Engineering

Electrical Engineering fields

Okayama University *Year* 2005

A Framework for Mobile Agent Systems with the Capability of Preceding and Following Users

Tokumi Yokohira
Okayama University

Kiyohiko Okayama
Okayama University

Takashi Murakami
Okayama University

Kayo Takarako
Okayama University

This paper is posted at eScholarship@OUDIR : Okayama University Digital Information Repository.

http://escholarship.lib.okayama-u.ac.jp/electrical_engineering/56

A Framework for Mobile Agent Systems with the Capability of Preceding and Following Users

Tokumi Yokohira[†], Kiyohiko Okayama[††], Takashi Murakami[†] and Kayo Takarako[†]
[†] The Graduate School of Natural Science and Technology,
Okayama University, 3-1-1, Tsushimanaka, Okayama, 700-8530, Japan
[††] The Information Technology Center, Okayama University, Okayama, Japan
E-mail : yokohira@cne.okayama-u.ac.jp, okayama@cc.okayama-u.ac.jp,
takashi-m@net.cne.okayama-u.ac.jp, takarako@net.cne.okayama-u.ac.jp

Abstract

As one of mobile agent applications, many systems which provide continuous service for users moving on a network have been proposed. In these systems, because a movement of mobile agents is performed after a user movement, users must wait for arrival of mobile agents. To reduce users' waiting time, we propose a fundamental framework for mobile agent systems where an agent can move precedently before a user movement. In our framework, it is assumed that computers are connected on a network and users with rewritable devices move on the network. The framework supports precedent movement of mobile agents based on prediction using movement history of users. Because the prediction may be wrong, the framework also provides the following movement of mobile agents. Moreover, the framework provides a recovery method of mobile agents in service in case that mobile agents disappear due to problems such as their bugs. Because we provide some APIs, via which various functions of our framework are accessed, developers of mobile agent systems can easily use our framework using the APIs. We implemented an experimental agent system using the APIs and confirmed that the framework performed correctly using the experimental system.

1 Introduction

A mobile agent is a software program which can autonomously move among computers on a network. When a mobile agent moves, it holds not only its program text but also its state of execution. Mobile agents are therefore expected to be used for many kinds of applications such as information gathering by moving among computers, distributed computing, and so on. As typical applications using mobile agents, some systems providing continuous service for users who move among computers on the system are considered. In the systems, continuous service can be performed by moving mobile agents automatically as users move, and user movement can be also automatically detected using sensing technologies such as an RFID tag technologies and image processing technologies. Thus, users can receive continuous service transparently to physical configuration of computers and/or networks.

As conventional systems providing continuous service, the system by Bates et al.[1], NetChaser[2], LocALE[3], f-Desktop[4] and FollowingSpace[5] have been proposed. In these systems, after a user receiving service by a mobile agent at a computer (say A) departs from the computer A and arrives at another computer (say B), the mobile agent starts to move from the computer A to the computer B. In other words, agent movement follows user movement in the conventional systems. Thus, such following mechanism forces users to wait for arrival of mobile agents, and consequently users suffer some waiting time until the service by mobile agents starts. Though the allowable value of the waiting time depends on the contents of the services provided by agents, the waiting time seems to become considerably large in such services that agents need to involve large amount of data like image data and agents need to perform large amount of computation before starting to provide services at the arrived computer, and consequently the waiting time of such service seems to exceed their allowable values frequently. Thus, it is important to consider a strategy to shorten the waiting time.

In this paper, we propose a fundamental framework with which agent application developers can easily implement mobile agent systems where mobile agents can arrive at a computer precedently before user's arrival at the computer. The framework automatically detects the user's departure from a computer, and then it immediately moves the copies of the corresponding mobile agent to computers (target computers), one of which the user probably arrives at, and consequently the waiting time drastically decreases. Because we use such the strategy that copies agents in order to provide the preceding mechanism, when such copy is repeatedly done, a situation that many unnecessary copies exist at the same time may occur. In order to avoid the situation, we introduce a generation number for each agent. Each agent initially has the generation number of zero when it is generated and each copy of an agent with the generation number n has the generation number $n+1$. An agent with the largest (newest) generation number serves their corresponding user, and the other agents are automatically removed. The framework also provides the following mechanism,

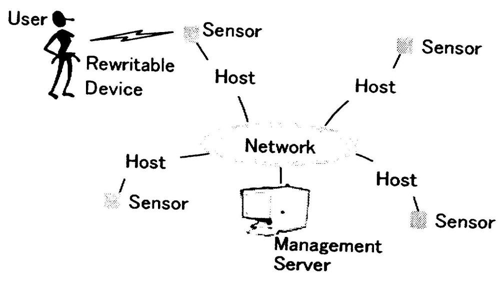


Figure 1: Underlying Environment

because target computers are predicted using movement history of the agent and the prediction may be wrong. Moreover, the framework provides agent backup and restore mechanisms against disappearance of agents due to problems such as their bugs. We have designed and implemented the preceding, following, backup and restore mechanisms and we provide some APIs, using which developers of agent systems can easily use the mechanisms. Moreover, we have implemented an experimental agent system using the APIs and we have verified the framework using the experimental system.

The rest of the paper is organized as follows. In Section 2, we describe the design of the mechanisms. In Section 3, we describe the implementation and APIs. In Section 4, the experimental system is described and the verification is performed.

2 Design of the Framework

2.1 Underlying Environment

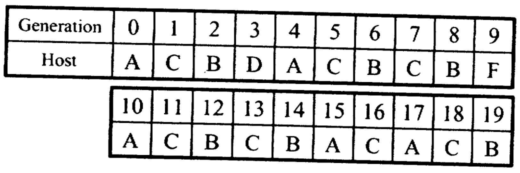
Figure 1 shows underlying environment of our framework. An agent platform such as AgentSpace[6] is implemented in each host and agents can move in a network using a moving function of the platform. A sensor attached to each host can communicate with a rewritable device attached to each user. Using the sensors and the devices, the framework can detect arrival and departure of users and move agents which provide services for users. The management server is used for management of the framework.

2.2 Required Mechanism

In the subsection, we describe the preceding, following, backup and restore mechanisms, and the generation management mechanism which guarantees that an agent with the newest generation number serves the corresponding user.

2.2.1 Preceding Mechanism

As described earlier, we predict hosts (target hosts), one of which a user probably arrives at. If the prediction is wrong with a high probability, then the preceding mechanism obviously does not perform well. Thus, it is very important to consider a method which can predict target hosts with a high probability. A trivial method which can predict target hosts with the probability of one is the method that all the hosts in the network are target hosts. However, the method can not be adopted from the view point of the scalability. In this paper, we adopt the following method based on an idea that users move in the same way as their past movement patterns with a high probability.



Generation	0	1	2	3	4	5	6	7	8	9
Host	A	C	B	D	A	C	B	C	B	F

10	11	12	13	14	15	16	17	18	19
A	C	B	C	B	A	C	A	C	B

Figure 2: An Example of Movement History

In order to know past movement pattern of each user, we record movement history of each agent. Movement history is a list of pairs of the generations of the corresponding agent and the hosts where the agent provides service for the user. Figure 2 shows an example of movement history. The example means that, the agent was originally generated at the host A and then moved to the host C, increasing the generation number by one, and so on, and currently the agent with the generation number 19 stays at the host B. We describe how to create movement history in the subsection 2.2.4.

Before an agent system using the framework runs, it tell the framework about the values of the two numbers x and y. The number x is the maximum number of target hosts, and y is the maximum number of hosts included in the past movement list which is defined as follows. When the value of a parameter y' is specified, the past movement list PML(y') is a list of y' hosts at which an agent arrived in the last y' generations. For example, PML(5) is (A, C, A, C, B) in Fig. 2. When the framework detects the departure of a user from the current host, it determines target hosts using the following procedure.

(0) Initialize the temporary variables $x' = x$ and $y' = y$.

(1) Find PML(y') from the corresponding movement history. For example, when x'= 3, PML(3) = (A, C, B) in Fig. 2.

(2) Excepted for PML(y') itself, find all sublists of the movement history which are identical to PML(y'), and find the hosts which are located at the next positions of the sublists. In Fig. 2, when y' = 3, we find the sublist (A, C, B) in the generations 0-2, (A, C, B) in the generations 4-6 and (A, C, B) in the generations 10-12 and find the host D in the 3rd generation, C in the 7th generation and C in the 13th generation, respectively.

(3) If the number of the hosts found in the step (2) is less than or equal to x', select all the hosts as target hosts and go to the step (4). Otherwise select x' hosts in the decreasing order of the occurrence number in the step (2) and select such hosts as target hosts. If the occurrence numbers of hosts are equal to each other, the host whose corresponding generation number is larger takes precedence over the other host. For example, we select the all hosts C and D when x' = 3, and we select the host C when x' = 1.

(4) If the number c of the current target hosts is equal to x or y' = 1, then terminate the procedure. Otherwise,

set $x' = x - c$ and $y' = y' - 1$, and go back to the step (1), where we ignore hosts which have been already selected as target hosts in the step (2). For example, when $x = 3$ and $y = 3$, the steps (1)~(3) first find the current targets hosts C and D ($c = 2$), and $x' = 3 - 2$, and $y' = 2$. And then the step (2) finds the host A and F. The step (2) selects the host A because the corresponding generation number of A is greater than that of F.

After the target hosts are selected, each copy (say a child agent) of the agent (say the parent agent) precedently moves toward each of the target host. If the generation number of the parent agent is n, that of each child agent becomes $n+1$. The parent agent remains at the current host during a certain period of time and then disappears. Because the parent agent remains, the design of the following mechanism become easy as described in the next subsection.

When a user arrives at a host, the generation number of the agent which the user requires has been already recorded in the rewritable device of the user as described in 2.2.4. If there exists an agent with the same generation number as the recorded number at the host, a pair of the generation number of the agent and the host's name is appended into the movement history, and the agent starts to serve the user. That is, it means that the precedent mechanism successfully works. Otherwise, that is, if there does not exist such an agent, it means the precedent mechanism fails, and the following mechanism takes over.

2.2.2 Following Mechanism

When the preceding mechanism fails, a host (say A) at which a user arrives first searches the host (say B) where the corresponding agent stays and bring the agent from the host B as follows. As described earlier, a list of the hosts where the agent served is recorded as the movement history. Thus, we can know the last host (the host B) where the user stayed from the movement history. And also as described earlier, the parent agent of the corresponding agent remains at the host B. Based on the facts, the host A first knows the host B and then requests the host B to move a copy (a child agent) of the parent agent to the host A. Receiving the request, the host B makes a child agent (whose generation number is greater than that of the parent agent by one) and send it to the host A. When the child agent arrives at the host A, the pair of the generation number of the child agent and the host's name is appended into the movement history, and the child agent starts to serve the user.

2.2.3 Backup and Restore Mechanism

As described in the subsection 2.2.1, the preceding mechanism makes copies of an agent. The backup mechanism transfers one of the copies to the management server as a backup of the agent, where the generation number of the transferred copy remains unchanged unlike the preceding mechanism. Some additional information, that is, *the user identifier*, *the agent name*, *the backup time* and *the removing condition* are also transferred to the server. The user identifier is the tag ID of the rewritable device of the user, and the agent name is the name of the executable program corresponding to the agent in the operating system on the host. The removing condition is one for removing the backup. We specify the maximum number of backups, the maximum disk capacity and backup duration as the conditions, and if one of the conditions is violated, some of the backup are removed.

In addition to such automatic backup operation, the framework also provides manual backup operation which performs a similar transfer to the automatic operation when it is explicitly called by an agent system on the framework.

The automatic restore operation is triggered when the framework detects the disappearance of an agent in service. The disappearance is detected as follows. Each agent in service regularly informs the framework of its existence using such a message "I-am-alive". If such information does not arrive at the framework, it determines that an agent disappears. When the framework detects the disappearance, it automatically transfers the most recent backup from the server to the host where the disappearance is detected.

We additionally provide manual restore operation. In the manual operation, an agent system on the framework can specify a backup on the server. For example, the agent system can specify the backup with a specified generation number. When the manual operation is explicitly called by an agent system, it transfers the specified backup from the server.

2.2.4 Generation Management Mechanism

In this paper, because only an agent with the same generation number as that of the rewritable device of a user can start to serve the user, the timing of changing the numbers of each agent and each rewritable device is important. As described in the previous subsection, the generation number of each agent is increased by one when the agent is duplicated from its parent agent. On the other hand, the generation number of each rewritable device is changed as soon as its owner starts to be served by the corresponding agent. That is, the generation number of each rewritable device is greater than that of the corresponding agent by one during the service. By doing that, even if the user departs from the host at an arbitrary time, we can keep the generation number of each rewritable device the newest and consequently an agent with the newest generation number can start to serve its user.

Disappearance of unnecessary agents is performed using the generation numbers of each agent and each rewritable device as follows.

Figure 3 shows an example of a family tree of agents. The agent A generates the child agents B, C and D and similarly only one child C generates its child agents E, F and G by the preceding mechanism. Note that only one agent in each generation generates child agents. Agents which dose not generate a child agents, that is, the agents B, E, D and G in Fig. 3, are unnecessary. Such agents automatically disappear as follows. We set a timer with a timeout value for every agent. Though the timer stops if the corresponding agent becomes a parent agent, that is, it

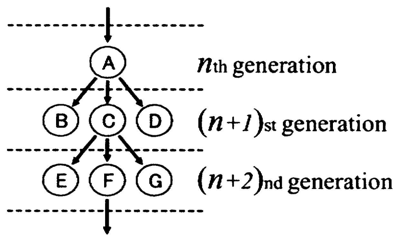


Figure 3: An Example of A Family tree

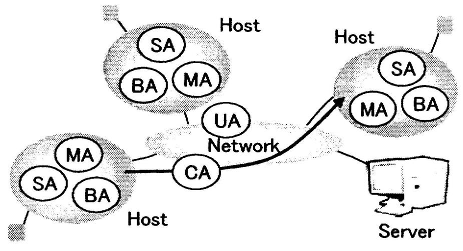


Figure 4: System Configuration

generates a child agent, otherwise the timer fires after the timeout duration and the agent automatically disappears with the firing as a trigger.

Disappearance of each agent which becomes a parent agent is performed as follows. As soon as a child agent starts to serve the corresponding user, the starting is notified to its parent agent. When the parent agent receives the notification, it automatically disappears.

Using the two disappearing mechanisms described above, every agent can automatically disappear when it becomes unnecessary. In this paper, in addition to the mechanisms, we provide two disappearing mechanism in order to remove unnecessary agents as soon as possible. One is the mechanism that an agent (say X) automatically disappears if there exists an agent (say Y) with larger generation number at a host and the agent X is not the parent of the agent Y, because the agent X is unnecessary. For example, if the agents B and F exist in a host at the same time, the former disappears. And if the agents A and E simultaneously exist in a host, the former also disappears. However, even if the agent C and E simultaneously exist in a host, the former does not disappear because C is the parent of E. The other additional disappearing mechanism is the mechanism that when a user with the rewritable devices with the generation number n+2 arrives at a host, if there exist an agent (say X) with smaller generation number and it is not the parent agent of an agent with the generation number n+2, then the agent X disappears, because the agent X is unnecessary. For example, in Fig. 3, when a user with rewritable device with the generation number n+2 arrives at the host where the agent B exists, B disappears. However, when the user arrives at the host where the agent C exists, C does not disappear because C is the parent agent of agents with the generation number n+2.

3 Implementation of the Framework

3.1 System Configuration

As described in the section 2, our framework provides the preceding, following, backup, restore and generation management mechanisms. The mechanisms are implemented using the following system side agents as shown in Fig. 4.

[Manager Agent (MA)] Each MA is a static agent which is stationed in each host and performs movement management of user agents (simply UAs hereafter) and the generation management mechanism.

Movement management of UAs is performed based on movement of users. In the preceding mechanism, the MA at a host where a parent UA exists receives a list of target hosts and hands the list to the parent UA which generates child UAs and each child UA moves to each target host. In the following mechanism, first the MA (say X) in a host (say A) at which a user with the rewritable device with the generation number n arrives receives the name of a host (say B) where the parent UA of child UAs with the generation number n exists from the server, and using the name of the host B, MA X requests the MA (say Y) in the host B to send a child UA of the parent UA to the host A. MA Y informs the parent UA of the name of the host A. The parent UA generates a child UA and the child UA moves to the host A. Using the movement management described above, the UA which a user wants to be served can wait for the user in the preceding mechanism or can arrive at the host at which the user arrives in the following mechanism. The UA starts to serve for the user as soon as the MA on the host orders the UA to serve. The backup and restore mechanisms are also performed using MAs as described later. As the generation management mechanism, MAs remove unnecessary UAs and maintain the generation numbers of writable devices and UAs.

[Sensor Agent (SA)] Each SA is a static agent which is stationed in each host. Using the sensor, after the SA detects an event of user arrival or departure, it informs the corresponding MA of the event. As soon as a UA starts to serve a user, the SA increases the generation number of the rewritable device via the sensor. Introducing SAs enables the framework to use various kinds of sensors by changing SAs only.

[Communication Agent (CA)] Unlike MAs and SAs, CAs are ephemeral mobile agents. MAs use CAs to send various information to other MAs. When an MA (say X) has information which should be sent to another MA (say Y), MA X generates a CA and hands the information to the CA. The CA moves to the host where MA Y exists and hands the information to MA Y, and then the CA disappears at the host.

[Backup and Restore Agent (BA)] BAs are static agents and perform the backup and restore mechanisms. As soon as a UA starts to serve in a host, the MA in the host generates the corresponding BA and informs the BA of control information such as the UA name, user identifier, the generation number. In the backup operation, the BA receives the executable image of the UA from the UA and creates a backup by merging the control information and the executable image together and then transfers the backup to the management server.

In the restore operation, the BA receives a backup from the server and extracts the executable image and then in-

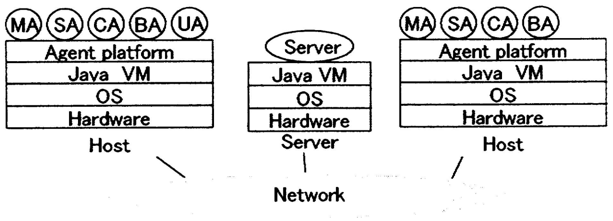


Figure 5: Implementation Environment

```
import java.awt.*;
import java.io.*;
public class Editor extends UABASE{
   private TextArea EditArea;
   public void startService(){
        EditArea.setEditable(true);
        EditArea.setBackground(Color.white);
        EditArea.setForeground(Color.black);
        show();
   }
   public void stopService(){
        dispose();
   }
   public void destroy(){
        dispose();
   }
   public void preService(){
        EditArea.setEditable(false);
        EditArea.setBackground(Color.black);
        EditArea.setForeground(Color.white);
        show();
   }
   public void create(){
        setTitle("Editor");
        EditArea = new TextArea(20,40);
        add("Center",EditArea);
        pack();
        EditArea.setText("Hello.");
   }
}
```

Figure 6: A Program using the Framework

forms the MA of the completion of the restore operation. Then the MA runs the executable image as a UA. BAs also detect disappearance of UAs using a kind of alive message from the corresponding UA and a timeout mechanism.

Figure 5 shows environment of the implementation of the framework. In this paper, as described earlier, we adopt AgentSpace as an agent platform. Because our framework and AgentSpace are both written in the Java language, they can be used on many kinds of operating systems.

3.2 APIs for Agent System Developers

We provide some APIs for agent system developers. They can easily implement agent systems using the APIs. The APIs are installed into our framework as a Java class library named UABASE. Thus, the developers can use the mechanisms of our framework by just inheriting the UABASE class as shown in Fig. 6. There are two most important class methods (functions) which must be written by the developers, *startService* and *stopService* methods. MAs use the former method to request UAs to start services and the developers write the content of the services in the method. On the other hand, MAs use the latter method to request UAs to stop services and the developers write the finishing operation of the services in the method.

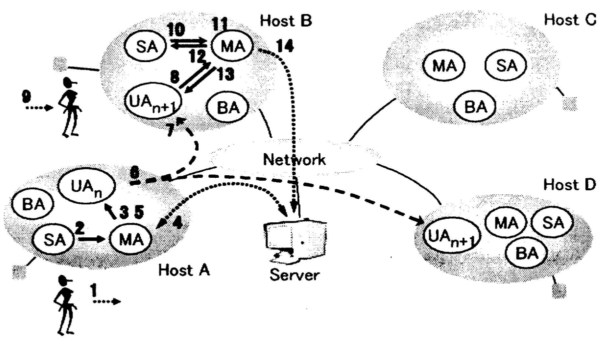


Figure 7: An Example of Agent Behavior

3.3 Example of Agent Behavior

Suppose that, as shown in Fig. 7, a UA with the generation number n (say UA$_n$) currently serves a user at the host A with the rewritable devices with the generation number n+1, and after the user departs from the host A, the user arrives at the host B. Focusing on agent behavior in the preceding mechanism, such situation can be described in detail as follows.

(1) The user departs from the host A.
(2) The SA detects the departure and informs the MA in the host of the departure.
(3) The MA informs UA$_n$ of the finish of the service.
(4) In order to perform the preceding mechanism, the MA obtains a list of target hosts from the management server. We assume that the MA obtains the hosts B and D as the target hosts.
(5) The MA informs UA$_n$ that the target hosts are B and D.
(6) UA$_n$ generates two child agents UA$_{n+1}$s, and they are transferred to the hosts B and D by the preceding mechanism.
(7) A UA$_{n+1}$ arrives at the host B.
(8) The UA$_{n+1}$ informs the MA in the host B of its arrival.
(9) The user arrives at the host B.
(10) The SA in the host B detects the user arrival and obtains the generation number n+1 which is recorded in the rewritable device.
(11) The MA confirms the existence of the UA$_{n+1}$.
(12) The MA requests the SA to increase the generation number of the rewritable device.
(13) The MA requests the UA$_{n+1}$ to start to serve the user.
(14) The MA requests the server to update the movement history.

4 Experiment for Verification of the Framework

Figure 8 shows experiment environment, where the OS on the host A is FreeBSD 4.9R, the OSs of the host B and C are Windows XP and the OS on the host D is Windows 2000. AgentSpace[6] is installed on the hosts A~C and the host D is used as the management server. RFID tags

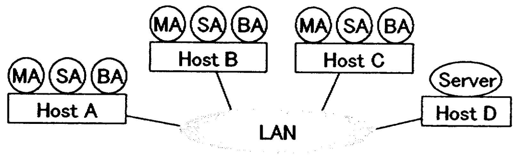


Figure 8: Experiment Environment

were used as rewritable devices and RFID reader/writer was used as the sensor of every host. We implemented a simple agent system called Editor Agent whose source code has already shown in Fig. 6. Each UA of the system is a kind of an editor window. We set the maximum number of target hosts with 2 and the maximum number of hosts in the past movement list with 1 in the preceding mechanism, and set a timeout value of each UA with 30 seconds, that is, every agent which dose not become a parent agent disappears after 30 seconds.

In the initial state of the experiment, the MA and SA are running on each host except for D and the server is also running on D, and the movement history is empty.

Under the environment and the initial state described above, we have confirmed that all the mechanisms described in the subsection 2.2 works correctly. For example, the following mechanism was verified by confirming the following steps (1)~(5) in this order.

(1) In the initial state, when a user arrived at the host B, a UA (say UA_0) was newly generated and started to serve.

(2) When the user departed form B, the preceding mechanisms failed, because only B was registered in the movement history and consequently the server could not determine target hosts.

(3) When the user arrived at the host C, a child agent UA_1 of UA_0 was generated at B, moved to C and started to serve at C according to the following mechanism.

(4) When the user departed from C, the preceding mechanism again failed, because the content of the movement history was (B, C) and consequently the server could not determine target hosts.

(5) When the user again arrived at B, a child agent UA_2 of UA_1 was generated at C, moved to B and started to serve at B according to the following mechanism.

The preceding mechanism was verified by confirming the following steps (6)~(11) in this order.

(6) Following the step (5), when the user again departed from B, the preceding mechanism successfully worked, that is, the server determined that the target host was C because the content of the movement history is (B, C, B) and a child agent UA_3 of UA_2 precedently moved to C.

(7) The user did not arrive at C but the host A, that is, prediction in the step (6) was wrong. Thus, a child agent UA_3 of UA_2 was generated at B, moved to A and started to serve at A according to the following mechanism.

(8) When the user departed from A, the preceding mechanism again failed, because the content of the movement history is (B, C, B, A) and consequently the server could not determine target hosts.

(9) When the user again arrived at B, a child agent UA_4 of UA_3 was generated at A, moved to B and started to serve at B according to the following mechanism.

(10) When the user departed from B, the preceding mechanism successfully worked, that is, the server determined that the target hosts were A and C because the content of the movement history is (B, C, B, A, B) and two child agents U_5^0 and U_5^1 of UA_4 precedently moved to A and C, respectively.

(11) When the user again arrived at A, UA_5^0 which precedently waited for the user started to serve.

5 Conclusions

In this paper, we have proposed a fundamental framework for mobile agent systems where a mobile agent can move to a computer precedently before a user arrives at the computer in addition to a classical following mechanism where an agent arrives at a computer after the arrival of the user. The framework predicts computers, one of which a user arrives at with a high probability, and moves an agent to each of such computers before the arrival of the user. The prediction is based on the movement history where the sequence of the computers at which the user visits, in order to correctly predict such computer with a high probability. The following mechanism is provided in case that the prediction is wrong. Moreover, the framework also provides backup and restore mechanisms in case that disappearance of agents due to problems such as their bugs. One of our future works is to evaluate our framework qualitatively and quantitatively. Another work is to improve the reliability of our framework. Because the management server of our framework becomes a single point of failure, its decentralization is required.

References

[1] J. Bates, D. Halls, and J. Bacon, "A Framework to Support Mobile Users of Multimedia Applications," *ACM Journal on Mobile Networks and Applications*, Vol. 1, No. 4, pp. 409–419, 1996.

[2] A. D. Stefano and C. Santoro, "NetChaser : Agent Support for Personal Mobility," *IEEE Internet Computing*, Vol. 4, No. 2, pp. 74–79, 2000.

[3] D. L. Ipina and S. L. Lo, "LocALE : A Location-Aware Lifecycle Environment for Ubiquitous Computing," *Proc. of ICOIN15*, pp. 419-426, 2001.

[4] K. Takashio, G. Soeda and H. Tokuda, "A Mobile Agent Framework for Follow-Me Applications in Ubiquitous Computing Environments," *International Conference on Distributed Computing System Workshop*, pp. 202–207, 2001.

[5] Y. Tanizawa, I. Sato and Y. Anzai, "A User Tracking Mobile Agent Framework "FollowingSpace"," IPSJ Journal, Vol. 43 No. 12, pp. 3775-3784, 2002.

[6] I. Sato, "AgentSpace : A Mobile Agent System," http://research.nii.ac.jp/~ichiro/agent/agentspace.html