing*, Cambridge, MA, MIT Press, 1987.

23 Chow TCK, Abraham JA. Load balancing in distributed systems. *IEEE Trans. Software Engineering*, SE-8, 4, Jul. 1982.

24 Chu WW, Holloway LJ, Lau MT, Efe K. Task allocation in distributed data processing. *IEEE Computer*, 13, 57-69, Nov. 1980.

25 Efe K. Heuristic models of task assignment scheduling in distributed sytesm. *IEEE Computer*, 15(6), Jun. 1982.

26 Wang Y-T, Morris RJT. Load sharing in distributed systems. *IEEE Trans. Comput*., C-34, 204-217, Mar. 1985.

27 Price CC, Krishnaprasad S. Software Allocation Models for Distributed Systems. In Proc. 5th Int. Conf. on Distributed Computing, 40-47, 1984.

28 Chow YC, Kohler WH. Models for dynamic load balancing in a heterogeneous multiple processor systems. *IEEE Trans. Computers*, C-28(5),354-361, 1979.

29 Chow YC, Kohler WH. Models for dynamic load balancing in homogeneous multiple processor systems. *IEEE Trans. Computers*, C-36, 667-679, May 1982.

30 Lin H-C, Raghavendra CS. A Dynamic Load-Balancing Policy with a Central Job Dispatcher (LBC). *IEEE Trans. Software Engineering*, 18(2), Feb. 1992.

31 Ni LM, Hwang K. Optimal load balancing in a multiple processor system with many job classes. *IEEE Trans. Software Eng.*, SE-11, 491-496, May 1985.

32 Stankovic JA, Sidhu IS. An adaptive bidding algorithm for processes, clusters and distributed groups. In *Proc. 4th Int. Conf. on Distributed Comput. Syst*., 49-59, 1984.

33 Eager DL, Lazowska ED, Zahorjan J. A Comparison of Receiver Initiated and Sender Initiated Adaptive Load Sharing. *Performance Evaluation*, Vol. 6, 53-68, 1986.

34 Winston W. Optimality of the shortest line discipline. *SIAM J. Appl. Prob*. 14, 181-189, 1977.

35 Zhou S. A Trace-driver Simulation Study of Dynamic Load Balancing. *IEEE Trans. Software Eng.*, 14(9), Sep. 1988.

36 Barak A, Shiloh A. A distributed load-balancing policy for a multicomputer. *Software: Practice and Experience*, 15(9), 901-913, 1985.

37 Douglis F, Ousterhout J. Transparent process migration: design alternatives and the sprite implementaiton. *Software: Practice and Experience*, 21(8), 757-785, Aug. 1991.

38 Eager DL, Lazowska ED, Zahorjan J. The limited performance benefits of migrating active processes for load sharing. In *Proc. 12th ACM Symp. on Oper. Sys Prin.*, 63-72, May 1988.

39 Powell ML, Miller BP. Process migration in DEMOS/MP. In *Proc. 9th ACM Symp. on Operaing System Principles*, 110-119, Dec. 1983.

40 Theimer M, Lantz K, Cheriton D. Preemptable remote execution facilities for the V-System. In *Proc. 10th ACM Symp. Operat. Syst. Principles*, 2-12, Dec. 1985.

41 Zhu W, Socko P, Kiepuszewski B. Migration Impact on Load Balancing - An Experience on Amoeba. *Operating Systems Review*, 31(1), 43-53, 1997.

42 Dasgupta P, Majumder AK, Bhattacharya P. V_THR: An adaptive Load Balancing Algorithm. *J. of Parallel and Distributed Computing*, 42, 101-108, 1997.

43 Kunz T. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. *IEEE Trans. Software Engineering*, 17(7), 725-730, Jul. 1991.

44 Shivaratri NG, Singhal M. A load index and a transfer policy for global scheduling tasks with deadlines. *Concurrency: Practice and Experience*, 7(7), 671-688, Oct. 1995.

45 Mehra P, Wah BW. *Load Balancing, An Automated Learning Approach*, World Sci., 1995.

46 Loh PKK, Hsu WJ, Cai W, Sriskanthan N. How Network Topology Affects Dynamic Load Balancing. *IEEE Parallel & Distributed Technology*, 25-35, Fall 1996.

47 Shivaratri NG, Krueger P, Singhal M. Load distributing for locally distributed systems. *Computer,* 33-44, Dec 1992.

48 Goswami KK, Devarakonda M, Iyer RK. Peerdiction-based dynamic load-sharing heuristics. *IEEE Trans. Parallel & Distributed Systems*, 4(6), 638-648, Jun. 1993.

49 Ahmad I. A semi-distributed load balancing scheme for large multicomputer systems. In *Proc., 10th Int. Conf. on Distributed Computing Systems*, 562-569, May 1990.

50 Ni LM, Xu CW, Gendreau TB. A distributed drafting algorithm for load balancing. *IEEE Trans. Software Eng.*, 11(10), 1153-1161, Oct. 1985.

51 Ramamritham K, Stankovic JA, Zhao W. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Trans. on Computers, C-38, vol. 1, 1110-1123, Aug. 1989.*

52 Mirchandaney R, Towsley D, Stankovic JA. Analysis of the Effects of Delays on Load Sharing. IEEE Trans. Computers, 38(11), Nov. 1989.

53 Cover TM, Thomas JA. Gambling and Data Compression. 125-140, *Elements of Information Theory*, John Wiley & Sons Inc., 1991.

54 Hui CC, Chanson ST. A Hydro-dynamic Approach to Heterogeneous Dynamic Load Balancing in a Network of Computers. In *Proc. 1996 Int. Conf. on Parallel Processing*, vol. III, 140-147, 1996.

55 Bonomi F. Adaptive Optimal Load Balancing in a Nonhomogeneous Multiserver System with a Central Job Scheduler. *IEEE Trans Computers*, 39(10), Oct. 1990.

56 Kleinrock L. *Queueing System, Vol. II: Computer Applications*, John Wiley and Sons, 1976.

57 Kleinrock L. *Queueing System, Vol. I: Theory*, John Wiley and Sons, 1975.