A Mobile-Agent-Based PC Grid

Munehiro Fukuda [1] Yuichiro Tanaka [2] Naoya Suzuki [3] Lubomir F. Bic [4]
Shinya Kobayashi [5]

[1] Computing & Software Systems, University of Washington, Bothell <*mfukuda@u.washington.edu*>
[2] Fujitsu Corp., Japan <*tanaka@fujitsu.co.jp*>
[3] Information Sciences & Electronics University of Tsukuba, Japan <*nas@is.tsukuba.ac.jp*>
[4] Information & Computer Science, University of California, Irvine <*bic@ics.uci.edu*>
[5] Computer Science, Ehime University <*kob@cs.ehime-u.ac.jp*>

Abstract

This paper proposes a mobile-agent-based middleware that benefits remote computer users who wish to mutually offer their desktop computing resource to other Internet group members while their computers are not being used. Key to this resource exchange grid is the use of mobile agents. Each agent represents a client user, carries his/her job requests, searches for resources available for the request, executes the job at suitable computers, and migrates it to others when the current ones have become unavailable for use. All the features of job migration will be encapsulated in a user program wrapper that is implemented on a Java layer between a mobile agent and the corresponding user program. The wrapper maintains the complete execution state of the user program, is carried by the mobile agent upon a job migration, and restores its user program at its destination. For this purpose, a user program is preprocessed with JavaCC and ANTLR to include check-pointing functions before its execution. These functions periodically save the execution state of a user program into its corresponding program wrapper, which can thus be carried by an agent smoothly.

1 Introduction

In spite of its ambitious goals, grid computing still benefits only a handful of researchers who are free to forage among supercomputers and high-performance workstations for their computation-intensive projects. The majority of users, on the other hand, have few opportunities to access such computing facilities and are assumed to pay little attention to computational grids in favor of their own desktop computing environments. However, if they could authorize each other to mutually use their computers, a collection of such desktop machines would consistently provide them with an enormous computing resource, especially for their time-critical needs (e.g., running multi-user game software, backing up their files upon computer replacement, running scientific simulations as instructive laboratory assignments, and stock market analysis). As one example, *Bayanihan Computing.NET* [17] is an academic endeavor that has been launched to collect and to use desktop computers for grid computing.

As an attractive option for forming a computational grid of desktop computers, we focus on an arrangement analogous to moderated Internet discussion groups, where members can offer their resources to their group members when they leave their computers unattended. To join such a computational grid, a user would provide a group moderator with his/her identity, computing resource information, and lease conditions. Resource search could take the same form as signing into a discussion group, followed by searching for group members to discuss with. At a first glance, the only important problem seems to be finding a method for job deployment to remote machines. However, desktop computers belonging to an Internet discussion group bring up aspects that are quite different from those connected using a SAN or LAN: they are powered on and off without a prior notice, they are under the full control of their owners, they may be located behind a gateway and thus have a private IP address, and their resource information may not be properly registered with the Internet group. One solution is periodic resource mining or job status probing conducted by job coordinating brokers, such as a web server in *Bayanihan Computing.NET* or a collection of static agents in NetSolve [15].

Contrary to this solution, our approach is to have a mobile agent represent a client user and coordinate his/her job over a computational grid of desktop computers. Mobile agents are capable of autonomously navigating over networks and launching network tasks, with which a user job can mine available resources and migrate from one computer to another whenever the current resource becomes unavailable. A central technical challenge is user process migration, extending its target to multiple processes working together in parallel, thus communicating with each other, which requires not only capturing a process state but also forwarding messages to migrating processes. We plan to encapsulate these migration features in a Java user program wrapper between the agent and the corresponding user program. This wrapper will capture the state and all messages to the user program, and will be carried by the agent upon job migration. For this purpose, a user program is preprocessed using JavaCC [12] and ANTLR [1] to implicitly interact with the user program wrapper.

As preliminary work, we have developed the first prototype that deploys a mobile agent to a PC cluster best fitting the client's request, executes a multiprocessor job there, and has the agent bring back the results. In this paper, we discuss our motivation for using mobile agents and the overview of our proposed system in Section 2, explain the key design in Section 3, show our preliminary and on-going work in Section 4, differentiate our design from other related work in Section 5, and present our final comments, future work and potential applications in Section 6.

2 Motivation and System Overview

We focus on the following three middleware features necessary to form a computational grid of desktop computers distributed over the Internet: (1) searching for and allocating to a job a system resource that may change dynamically, (2) migrating a process to a newly available machine, in particular when the current machine is attended by its owner or even powered off, (3) supporting network-transparent communication among migrating processes, some of which may run at a machine located behind a gateway or a firewall.

Mobile agents, since their emergence in the mid 90's, have been considered as a mechanism to automate information retrieval, e-commerce, and network device management. Unfortunately, since almost all of those application domains are feasible using traditional client-server script programming and too broad to dispatch mobile agents, they have yet to find their "killer applications". However, we feel that mobile agents can reach their potential by implementing the above middleware features for the following reasons: (1) all desktop computers in a grid are preregistered to a trusted Internet group, which thus firmly defines the range where mobile agents may migrated as well as the interface through which they communicate; (2) their state-capturing and migration mechanisms help a user job to smoothly check-point and migrate to another site; and (3) with an HTTP-tunneling transfer technique, mobile agents may carry with them messages exchanged among computers that are engaged in the same job. Although the traditional scripts could still address some of the problems discussed above, mobile agents can demonstrate their superiority in dispatching, check-pointing, and migrating user jobs.

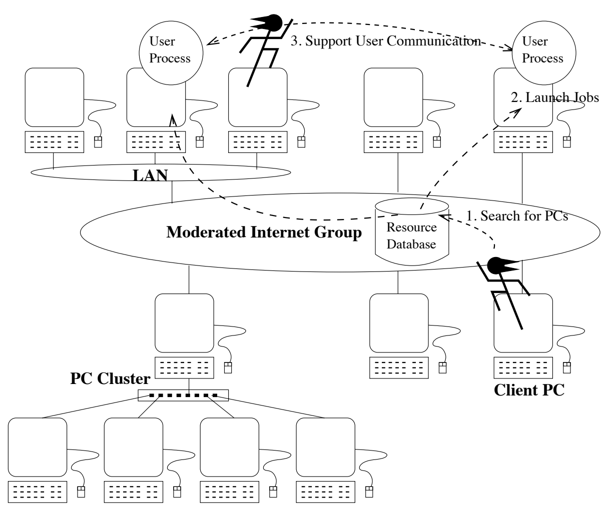


Figure 1. The proposed PC grid overview.

Figure 1 shows an overview of the middleware for a computational grid of desktop computers implemented with mobile agents, each carrying out a job submitted from its client user. A computation scenario in our proposed middleware is as follows:

To form a PC grid society, a moderator starts an Internet group that maintains each group member's information in its database. Needless to say, multiple Internet groups can be organized using the Globus Metacomputing Directory Service (MDS) [5]. Each user creates a guest account, runs a web server, registers his/her computer information to the Internet group, and receives a mobile-agent execution engine that can be triggered through a web access. After signing into a grid Internet group, a user invokes a mobile agent that opens a menu window to prompt for user instructions for resource search. A user job and its related file names will be passed to the mobile agent.

Upon job submission, a mobile agent accesses the group database to look for computers satisfying its client request. In cases where there are no such machines, the agent repeatedly visits other group databases by traversing MDS or every participating computer until it mines the machine best fitted to run a job. For migration, the agent uses an HTTP tunneling technique in which the agent is delivered as an HTTP message and thereafter is resumed by a servlet at each destination. All the information retrieved through resource search could be reflected to the group database.

Upon locating a computer to run a user job, the mobile agent uploads all files necessary for execution from its client to this target computer. If the client needs two or more machines, the agent spawns child agents and dispatches each child to a different computer where it in turn launches and monitors a user process. When a remote computer becomes unavailable during job execution, the monitoring mobile agent migrates the corresponding process to another available machine. Such process migration requires an execution snapshot. For this purpose, a language preprocessor is needed to automatically insert check-pointing functions into user source code at compilation time. A back-up snapshot is periodically stored in several machines in the grid, so that a mobile agent can retrieve a suspended process from the latest snapshot. Upon a termination of the client's job, a mobile agent carries back to the client all updated files including computation results.

3 Key Design

As shown in Figure 2, our proposed system consists of four layers: (1) a collection of client nodes and group servers, all connected to a network, (2) mobile agents representing a client user and taking care of its job execution, (3) Java wrappers, each wrapping a user program, and (4) user programs written in Java or C/C++.

There are five technical challenges to implement our mobile-agent-based middleware. The first is how to deploy mobile agents to remote computers under the familiar restriction where only a few common TCP/IP ports are available for use. The second is how to take process snapshots. Our strategy is to insert snapshot functions into a user program at compile time, and thus the third challenge is how to preprocess a user program. Since we assume each participating desktop computer may be powered off at any time, process snapshots must be maintained in a distributed fashion and retrieved by mobile agents at newly available machines. This is the fourth challenges. Finally, the fifth challenge is how to facilitate communication among processes that execute the same job but may be migrated. The following introduces our solutions to these technical issues.

3.1 Agent Transfer

The current security trend closes all but a few major TCP/IP ports, such as sftp, ssh, and http. It is therefore getting infeasible to assign a new port to agent transfer. In addition, a user may be reluctant to run an additional daemon process in charge of handling mobile agents. An HTTP tunneling technique addresses this problem by enclosing a mobile agent into an HTTP message and sending it to a destination HTTP server. For this purpose, an HTTP server at each desktop machine must prepare a specific web page where a servlet receives, extracts and interprets such

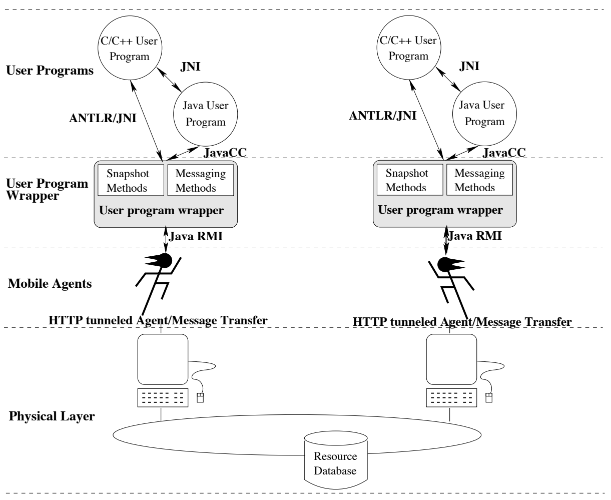


Figure 2. The four-layered PC grid environment.

HTTP-enclosed mobile agents. For security, we will actually use secured HTTP communications and the Apache-SSL server, both of which deal with client/server authentication and spontaneous message encryption.

3.2 Execution Check-Pointing

Our check-pointing scheme is to insert state-capturing functions into a user source code with a language preprocessor. Since a user program itself is not a mobile agent and thus does not include any migration-initiation statements, such as go( ) or hop( ), we must insert such statements between consecutive blocks of computation, such as for/while loops. For this purpose, we plan to use two compiler-compiler tools: ANTLR for C/C++ and JavaCC for Java source code. The reason we include Java is that we assume two levels of job execution. One is native execution where applications are coded in C/C++, possibly using PVM and MPI, compiled thoroughly, and executed directly. The other is Java-based execution where applications are outlined in Java or even in JPVM to invoke existing C/C++ library modules and executables.

Our blueprint for check-pointing is based on the four levels of implementation: mobile agents, user-program wrappers, Java user programs, and C/C++ user programs.

Mobile agent level: The mobile agent is mainly in charge of job coordination, i.e., controlling where and when to run a user job. The actual check-pointing work will be performed above the mobile agent level.

User-program wrapper level: This is a Java object, instantiated by a mobile agent at a remote computer. The user-program wrapper monitors the execution state of a user process and funnels all TCP/rsh-based messages from the user process to it. The wrapper thereafter transfers those messages to the mobile agent that can then deliver the messages to other computers participating in the same job execution. For this purpose, a wrapper provides state-capturing and message-forwarding methods that are to be called from the user program. Whenever a process needs to be migrated, the mobile agent serializes, carries with it, and de-serializes this wrapper object to a new destination, from which the process can be resumed.

Java user program level: A user program may be outlined in Java to execute precompiled native code, in which case a snapshot can be taken between any two native code invocations (but not during each native execution). This will be implemented by inserting the user-program wrapper's check-pointing methods into the Java source code. At a very early stage of our implementation, a user will be advised to manually add such methods in his/her Java program, however our ultimate implementation will use JavaCC for automatic method insertions.

Similar to most existing Java-based mobile agents [16, 14], the main restrictions are two-fold: one is that all child Java threads will be terminated upon a migration, and the other is that all I/O operations local to the previous machine will not be forwarded.

C/C++ user program level: A user program may be entirely coded in C/C++ and executed in native mode. Our implementation at this level takes the same strategy as the Java user program level, and uses ANTLR and JNI to automatically insert user program wrapper's methods into C/C++ source. This state-capturing however incurrs more restrictions in multithreading, I/O, memory, and IPC operations than the Java level.

We will address some of those restrictions, using our former research outcome. Multithreading at native level is made mobile with our M++ self-migrating threads [7]. Given a user program spawning pthreads, our language preprocessor will convert all those pthread functions to the corresponding M++ thread functions that internally capture each thread status with *setjmp* and *longjmp* functions. Dynamic memory can be also carried with a user program by facilitating the allocation and deallocation of a system-predefined data type whose instances are all serialized upon a migration [6].

As a result, the proposed system applies some restrictions to Java and native programs as summarized in Table 1.

Items	Java level	Native level
Threads	Terminated upon a process migration	Emulated by M++ threads but restricted to homogeneous architecture
Local I/O	Not forwarded	Not forwarded
Memory	Serializable objects only	System-predefined objects only
IPC	TCP messages only	TCP messages only Unix pipe/msq excluded

Table 1. Programming restrictions.

3.3 User Program Preprocessing

Java and C/C++ applications must be preprocessed with our JavaCC/ANTLR-based preprocessor to include check-pointing methods, with which the corresponding user-program wrapper takes an execution snapshot. The implementation of check-pointing methods is a similar issue in strong migration of mobile agents. There are three possible implementations:

1. *Inserting setjmp and longjump:* These functions are effective for saving and resuming an execution environment, including a program counter [7]. They are however available only for native programs based on a homogeneous architecture.
2. *Inserting if-else clauses:* Upon its initial execution, a user program executes the *if* clause that takes a snapshot of variables. Upon migration, the *else* clause is chosen to restore the variables [11]. This scheme is useful for Java, which does not allow manipulation of a program counter.
3. *Dividing user code:* This is our original algorithm implemented in the UCI-MESSENGERS mobile agent system [2, 22]. A user program is divided into two different functions before and after a state-capturing statement. A process initiates its migration upon termination of the first function execution, and it calls the second function after it has migrated.

We will take the options 1 and 3 for capturing execution state at native and Java level respectively. Option 1 allows C/C++ user programs to migrate within only homogeneous desktop machines, which is not a severe restriction, if we take into a consideration the following two factors: a plenty of homogeneous desktop computers are to join a grid, and we can remove overhead, such as program recompilation and data conversion, incurred by process migration to heterogeneous computers.

3.4 Snapshot and Consistent Cut Maintenance

All snapshots taken by each user-program wrapper must be maintained in non-volatile storage for anticipated job recovery. To avoid disk overflow or any failure stops, such snapshot maintenance should involve more computers than a single client or group server.

The quorum-based protocol is a well-known replication algorithm that can retrieve the latest snapshot from a group of back-up storage devices. One easy implementation is saving/retrieving a snapshot copy into/from more than half of all participating processes. However, as more computers are involved in the same job, the required communications also become more frequent. To address this problem, we need to restrict the number of snapshot-maintaining computers ultimately down to those communicating with each other within a certain time quantum. This may, on the other hand, retrieve an older snapshot (although this occurs only in pathological cases). We have started to evaluate such miss-retrieval possibilities under various conditions. In our proposed middleware, a mobile agent receives user input that accepts or rejects future possibility of an older snapshot retrieval and repetition of identical computation. In the latter case, the agent uses the quorum-based protocol for snapshot maintenance.

To safely garbage-collect unnecessary old snapshot from back-up storage, we plan to use the global consistent cut algorithm we have proposed in [13]. This algorithm logically forms a hierarchical ring connection of computers, each reporting its latest snapshot to and receiving a new consistent cut from the upper layer of the ring. The top ring includes multiple machines, so that no one centralizes all snapshots. Furthermore, each ring is re-configurable when detecting faulty computers. While we have shown (by simulation) its performance to be competitive with other major algorithms, this mobile-agent-based middleware will be its first practical implementation and empirical evaluation.

3.5 Inter-Process Communication

Since a process may migrate to another site or may run at a machine located behind a gateway, thus with a private IP address, it can be quite difficult to locate a user process with a physical IP address. Instead, we propose that each process involved in the same job be addressed with a sequential number only valid inside this job, (termed a middleware-unique ID). Such ID must be translated into an appropriate IP address every time a job is migrated. If a mobile agent has launched a user job at a computer residing behind a gateway, namely one that is IP-masquerading, it first sends an empty message to the group server or the client machine, so that the masquerading machine is given a temporary IP address and is thereafter identified from outside of the gateway.

The user-program wrapper is in charge of supporting inter-process communication based on middleware-unique IDs. A user process calls the wrapper's message-forwarding method, passing the middleware-unique ID of its destination process. Each message is first passed to the corresponding mobile agent that tracks all machines participating in the same job execution. This means that the agent maintains a table that translates a middleware-unique ID to the corresponding IP address. The agent then delivers the message over HTTP to the actual destination.

Previously, the Mach operating system has supported such location-independent IPC mechanism in its port migration mechanism [10] that broadcasts the latest port position to every node in the system with a periodic token message. This may however not be extensible to a grid middleware. Some scalable algorithms have been introduced in [18] for message delivery: (1) tracing the footprints of a migrating process en route to the latest destination, (2) having a central server forward a message to the latest destination, and (3) inquiring of the latest destination from where the process was originated only when a message has been returned. Because of the distributed and independent nature, we will extend algorithm 3. Our extension is that, whenever a mobile agent migrates its user process to another machine, it reports the new address to only those that have communicated with this process within a certain time quantum. This does not require that any machine become a single point of data collection. If a job runs as multiple processes, some of them must communicate with each another at least once, which guarantees that each process destination is tracked by mobile agents managing those processes. One drawback to this is that, when a process initiates its first communication with another process that might have been migrated somewhere, it must query the correct location from all processes.

4 Project Status

4.1 Preliminary Experiment

We have developed a CoordAgent prototype [8], which deploys a mobile agent to a PC cluster best fitting its client's request, executes a multiprocessor job there, and has the agent bring back the results. The CoordAgent system consists of five main modules: (1) XML as a resource description language, (2) servlets as user and system administrator interface, (3) per-cluster agents as resource managers, (4) mobile agents as user representatives, and (5) FastCGI processes as inter-cluster communication proxies. Broadly speaking, each user orders a job request through a servlet that converts it into XML and deploys a mobile agent with this XML description. The agent negotiates with a per-cluster agent every time it visits a new cluster, launches a user job, and reports results to the client.

The CoordAgent system is based on a cluster configuration that may consist of multiple computing nodes; all these nodes have the identical CPU architecture; they are under the control of NFS; and at least one of them has a public IP address, and thus is accessible from the external network (termed a cluster gateway). Figure 3 illustrates the environment presented in the gateway of each cluster. (Non-shaded

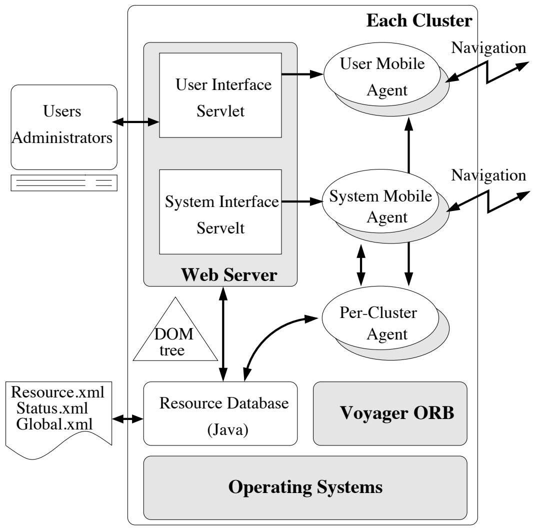


Figure 3. Prototype configuration.

Resource types	Cluster 1	Cluster 2	Cluster 3
CPU architecture	386	Sparc	386
#Nodes	8	8	12
Memory	128	64	256
Network card	Myrinet	Ethernet	Myrinet
Bandwidth (Mbps)	1000	100	1000
OS type	Solaris	Solaris	Linux
OS version	2.7	2.7	2.47
Application	MPI	MPI	OpenMP
	M++	OpenMP	

Table 2. Experimental machine specification.

components have been designed by us, while shaded ones are freeware.) The Voyager ORB [16] was used as the underlying mobile agent workbench. The main reason behind this choice was its dynamic aggregation feature permitting mobile agents to dynamically load new class objects into themselves and thus used to evolve agent behavior in response to runtime system reconfiguration. On top of the Voyager ORB is a per-cluster agent that schedules a user job launched from his/her mobile agent. In addition, each cluster gateway is supposed to run an HTTP server where two servlets are registered to provide interface to users and administrators respectively. The former submits a user mobile agent upon receiving a job request, while the latter may deploy a system mobile agent in case the remote clusters need to update their configuration. Those two types of mobile agents navigate over a predefined region of clusters to complete their missions.

We have evaluated the preliminary system performance using one front-end desktop computer and three clusters, each specified in Table 2. The experiment was to submit a parallel program that randomly exchanges MPI messages among processors. As shown in Figure 4, such job submission has requested four nodes of a cluster that is based on the Intel-386 architecture, controlled by Solaris, and equipped with 128M-byte memory per node. The user interface servlet converts this request into a job-descriptive DOM tree and passes it to a user mobile agent. Through network navigation, a user mobile agent finds that Cluster 1 satisfies this job request, executes an MPI program there, and displays a result onto a monitor. It took 100.53 seconds for this MPI program to complete its execution at Cluster 1. The turn-around time from the agent deployment to the outcome presentation was 109.29 seconds. Therefore, the CoordAgent system itself incurred 8.76 seconds as its overhead, which occupied only 8% of the total cost. It is true that such overhead depends on application size, server performance, and network latency. More overhead is anticipated as we plan to have an agent migrate its corresponding user process to a more suitable machine at run time. Therefore, in order to complete a full implementation of the proposed middleware, we need additional performance improvement and software development as discussed in the following subsection.

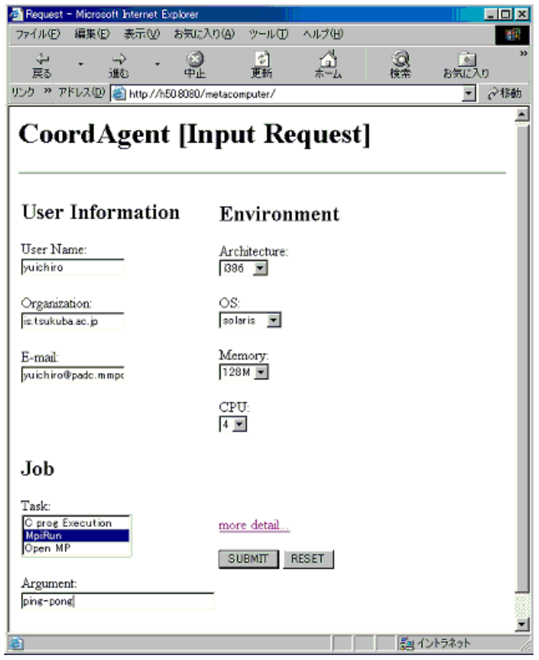
4.2 Current Status

Based on the preliminary experiment, we have started our implementation of the proposed middleware with the following work:

1. *Resource database reconstruction:*

   In the CoordAgent prototype, a user mobile agent had to visit each cluster until finding the one best suited to execute its user job. This resource search is obviously not scalable for a grid environment involving a large number of desktop computers. Therefore, as in the specification of the proposed middleware, a grid resource database is necessary to manage all the resource information on a group of grid-participating machines, which will allow a mobile agent to reach suitable computers in most cases immediately after visiting the database. However, we still have each machine maintain its local resource database that is useful for a user mobile agent to monitor the local resource.

   Another problem is XML-based resource descriptions. XML files may be still feasible for describing a client's job request but not for all computing resource of the participating computers. Because of this reason, such resource information should be maintained using a


Request - Microsoft Internet Explorer
ファイル(F) 編集(E) 表示(V) お気に入り(A) ツール(T) ヘルプ(H)
戻る 進む 中止 更新 ホーム 検索 お気に入り
リンク アドレス(D)
移動
イントラネット

http://h50:8080/metacomputer/

CoordAgent [Input Request]

User Information

User Name: yuichiro

Organization: is.tsukuba.ac.jp

E-mail: yuichiro@padc.mmpc

Environment

Architecture: i386

OS: solaris

Memory: 128M

CPU: 4

Job

Task: C prog Execution / [✓ MpiRun] / Open MP

Argument: ping-pong

more detail...

SUBMIT RESET

Figure 4. A job submission.

more performance-oriented database with JBDC interface.

2. *Mobile-agent workbench re-development:*

   The CoordAgent prototype used the Voyager ORB as our underlying mobile-agent workbench in order to take advantage of its dynamic aggregation feature. However, to complete the proposed middleware, we must encapsulate a mobile agent in an HTTP message and keep track of the latest position of all mobile agents involved in the same user job. It is not realistic to modify Voyager to meet our specification, especially in the current situation where it is now provided only on a commercial basis. Therefore, we decided to modify UCI-MESSENGERS, our original C-based mobile agent system, into a Java-based system and to enhance its features.

Although our implementation is still in a very preliminary stage, the current status includes: MESSENGERS' migration features have been re-implemented on a Java layer; grid and local resource databases are under construction using MySQL; and a user program wrapper is in a design phase.

5 Related Work

This section differentiates our system from other middleware systems based on process migration or agents.

Process Migration: The Condor system [4] is one of the most popular job schedulers/dispatchers used in collections of distributed workstations and dedicated clusters. It is based on process migration. Running on each computing node, the Condor daemon locates and dispatches a submitted job to the node best suited to performing the job. If a participating node has been overloaded or can no longer keep providing its requested resource, the Condor daemon takes a snapshot of the job and resumes it on another node. Job submission takes a description including not only the requested computing resource, (e.g., OS type, number of processors, etc.) but also job dependencies and suspension/resumption control.

Due to its snapshot algorithm using a process core dump, Condor is suitable for scheduling a collection of independent processes in a batched manner. However, some applications may need to run interactive, communicating, and/or multithreading processes. Our proposed middleware will address such execution by having a mobile agent maintain a communication channel with its client user and spawns child agents, each managing one user process and supporting inter-process communication.

Agent-Based Resource Allocators: NetSolve is a well-known system that uses an agent-based approach for resource allocation [15]. The system provides user programs with an RPC environment. A NetSolve agent accepts client requests and dispatches each request to the most suitable server. Working at a particular site, it maintains a directory of available remote resources and monitors their status. Each agent dynamically locates the servers for, flexibly marshals arguments of, and schedules the sequential/parallel invocation of desired RPCs.

The main difference with our proposed middleware is that a NetSolve agent is local to one site and intended to orchestrate each application's RPCs over the network, while we use mobile agents to dispatch an entire job.

Mobile-Agent-Based Software Installation: MASA is a proposed system that allows those involved in IT management to dispatch their mobile agents to a target machine where the agents refer to a site-specific access list, authorize each others, and complete a software installation cooperatively [21]. Notable is their emphasis on multiple authorization that is situation-sensitive but not task-oriented. Such a software installation will be performed successfully only when all agents representing an IT manager, an installer, and users have agreed to the installation in a predefined sequence.

In MASA, mobile agents representing different users perform a task at a specific site cooperatively. On the other hand, each mobile agent in our middleware searches available sites and launches a parallel job there by spawning child agents.

Mobile-Agent-Based Process Management: Catalina: a smart application control and management [9], an active grid-monitoring system using MAP agents [20], and a grid-computing model with J-SEAL2 [3] are mobile-agent-based systems proposed for computational grids.

In Catalina, an application-delegated manager (ADM) assigned to each application launches a mobile agent termed a *task agent* to monitor and manage each task execution. Their agents are characterized as sentinels reporting their job status to and ordered a next behavior by ADM, the commander which centralizes the entire system's status. Each agent of Catalina is capable of monitoring and check-pointing its job, so that a job can be resumed at another available computing node for the purpose of fault tolerance and load balancing. Since ADM works as a message center and a snapshot database, it is in charge of not only relaying all messages delivered to each job but also collecting all job execution history.

In MAP, each agent has its different mission — such as network monitoring or event handling. Cooperative work among such independent agents maintains quality of service in executing an entire user application. The system focuses on improving network performance rather than taking care of each user's job execution, and therefore its main task is system monitoring and message rerouting. MAP uses a central directory for resource search and a static resource allocation algorithm.

In J-SEAL2, a client user contacts an *operator*, a job-coordinating server for dispatching his/her job to a group of available computing nodes termed *donators*. The *operator* deploys a J-SEAL2 agent, termed a *deployment mobile agent* to each *donator*. A *deployment agent* passes its user job to the *donator* where a *mediator* process controls job launching, monitoring, suspension, and resumption. The *deployment mobile agent* reports back job execution status and results to the *operator* that makes further decision on job management. This computation model assumes that each mobile agent acts as a relay between the *operator* and the *donators*.

These systems' design principle overlaps with our proposed middleware in terms of using mobile agents, however the originality of our approach lies in having each mobile agent independently and entirely take care of a different client job from resource search to job migration. We are more concerned with job migration in order to cope with computing nodes that are powered off frequently.

6 Conclusions

This paper has proposed a mobile-agent-based middleware that mainly benefits remote desktop computer users as a mutual computing-resource exchange. The project combines the enhancement of a mobile agent platform and the behavioral design of mobile agents to keep allocating computing resources available to a user job.

Our target is distributed computing applications that tend to execute a long-running process or to repeat the same computation with different parameters over a collection of computing nodes. These include distributed scientific data management, parallel and distributed simulation, and instructive laboratory work in distance education. As a particular application, the proposed middleware will be used to build UW Bothell's LOGOS system [19] that automates a series of scientific data management processes such as query interpretation, knowledge (in the form of rules), data, and algorithm retrieval, plan generation, and data analysis, in each phase of which mobile agents will allocate computing nodes best fitting LOGOS' variety of requirements.

Based on our preliminary experiments, we have started updating CoordAgent toward the proposed middleware. The entire system including a user program wrapper, a language preprocessor, and check-pointing/snapshot maintenance features will be implemented in collaboration among UW Bothell, UC Irvine, and Ehime University.

References

[1] ANTLR Website. http://www.antlr.org/.

[2] L. F. Bic, M. Fukuda, and M. B. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, Vol.29(No.8):55–61, August 1996.

[3] W. Binder, G. D. M. Scrugendo, and J. Hulaas. Towards a secure and efficient model for grid computing using mobile code. In *Proc. 8th ECOOP Workshop on Mobile Object Systems: Agent Application and New Frontiers*, Malaga, Spain, June 2002.

[4] Condor Project. http://www.cs.wisc.edu/condor/.

[5] I. Foster and C. Kesselman. The Globus Project: A Status Report. In *Proc. IPPS/SPDP'98 Heterogeneous Computing Workshop*, pages 4–18, 1998.

[6] M. Fukuda, N. Suzuki, and L. F. Bic. Introducing dynamic data structure into mobile agents. In *Proc. of the 1999 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications - PDPTA'99*, pages 1854–1860, Las Vegas, NV, June 1999.

[7] M. Fukuda, N. Suzuki, L. M. Campos, and S. Kobayashi. Programmability and performance of m++ self-migrating threads. In *Proc. the IEEE Int'l Conference on Cluster Computing - Cluster2001*, pages 331–340, Newport Beach, CA, October 2001. IEEE-CS.

[8] M. Fukuda, Y. Tanaka, L. M. Campos, and S. Kobayashi. Inter-cluster job coordination using mobile agents. In *Proc. 3rd Int'l Workshop on Active Middleware Services*, San Francisco, CA, August 2001. IEEE CS.

[9] S. Hariri, M. Djunaedi, Y. Kim, R. P. Nellipudi, A. K. Rajagopalan, P. vdlamani, and Y. Zhang. CATALINA: A smart application control and management environment. In *2nd Int'l Workshop on Active Middleware Services*, 2000.

[10] J. B. III. A fast mach network ipc implementation. In *Proc. 2nd USENIX Mach Symposium*, pages 1–10, Monterey, CA, November 1991. USENIX Association.

[11] T. Illmann, F. Kargl, M. Weber, and T. Kruger. Migration of mobile agents in java: Problems, classification and solutions. In *Proc. of the Int'l ICSC Symposium on Multi-Agents and Mobile Agents in Virtual Organizationa and E-Commerce (MAMA'00)*, Wollongong, Australlia, http://cia.informatik.uni-ulm.de/papers/mama00.pdf, December 2000. Springer.

[12] JavaCC. http://www.webgain.com/products/java_cc/.

[13] S. Kobayashi and M. Fukuda. Hierachical multi rings gvt algorithm for time warp mechanism. *IPSJ Transactions*, Vol.41(No.11):3161–3172, November 2000.

[14] D. B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley Professional, 1998.

[15] NetSolve Web Site. http://icl.cs.utk.edu/netsolve/.

[16] Objectspace. Voyager orb 3.3. Developer guide, Dallas, TX, 1999.

[17] Project Bayanihan: Web-Based Volunteer Computing Using Java. http://www.cag.lcs.mit.edu/bayanihan/.

[18] P. K. Shinha. *Distributed Operating Systems: Concepts and Design*, chapter 8.2.2, Process Migration Mechanisms, pages 384–393. IEEE CS Press, New York, 1997.

[19] M. Stiber, G. J. D. Swanberg, J. Miller, M. Ellisman, S. Young, and R. Jain. LOGOS: A system for neurobiological structure/function analysis. In *Proc. 9th Int'l Conf. Scientific & Statistical Database Management*, pages 212–222, Olympia, WA, July 1997.

[20] O. Tomarchio, L. Vita, and A. Puliafito. Active monitoring in grid environments using mobile agent technology. In *2nd Int'l Workshop on Active Middleware Services*, 2000.

[21] G. Vogt. Delegation of tasks and rights. In *Proc. the 12th Annual IFIP/IEEE Int'l Workshop on Distributed Systems: Operations & Management DSOM 2001*, pages 327–337, Nance, France, October 2001. INRIA.

[22] C. Wicke, L. F. Bic, M. B. Dillencourt, and M. Fukuda. Automatic state capture of self-migrating computations in MESSENGERS. In *Proc. 2nd International Workshop, MA'98*, pages 68–79, Shuttgart, Germany, September 1998. Springer.