Computer Technology and Application 4 (2013) 356-363

Evaluation of Program Code Caching for Mobile Agent Migrations

Masayuki Higashino, Kenichi Takahashi, Takao Kawamura and Kazunori Sugahara

Department of Information and Electronics, Graduate School of Engineering, Tottori University, Tottori 680-8552, Japan

Received: May 27, 2013 / Accepted: June 27, 2013 / Published: July 25, 2013.

Abstract: Mobile agents are able to migrate among machines to achieve their tasks. This feature is attractive to design, implement, and maintain distributed systems because we can implement both client-side and server-side programming in one mobile agent. However, it involves the increase of data traffic for mobile agent migrations. In this paper, we propose program code caching to reduce the data traffic caused by mobile agent migrations. A mobile agent consists of many program codes that define a task executed in each machine they migrate; thus, the mobile agent migration involves the transfer of their program codes. Therefore, our method reduces the number of the transfer of program codes by using program code cache. We have implemented our method on a mobile agent framework called Maglog and conducted experiments on a meeting scheduling system.

Key words: Mobile agent, agent migration, cache.

1. Introduction

Advances of wireless connection technologies enable us to connect to the Internet anywhere and anytime. Nowadays, not only personal computers but also various appliances, such as personal computers, mobile phones, car navigation systems, and televisions, are connected to the Internet. Thus, ubiquitous network environment would be realized in near future. In a ubiquitous network environment, their numerous appliances will communicate and work together for providing helpful services to us. Such an environment requires us to implement complex network-based systems.

Socket programming and RPC (remote procedure call) are most traditional ones to implement such a network-based system. We, however, need to implement two programs, client-side program and server-side program. This complicates the maintenance of the system because the modification of the system involves the modification of both client-side and server-side program. As the countermeasure of this complication, many researchers pay attention to mobile agent systems.

A mobile agent system is constructed from many mobile agents. A mobile agent is able to migrate among machines to achieve its tasks. We can use mobile agent migration instead of communications between the server and the client. Since the mobile agent can continue its works over the machines, we do not need to elaborate a client- and server-side program pair. Thus, we can implement a server-side and client-side program as one mobile agent. This enables us to implement a network-based system in a ubiquitous computing environment without being aware of communications APIs and protocols. Such a mobile agent system is attractive to design, implement and maintain a distributed system; however, it involves the increase of data traffic caused by mobile agent migrations.

Therefore, many researchers have proposed to mitigate data traffic caused by mobile agent migrations. Ref. [1] proposes a method to select an efficiency

Corresponding author: Masayuki Higashino, doctoral student, research field: distributed computing. E-mail: s032047@ike.tottori-u.ac.jp.

migration route calculated from the round-trip time and average packet loss rate of ping messages. Ref. [2] proposes that a machine transfers program codes before the migration of the agent happens actually. It is, however, difficult for a machine to predict where the agent migrates. Ref. [3] proposes to use a multicast message to distribute a mobile agent. This enables to use network bandwidth efficiently because the mobile agent is distributed on some machines simultaneously. However, the situation the multicast message enables to be utilized is limited. Thus, these approaches restrict situations to be applied within a specific situation and/or network environment. Further, these approaches require us to change the behaviors of each mobile agent.

OMG (Object Management Group) [4], which is a standards consortium of a mobile agent, mentions to apply a cache mechanism to mobile agent migration. Agent Space [5] also mentions a cache mechanism. They, however, do not show the detail of the mechanism and protocols. Further, the effectiveness of a cache mechanism is not well-studied even if some agent systems such as Refs. [6，7] implement a cache mechanism. Thus, in this paper, we discuss the details of a cache mechanism and protocol.

Before we discuss a cache mechanism and protocol, we should discuss mobile agent systems because the cache mechanism and protocol are for mobile agent migrations. Therefore, we define the internal structure of a typical mobile agent in Section 2. After that, in Section 3, we design the cache mechanism and protocol for an effective mobile agent migration. We implement our mechanism on a mobile agent framework called Maglog, and we show its effectiveness on various patterns of agent migrations and a practical application in Section 4. Finally, we conclude the paper in Section 5.

2. Mobile Agent System

A mobile agent system is usually structured from mobile agents and Agent Runtime Environments AREs (agent runtime environments), as shown in Fig. 1. This structure is a typical structure mentioned in Ref. [8].

An ARE is installed in each machine distributed on a network, and provides some functions to help the executions of tasks of mobile agents. A mobile agent migrates among the network by using the function of the AREs, and accomplishes its task using resources distributed on the network. For example, when a mobile agent tries to make a plan of sightseeing, the agent, firstly, goes on airline company site for the reservation of an air ticket; and then, goes on hotel reservation sites to reserve a room; finally, shows the plan to a user.

As mentioned here, a mobile agent system consists of AREs and mobile agents. Almost functions implemented in AREs are prepared to help the executions of tasks of mobile agents, thus, their functions are almost same in each AREs. However, tasks of each mobile agent are naturally different in each. For example, a mobile agent mentioned above would have a task to make a trip plan. A mobile agent to adjust meeting schedules would have a task to adjust schedules among attenders. Thus, the program codes implementing in each mobile agent are different.

Considering the installation of a cache mechanism, there are two candidates a cache mechanism to be installed in, AREs and mobile agents. However, it is unsuitable to install a cache mechanism in mobile agents, because each mobile agent has different tasks. It will involve the change of the implementation of each mobile agent as same as approaches proposed in Refs. [1-3]. On the other hand, when we consider the installation of a cache mechanism into AREs, we do

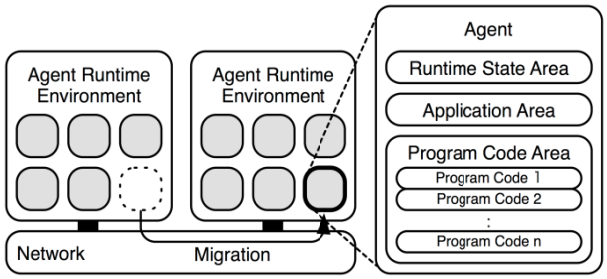


Fig. 1 Overview of a mobile agent system.

not need to pay attention to the difference of each AREs because almost AREs are composed from common functions. Therefore, we design a cache mechanism to install in AREs. Thus, programmers of a mobile agent application need not take care of a cache mechanism because the cache mechanism is working on the ARE layer.

2.1 Internal Model of a Mobile Agent

We propose to apply a cache mechanism for a mobile agent migration. What enables to be cached? To answer this question, we have to discuss the internal model of a mobile agent. We can make the typical model of a mobile agent from the discussion in Refs. [4, 8-10]. This model is much conformed to a lot of mobile agent systems such as Ref. [11], Agent Space [5], Aglets [12], and so on.

A mobile agent consists of a runtime state, application data, and program codes. The runtime state manages variables such as call stack pointers, program counters, and so on. Data in the runtime state constantly change while a mobile agent is working. The application data are related to an application. For example, in a meeting scheduling system, the application data may include users' preferences and attenders' list of meetings. These data depend on each mobile agent. The program codes represent the tasks of a mobile agent. The mobile agent proceeds with its tasks by the execution of program codes.

When we try to apply a cache mechanism to mobile agent migration, we have to classify them into cacheable or un-cacheable data. A cache mechanism stores the duplication of original data, the duplication data are reused later instead of getting the original data. If the duplication data change from the original data, its duplication is useless anymore.

A runtime state is meaningless to be cached because it constantly changes according to mobile agent's behavior. Thus, we conclude a runtime state is un-cacheable. A program code is cacheable because it does not change. However, we cannot conclude application data is cacheable or un-cacheable because it depends on an application. If we deal with the application data as the object of a cache mechanism, we have to entrust the classification of the application into cacheable or un-cacheable to programmers of a mobile agent application. This would not be preferred because programmers want to devote only to the implementation of an application. Thus, we focus on only program codes as the object of a cache mechanism.

2.2 Migration of a Mobile Agent

When an agent migrates from a source ARE to a destination ARE, the source ARE has to transfer a runtime state, application data and program codes to the destination ARE. The steps for agent migration are usually modeled as Fig. 2.

1. A source ARE connects a destination ARE, an agent tries to migrate to the destination ARE.
2. The destination ARE responses whether it allows the migration of the agent or not.
3. If the destination ARE allows the migration, the source ARE stops the agent.
4. The source ARE transfers a runtime state, application data and program codes to the destination ARE.
5. The destination ARE reconstructs an agent from program codes, a runtime state, application data received from the source ARE.
6. The destination ARE activates the agent.

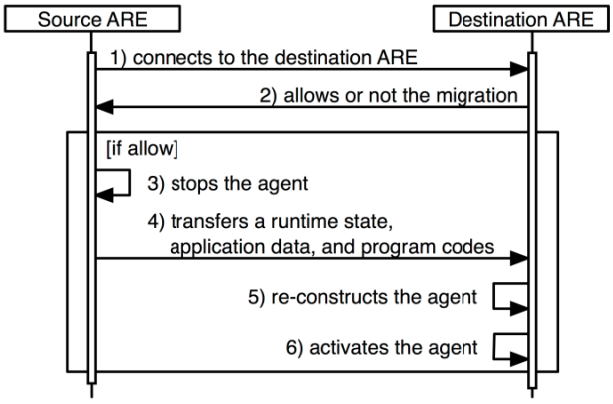


Fig. 2 Steps for mobile agent migration.

When we apply a cache mechanism to mobile agent migration, we have to add steps for a cache mechanism in these steps.

3. Cache Mechanism for Mobile Agent Migration

A cache mechanism stores the duplication of original data (cached data), and provides cached data to a requester instead of getting the original data. A cache mechanism is used as CPU (central processing unit) cache, web cache, DNS (domain name server) cache, and so on. These mechanisms cut the transfer of data from a requester (e.g., web server in web cache) to a requester (e.g., web browser); thus, cached data are managed by the requester. On the contrary, data are transferred from a requester to a requester in mobile agent migration because a mobile agent migrates from a source ARE (requester) to a destination ARE (requester). When we apply a cache mechanism to mobile agent migration, a source ARE, first, has to check which data is cached. As discussed in Section 2.1, the object of our cache mechanism is program codes. Therefore, we need to identify each program code for the check of a program code cached in a destination ARE.

3.1 Identification of Program Code

One mobile agent usually consists of many program codes. Some of these program codes are implemented by the programmer of a mobile agent application, but some may be OSS (open source software) downloaded from the Internet. Further, different programmers may use a same name in different program codes. If we create the identifier of each program code from a name such as a file name, a function name, and so on, it may cause cache poisoning. If cache poisoning occurs, a mobile agent migrated can not work well anymore. The most important point is that anomaly can harm a mobile agent easily by causing cache poisoning. Thus, it is risky to identify program codes by a name. In order to avoid cache poisoning, program code identifiers must satisfy following conditions:

1. It is difficult to create same identifier from different program codes.
2. Identifiers of same program codes must be same.

In order to satisfy above conditions, we use a hash value created from a content of program codes as a program code identifier. Since the identifier is a hash value created from a content of program code, the identifier changes if a program code changes; the identifiers are same whatever program codes are same.

We use the SHA-1 (secure hash algorithm 1) [13] as a hash function. SHA-1 takes a program code less than 2^{64} bits in length and can produce a 160-bit identifier. The probability of that same hash value created from different program codes is extremely small even when anomaly tries to find it. Even if the collision of identifiers occurs, it would throw an error. If an error occurs, the identifier of the two collided program codes is marked as pollution. A program code marked as pollution is regarded as if it is not cached. This prevents an anomaly from causing cache poisoning attacks.

3.2 Local Cache Table

We design a local cache table to store the pair of a program code and its identifier. The local cache table is prepared in each ARE. All program codes are stored in the local cache table with their identifiers. An ARE knows whether the program code corresponding to an identifier is cached or not to see the local cache table, and gets its program code.

3.3 Mobile Agent Migration with a Program Code Cache

In order to apply a cache mechanism to mobile agent migration, we add steps to check program codes cached in a destination ARE before the transfer of program codes. Program codes are transferred with a runtime state and application data at the Step 4 in the typical model of mobile agent migration in Fig. 2. Therefore, we extend the Step 4 as following:

(1) The source ARE transfers a runtime state, application data, and the identifiers of program codes

to a destination ARE.

(2) The destination ARE refers its local cache table, finds program code identifiers which are not listed in, and requests their program codes to the source ARE.

(3) The source ARE finds requested program codes from its local cache table, and transfers their program codes to the destination ARE.

(4) The destination ARE receives the program codes and stores them with their identifiers into the local cache table.

The first migration would involve the transfer of almost program codes because their program codes are not listed in the local cache table of a destination ARE. However, after that, the identifiers of their program codes are listed in the local cache table. Therefore, second migration does not involve their transfer. This mitigates the increase of data traffic caused by mobile agent migrations.

4. Evaluations

4.1 Experimental Environment

We have implemented our cache mechanism on Maglog [11], which is a Java-based mobile agent framework. ARE of Maglog has a Prolog-to-Java source code translator and a Prolog interpreter. The agent is implemented by Prolog language, and works by using the Prolog interpreter. Therefore, ARE can access the runtime state of an agent from a Prolog interpreter's internal states. Program codes are Java byte codes translated from a program implemented by Prolog. Application data are the serialized data of a Java object by Java Object Serialization [14]. The identifier of a program code is generated from Java byte codes by using java security. Message Digest [15], a local hash table, is implemented by java.util.HashMap [16].

In experiments, we used a computer of Intel Core i7-2600 Processor (8 MiB Cache, 3.40 GHz) and 32 GiB RAM. Each ARE of Maglog runs on JRE 6 [17] on Debian GNU/Linux 6.0.4 (Kernel 2.6.32-5-686-bigmem) and is connected via a network. The network is constructed by a physical NIC (network interface card), which is assigned multiple IP addresses by IP Aliasing of Linux, and each IP address is connected by emulated Ethernet which can change the speed to 10 BASE-T, 100 BASE-T, and 1000 BASE-T by Dymmynet [18]. A mobile agent consists of 4,096 program codes, the size of each program code is 1,024 KiB, the size of a runtime state is 70 KiB, and the size of application data is 0 KiB.

4.2 Overhead of Cache Mechanism

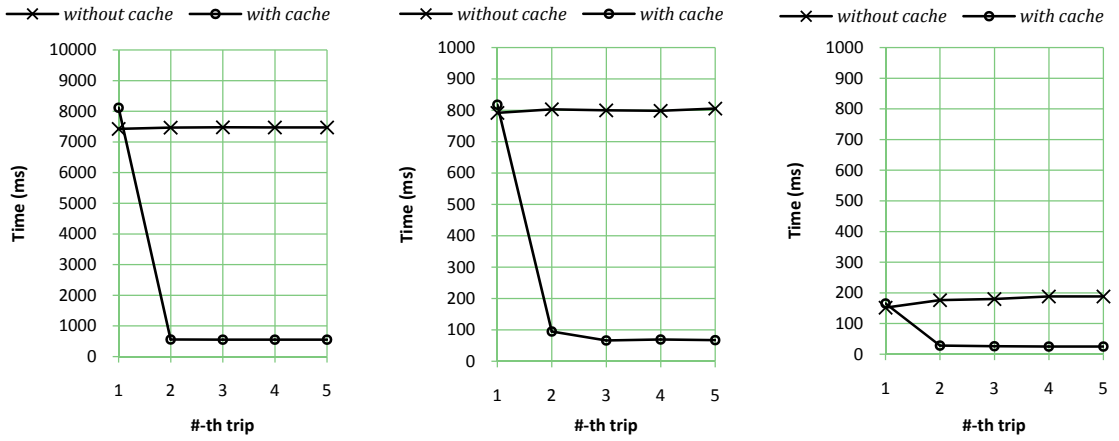
Our cache mechanism involves transferring of the identifiers of program codes, and checking of a local cache table. Firstly, we should clear the overhead of the cache mechanism. Therefore, we measured the time between an agent which tries to start migration and finishes it.

The results are shown in Fig. 3. In this figure, with cache represents the migration time in where the cache mechanism is installed; without cache represents it is not installed. The migration time of with cache at the first migration is slower than without cache. This difference means the overhead of the cache mechanism. The overhead is only 8.5% in 10 BASE-T, 3.2% in 100 BASE-T, 8.1% in 1,000 BASE-T. Thus, the overhead of the cache mechanism is relatively small. After the second migration, the migration times is dramatically improved in with cache. In 10 BASE-T, the migration time of with cache is 7.4% of without cache; 8.4% in 100 BASE-T; 13.3% in 1,000 BASE-T.

4.3 Evaluations on Agent Migration Patterns

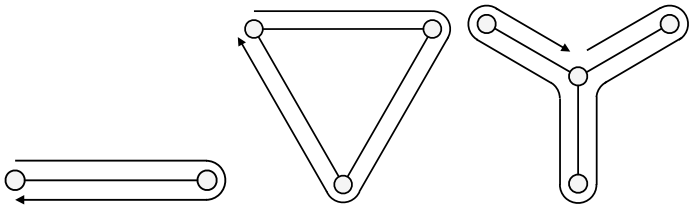
In order to clear the impact on various patterns of mobile agent migration, we measured the time of agent migrations on a shuttle, a round, and a star pattern as shown in Fig. 4.

In the shuttle pattern, cache miss occurs at the outward migration in the first trip, but it not occurs in other trips. In the round pattern, cache miss occurs in all the migrations of the first trip except the last migration, but it not occurs in all the migration after the second trip. The star pattern is repetition of a shuttle pattern.



(a) 10 BASE-T (b) 100 BASE-T (c) 1000 BASE-T

Fig. 3 Times between an agent starts and finishes the migration.



(a) Shuttle (b) Round (c) Star

Fig. 4 Agent Migration Patterns.

The experimental results conducted on 100 BASE-T are shown in Fig. 5. As shown in this figure, the migrations of *with cache* finished faster than without cache even in the first trip except the round pattern. After the second trip, the migration time of with cache is improved further in all patterns.

4.4 Evaluation on Random Agent Migration

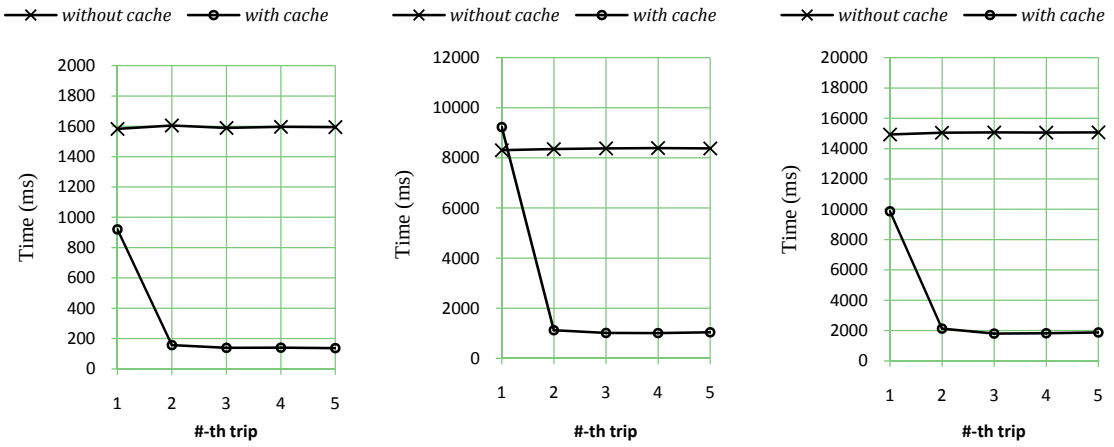
We evaluate the agent migration time when an agent migrates to somewhere randomly. In this experiment, we prepared 10 AREs. The agent migrates other ARE that is randomly selected from 9 AREs (10-current ARE). Fig. 6 shows results of 100 migrations.

As shown in this figure, the migration times of *without cache* are around 840 msec even if the number of times of migrations increases. In contrary, the migration time decreases in with cache. It is approaching 126 msec.

4.5 Evaluation on a Meeting Scheduling System

We evaluate our mechanism on a meeting scheduling system [19, 20]. The meeting scheduling system is developed by Maglog and used in our university. In this system, a user's agent of an inviter migrates among AREs to gather the schedules of invitees. If there is no candidate day by a conflict with the schedules of invitees, a user's agent negotiates with invitees to decide the day of a meeting. Thus, a user's agent migrates over AREs to gather the schedules of invitees, to negotiate with invitees, and to decide the day of a meeting.

In this experiment, we prepared an automatic manipulation program which substitutes for the operations of a human being. The program selects invitees randomly, tries to decide the day of a meeting with the invitees, makes users' schedule, replies its schedule to an inviter, and sometime makes a concession to the inviter, at a random time. We conducted this experiment on 11 computers that are installed Intel Pentium 4 processor (3.0 GHz) and 1 GiB RAM and are connected via 100 BASE-T Ethernet. An ARE of Maglog runs on JRE 1.5 on Turbolinux 10 (Kernel 2.6.0) on this computer.



(a) Shuttle (b) Round (c) Star

Fig. 5 Results on each agent migration pattern.

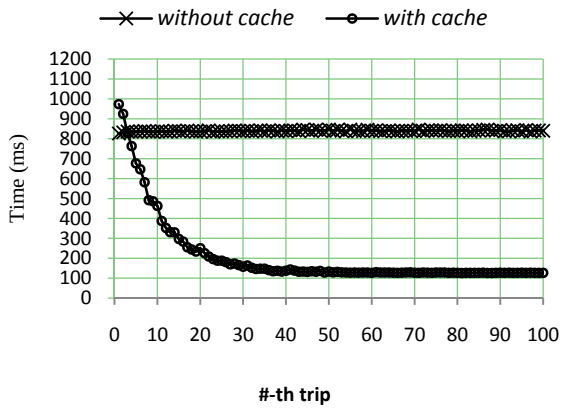


Fig. 6 Results when mobile agents migrate randomly.

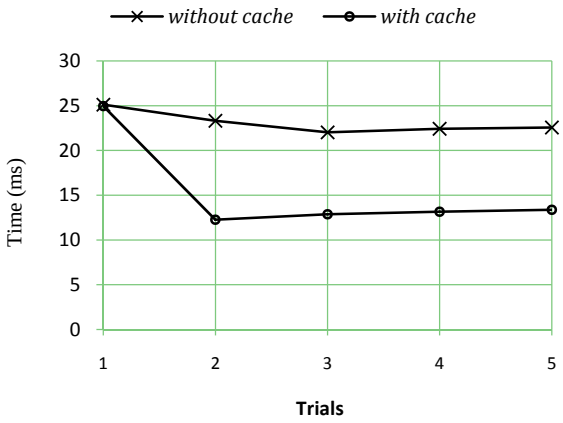


Fig. 7 Result on a meeting scheduling system.

The experimental result is shown in Fig. 7. As shown in figure, even in the decision of first meeting day, the performance of the system of with cache is almost same with without cache. From the second decision, it is about 52% of without cache. This result shows that the cache mechanism is efficient in not only toy problems but also practical mobile agent applications.

5. Conclusions

In this paper, we discussed to apply a cache mechanism to mobile agent migration. Our cache mechanism does not influence on the implementation of mobile agent applications because it works on an agent runtime environment. We implemented the cache mechanism on Maglog, which is a mobile agent framework, and conduct experiments. The result on a meeting scheduling system shows that the mechanism of the cache enables to improve its performance by 52%.

References

[1] Y. Lee, K. Kim, Optimal migration path searching using path adjustment and reassignment for mobile agent, in: Proceedings of the 4th International Conference on Networked Computing and Advanced Information Management, 2008, pp. 564-569.

[2] G. Soares, L.M. Silva, Optimizing the migration of mobile agents, in: Proceedings of the 1st International Workshop on Mobile Agents for Telecommunication Applications, 1999, pp. 161-178.

[3] D. Gavalas, An experimental approach for optimising mobile agent migrations, Mediterranean Journal of Computers and Networks 1 (1) (2005) 47-56.

[4] Mobile Agent System Interoperability Facilities Specification, Object Management Group, Inc., 1997.

[5] I. Satoh, Agent Space: A higher order mobile agent system, The Special Interest Group Notes on Programming of Information Processing Society of Japan 98 (30) (1998) 41-48.

[6] P. Braun, I. Muller, R. Kowalczyk, S. Kern, Increasing the migration efficiency of Java-based mobile agents, in: IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2005, pp. 508-511.

[7] T. White, Mobile Code Toolkit v1.6.2 [Online], http://www.sce.carleton.ca/netmanage/mctoolkit/mctoolkit162/mct.html.

[8] A. Fuggetta, G.P. Picco, G. Vigna, Understanding code mobility, IEEE Transactions on Software Engineering 24 (5) (1998) 342-361.

[9] FIPA Agent Management Specification (SC00023K), Foundation for Intelligent Physical Agents, 2004.

[10] FIPA Abstract Architecture Specification (SC00001L), Foundation for Intelligent Physical Agents, 2002.

[11] S. Motomura, T. Kawamura, K. Sugahara, Logic-Based mobile agent framework with a concept of “Field”, IPSJ Journal 47 (4) (2006) 1230-1238.

[12] Aglets Homepage, http://aglets.sourceforge.net/.

[13] D. Eastlake, P. Jones, US Secure Hash Algorithm 1 (SHA1), Request for Comments 3174, Internet Engineering Task Force, 2001.

[14] Oracle and/or its affiliates, Java Object Serialization Specification [Online], http://docs.oracle.com/javase/6/docs/platform/serialization/spec/serialTOC.html.

[15] Oracle and/or its affiliates, Message Digest (Java Platform SE 6) [Online], http://docs.oracle.com/javase/6/docs/api/java/security/MessageDigest.html.

[16] Oracle and/or its affiliates, Hash Map (Java Platform SE 6) [Online], http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html.

[17] Oracle and/or its affiliates, Java SE 6 Documentation [Online], http: //docs.oracle.com/javase/6/docs/.

[18] L. Rizzo, Dummy net: A simple approach to the evaluation of network protocols, ACM SIGCOMM Computer Communication Review 27 (1) (1997) 31-41.

[19] T. Kawamura, S. Motomura, K. Kagemoto, K. Sugahara, Meeting arrangement system based on mobile agent technology, in: Proceedings of the 2nd International Conference on Web Information Systems and Technologies, Setúbal, Portugal, 2006, pp. 117-120.

[20] T. Kawamura, Y. Hamada, K. Sugahara, K. Kagemoto, S. Motomura, Multi-agent-based approach for meeting scheduling system, in: Proceedings of IEEE International Conference on Integration of Knowledge Intensive Multi-Agent Systems, Waltham, Massachusetts, 2007, pp. 79-84.