No. 898

A Load Balancing Facility Using Aglets, a Java-based Mobile Agent System

by

Taijun SAWANO, Yongbing ZHANG, and Hideaki TAKAGI

February 2001

A Load Balancing Facility Using Aglets, a Java-based Mobile Agent System

Taijun SAWANO† Yongbing ZHANG‡
Hideaki TAKAGI‡

† Doctoral Program in Policy and Planning Sciences, University of Tsukuba, Tsukuba-shi, 305-8573 Japan
‡Institute of Policy and Planning Sciences, University of Tsukuba, Tsukuba-shi, 305-8573 Japan

Abstract

Load balancing in distributed systems allows users to access large amount of computing resources distributed around the system and may provide substantial performance improvements to applications. A distributed systems, however, may contain various hosts that are heterogeneous in hardware architecture or software platform. One has to sacrifice the transparency degree for the task remote execution or make modifications on the user interface or the core system in order to overcome restrictions on user tasks to support heterogeneity.

In this paper, we develop a load balancing facility using Aglets, a Java-based mobile agent system proposed by IBM over a local area network in order to improve the imbalance of workload among the nodes. When a user task arrives at a node the load balancing facility tries to assign it to a node that appears to result in the shortest response time. Primary advantages of our facility are that it is totally independent of the system architecture as far as the system provides a Java Virtual Machine (JVM) and that it does not let users to take care of load balancing when developing an application program. The experiments show that our facility improves the response time of tasks substantially when there are multiple computers available in the system and that it is very effective for CPU-bound tasks.

1 Introduction

The revolution of communication technologies has conducted the distributed processing possible across the network. The benefits of distributed processing can be achieved by balancing the system workload among the computers (hosts) so that the resources (CPU, memory, etc.) of the system are utilized effectively. The benefit function is usually stated in terms of response time reduction or throughput maximization [9, 11, 16]. A distributed system, however, may contain various hosts that are heterogeneous in hardware architecture or software platform. One has to sacrifice the transparency degree for the task remote execution or make modifications on the user interface or the core system in order to overcome restrictions on user tasks to support heterogeneity.

In programming paradigm for distributed processing, there has happened recently, another revolution. The programming paradigm for distributed applications has changed from the client-server paradigm [2] to code-on-demand paradigm [4] and mobile agent paradigm [6, 7, 10, 12, 13]. In the client-server paradigm [11, 14], a server provides a set of services accessible to some resources, e.g., databases and the code of a service is hosted locally by the server. The relation of the client and the server is permanently fixed in this paradigm. A server has all of the hardware and software resources, and therefore it may become overloaded if the requests arrive at it intensively. Transmission of data between the client and the server may also waste a large amount of network bandwidth, causing network congestion.

In code-on-demand paradigm, the service codes are located on the server and a client initially only knows where the service codes are but is unable to execute these codes. When a client requests a service, it sends a request to the server for receiving the required code. Once the code is received by the client, the computation is carried out at the client using the local resources to achieve the required service. This paradigm alleviates the load of the server but the relation between a client and a server is still fixed.

The mobile agent paradigm, on the other hand, is a new paradigm whereby any host on the network is allowed to have a high flexibility of where to execute the code and what to use the resources. That is, the execution of the code is not tied to a single host but rather is available throughout the network. Therefore, a client and its server can be merged and reside on the same host, and they can also be moved to any other hosts on the network. This paradigm provides many benefits because of its flexibility and can be utilized in electronic commerce, distributed information retrieval, parallel processing, etc. In this paper, we explore the last perspective of the mobile agent paradigm. That is, we make use of the replication function of the mobile agent paradigm to clone multiple agents and spread them throughout the network to achieve load balancing in distributed systems.

In this paper, we develop a load balancing facility over a local area network (LAN) using *Aglets*, a Java-based mobile agent system provided by IBM [8, 10]. In this facility, a user has no need to

take care of what architecture he/she is using as far as the system provides a Java Virtual Machine (JVM) and how load balancing is performed when developing a application as far as it is a Java application. A user also has no need to write an Aglets-dependent application either. Therefore, an efficient multi-task processing environment over a network can be realized.

The rest of the paper is organized as follows. In Section 2, we describe our load balancing facility. The implementation details of the system are given in Section 3. The evaluation of the load balancing facility is presented in Section 4. Our conclusion and future research are described in Section 5.

2 Load Balancing Facility Design

The load balancing facility proposed in this paper focuses on a local area network (LAN) environment as shown in Figure 1, where there exist various host computers with different processing capability or functions connected by a fast local area network. The communication delay can therefore be neglected and the security issue is not critical here. The load balancing facility is developed by using a Java-based mobile agent system, Aglets, and therefore the heterogeneity problem of the system in hardware and software is resolved and it can be easily extended to the case in wide area networks (WANs). The load balancing facility clones working agents if necessary and dispatches them on the network for achieving distributed processing. The agents can work and move autonomously over the network and therefore a more flexible system can be realized. There exist some load balancing facilities for distributed processing and, among them, the Load Sharing Facility (LSF) proposed by Zhou et. al may be the best [15, 18]. However, since LSF depends on the system architecture, although minor and limited, it needs to modify the user interface and some of the core local system to run tasks on a heterogeneous environment.

2.1 Basic Architecture

Our system consists of various host computers, called nodes, connected via a communications network, as shown in Figure 1. Each node may have various processing capability and different

function, but should have a Java virtual machine (JVM). Each task, i.e., Java application, can arrive at any node and be processed at any node in the network. It can be processed locally at where it initially arrived and can be transferred to another node for remote processing. In the latter case, the processing result is transferred back to the arrival node. A key characteristics of our system is that the processing location and the transferring process of a task is transparent to the user. Therefore, it appears to the user that the whole distributed system is a single parallel computer.

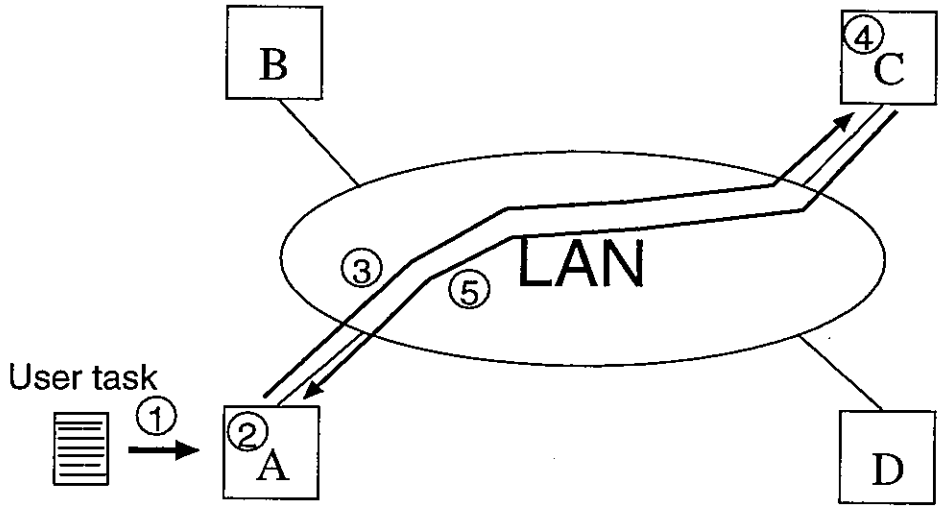

Figure 1: A load balancing system over a LAN.

The main components consisting of the system are shown in Figure 2.

- Information Collector

  Information collector is a component used for collecting the load information of the system for the master agent in order to determine the processing node for arrival tasks and for providing its own load information to other nodes.

- Master Agent

  A master agent is a key component resided on each node and provides the interface communicating with users. It is used for receiving requests from the users, for deciding the

processing nodes, for creating slave agents, for receiving the results from the slave agents, and for passing the results to the users. Before remote task initiation, a connection is established between the local master agent and the remote agent system so that remote system gets ready to accept foreign slave agents.

- Slave Agent

  A slave agent is a component created by a master agent when the master agent decides a user task should be processed remotely. The slave agent carries the user task to the remote node (server), processes the task at the server, and then carries the results back to its master again. The slave agent can stay at the same remote host until the task processing finishes, move to another host autonomously during the task execution, or move to a series of hosts according to a given travel itinerary.

- User Task

  A user task is a Java class file, i.e., a series of byte codes, that a user intends to execute. It is written as a common Java application and when developing it there is no need to take care of load balancing.

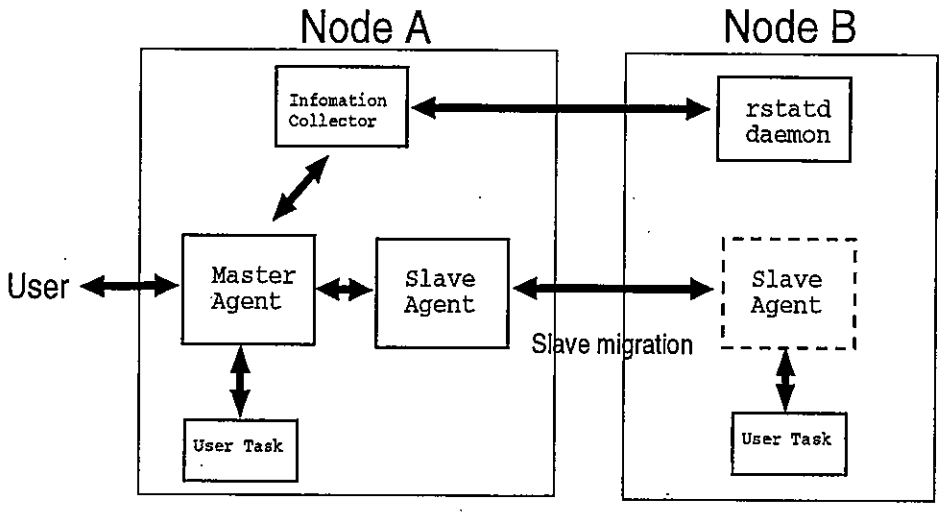

Figure 2: Organization of a load balancing system.

2.2 Policies for Load Balancing

In load balancing strategies [9, 11, 17], we have *static load balancing strategies* which balance the workload based on only the statistical information of the system. On the other hand, we also have *dynamic load balancing strategies* which balance the workload dynamically depending on the system state fluctuation. Furthermore, in dynamic load balancing, we have a *task assignment strategy* in which the processing node of an arriving task is determined before its execution and a task in execution cannot be transferred. On the other hand, we also have a *process migration strategy*, in which a task in execution can be transferred further according to the system state. The latter strategy is believed to be more effective than the former but should be more expensive and complicated in implementation. Eager et. al [3] showed that the process migration strategy has only limited improvement over the task assignment strategy. The load balancing facility proposed in this paper, addition to LSF, belongs to the category of task assignment strategies.

2.3 Load Indices

For each type of resource, a *load index* is defined to show the load condition. In this paper, we use the processing capability and the amount of memory available at a node as our load indices. The slowest CPU speed and the least amount of memory at a node are taken as the basis for the load index and the load of each node is denoted by a *load vector* of the normalized CPU and memory indices using the base load index. For example, a node has processing power of 2 and memory capacity of 3 means that it is two times faster than the slowest host in processing power and has three times more memory than the host of the least memory.

2.4 Determination of Task Transmission

Even though dynamic load balancing has the potential to improve the system performance further than static load balancing, the load balancing decision may not be correct and cause the system to become unstable because of the wrong information and the inherent dynamic nature of the system. In order to alleviate the load fluctuation and avoid the possible wrong decision of load balancing,

we introduce a threshold by which a node determines whether it can transfer an arriving task based on the load difference between the node and the server. In this paper, the value of the threshold is 1. That is, a node can transfer a task to a server only if the difference of the load vector is greater than 1. Since the communication delay is negligibly small in a LAN environment, it is not taken into account in the load balancing decision.

2.5 Determination of Eligible Tasks for Transfer

Although the load balancing facility has no restriction on remote processing to a task as far as it is a Java application program, the efficiency for execution should be taken into consideration. The computation time of a task has an important impact on load balancing strategies [1]. In this paper, we classify user tasks into two categories, *eligible for transfer* and *ineligible for transfer* and let each node to hold an eligible task list. An eligible task is usually the one with a long computation time, i.e., CPU-bound tasks. Tasks with short computation time or many I/O operations, i.e., I/O-bound tasks, on the other hand, are not eligible for transfer, because running I/O tasks results in limited benefits. In most cases, it is easy to distinguish a task from eligible to inegligible [5]. As shown in [5], there are not many tasks eligible for transfer and therefore a user can register these tasks to the eligible task list manually.

3 Implementation of the Load Balancing Facility

The load balancing facility consists of three main components as described in Section 2 and the master agent plays a key role in load balancing, receiving requests from users, dispatching tasks to servers, creating slave agents, and advising the results to the users. The communication mechanism between the master and the slave agents is implemented using a programming design pattern, *Master-Slave pattern*, provided by Aglets. For sake of simplicity, we only implemented the system on a Unix workstation cluster with a fast local area network. The system, however, can be easily extended to the MS-Window environment with minor modification in the information exchange mechanism.

3.1 Information Collector

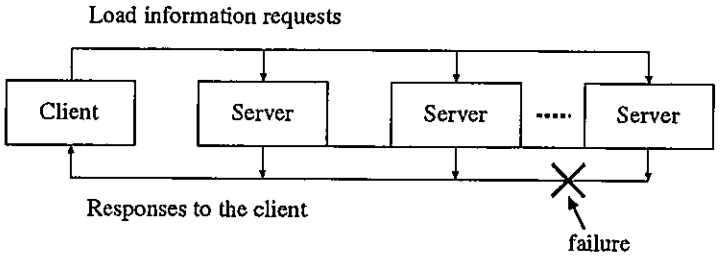

Figure 3: Collection of load information in the system.

Each node has a *node list* showing the identifiers of all the nodes that can be used for load balancing in the system and a *available node list* showing the nodes that can be used right away. When a node comes up, it broadcasts a request over the network to search for the available nodes on its node list. If a remote node responds to its request, it registers the remote host on its available node list. It also invokes the load statistical server, a Unix daemon *rstatd*, in order to respond to the requests from other nodes. When a new user task arrives at a node, the node (client) broadcasts a request to the nodes (servers) on its available node list to collect the load information using the Unix command, *rup*, as shown in Figure 3. In order to avoid network and server failures, a time-out limit (0.5 or 1s) in our facility is used for waiting the reply from a server. The information collector is implemented by using a multi-threaded scheme. For sake of simplicity, the total amount of memory other than the available memory at any given instant at a node is used as the memory element in the load vector. The CPU load is obtained by using the command, *rup*, and the *1-minute average load* is used in making load balancing decisions.

3.2 Master Agent

A master agent is responsible for interacting with users via MasterWindow, determining the node with the least load, creating slaves to carry out tasks, and receiving the results from the slaves. A master agent activates the information collector to collect the system information when it receives a task from its user via the MasterWindow as shown in Figure 4. It then determines the best

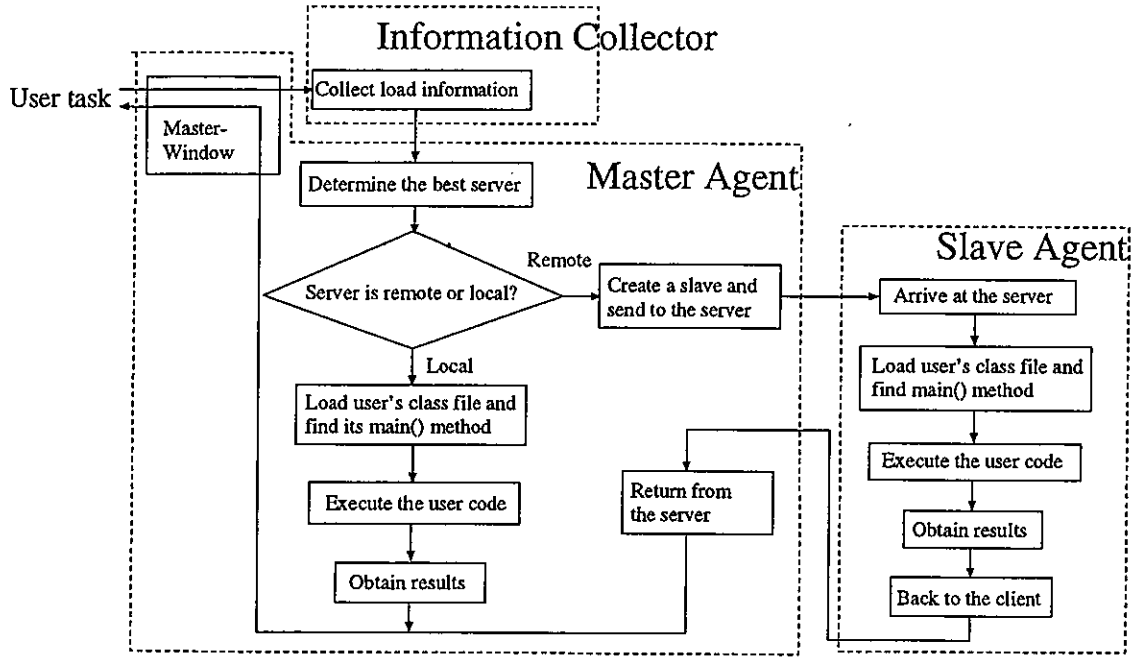

Figure 4: Processing a user's task.

processing node for the task based on the collected information. If the task should be processed locally, the master agent loads the byte code of the task and executes its main method right away. The execution of the task is carried out by using Java Reflection class. It then passes the results back to the user via MasterWindow again. If the processing node is a remote node, on the other hand, the master agent creates a slave agent and lets it carry the byte code of the task toward the remote node. When the slave comes back with the results, the master agent passes the results to the user through the MasterWindow.

3.3 Slave Agent

A slave agent is created by its master agent to carry a user task toward the server and execute the task at the server. It holds the byte code of the task and moves to the server, and then executes the task main method soon as it arrives at the server as shown in Figure 4. The execution of the task is implemented as in its master agent by using Java Reflection class. A slave agent can no longer move to any other nodes during the execution. When the execution finishes, the slave agent goes back to the client with the processing results.

3.4 User Interface

Figure 5 shows the user interface of the system, which contains four fields. A user input what he/she intends to execute and the appropriate options in Field (1). The execution of the task begins when pressing the button "GO". Field (2) shows the information collected by the information collector and Field (3) shows where the task is executed, what the processing result are , and how long the computation time is. Field (4) gives the logs for executing the user task.

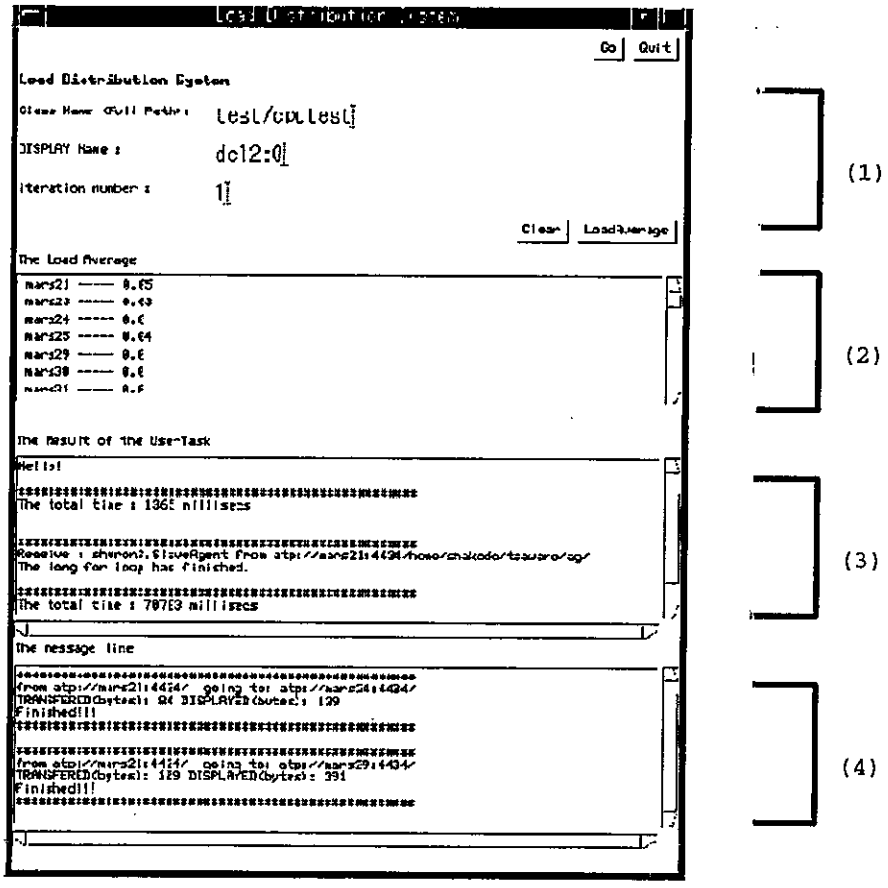

Figure 5: User interface of the load balancing facility.

4 Performance Evaluation

In this section, we evaluate the performance of the load balancing facility and examine when we can obtain benefits through load balancing. The task response time, i.e., the duration from the instant at which the button, "GO", is pressed to the instant at which the results are displayed in Field (3) in Figure 5, is used as the *performance index*. We firstly examine the overhead cost for running the load balancing facility. Secondly, we compare the task response times, using a CPU-bound task, for the case of running the task on the local JVM and the case of running the task remotely using our load balancing facility. We finally examine the effect of the number of nodes in the system on the task response time. The experiment has been carried out on a Unix workstation cluster with 40 hosts, which are connected with each other by a 100-Mbps local area network. The communication delay is not taken into account as described in Section 2.4.

4.1 Estimation of the Overhead Cost

Table 1: Overhead of the load balancing facility.

local execution	distributed execution
0.1s	1.5s

The overhead cost for running a task at a remote host contains the time of deciding the processing node, the time of creating a slave agent, and the time that the slave agent goes to and comes back from a server. In order to estimate the overhead cost for load balancing, we ran a very short Java program that is less than 500 bytes and outputs only a "Hello" message. Therefore, the communication delay for transferring the task and the time for packing and unpacking the packet for the task should be very short and can be neglected. Table 1 shows the average response time for 10 execution runs when there is no load balancing and when our load balancing facility is used, respectively. We can see that from Table 1 the overhead cost for using the load balancing facility is lower than 1.5s. When the size of the task increases the communication delay will also increase. In our experiments, however, transferring a program of a size around 100K bytes between two hosts

was always within a few seconds. This indicates that the overhead cost both for running the load balancing facility and for transferring a moderate size of tasks is very low.

4.2 Execution of a Long Computation Task

Table 2: Response time for a task with long execution time.

local execution	distributed execution
62.0s	64.4s

Since the overhead for load balancing is less than 1.5s as described in Section 4.1, running a task with a execution time longer than this overhead cost at a remote node leads benefits. It is preferable to run tasks remotely with long computation time, i.e., CPU-bound tasks. We examine here the response time of a small (less than 1K bytes) but long computation task. The task is a numerical calculation program, and has no input but a single line of output.

Table 2 shows the average response time of the task for 10 execution runs. From this table, we see that the overhead for running the task is around 2s, which is negligibly small compared with the task computation time of 62s. Therefore, the effect of the overhead for load balancing on the task response time can be neglected when running long computation time tasks.

4.3 Effect of the Number of Nodes

In this section, we examine the effect of the number of nodes on the task response time when running multiple tasks in parallel. Figure 6 shows the average response time for running the task shown in Table 2 in Section 4.2 when changing the number of nodes in the system.

We see from Figure 6 that the results agree with our intuition. That is, the higher the parallel degree the better the efficiency of the load balancing facility. It is also observed that the task response time remains constant when the number of tasks is less than or equal to the number of nodes. Furthermore, when the number of tasks exceeds the number of nodes, the task response time roughly equals to that with no load balancing divided by the number of nodes. Therefore, if there are multiple nodes available for load balancing in the system, a substantial performance

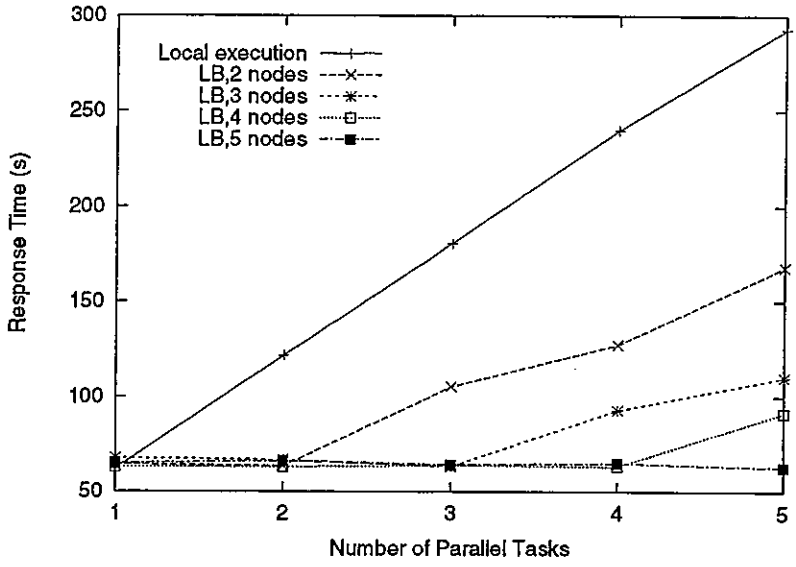

Figure 6: The effect of the number of nodes on the task response time.

improvement can be achieved.

5 Conclusion and Future Research

In this paper, we developed a network load balancing facility in a LAN environment using Aglets, a Java-based mobile agent system proposed by IBM. We showed that the load balancing facility improves the task response time dramatically if there are multiple nodes available in the system. It is observed that the overhead for running the load balancing facility is very low (lower than 2 seconds) and can be neglected for tasks with long computation time.

Since the load balancing facility proposed in this paper was implemented by using Java, a platform-independent language, it is easy to extend the facility to a large-scale, heterogeneous system in WAN. In this case, however, we need to take into account of the factors of security and system management. In a large-scale system across WANs, a scheme is needed that guarantees the system security and the limited usage of the local resources at a node. Furthermore, load balancing across WANs should be treated differently from that within a LAN in order to reduce the implementation overhead and ease the system maintenance. The following functions should

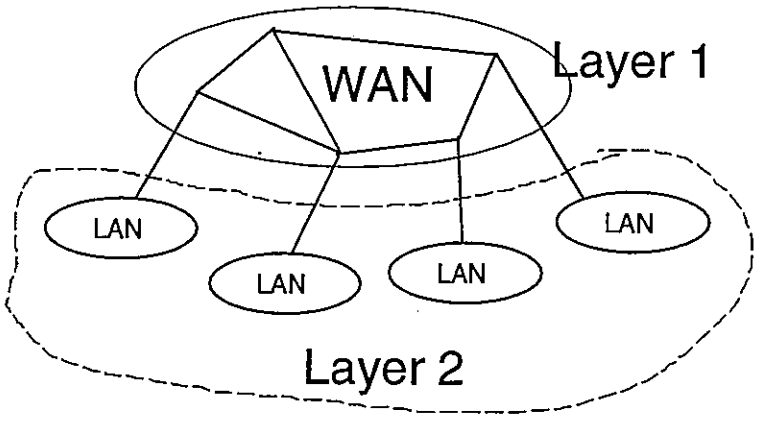

Figure 7: A load balancing system over a WAN.

be added to the facility in order to adapt to the WAN environment.

- Security. In order to avoid the excess or illegal usage of the local resources by users from other nodes across WANs, e.g., illegal access to local files or overuse of processing power, a mechanism is needed that can provide various levels of resource services and user authentication.

- Portability. In order to adapt to the heterogeneity of WANs, it needs to separate the system components that are dependent on the system architecture into the independent modules and keep the system easy to maintenance and update.

- Hierarchy. The efficiency problem for load balancing should also be addressed. The load balancing and the information collection across WANs are difficult to realize and their cost should also be expensive. A hierarchical architecture shown in Figure 7 demonstrates a solution where the system is divided into two layers. The sub-system at layer 2 behaves similarly to the current system for LANs but it needs a leader node at this level that is responsible for exchanging information and making load balancing decisions within a LAN. If a node cannot find an appropriate server in its local LAN, it asks its local leader node for possible remote processing across WANs. Each node knows only the information of all the nodes within the same LAN. The leader node in a LAN, on the other hand, knows the

locations of other leader nodes and their average load information. Each node, except the leader node, in a LAN is invisible to other networks so that the load balancing is achieved in a hierarchical way.

References

[1] L.F. Cabrera. The influence of workload on load balancing strategies. In *Proc. USENIX Summer Conf.*, pages 446–458, 1986.

[2] D.T. Dewire. *Client/Server Computing.* McGraw-Hill, Inc., 1993.

[3] D.L. Eager, E.D. Lazowska, and J. Zahorjan. The limited performance benefits of migrating active processes for load sharing. In *Proc. 12th ACM Symp. Oper. Syst. Principles*, pages 63–72, May 1988.

[4] B. Eckel. *Thinking in Java.* Prentice Hall PTR, New Jersey, 2000.

[5] D. Ferrari and S. Zhou. A load index for dynamic load balancing. In *Proc. 1986 Fall Joint Comput. Conf.*, pages 684–690, November 1986.

[6] Fujitsu, Kafka: A multi-agent library for Java (in Japanese), http://www.fujitsu.co.jp/hypertext/free/kafka/jp/.

[7] S. Honida, T. Iijima, and A. Ohtsuka. *Agent Technology (in Japanese).* Kyoritsu Publishing, Inc., 1999.

[8] IBM, Aglets, http://www.trl.ibm.co.jp/aglets/index.html/.

[9] H. Kameda, J. Li, C.G. Kim, and Y. Zhang. *Optimal Load Balancing in Distributed Computer Systems.* Springer-Verlag London Ltd., 1997.

[10] D.B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets.* Addison-Wesley Longman, Inc., 1999.

[11] D. Milojicic, F. Douglis, and R. Wheeler, editors. *Mobility Processes, Computers, and Agents.* Addison-Wesley Longman, Inc., 1999.

[12] Mitsubishi, Concordia, http://www.meitca.com/HSL/Projects/Concordia/.

[13] ObjectSpace Inc., Voyager, http://objectspace.com/products/Voyager/.

[14] R. Orfali and D. Harkey. *Client/Server Programming with Java and CORBA.* John Wiley & Sons, Inc., 1997.

[15] Platform Computing Inc., Load Sharing Facility (LSF), http://www.platform.com/.

[16] A.S. Tanenbaum and R. van Renesse. Distributed operating systems. *ACM Comput. Surveys*, 17(4):419–470, December 1985.

[17] S. Zhou and D. Ferrari. A measurement study of load balancing performance. In *Proc. 7th Int. Conf. Dist. Computing Syst.*, pages 490–497, Berlin, West Germany, September 1987.

[18] S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: A load sharing facility for large, heterogeneous distributed computer systems. *Software–Practice and Experience*, 23(12):1305–1336, December 1993.