On Load Balancing for Distributed Multiagent Computing

Ka-Po Chow and Yu-Kwong Kwok, *Member*, *IEEE*

Abstract—Multiagent computing on a cluster of workstations is widely envisioned to be a powerful paradigm for building useful distributed applications. The agents of the system span across all the machines of a cluster. Just like the case of traditional distributed systems, load balancing becomes an area of concern. With different characteristics between ordinary processes and agents, it is both interesting and useful to investigate whether conventional load-balancing strategies are also applicable and sufficient to cope with the newly emerging needs, such as coping with temporally continuous agents, devising a performance metric for multiagent systems, and taking into account the vast amount of communication and interaction among agent. This paper discusses the above issues with reference to agent properties and load balancing techniques and outlines the space of load-balancing design choices in the arena of multiagent computing. In view of the special agent characteristics, a novel communication-based load-balancing algorithm is proposed, implemented, and evaluated. The proposed algorithm works by associating a credit value with each agent. The credit of an agent depends on its affinity to a machine, its current workload, its communication behavior, and mobility, etc. When a load imbalance occurs, the credits of all agents are examined and an agent with a lower credit value is migrated to relatively lightly loaded machine in the system. Quasi-simulated experiments of this algorithm show load-balancing improvement compared with conventional workload-oriented load-balancing schemes.

Index Terms—Load balancing, distributed systems, cluster computing, multiagent computing, object-based systems, communication.

1 Introduction

Multiagent systems have recently been widely employed in developing scalable software systems on heterogeneous networks [6], [13], [23]. Indeed, using a cross-platform language (such as Java in most cases), distributed systems based on agents are very attractive because of the inherent scalability and autonomy. Viewing the software agents (usually in the form of objects) as "brokers" and the interactions among agents as the exchange of "services," a multiagent system closely resembles a community of human beings doing business with each other, and are widely envisioned to be able to perform many commercial activities on behalf of human beings (see Table 1 [5] for a list of properties that agents generally have in common and distinguish them from traditional processes or threads). Thus, many financial applications are being built using such a multiagent paradigm [8], [23]. Examples of notable multiagent systems include those reported in [9], [10], [12], [19], [25]. However, the end-users (e.g., the customer who employs the software agents to locate the cheapest prices of certain commodities in the financial markets) definitely demand a quick response from such a multiagent system. Because of the unique features (e.g., mobility, autonomy, etc.) of a software agent (detailed below), load imbalance is inevitable over time and a longer response time results. Specifically, if the agents are fully autonomous in the migration from machines to machines without any coordination with respect to the load levels on the machines, it is very likely that some machines are overloaded and become bottlenecks in the distributed computation process. Thus, load balancing, as a system service, is very much needed to make such a multiagent distributed computing paradigm attractive and useful for implementing large-scale applications.

In dynamic load balancing research [1], [2], [15], [16], [17], the whole cluster of machines (PCs and/or workstations) is often viewed as a common resource to which users submit their own jobs through the workstations (see [4], [27], [28], [33], for a good overview of the general load-balancing problem). Dynamic load-balancing schemes distribute the jobs among the workstations so as to balance the workload. These jobs run until they are finished or killed. The job assignment decisions require no a priori job information, such as execution time, resources needed and communications to be performed. New jobs may be added to the cluster of machines [11], [24] at any time by the users, and are scheduled by the load-balancing system.

The above scenario, nevertheless, cannot be applied to a distributed multiagent system. To begin with, agents are temporally continuous. They do not cease to exist unless the whole system is shut down or they are extinguished out of the needs of their internal agent management. In addition, very often, nearly all the agents are specified and created at system startup. Seldom is there a need to introduce more agents to the system. Furthermore, agents interact among themselves through highly variable communication patterns, while the communication between the jobs in a cluster are usually with a static pattern. The focus of agent research has been on its philosophy, theory, language, and architecture. There are also some multiagent systems which aim at load-balancing problems [29]. Very little published

• *The authors are with the Department of Electrical and Electronic Engineering, The University of Hong Kong, Pokfulam Road, Hong Kong. E-mail: {kpchow, ykwok}@eee.hku.hk.*

Manuscript received 1 Jan. 2001; revised 7 Sept. 2001; accepted 22 Jan. 2002. For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 113378.

TABLE 1
Agent Properties

Property	Meaning
reactive	responds in a timely fashion to changes in the environment
autonomous	exercises control over its own actions
goal-oriented/proactive	does not simply act in response to the environment
temporally continuous	is a continually running process
communicative/socially able	communicates with other agents, perhaps including people
learning/adaptive	changes its behavior based on its past experience
mobile	able to transport itself from one machine to another
flexible	actions are not scripted
cloning	duplicates itself to achieve better performance
character	believable "personality" and emotional state

research is done on the load-balancing aspect to capture the essence of agent systems in such a distributed environment. The introduction of agent technology has opened up new opportunities for further research of load-balancing with respect to agent technology. This research aims at bridging this gap as a first attempt by examining load-balancing strategies with respect to their application to multiagent systems.

This paper is organized as follows: As detailed in Section 2, we translate such characteristics into corresponding load-balancing strategies which are implemented in our newly proposed load-balancing model for evaluation. We constructed an agent test platform so that experimental results can be obtained with variation of system parameters, as described in Section 3. In Section 4, analysis of the results shows an improvement made by our new communication-based load-balancing algorithm, called Comet, in the place of an agent-based system, compared with traditional workload-based load-balancing algorithms. Several future research directions are also identified in Section 5.

2 Credit-Based Load-Balancing Model

In this section, we introduce a new load-balancing model, Credit-Based Load Balancing Model, which aims at capturing some of the necessary agent characteristics. It lets us analyze the dynamics of load-balancing operations with respect to multiagent systems.

2.1 Load-Balancing Model Overview

In dynamic load-balancing schemes, the two most important policies are selection policy and location policy. As previously discussed, the selection policy deals with which task or agent is migrated whenever there is a need, while the location policy determines to which destination machine the selected agent is migrated to. If the selection policy is formulated carefully, the desirable effects are that the agent selected to migrate will not make the overall situation worse by making the destination more overloaded than the source, and the cost of the migration will be compensated by the gain in performance. Likewise, should the location policy be properly planned, the overall system workload will be more averaged after the migration.

The credit-based load-balancing model focuses on these two policies. We assign a numerical value, called credit, to every agent. The credit indicates the tendency of the agent to remain undisturbed in case migration is under consideration. To an agent, the higher its credit, the higher its chance to stay at the same machine, which is equivalent to saying that its chance to be selected for migration is lower.

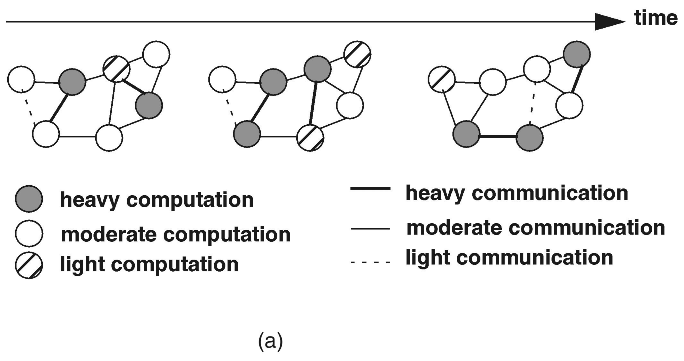
The credit of each agent changes in accordance with the behavior of the agent system and agent interaction. The credit of an agent increases in the following ways:

- its workload decreases,
- it communicates with other agents also residing at the same machine, or
- it has a high affinity with the local machine. For example, it requires a special type of processors, I/O devices, or large amounts of data localized at the machine.

On the contrary, the credit of an agent decreases in the cases below:

- its workload increases,
- it communicates with other agents residing at other machines,
- it has a high mobility, i.e., it can be migrated elsewhere very easily, or
- it has just sent or received an agent message which indicates that the agent will probably become busier in a short while.

Given such a credit, we now have a common standard to describe the credibility of each agents tendency to stay at the same machine. As the interactions and communications among the agents continue, the credit of each agent changes accordingly. Such a credit can be used in the selection



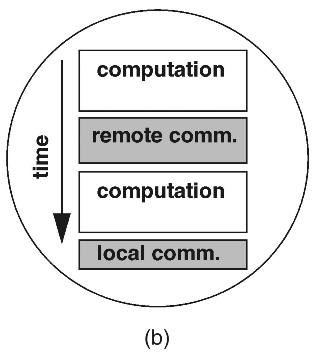


Fig. 1. (a) Iterative and dynamic nature of a multiagent application. (b) The structure of an agent.

policy, where the agent with the lowest value is identified and migrated.

One interesting aspect of the Comet algorithm is that, if an agent has a higher computation workload, it will have a higher chance of being selected for migration. On the surface, this design choice seems to be counterintuitive. However, we incorporate this selection mechanism based on our observations on the variation of overall system workload in that when an agent has a higher workload, such an increase is usually a localized event and the duration can be quite significant. Thus, it makes sense to move such an agent to a relatively lightly loaded machine.

The remaining question is the location policy. This is not addressed by the credit model. It is, however, determined indirectly during the computation of the credits. The computation of an agents credit requires the communication loading that this agent makes with every other agent. The information of communication loading with each agent is gathered in a vector. After the selection policy has determined which agent is to be migrated, the location policy will be in effect. The location policy first identifies which remote agent will perform the most communication with the agent to be migrated. The machine at which this remote agent resides is selected as the destination machine.

Based on this model, the load-balancing problem in a multiagent system can be viewed as the periodic calculation of credits for each agent. No actions are needed until the loading level of a machine exceeds a certain predefined threshold. At this point, the selection policy makes the decision based on each agents credit. The location policy also inherently determines which machine receives the migrated agent.

In [34], Zhou points out that the failure of a central host should be detectable in a rapid manner. A new host can thus start up subsequently to assume the load collection duty, or another running machine can take up the responsibility instantly. The loss of load balancing for a short period of time is not critical and detrimental since all work arriving in the interim may be scheduled locally. As regards the performance bottleneck problem, Zhou's experimental results indicate that the marginal increase in the likelihood of finding an underloaded host is offset by the linearly increasing cost of state dissemination. He suggests that a reasonable cluster size for visible gain in aggregated response is about 30 and a slower time scale in state dissemination process may well prove a more effective approach. With such a cluster size, the load-balancing information flow is not high enough to saturate the central host. This follows that the possible threat of bottleneck posed by it does not have sufficient grounds.

2.2 The Comet Algorithm

In the multiagent system considered in our study, we assume an application is composed of agents executable on any of the p machines of the cluster. The structure of the application is modeled by the interdependence relationships among the agents. More specifically, we use an undirected graph to model the application structure. An undirected graph is an appropriate generic model because a multiagent application executes perpetually and produces results continuously in response to user queries (e.g., financial database queries).

One particular feature in our multiagent system is that the communication pattern among the agents is known. Even if it changes, all such changes are registered on the Agent Name Server and JATLite Message Router. Using this feature, we can arrange the agents to minimize communication overhead through the interconnection network. Notice that the computational load of an agent and the communication load between two agents may be different for processing different queries. Thus, the data or control dependencies among agents are not constant. Hence, the communication dependency relationship between any two agents cannot be modeled by a directed edge. Indeed, an edge between two agents only indicates that the agents communicate during the processing of a particular query but does not imply a precedence relationship. Given these agent characteristics, we can also see that the application is inherently iterative in nature in that each iteration corresponds to the execution of the application for one particular query. Fig. 1 illustrates the dynamics and structure of the multiagent application.

Traditional load-balancing strategies commonly use the computational load of a process as the load index based on the assumption that computation is the dominant activity in a process and communication can be fully overlapped with computation [21], [28], [32]. While this approximation might be valid for heavy weight processes in a distributed system, such a load index is clearly not appropriate for the multiagent system considered in our study. As mentioned earlier, the computation load within an agent is sometimes

TABLE 2
Definitions of Notations

Symbol	Definition
m_k	machine (host) k ($k = 1, \ldots, p$)
a_i	agent i ($i = 1, \ldots, n$)
w_i	computation load of a_i in terms of clock cycles
u_i	communication load of a_i in terms of clock cycles
h_i	intra-machine communication load of a_i
g_i	inter-machine communication load of a_i
$c(a_i, a_j)$	the number of clock cycles needed for communication between a_i and a_j if the two agents are mapped to the same machine
L_k	total load of m_k
f	remote communication scaling factor (the bandwidth ratio of inter-machine communication to intra-machine communication)
$M(a_i)$	the machine index on which a_i is residing
C_i	credit value of a_i

minimal (e.g., a fast execution of a certain financial formula) and a BSP style of multithreaded programming model [31] is more accurate. Thus, we propose a composite attribute to indicate the load of an agent that takes into account the effect of remote and local communications among agents. Specifically, the load of an agent executing on machine is defined as the sum of its computational load and the communication load (see Table 2 for a summary of notations), where:

$$u_i = h_i + g_i = \sum_{M(a_i)=M(a_j)} c(a_i, a_j) + \frac{f}{2} \sum_{M(a_i)\neq M(a_j)} c(a_i, a_j)$$

(note that a_j may be local or remote depending upon the value of $M(a_j)$). Here, h_i and g_i represent the intramachine and intermachine communication load, respectively. The factor 2 is included to avoid doubly counting the inter-machine communication. The value of w_i is computed dynamically by examining the clock ticks in a specified period. Note that the communication cost $c(a_i, a_j)$ can be computed by each agent using the message sizes. The scaling factor (> 1) is system dependent and calculated based on the network bandwidth of the system. With networking technology nowadays, intermachine communication is often approximately more than an order of magnitude slower than intramachine communication. The load L_k of a machine m_k is defined as the sum of all its local agents load. More specifically,

$$L_k = \sum_{M(a_i)=k} (w_i + u_i).$$

The goal of a load-balancing algorithm is to minimize the variance of the load among all the machines in the cluster. This will, in turn, minimize the average response time of serving users queries.

With the above definition of load index, an overview of the proposed Comet load-balancing algorithm is in order. Below, we describe the important aspects of the Comet system.

Information policy: In the Comet system, we employ a distributed periodic information gathering policy. Specifically, the machines in the system synchronize periodically and check their local aggregate load against the load threshold, which is the estimated high (T_H) load level computed based on measured load values obtained by profiling the system. Certainly, the load threshold is related to the given problem size of the application (i.e., number of agents involved and the activity levels of the agents as determined by the load exerted by the users, e.g., number of queries submitted) and the average load required to give a reasonably fast response time. In our implementation, we allow the administrator of the system to change the load threshold online. That is, although the system starts with a load threshold (specified as a command line argument) determined a priori by profiling the application, the threshold (and the period of information collection) can be changed when needed (e.g., for some administrative reasons). In fact, to assist the administrator, our system reports the instantaneous average loads and the variances before and after each load-balancing phase. Finally, a centralized approach is used in decision policy, i.e., a central entity determines whether there is a need to perform agent migration. In short, our model employs distributed information gathering policy and centralized migration decision-making policy.

Selection policy: Each agent a_i is associated with a credit C_i which is defined as: $C_i = -x_1 w_i + x_2 h_i - x_3 g_i$, where x_1, x_2, x_3 are positive real numbers and are application-dependent coefficients. The rationale of this credit attribute is to capture the *affinity* of an agent to the machine in that the intramachine communication component contributes positively to the credit, whereas the reverse is true for the

```
ALGORITHM: Comet
Input: threshold load value $T_H$ (determined by profiling), information collection period
$\tau$ (in seconds), number of agents $n$, hosts' characteristics (address and other system
information) and number of hosts $p$.
1.  Initialization: check the current locations of the agents;
2.  while (true) do {
3.      Information collection: compute the credits of each agent and compute the
        load on each host;
4.      for each host with load greater than $T_H$ do {
5.          migrate the agent with the smallest credit according to the selection and
            location policies;
6.          repeat the process until the load is smaller than $T_H$;
7.      }
8.      sleep ($\tau$); // wait for wakeup signal also
9.  }
```

Fig. 2. Algorithm: Comet.

intermachine communication. In the sender machine, the agent with the smallest credit is selected for migration because such an agent spends a dominant amount of time communicating with a remote agent and, hence, is a suitable candidate for migration in order to reduce the local load level. After migration, the heavy intermachine communication becomes local communication in the receiver machine. Although some local communication in the sender machine also becomes intermachine communication, the overall effect is still desirable because the senders load is reduced. Finally, note that we assume all agents are equally "mobile," meaning that they can be migrated to another machine with the same ease. If, in the scenario, there are some agents that are relatively not mobile (i.e., they have affinity to certain machines which may have special I/O facilities), we can add a constant to computation of the values of C_i.

Location policy: After a migrating agent is selected, we need to determine the target machine. In the Comet system, each agent a_i keeps track of a p-element vector storing the value of remote communication (the local communication component is stored as the $M(a_i)$th element) between the local machine and all other machines in the network. Specifically, each element v_s of the vector is $\sum_{M(a_j)=s} c(a_i, a_j)$. Suppose the yth element ($y \neq M(a_i)$) is the largest element. Then, machine m_y will be chosen as the receiver of the agent.

With the above descriptions, the Comet algorithm is outlined in pseudocode format in Fig. 2. Notice that the proposed location policy is novel in that we implicitly specify a receiver machine in the network for a migrating agent. By contrast, most existing load-balancing schemes require the system to determine a receiver which is usually the one with the lowest load. It should be noted that, for multiagent systems in which communication is the dominant event, choosing the lowest load machine may not be a suitable strategy because such a machine may not necessarily reduce the intermachine communication overhead, which is a dominant part of the aggregate load. The rationale of selecting the lowest load machine as the receiver is to avoid *thrashing*. However, the Comet system is inherently free of thrashing because the total load across all machines is reduced after each migration. To see this, consider the situation shown in Fig. 3.

Here, agent a_i is selected for migration from machine m_k to machine m_l due to its heavy communication with remote agents on m_l (e.g., agent a_j). Now, the aggregate load on m_k before migration is given by: $L_k = \sum_{M(a_z)=k}(w_z + h_z + g_z)$. Let W, H, and G denote $\sum w$, $\sum h$, and $\sum g$, respectively. Also, let $\sum_{M(a_j)=l} v(a_I, a_j) = \beta$ and $\sum_{M(a_z)=k} c(a_i, a_j) = \alpha$. Then, after migration, the new values of load are given by:

$$L'_k = (W_k - w_i) + (H_k - \alpha) + \left(G_k + \frac{f\alpha}{2} - \frac{f\beta}{2}\right)$$
$$L'_l = (W_l + w_i) + (H_l + \beta) + \left(G_l - \frac{f\beta}{2} + \frac{f\alpha}{2}\right).$$

Thus, the total load[1] of the two machines after migration is given by: $L'_k + L'_l = L_k + L_l + (\beta - \alpha)(1 - f)$ and, hence, the total load is reduced because $f > 1$ and $\beta > \alpha$ (otherwise, a_i will not be selected). Thus, thrashing will not occur in the Comet system because thrashing occurs if and only if the aggregate load across the whole system is nondecreasing over time. Given this stability characteristic and the incremental load reduction process (through migration), the Comet scheme is able to minimize the variance of load across all the machines and, hence, optimize the average response time of user queries.

3 Experimental Environment

This section describes the experimental environment of this research in which measurements were taken, what inputs were used to drive the system, and what outputs were collected for subsequent analysis and evaluation. Implementation details of the experimental system are illustrated. System operations, like information dissemination, agent migration, and results collection, are also explained.

1. Strictly speaking, there should be a cost (in terms of time, for example) associated with the migration. However, as we are interested in the "steady state" overall workload of the system, such a transient cost is ignored in the analysis.

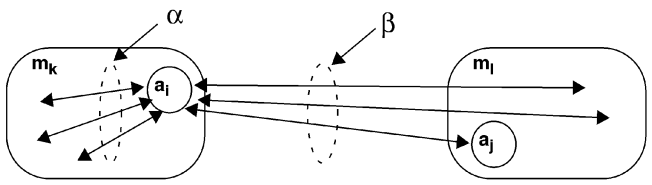


Fig. 3. The situation before agent a_i migrates to machine m_l.

3.1 System Overview

To investigate the effects of Comet, a multiagent system has been implemented to provide a testing platform of the credit-based load-balancing model. The whole system is written in Java using JDK 1.17 and is implemented on a cluster of 16 machines, which comprises eight PCs running Linux 2.0.36 (RedHat 5.2 distribution) and eight Sun UltraSparc I Workstations, connected through a 155 Mbps ATM network. The configurations of these 16 machines are listed in Table 3. These machines reside at the High Performance Computer Research Laboratory of the Department of Electrical and Electronic Engineering, and Systems Research Group Laboratory of the Department of Computer Science and Information Systems, the University of Hong Kong. All the machines have separate hard drives, but the same set of files are mirrored on every one of them with the same directory structure. This is to ensure that all the related initialization and configuration files are accessible from every machine, and network file systems are not adopted to avoid I/O bottleneck.

3.2 JATLite: A Java Agent Development Package

An agent package, called JATLite [14], is used to realize the agent interactions and communications. JATLite is an open-source package developed by Stanford University and currently of version 0.4 beta. It provides a set of Java packages that facilitate the agent framework development using the Java language.

In a system implemented with JATLite, there is an entity, called agent message router (AMR), which acts as the central backbone for agent communication. Each agent has to register its name with this AMR before they can interact with other registered agents. After successful registration, the agent can send messages to and receive from other agents through AMR, regardless of the location of the destination machines. In each message transmission, the whole message is transferred from the source agent to the AMR first, and the AMR looks up its registration database to find out the location of the destination agent. After that, the message is forwarded to the destination agent. No direct connection between the two agents is established.

This way of message transfer is in contrast to other agent packages agent name server (ANS), which only resolves agent names into their real locations. In a system with ANS implementation, every agent has to query the ANS about the physical location of a destination machine. Afterwards, it still has to make a separate connection with the destination agent before it starts the message delivery. The ANS can be conceived as the name resolution server in a TCP/IP network. Each machine just needs a symbolic name of another machine without noting the physical location. As such, changing the physical location address will not affect the system functioning.

In JATLite, since every message has to be transferred to the AMR first before it is forwarded to the destination agent, the incurred overhead compared with the case of direct connection may be very significant. To cite an example, both the source and the destination agents may reside at the same host, while the AMR is in a remote host. Although the source agent can send the message directly to the other party without using the external network, the AMR implementation still makes no exception and needs to go for a round-trip. In the light of this potentially undesirable overhead, we decided to abort the utilization of such an AMR in the JATLite implementation. Rather, we implement a distributed name resolution (DNR) mechanism to replace the original ANS function carried out by the JATLite AMR. This makes our implementation slightly more complicated, but brings a great degree of flexibility and adaptability. Another advantage of using JATLite is that it has incorporated a KQML layer [20]. This allows us to exchange KQML, one of the de facto standard agent communication languages, without further effort. The KQML layer provides encapsulation, transmission, and extraction functionalities which are indispensable to agent interactions.

3.3 Classification of Hosts

The overall configuration is shown in Fig. 4. There exists a central host. Its role is to initiate system startup, suspension, and termination. It is also responsible for providing an interface for query of the current system loading and agent distribution. The central host does a number of lightweight tasks, and the volume of information exchanged with it

TABLE 3
Machines in the Experimental System Setup

Model	Intel PC	Sun UltraSPARC I Model
Processor	450 MHz Dual P6	143 MHz UltraSPARC I
OS	Linux 2.0.36	SunOS 5.6
Main Memory	128 MB	64 MB
Number of Machines	8	8

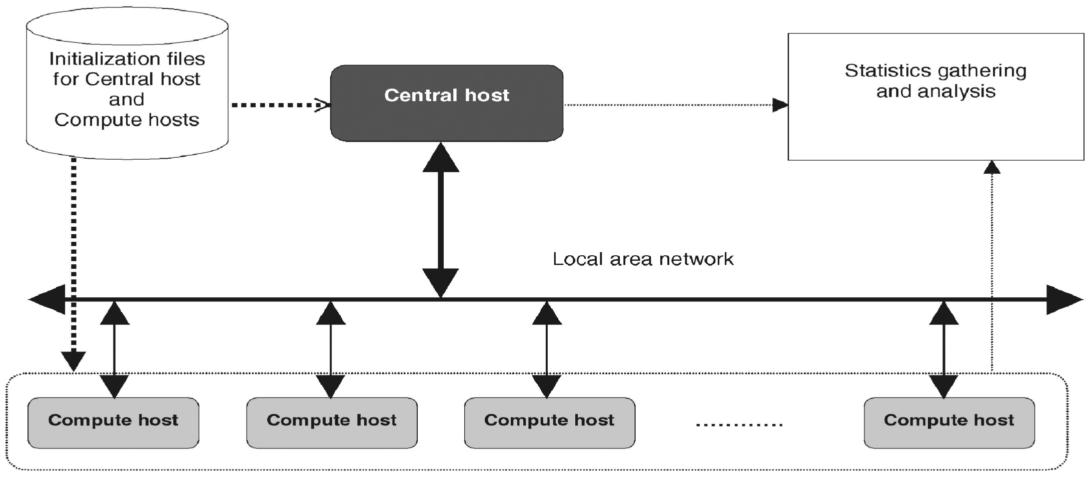


Fig. 4. System overview.

should not be high. Therefore, it is not considered as a serious bottleneck.

Central host is also the decision-maker about whether there is a need for agent migration, and also the chief commander of selection and location policies. Other than the central host, there are a number of compute hosts. These are the actual hosts which do the computation and among which agents are started and run. Migration of agents may occur as needed in-between the compute hosts. A set of initialization files are needed to inform the central host which compute hosts are available and participate in the system operation. They also contain the computation parameters being used by the compute hosts. The number of agents being started is also specified in the files. During every system run, statistics about the workload variations and migration decisions are recorded. At each migration point, the workload distribution situation before and after migration provide useful indicators about whether the current policies can really offer good load-balancing judgment.

3.4 Classification of Agents

This system consists of three types of agents. The majority of agents belong to the type Work Agent. It is the type which does all the computation and resides at the compute hosts. There may be one or more Work Agents in each compute host. The second type is the Communication Agent (Comm Agent). There is one Comm Agent at each compute host. It needs to startup those Work Agents destined to live at the same compute host, collects the credit values of each of them, and submits the values to the Central Agent, which is the third type of agents. There is only one Central Agent residing at the central host. It gathers all the information submitted by each Comm Agent, detects whether it is a suitable time to initiate a migration process, and determines which Work Agent should be migrated to which compute host. All Comm Agents are, in turn, started up by the Central Agent.

Being the administrative agents, all Comm Agents and the Central Agent are not involved in the migration. They remain at their original locations until the system is shut down. In our implementation, all the Work Agents are actually identical in the program code. They differ in the coefficients supplied to them, which, in turn, will vary their behavior. The Work Agents interact among themselves according to the parameters. The initialization files also specify which other Work Agents each of them has to interact with. In other words, the files define the communication pattern. Fig. 5 illustrates the relationship of the hosts and the three types of agents. The Central Agent has a close connection with all Comm Agents. Each Comm Agent closely monitors all Work Agents residing at the same machine. All the Work Agents interact with one another as specified by some initialization files. We can see that this multiagent system is itself monitored by agents implemented by the same agent package.

3.5 System Hierarchy

Our system is built on top of the JATLite KQML Layer as shown in Fig. 6. JATLite does offer a Router Layer, but as said previously, it requires all message delivery to pass through the router, we did not implement our system upon the Router Layer. The hierarchy of our system is described as follows: The lowest layer is the Java virtual machine, which interprets the Java bytecodes. The second lowest layer is the core system of the JATLite package. It is written in Java. As a result, it is placed on top of the Java virtual machine. The abstract layer provides a collection of abstract classes necessary for JATLite implementation. The base layer provides basic communication based on TCP/IP and the abstract layer. It is independent of the protocol or message language used in the higher layer. The KQML layer provides the encapsulation and parsing of KQML messages. The Agent management layer provides a basis to initiate the Work Agents in the above layer and acts as a coordinator for all agent-related management. The highest layer is where Work Agents do computations and interact. The hierarchies may pose a threat of too much overhead. It should be noted, however, that the most serious overhead lies in the implementation of the Java Virtual Machine, the basis of all Java programs. All the above layers represent one or two function calls and are minimal compared with the JVM layer.

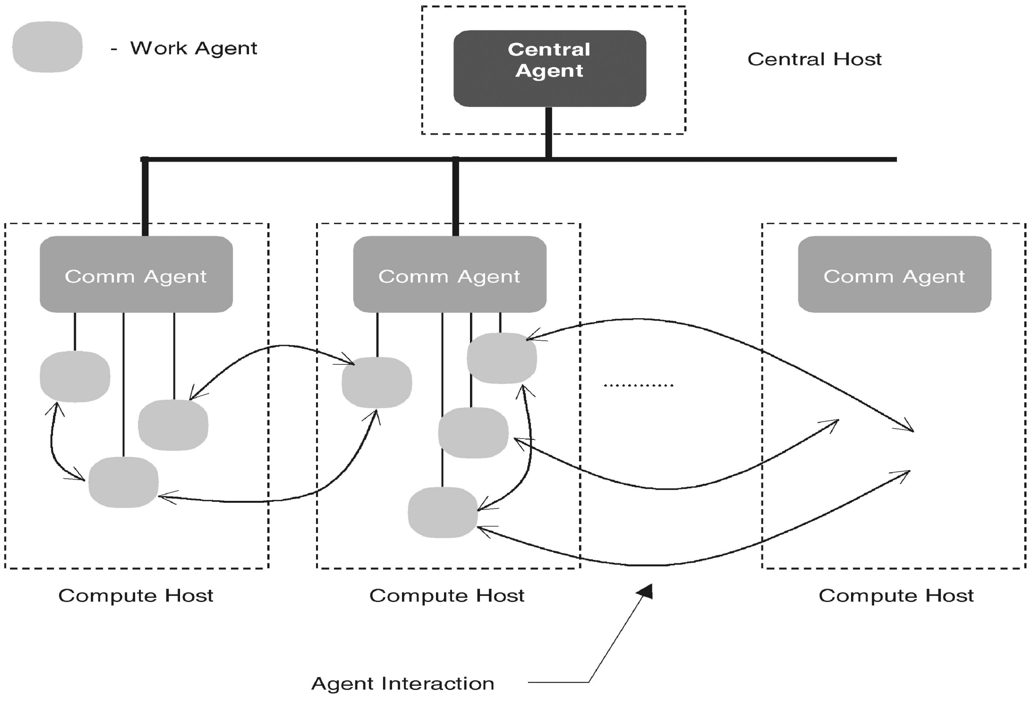


Fig. 5. Agent interactions.

3.6 Load Information Dissemination

The flow of load information is an implementation of information policy. All the Work Agents maintain their own credit values derived from various policies. Their credit values are gathered periodically by the Comm Agent resident at that compute host. Finally, the Central Agent at the central host obtains the credit values gathered by each Comm Agent. In summary, each Work Agent is responsible for its own credit value at the agent level, while each Comm Agent is responsible for collecting the credits from the Work Agents at the same machine. The Central Agent makes load-balancing decisions after it has acquired all the necessary information.

3.7 Migration Operations

As shown in Fig. 7, it is the Central Agent that determines whether there is a need to perform agent migration, judging from the collected credit values. Thus, the Central Agent realizes the migration initiation policy. After an agent is selected to migrate from its current host (the source machine) to another compute host (the destination machine), the Central Agent contacts the Comm Agents at both the source machine and the destination machine.

1. **Source machine**: At the source machine side, the Comm Agent informs the selected Work Agent about the migration decision. In our system, we assume that negotiation is implicit and can be represented by the communication patterns among agents. Effectively, whenever the Central Agent wants to migrate a Work Agent, it is able to do so without the Work Agent expressing any disagreement.

   When the Work Agent has been informed about the migration needs, it enters a suspend state and ceases to serve other Work Agents queries. Instead,

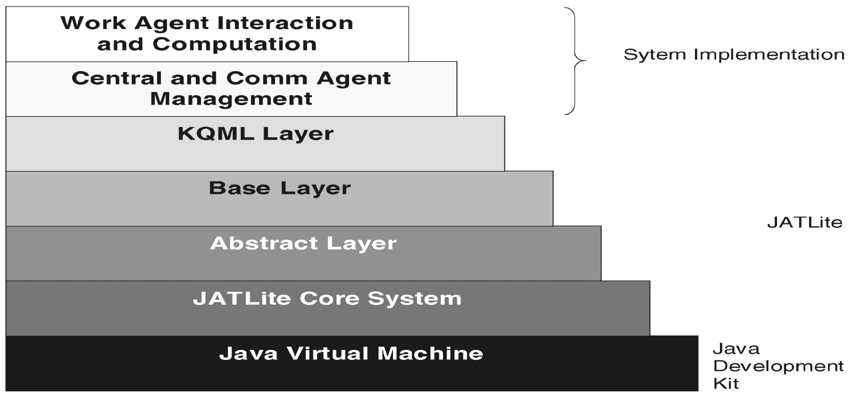


Fig. 6. System hierarchy.

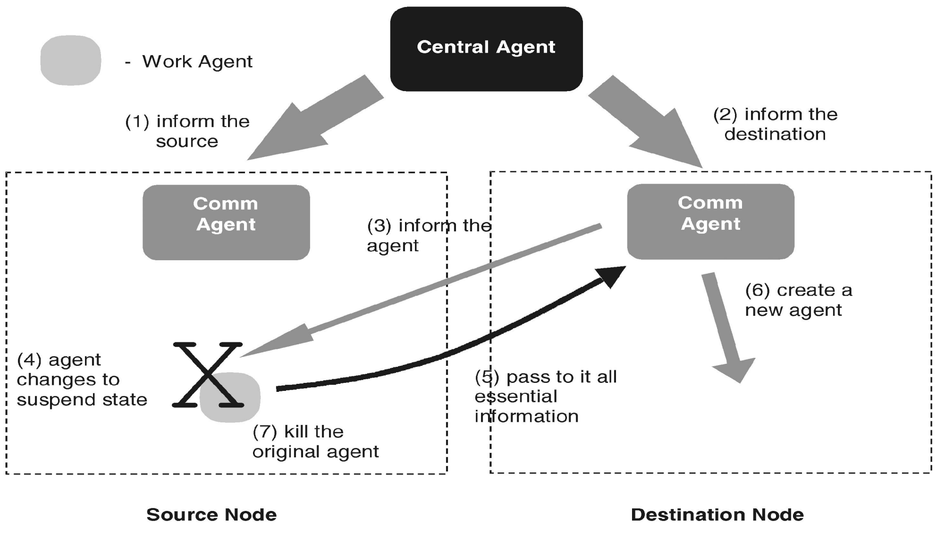


Fig. 7. Agent migration.

it responds with a suspend KQML message. Other Work Agents need to submit another query after some time.

2. **Destination machine**: The Comm Agent at the destination machine side starts up another Work Agent. Since the program code is the same for all Work Agents, the process of migration in our system is, in fact, an initialization of another identical agent, accompanied by a termination of the original one.

   After the Work Agent is started at the destination machine, a connection channel is built between the resurrected and the dying Work Agents. All the necessary and useful information is transferred from the dying agent to the resurrected one, including current states, coefficients for computation, and information about connection channels established. Consequently, the original Work Agent is terminated. This follows that the resurrected Work Agent functions exactly the same as the dead agent, essentially realizing the whole migration process.

3.8 Synthetic Workload

The two main approaches for feeding workload as inputs to load-balancing algorithms are synthetic workload and trace workload. Many experiments on load-balancing algorithms were done with a traced workload. These workloads were obtained by recording the variations of workload in real-life systems, like a cluster of workstations provided for general use. Synthetic workloads are created by crafting artificial workloads with mathematical characteristics. Some examples are exponential distribution [26], [30], [22] and two-stage hyperexponential distribution [3], etc.

There was no trace workload data available for use in this multiagent system because there does not exist a representative workload model, and the synthetic workload is used in our experiments instead of traced workload. Specifically, all the workloads were a random amount of program loops in which there are large amounts of numerical calculations. Each Work Agent in the system is given a range within which a random number is generated. Workloads and communication are performed according to this random number. The random number range stays constant for the same Work Agent, so as to maintain the consistent behavior of individual agents. Because of the above nature of the workload fed into the system, our experiments are quasisimulated, in the sense that the system is not simulated theoretically using a programmable simulator, but rather, it is implemented realistically, but with artificial workload.

3.9 Workload Parameters

In the experiments conducted, several parameters have been identified to formulate the workload for our purpose:

- Agent Computation Load: This represents the level of computation workload in each agent. This workload is generated by a combination of loops and calculations. The amount of workload is controlled by a random number generated as described in the previous section.
- Message Size: This represents the size of messages transferred during each communication process. It is also controlled by a random number, like the case for computation load.
- Intermessage Duration: This is the duration elapsed between two message exchanges. It defines how frequently the communication is performed between two Work Agents. Based on our prior observations on the behaviors of the agents (we used agents that perform financial type computations) in our Ethernet network, the intermessage duration is specified to be 10 seconds for all Work Agents in this system.
- Computation-Communication Correlation: To maintain a consistent agent behavior, we correlate the computation and communication of an agent such that an agent with a higher level of computation workload, its communication performed also contains larger-sized messages, and vice versa. This is achieved in the system by making use of the same random number to generate workload and message size.

3.10 Experimental Parameters

For each experiment, several parameters have to be specified. They are described in this section.

Credit Coefficients: In our study, we have adopted a linear model in the calculation of credits, $C_i = -x_1 w_i + x_2 h_i - x_3 g_i$, where x_1, x_2, and x_3 are the coefficients which aim to capture the degree of the affinity of an agent to the machine. The values of these three coefficients can be varied as to what extent we want to emphasize different aspects. For example, if the effect of intermachine communication is to be emphasized, more, x_3 should be increased. Another use of changing the values of the coefficients is to adjust the weighting between the two different natures of measurement: workload and message size. To make them comparable, some adjustments must be made to the coefficients. In our experiments, we took all of them as unity but we would like to emphasize that these coefficients could be fine tuned for specific applications. Indeed, as detailed in Section 4, some experimental results are also gathered by varying these coefficients.

Number of Agents: It is easily seen that the number of agents is one of the most important parameters in a multiagent system experiment. When the average number of agents in a machine is small, less choices are available for the selection policy and the effect of adding an agent to a machine or removing one is very significant, directly making the load-balancing algorithm less efficient. However, when the average number of agents in a machine becomes large, the main memory of a machine may be insufficient to house all the agents. This is especially true for Java agents since each Java process must be accompanied with a Java virtual machine, which is notorious for its high memory usage and sluggish performance. When some Java agents are forced to swap out of the main memory, the running time will be greatly increased, which is out of the controlling scope of the system. From our trial experiments, we found that a Linux PC box with 128 MB can house only about 12 agents without extensive swapping. These numbers vary from one Java version to another, from one OS to another, and from one machine configuration to others, too. Because of this, there is a limited range for the number of agents in order to obtain meaningful and comparable results.

Number of Hosts: The number of hosts is limited by the available resources only. In this research, we are only concerned about homogenous machines for easier comparison of raw UNIX loading values from different hosts. Heterogeneous clusters can be included if some conversion mechanism is available so that raw loading values can be normalized before comparison. It should also be noted that when the number of hosts increases while average number of agents in each host remains unchanged, the number of edges in their communication pattern may increase in a greater magnitude, and the workload collection cycle will take a longer time.

Period of Workload Collection: From the first set of trial experiments, we found that the period of workload collection has a great impact on the efficiency of the system and the load-balancing algorithm. If we want to keep the workload as balanced as possible, it follows immediately that we need to keep our information about the present situation of the system most up-to-date and, thus, the period of workload collection should be very small. Nevertheless, when the workload information is to be collected incessantly, such a collection process itself will contribute tremendous workload to the multiagent system under investigation, jeopardizing the effectiveness of the algorithm. Toward another extreme, when the period is large, the information used for load-balancing decision making will be already obsolete. It is by no means a good indicator of the present loading situation of the whole system. In view of the above, it is necessary to strike a balance between the two implications. After some trials, we decided to take 60 seconds as the data collection period in our experiments. This is also the period when the Central Agent makes decisions about whether an agent should be migrated or not.

Communication Pattern: Since agents must communicate with one another, they form a communication pattern. Agents establish connections and transmit messages according to this pattern. Such a communication pattern has profound significance to the workload situations and load-balancing decisions. The reason is that when an agent has a communication pair with another agent and they exchange a large amount of messages, the vector used in location policy will tend to migrate the two agents together once one of them is selected in the migration policy. A communication pattern cannot be translated into simple parameters directly. In our experiments, we have made use of a new parameter, called pair ratio, which defines the portion of communication pairs that will exist among all possible communication pairs. After defining the number of agents, a communication pattern is generated with this pair ratio as a reference. Communication pairs are created among randomly selected pairs of agents. The pair ratio in our experiments ranged from $\frac{1}{4}$ to $\frac{1}{8}$. A smaller pair ratio was used for a higher number of agents while a large pair ratio was for a smaller number of agents. The objective is not to overwhelm the agent system with message transfers. Otherwise, one cycle of message transfers will be too long, making our workload data collection ineffective. Moreover, the experiments aim to investigate the effects of load-balancing with different paradigms, the choice of pair ratio does not infringe this basic objective.

3.11 Points of Examination

In our experiments, we measured two values which are our points of examination. They are variations of normalized standard deviation over time for each run, and the average normalized standard deviation for each set of experimental parameters.

Workload Data Normalization: When different algorithms are tested, even though they are tested with the same set of program and data using the same cluster of machines, the workload of the background user-level and kernel-level jobs may affect the workload exhibited by the agents concerned. A normalization process is therefore required to convert the value to enable direct comparison. The normalization is done by dividing the measured value by the mean workload during the same run. This has the similar effect as excluding the white noise in spectral analysis, with the value of white noise obtained from the mean measured value.

Normalized Standard Deviation: One of the aims of a load-balancing algorithm is to minimize the variations in workloads of all machines in a cluster. Regarding this, standard deviation in workload is often taken as the performance metric of a load-balancing algorithm. The smaller the standard deviation, the better the load-balancing scheme is. Normalized standard deviation is obtained from dividing the measured standard deviation by the average workload value of the same experiment. By looking at the changes in standard deviation of workload with respect to time, it is easier to visualize the effect of load-balancing upon the system.

Average Normalized Standard Deviation: As said above, the variations of normalized standard deviation over time are recorded for each experiment. Among this set of standard deviation, the mean can be calculated. After a number of the same experiments, a set of normalized standard deviations can be calculated. Finally, all such normalized data are averaged to give the final performance metric for a particular set of experimental parameters.

3.12 Performance Metric

In other research work on load balancing, the overall execution time of a set of jobs is often taken as the performance metric. This is possible only if all jobs have a finite execution time so that the whole set of jobs also have a finite execution time, and can be fed into different load-balancing schemes for comparison. For the case of multiagent systems, the above strategy cannot apply. The reason is that no overall execution time can be measured because of the immortality nature of agents.

To remedy the need for a performance metric, we can define another performance metric which does not depend on the finite execution time of jobs. One is the execution time of a query which consists of a set of agent operations and interaction. The other is the workload distribution situation across all machines in a cluster. For the former proposed performance metric, some representative query to a system is needed. That means, if we adopt such a performance metric, a representative query must be devised in order that this metric can be obtained generically in any multiagent systems. Such a query should specify what tasks agents will perform, how and with whom each of them will communicate, and when to do all these operations. The main difficulty of this performance metric is that there does not exist any well-accepted representative query. The situation is made worse when considering the lack of a well-accepted multiagent system model. The second proposed performance metric, the workload distribution situation across the machines in a cluster, is relatively more universal and implementable in different cases. This metric can be measured in any system with all kinds of load-balancing schemes. More specifically, the workload distribution is measured as the average normalized standard deviation. As a result, this metric is the major parameter considered in our experimental results analysis. Nevertheless, as detailed in Section 4.4, we also performed experiments with randomly generated queries to investigate the performance of the algorithms in terms of query response time.

To illustrate the performance of the Comet algorithm, we also implemented the symmetrically initiated distributed load-balancing algorithm [28], which is well-known to be efficient in handling the load balancing of tasks in a traditional distributed computing system. Because this algorithm is purely workload based, we call it workload based load balancing (WBLB) here to clearly reflect its difference in design philiosophy in contrast to that of the Comet algorithm. Due to space limitations, we do not repeat the description of the WBLB algorithm here and the details about its operations and features can be found from [28].

4 Results

This section describes experimental results obtained from feeding the system with various combinations of parameters. First, the effect of varying the number of agents and hosts is investigated. After this, we study the effect of varying the coefficients used in the calculation of the credit values. We will see that this effect is not very pronounced even when there is a large difference in the coefficients. At last, the algorithm is applied to a special mathematical data structure, a binary tree. We found that Comet performs similarly with both random communication structure and trees.

4.1 Varying Number of Agents and Hosts

The first set of results was obtained by varying the number of agents, n, and the number of hosts, p. We have tested the system with the following sets of values: $n = 48$, 64, 80, 120, and $p = 8$, 12, 16. However, not every case of different number of agents is applied on every case with different number of hosts. The case of 24 agents is only applied to the case of eight hosts since there are too few agents per hosts for the case of 12 hosts and 16 hosts. Likewise, the case of 120 agents is not applied to the case of eight hosts since too many agents in one single host will use up all the available physical memory very quickly, thus jeopardizing the reliability of the results. In summary, Table 4 below shows the performance metric results, the *average normalized standard deviation* (ANSD), of every test case, for WBLB and Comet. In the table, each ANSD value is obtained by averaging the ANSD results of 50 different runs. At first glance, the comet load-balancing algorithms produce a smaller ANSD in all cases, hence indicating an improved load-balancing effect. To give a better visualization of the comparison between WBLB and Comet, selected traces of ANSD are plotted against time, as illustrated in the graphs in Fig. 8, which only shows one of the workload traces in one selected experiment for a particular combination of n and p.

For the same number of agents, the ANSD for Comet is lower than WBLB. This means the workload across all hosts is less varied in the case for Comet, thus indicating an enhanced load-balancing performance. As the number of agents increases, ANSD decreases, which means the load-balancing effect is more pronounced. This is due to the fact that there are more options for both location and selection policies. There is a higher chance of selecting an agent to be migrated to another computing host such that the overall

TABLE 4
Summary of Experimental Results by Varying n and p

WBLB ANSD results

	n				
p	24	48	64	80	120
8	0.78	0.57	0.46	0.44	
12		0.57	0.50	0.45	0.40
16		0.78	0.65	0.58	0.51

Comet ANSD results

	n				
p	24	48	64	80	120
8	0.60	0.50	0.39	0.31	
12		0.47	0.41	0.36	0.28
16		0.58	0.56	0.41	0.33

(a) WBLB ANSD results. (b) Comet ANSD results.

workload is more balanced. The same situation occurs in both WBLB and Comet systems.

In Table 4, we can see that when the number of agents increases, the difference in performance of WBLB and Comet also increases. This can be explicated as follows: Recall that the agents are not uniform tasks. Indeed, each of the agents contributes different extents of load increases to its host at different time instants in its execution life time. Precisely because of such a dynamic behavior, when there are more agents in the system, the load-balancing problem becomes more difficult, as there are more possible variations in the system state. Thus, some load-balancing algorithms may perform better than others under a more difficult situation (in other words, some algorithms do not scale).

Furthermore, from Table 4, it can be observed that for the same n, when p increases, the load-balancing effect becomes worse, i.e., ANSD increases. This is because when p increases, less agents reside at each host and, therefore, the choice of an agent being migrated according to selection policy is more restricted. Moreover, when there are less agents in one host, the fluctuations of the workload of some agents become significant. This, in turn, affects the overall standard deviation of the whole host. It can be seen that, for a small number of agents, e.g., WBLB (with 48 agents) and Comet (with 48 agents), the data shows some increase when the number of hosts decreases. The reason is that as the number of hosts falls, the choice available for location policy, i.e., selection of a destination host to migrate an agent, will be smaller. In this connection, the load-balancing effect will be weaker. This weak effect is not prominent when the number of agents is high, as the high number of agents increases the choice in selection policy and, thus, can compensate the performance loss.

In summary, the Comet load-balancing algorithm performs better when both n and p are large. In that case, since there are many agents and hosts, it is more possible to select an agent and a host to make the overall loading of the system more balanced.

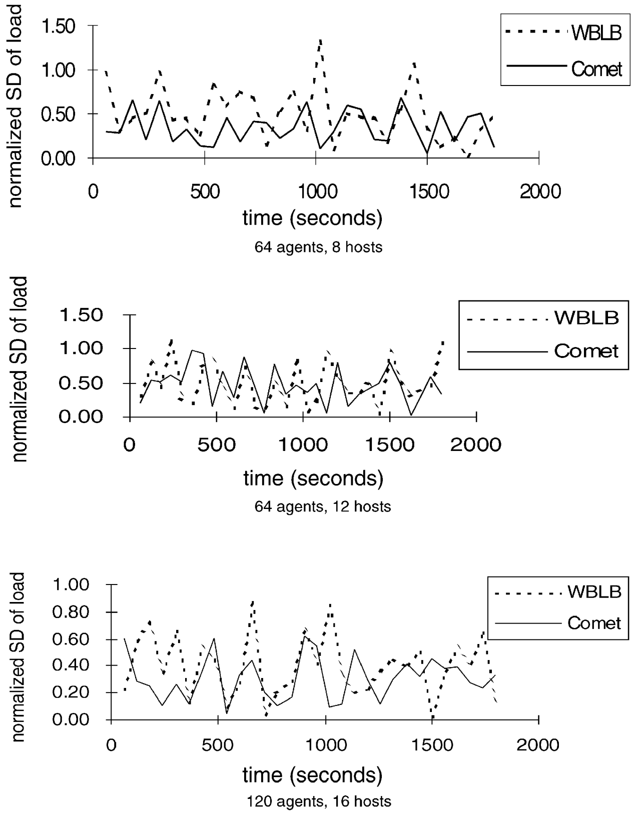


Fig. 8. Sample workload traces in three cases. (a) 64 agents and eight hosts. (b) 64 agents and 12 hosts. (c) 120 agents and 16 hosts.

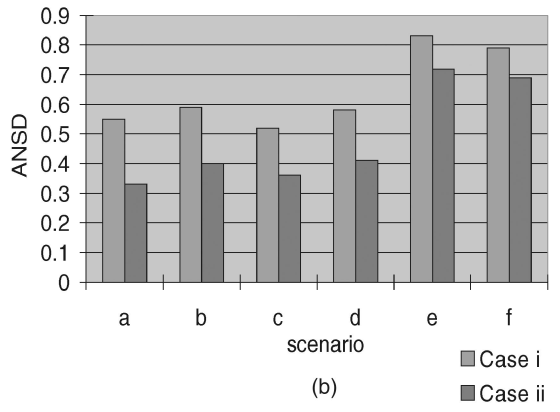
4.2 Varying Credit Coefficients

The calculation of the credit values involve the specification of three coefficients. Previous experimental results take all coefficients as unity. This section investigates the effect of varying those coefficients. Let us take a review of the credit formulation: $C_i = -x_1 w_i + x_2 h_i - x_3 g_i$. In this equation, all x_1, x_2, and x_3 are taken as unity for all previous experiments but we would also like to investigate the effects of these coefficients and, thus, we have done test scenarios with the values shown in Fig. 9a.

The first four sets of experiments aim at investigating the effects of x_3, which is responsible for symbolizing the effect due to workload. The larger is this coefficient, the greater tendency is given to a high-workload agent to migrate out of the current host. This effect is tested on the cases $(x_1, x_2) = (1, 1)$ and $(x_1, x_2) = (10, 10)$. The last two sets of experiments are designed to prove the effects of intrahost communication and interhost communication. The larger x_1 is, the more emphasis is placed on intrahost communication, and the

scenario	x1	x2	x3
a	1	1	1
b	1	1	10
c	10	10	1
d	10	10	10
e	1	10	1
f	10	1	1

(a)



(b)

Fig. 9. Scenarios and results of the different combinations of n and p with various values of the coefficients. (a) Test scenarios and (b) results of two cases.

greater affinity the agent has to stay at the same host. On the contrary, if x_2 is larger, the agent will have a greater tendency to migrate out because of interhost communication. Because of the limitations of resources, we have not implemented every experiment with different n and p. We only experimented two combinations of n and p for each set of coefficients, namely: Case 1) $n = 48$ and $p = 8$; and Case 2) $n = 120$ and $p = 16$. The results are shown in Fig. 9b.

For the first four cases, they do not show much significant difference. The reason is that the increased coefficient in x_3 induces the same effect to all agents alike. Consequently, the decision-making process during migration is not very different. Looking at Cases (e) and (f), we can see that the ANSD is particularly higher than other cases. Cases (e) and (f) are special in that the coefficients for intrahost and interhost communications are diverging greatly. This means that the weights of both kinds of communication are unbalanced. The relatively higher value of ANSD may be caused by some agents constantly being migrated from one host to another, giving rise to exceedingly varied workload. In summary, the coefficients x_1 and x_2 should be equal so that the weights of intrahost and interhost communications are balanced. Taking into unbiased account both types of communications, the performance in terms of ANSD will be better in this situation.

4.3 Communication Pattern

The previous communication pattern formed among the agents is randomly constructed. However, in many occasions of computation, a tree communication pattern is required. Examples are sorting, searching, and other kinds of divide-and-conquer algorithms. It is interesting as well as useful to see the effects of applying Comet to those algorithms. The only parameter in a tree communication pattern is the tree order. We have tested order 4 and 5, which translate into 31 and 63 agents. A set of agents with a tree communication pattern was fed into the system and again the values of ANSD were collected. The case of eight hosts is tested. Each test is run for 10 times 1,800 seconds each. Random communication patterns are also used for comparison. The results are listed in Table 5 below.

These results should be compared with Comet for the same number of agents in the event of randomly generated communication patterns. We did not have experimental results for 32 agents with random communication patterns; nevertheless, we can still compare the results with the linearly interpolated case of 24 and 48 agents. For the case of 63 agents, we can make a direct comparison with 64 agents, assuming no significant difference. Thus, we obtain the comparison graph as shown in Fig. 10.

We see that for the case of Comet, the tree communication pattern results in a better load-balancing performance than a randomly generated pattern. This can be explained by the fact that tree structures are inherently regularly formed, with certain degree of clusteredness. Hence, the agents can be arranged in such a way that interhost communications are reduced to a greater extent. On the contrary, in the case of a randomly generated pattern, each agent has the same probability to establish a communication link with another agent, thus making the overall pattern irregular. This also makes it impossible to achieve an arrangement of agents such that the interhost communication can be minimized as in tree pattern. For the case of WBLB with respect to tree pattern, the results do not indicate any obvious and consistent conclusion. The value

TABLE 5
Results of Comet with Agents in Tree Communication Pattern

n	Comet		WBLB	
	Tree	Random	Tree	Random
31	0.52	0.55	0.71	0.68
63	0.32	0.39	0.45	0.46

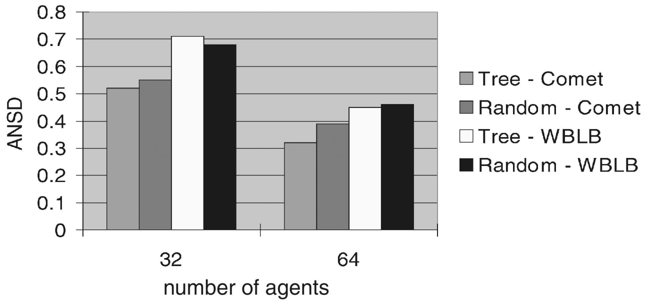


Fig. 10. Performance comparison of tree and random patterns.

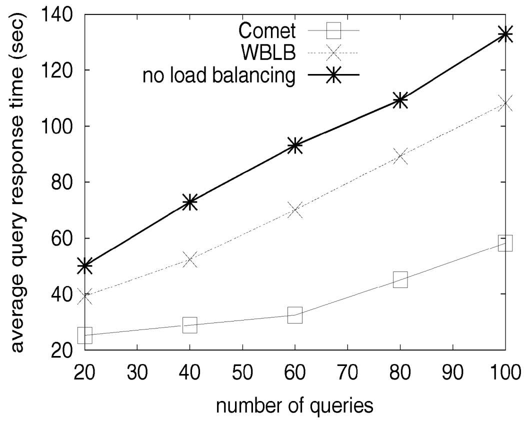


Fig. 11. Average query response times of the three approaches.

of ANSD is higher in one case, but lower in another. In fact, they are very close together. We can, therefore, assume that for WBLB, the communication pattern does not make any remarkable difference in ANSD.

4.4 Query Response Time

To quantify the performance of the Comet algorithm from an end-user's point of view, we also performed experiments with hypothetical queries, which are randomly generated by selecting a random subset of agents to perform a query with a random communication pattern. We performed experiments using our prototype system with 20, 40, 60, 80, and 100 queries and we measured the average query response times. We used 48 agents on eight machines. For each query set, we ran the experiment 10 times with different initial configurations of the agents. In these experiments, to illustrate the effect of load balancing, we also included a "no load balancing" method, which let the agents to fully autonomously migrate (specifically, each agent will flip a coin after finishing a query to decide whether it will migrate to a randomly selected machine). These results are shown in Fig. 11. As can be seen, the Comet approach outperformed the other two methods and the average query response times increase with the number of queries. We can see that the increase in the response times of the Comet algorithm is smaller while the other two approaches have a sharper increase.

4.5 Summary

As can be seen from the above results, for a different number of agents and hosts, Comet consistently demonstrates a stronger capability to reduce the workload variation across all the hosts in a cluster. Results show that for the same number of hosts, the performance of Comet will be better for a larger number of agents. Likewise, for the same number of agents, Comet will be better when the number of hosts increases. In general, both Comet and WBLB will be better for a larger number of agents and hosts, and Comet consistently performs better than WBLB. As regards the calculation of credit values, three coefficients are involved. The effect of varying the coefficients was experimented and studied. It is found that the coefficients for intermachine communication and intramachine communication should be equal in order to achieve the best performance. In doing so, the effects of both communication will be balanced. It is also found that the coefficient corresponding to workload does not affect the performance noticeably. The explanation proposed is that the same weighting effect of workload is applied to all agents alike. No significant difference is, thus, observed. At last, a ubiquitous data structure, a binary tree, is used to model the agent communication pattern, and the agents are fed to the system. It is shown that the Comet algorithm produces a better performance in the face of a binary tree communication pattern. It can be explained that Comet succeeds to capture the clusteredness of a tree pattern and, subsequently, translates the pattern into a mapping between the agents and the hosts such that the workload variation is minimized.

5 Conclusions and Future Work

This research collocates the concept of load balancing and agent theory into one perspective. The load-balancing strategies were discussed in the light of the characteristics of agents. The discussion was followed by an introduction of multiagent cluster system test platform built from scratch and the design of a novel load-balancing algorithm, Comet, which is communication-based, apart from taking workload into account. We implemented the algorithm on the test platform. Different system parameters were fed into the test platform and experimental results were collected and evaluated accordingly. Based on the temporal continuity of agents, one of the prominent agent characteristics, we have defined the performance of a load-balancing scheme as the average normalized standard deviation (ANSD) in a specified period of time. ANSD is taken as the performance metric in the test platform which minimizes the subtle system background workload differences. This makes the metric widely comparable for experiments with diverging system configuration parameters. We compared the results with the conventional workload-based load-balancing algorithm. The results show that Comet performs generally better than WBLB. The performance is better with a larger number of agents and hosts. This implies better scalability.

As for the internal coefficients of the credit value calculation in the Comet algorithm, some experiments were designed to testify their effect toward the overall performance. They demonstrated that the weighting of both intermachine and intramachine communication should be the same in order to achieve the lowest system workload variation. Results also suggest that the weighting of computational workload has no significant influence on the load-balancing performance. An interesting data structure, binary trees, is incorporated in the agent communication pattern in the last set of experiments. It is shown that Comet produces a better performance for a tree communication pattern than a randomly generated one. This demonstrates the ability of Comet to capture the clusteredness of the communication pattern and formulate an optimized agent-host mapping dynamically.

One limitation of our study in this paper is that we assume a static communication pattern. In reality, agents may change their communication patterns dynamically. Thus, a further research direction is to investigate the effectiveness of the Comet algorithm under such a more dynamic situation. Another avenue of further research is to incorporate language constructs in devising the load-balancing strategies and agent migration decisions. We are currently working on some further experiments with workload variation prediction based on the characteristics of different language constructs.

Acknowledgments

The authors would like to thank all the anonymous reviewers (in particular, reviewer E) for their insightful comments and careful reading of the paper. The authors would also like to thank Professor Kai Hwang and Dr. Hai Jin for their valuable suggestions in the initial stage of this research. This research was supported by a grant from the Hong Kong Research Grants Council under contract number HKU 7124/99E. A preliminary version of portions of this paper was presented at the ParCo99, Delft, The Netherlands, August 1999.

References

[1] I. Ahmad and A. Ghafoor, "Semi-Distributed Load Balancing for Massively Parallel Multicomputer Systems," *IEEE Trans. Software Eng.*, vol. 17, no. 10, pp. 987-1004, Oct. 1991.

[2] I. Ahmad, A. Ghafoor, and K. Mehrotra, "Performance Prediction and Distributed Load Balancing on Multicomputer Systems," *Proc. Supercomputing '91,* Nov. 1991.

[3] R.M. Bryant and R.A. Finkel, "A Stable Distributed Scheduling Algorithm," *Proc. Second Int'l Conf. Distributed Computing Systems,* Apr. 1981.

[4] T.L. Casavant and J.G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Trans. Software Eng.*, vol. 14, no. 2, pp. 141-154, Feb. 1988.

[5] S. Franklin and A. Graesser, "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents," *Proc. Third Int'l Workshop Agent Theories, Architecture, and Languages,* 1996.

[6] S. Fricke, K. Bsufka, J. Keiser, T. Schmidt, R. Sesseler, and S. Albayrak, "Agent-Based Telematic Services and Telecom Applications," *Comm. ACM,* vol. 44, no. 4, pp. 43-48, Apr. 2001.

[7] M.R. Gary and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[8] M.L. Griss and G. Pour, "Accelerating Development with Agent Components," *Computer,* vol. 34, no. 5, pp. 37-43, May 2001.

[9] T.E. Senator, H.G. Goldberg, J. Wooton, M.A. Cottini, A.F.U. Khan, C.D. Klinger, W.M. Llamas, M.P. Marrone, and R.W.H. Wong, "The FinCEN Artificial Intelligence System: Identifying Potential Money Laundering from Reports of Large Cash Transactions," *AI Magazine,* vol. 16, no. 4, pp. 21-39, 1995.

[10] M.N. Huhns and M.P. Singh, "Distributed Artificial Intelligence for Information Systems," CKBS-94 Tutorial, Univ. of Keele, UK, June 1994.

[11] K. Hwang and Z.W. Xu, *Scalable Parallel Computing: Technology, Architecture, Programming.* McGraw-Hill, 1998.

[12] N.R. Jennings, P. Faratin, M.J. Johnson, T.J. Norman, P. O' Brien, and M.E. Wiegand, "Agent-Based Business Process Management," *Int'l J. Cooperative Information Systems,* vol. 5, nos. 2/3, pp. 105-130, 1996.

[13] N.R. Jennings and M.J. Wooldridge, *Agent Technology: Foundations, Applications, and Markets,* Berlin, Germany: Springer-Verlag, 1998.

[14] JATLite Web Site, http://java.stanford.edu/, 2002.

[15] M. Kafeel and I. Ahmad, "Optimal Task Assignment in Heterogeneous Distributed Computing Systems," *IEEE Concurrency,* vol. 6, no. 3, pp. 42-51, July 1998.

[16] P. Krueger and M. Livny, "A Comparison of Preemptive and Non-Preemptive Load Distributing," *Proc. IEEE Int'l Conf. Distributed Computing Systems,* pp. 123-130, 1988.

[17] T. Kunz, "The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme," *IEEE Trans. Software Engineering,* vol. 17, no. 7, pp. 725-730, July 1991.

[18] Y.-K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys,* vol. 31, no. 4, pp. 406-471, Dec. 1999.

[19] M. Ljunberg and A. Lucas, "The OASIS Air Traffic Management System," *Proc. Second Pacific Rim Int'l Conf. AI (PRICAI-92),* 1992.

[20] S.A. Moore, "KQML and FLBC: Contrasting Agent Communication Languages," *Proc. 32nd Ann. Hawaii Int'l Conf. Systems Sciences,* p. 10, 1999.

[21] M.P. Nuttall, "Cluster Load Balancing using Process Migration," PhD thesis, Department of Computing, Univ. of London, June 1997.

[22] L.M. Ni, C. Xu, and T.B. Gendreau, "A Distributed Drafting Algorithm for Load Balancing," *IEEE Trans. Software Eng.,* vol. 11, no. 10, pp. 1153-1161, Oct. 1985.

[23] M.P. Papazoglou, "Agent-Oriented Technology in Support of E-Business," *Comm. ACM,* vol. 44, no. 4, pp. 71-78, Apr. 2001.

[24] G.F. Pfister, *In Search of Clusters,* second ed. New Jersey: Prentice Hall, 1998.

[25] The Pleiades Project, http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-5/www/pleiades.html, 2002.

[26] K. Ramamritham and J.A. Stonkovic, "Dynamic Task Scheduling in Hard Real-Time Distributed Systems," *IEEE Software,* vol. 1, no. 3, pp. 65-75, July 1984.

[27] B.A. Shirazi, A.R. Hurson, and K.M. Kavi, *Scheduling and Load Balancing in Parallel and Distributed Systems.* Los Alamitos, Cailf.: IEEE Computer Soc. Press, 1995.

[28] N.G. Shivaratri, P. Krueger, and M. Singhal, "Load Distributing for Locally Distributed Systems," *Computer,* vol. 25, no. 12, pp. 33-44, Dec. 1992.

[29] A. Schaerf, Y. Shoham, and M. Tennenholtz, "Adaptive Load Balancing: A Study in Multi-Agent Learning," *J. Artificial Intelligence Research,* 1995.

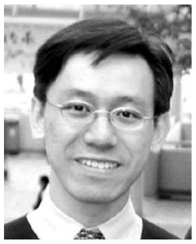
[30] J.A. Stankovic, "Simulations of Three Adaptive, Decentralized Controlled, Job Scheduling Algorithms," *Computer Networks,* vol. 8, pp. 199-217, 1984.

[31] L.G. Valiant, "A Bridging Model for Parallel Computation," *Comm. ACM,* vol. 33, no. 8, pp. 103-111, Aug. 1990.

[32] P. Williams, "Dynamic Load Sharing within Workstation Clusters," honors dissertation in Information Technology, Univ. of Western Australia, Oct. 1994.

[33] C. Xu and F.C.M. Lau, *Load Balancing in Parallel Computers: Theory and Practice.* Boston: Kluwer Academic, 1997.

[34] S. Zhou, "Performance Studies of Dynamic Load Balancing in Distributed Systems," Tech. Report UCB/CSD 87/376, EECS, Univ. California, Berkeley, Oct. 1987.



Ka-Po Chow received the BEng degree in computer engineering and the MPhil degree from the Department of Electrical and Electronic Engineering at the University of Hong Kong in 1997 and 2001, respectively. At present, being a cofounder, he works as a software engineer in a cluster computing company in Hong Kong. His technical interests lie mainly in scalable computer systems and software support for distributed computing.

Yu-Kwong Kwok received the BSc degree in computer engineering from the University of Hong Kong in 1991, and the MPhil and PhD degrees in computer science from Hong Kong University of Science and Technology in 1994 and 1997, respectively. He is an assistant professor in the Department of Electrical and Electronic Engineering at the University of Hong Kong. Before joining the University of Hong Kong, he was a visiting scholar in the Parallel Processing Laboratory at the School of Electrical and Computer Engineering at Purdue University, West Lafayette, Indiana. His research interests include software support for heterogeneous cluster computing, distributed multimedia systems, adaptive wireless communication protocols, and mobile computing based on short range wireless technologies. He is a member of the ACM, the IEEE, the IEEE Computer Society and the IEEE Communications Society.