2012 26th International Conference on Advanced Information Networking and Applications Workshops

Mobile Agent Migration Based on Code Caching

Masayuki HIGASHINO, Kenichi TAKAHASHI, Takao KAWAMURA, and Kazunori SUGAHARA
Graduate School of Engineering
Tottori University
Tottori, Japan
{*s032047, takahashi, kawamura, sugahara*}*@ike.tottori-u.ac.jp*

***Abstract*—Network-based system requires us to implement both a client-side and server-side program. The update of a client-side program involves the update of a server-side program, and vise versa. To reduce this inconvenience, mobile agent-based programming is attractive to design, implement and maintain distributed systems. Because a mobile agent migrates from one computer to other computer and can continues its execution, both a client-side and server-side program is not required to be implemented. The migration of a mobile agent, however, causes increase of data traffic. Therefore, many researchers proposed methods to reduce a number of agents migration. However the effectiveness of these approaches is limited because they depend on mobile agent behaviors. Furthermore, they restrict the implementation of mobile agents. In this paper, we focus on an agent runtime environment and try to reduce data traffic in mobile agent migrations. In our proposal, an agent runtime environment caches agent codes and agent status. Cached codes and status are reuse when a mobile agent comes back again. Thus, our method enables to reduce data traffics caused by mobile agent migration at the agent runtime environment level. Moreover, our proposed method allows us flexible implementations of mobile agents, since an agent runtime environment is independent from the mobile agent behaviors. We have applied our method on a mobile agent framework, called Maglog, and conducted experimental results. The results show 52% improvement of mobile agent migration time.**

***Keywords*-cache; mobile agent; agent runtime environment; data traffic;**

I. INTRODUCTION

Nowadays, not only personal computers but also various appliances such as personal computers, mobile phones, car navigation systems, televisions and etc., are connected to the Internet. Further, advances of wireless connection technologies enable such appliances to connect to the Internet in anywhere and anytime. Thus, ubiquitous network environment is ready to be in a real. In ubiquitous network environment, numerous appliances, including users' mobile terminals, around our town will work together autonomously and provide appropriate services to us.

The realization of such an environment requires us complex network programming techniques. Using IPC (Inter-Process Communication), such as Socket and RPC (Remote Procedure Call), is most traditional one. Since it usually becomes a server-client model, we need to implement both a server-side and a client-side program.

Therefore, many researchers pay attention to mobile agent programming. A mobile agent can migrate among computers, exactly, agent runtime environments (ARE). We can use mobile agent migration instead of communications between a server-side and a client-side program. This enables us to develop distributed systems, such as a ubiquitous computing environment, without being aware of communications APIs (Application Programming Interface) and protocols. Further, we are able to put together both tasks, which should be implemented in a server-side and a client-side program separately, into one mobile agent. Thus, mobile agent programming makes simplify distributed network programming by using mobile agent migration instead of IPC.

However, the mobile agent migration increases data traffics because it involves the transmission of not only messages but also execution state, program codes, etc. Thus, the performance of mobile agent systems becomes sometimes lower than IPC-based systems. To mitigate such case, some of researchers try to reduce a number of mobile agent migrations [1], [2], [3], [4], [5]. These, however, approaches depend on agent behaviors. Thus, they are not always adaptive to applications. As the result, the easiness of the distributed system development, which is a representative advantage of mobile agent programming, would be spoiled.

In this paper, we propose a cache mechanism to reduce data traffic when a mobile agent migrates. Our cache mechanism works on ARE layer, not on agent layer. Therefore, developers do not need to care of our cache mechanism; our mechanism is compatible to all applications. In our cache mechanism, we assume a mobile agent consist of execution state, program codes, and application data. The execution state and application data dynamically changes by mobile agent behaviors, but program codes are usually static. Therefore, we focus on program codes and try to cache them.

The rest of this paper is organized as follows. Next, Section II contains description of a typical mobile agent system and its internal structure of mobile agent. Our cache mechanism is applied to the typical mobile agent system. Section III presents our cache mechanism. Section IV describes the experimental results of the mechanism on various mobile agent migration patterns and a mobile-agent-based meeting scheduling system as a practical application. Finally, Section V draws the conclusions.

978-0-7695-4652-0/12 $26.00 © 2012 IEEE
DOI 10.1109/WAINA.2012.127

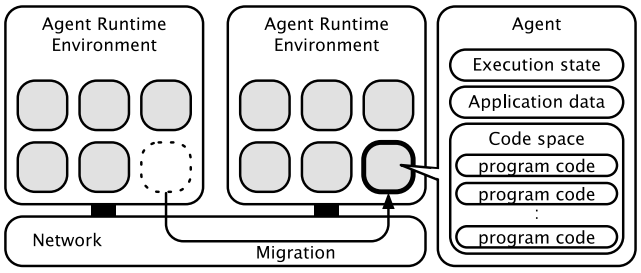


Figure 1. Overview of a mobile agent system.

II. MOBILE AGENT SYSTEM

In mobile agent systems, an ARE is installed in each machine distributed on a network. Each ARE enables a lot of mobile agent to work on. A mobile agent migrates among mobile agent AREs, and accomplishes task in each ARE. For example, when a mobile agent plans to make the schedule of a trip, the mobile agent first goes an airline company site and reserves a ticket; it goes a hotel reservation site and reserves it. Like this, a mobile agent migrates among machines and accomplishes a task.

Figure 1 shows the structure of a mobile agent, which is a typical structure according to [6]. A mobile agent consists of execution state, application data, and program codes. The execution state manages variables such as call stack pointers, program counters and so on. These variables change constantly while a mobile agent is work. The application data depends on each mobile agent. For example, the application data may include of user's preference, and a plan which a mobile agent made. The program codes define the tasks of the mobile agent. The mobile agent proceeds with its tasks by the execution of program codes. In order to continue the tasks of a mobile agent, an ARE has to transfer execution state, application data, and program codes to a destination ARE.

III. CODE CACHE FOR MOBILE AGENT

We propose a cache mechanism for mobile agent migration. After we discuss which data should be cached in our cache mechanism, we propose a cache mechanism.

A. Cacheable Data in Mobile Agent

The mobile agent consists of execution stat, application data, and program codes. These are classified into *cacheable* and *un-cacheable* data. *Cacheable* data must be static because data continuously changed is difficult to be reused.

Execution state is *un-cacheable* because it changes continuously according to mobile agent's behavior. We cannot say application data is *cacheable* or *un-cacheable* because it depends on the implementation. It is *cacheable* if data rarely change, *un-cacheable* if data change continuously. Program codes are *cacheable* because it usually does not change.

We aim at a general cache mechanism. Therefore, we focus on only the program codes for cache.

B. Program Code Cache Mechanism

Cache mechanisms (e.g., cache memory in a computer, web cache in the Internet) avoid an unnecessary transfer of data from a remote area to a local area by using data cached in the local area. On the contrary, cache mechanism for mobile agent migration avoids an unnecessary transfer of program codes from a remote area to a local area by using program codes cached in the remote area. Thus, at the first connection in mobile agent migration, the local area has to check which program codes are cached or not in the remote area.

In our mechanism, at the first connection in a mobile agent migration, a source ARE sends a set of program code identifiers to a destination ARE. The destination ARE checks which program codes are already cached in a local cache space by program code identifiers. If uncached program are found, the destination ARE requests to transfer uncached program codes to the source ARE. The source ARE sends program codes requested from the destination ARE.

Figure 2 shows a sequence diagram of a mobile agent migration with the program code cache mechanism. The step of our mechanism is as follows:

1) The source ARE sends execution state, application data, and identifiers of the program codes to the destination ARE. Usually the source ARE stops threads and close files, and etc. of the agent before this step, but this depends on mobile agent frameworks.
2) The destination ARE checks which program codes are cached in its local cache space by the identifiers.
3) If the destination ARE finds identifiers of uncached program codes, goto step 3a.
    a) The destination ARE sends the identifiers of uncached program codes back to the source ARE.
    b) The source ARE requests program codes from its local cache store.
    c) The source ARE gets program codes from its local cache store.
    d) The source ARE sends program codes corresponding to identifiers requested from the destination ARE.
    e) The destination ARE stores the program codes into its local cache space.

After that the destination ARE reconstruct a mobile agent from program codes cached in its local cache space, execution state, and application data.

First migration to a destination ARE involves transfer of program codes because almost program codes are not cached in the destination ARE. However, after second migration, since program codes constructing the mobile agent are already cached in the destination ARE, transfer of the program codes (step 3a and 3e) are not required.

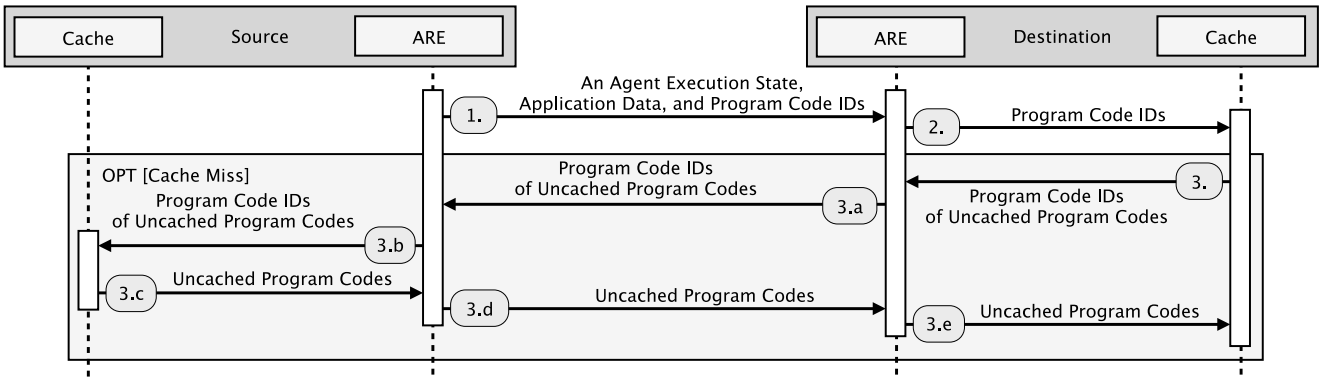


Figure 2. A sequence diagram of an agent migration with program code cache.

C. Program Code Identifier

One mobile agent is usually constructed of many program codes. Some of these program codes are implemented by programmers of the mobile agent, but some are downloaded from the Internet as program libraries for reuse. Then, programmers may revise a downloaded program, but reuse it with same name such as a file name, a class name, a method name, etc. Also, venders of the program libraries revise program codes but may use the same name. In addition, different venders may use the same name in different program codes.

In these cases, program code identifiers made from these names cause cache poisoning. If cache poisoning occurs, a mobile agent migrated cannot work well anymore. Therefore, we have to strictly determine a creation rule of identifiers. The creation rule of identifiers must satisfy following conditions:

1) Identifiers of same program codes must be same.
2) Identifiers of different program codes must be different.

Our system uses a hash value created from program codeself. Therefore, whatever a program code has been revised, the hash value of a program code changes. We use SHA-1 [7] 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 is created from different program codes are extremely small (even a collision attack have been discovered). When collision occurs, an error would be detected from a mobile agent constructed from the collided program code. If collision is detected, the identifier of the collided two program codes is marked as pollution. It is regarded as uncached program code even if a program code marked as pollution is already cached. Therefore, a program code with pollution mark is always transferred from a destination to a source ARE. This scheme cannot completely eliminate the possibility of collisions, but would be practical approach because the probability collision occurs is extremely small.

D. Cancellation of Program Code Transfer

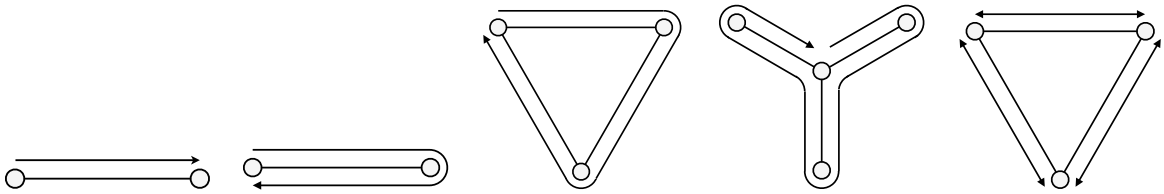
A mobile agent is often cloned and their cloned mobile agents work independently. Thus, many similar mobile agents are working simultaneity on a mobile agent system. In this situation, when a new ARE joins in the system, these similar mobile agents may try to the new ARE simultaneously. For example, some mobile agents may try to migrate to the new ARE for load balancing. In this case, same program code are transferred from different AREs to the new ARE at the same time. However, the new ARE does not need to get plural same program codes, it is enough to get only one program code. Therefore, we introduce a cancellation mechanism of program code transfer.

In the cancellation mechanism, when a destination ARE sends program code identifiers at step 3a in section III-B, the destination ARE stores their identifiers marked as *requested* with a source ARE list into its local cache space. When the destination ARE completes to get the program code at step 3e, the destination ARE sends a cancellation request message to AREs listed in the source ARE list. The source ARE received the cancellation request message stops the program code transfer.

IV. EXPERIMENTAL RESULTS

We implemented our cache mechanism on Maglog (Mobile AGent system based on proLOG) [8], [9], which is our proposed mobile agent framework. Maglog is implemented in Java and runs on any platform providing Java Runtime Environment (JRE). A mobile agent is a java object which is able to run concurrently by using threads. A program code identifier is generated by a hash function (SHA-1) from a Java bytecode.

The agent of Maglog is implemented by extending PrologCafé [10]. PrologCafé is a 100% pure Java implementation of the Prolog programing language, which contains a Prolog-to-Java source-to-source translator and a Prolog interpreter. Therefore, Java programmers are able to completely access to a Prolog interpreter's internal execution states such as



(a) One-way pattern. (b) Shuttle pattern. (c) Round pattern. (d) Star pattern. (e) Random pattern.

Figure 3. Agent migration pattern.

choice point stack, trail stack, a set of variable bindings, and etc. When an agent migration, agent execution state, application data, program codes, and identifiers are converted to byte array by Java Object Serialization [11], and transferred among AREs through HTTP/1.1 protocol.

A. Overhead of Cache Mechanism

The performance of a system implemented our mechanism becomes worse than the system without our mechanism because our cache mechanism involves the transfer of program code identifiers and the reference to cache space. Therefore, we first clear overhead of our mechanism. To clear the overhead, we use a one-way migration pattern (Figure 3(a)) and measured the mobile agent migration time using three different transmission speeds.

The results ware obtained using two computers of a Intel Core 2 Duo processor 2.66 GHz and 4 GB RAM. These two computers are connected via Ethernet of 10 BASE-T, 100 BASE-T, and 1000 BASE-T. An ARE of Maglog runs on a JRE 1.6.0_17 on Mac OS X 10.6.2. A mobile agent consists of 1020 program codes and their total size is 842 kB.

Figure 4 shows the results. The *no transfer* shows the results of when the all of program codes are shared in advanced, and besides cache mechanism is not deployed. At first migration, the migration time with our cache mechanism (*with cache*) was slower than without cache mechanism (*without cache*). However, after second migration, the migration time are dramatically improved. In 10 BASE-T, the migration time of *with cache* is 14.8% of the results of *without cache*; In 100 BASE-T, it is 16.8%; in 1000 BASE-T, 50%. They are almost same with the result of *no transfer* mechanism. These results show that as the network speed slower, our cache mechanism is more effective.

B. Effectiveness on Agent Migration Patterns

In order to confirm the impact on various mobile agent migration patterns, we conducted agent migration time on shuttle, round, star, and random patterns (Figure 3). In this experiment, we use 1000 BASE-T.

Figure 5(a) shows the result of the shuttle pattern. This pattern causes cache miss hit in outward migration and cache hit in homeward migration. Thus, the percentage of the shuttle pattern's migration time is shorter than the one-way pattern.

Figure 5(b) shows the result of the round pattern. This pattern is a repetition of the one-way patterns, except the last migration. Therefore, at first trial, all the migrations except the last cause cache miss hit. Thus, the most of the migrations causes the cache miss hit, the percentage of the round pattern's migration time is longer than the other migration patterns.

Figure 5(c) shows the result of the star pattern. This pattern is a repetition of the shuttle patterns. Thus, the percentage of the star pattern's migration time is shorter than the shuttle pattern.

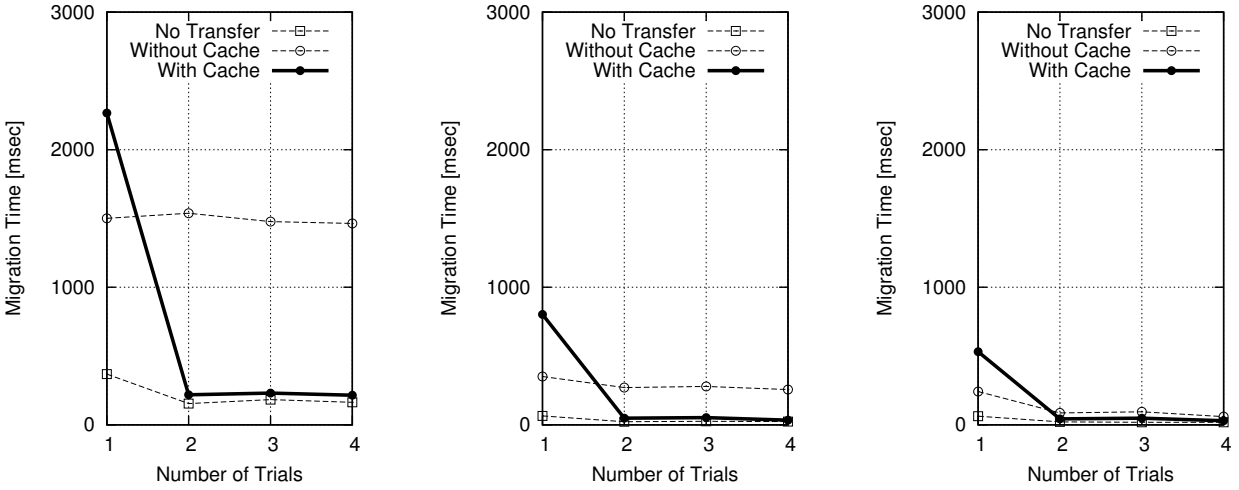
In these results, at the first migration, the *with cache* is slower than the *without cache*. However, after second migration, the *with cache* is faster than the *without cache*. Moreover, averages of migration times with cache from the first to the fourth trial are always better than the average of without cache. Therefore, it is better to deploy our mechanism if mobile agents use any same migration pattern more than 4 times.

Further, we plot migration times of the random pattern and its regression curve in Figure 6. In this pattern, we prepared only one agent and 10 nodes. The agent randomly selects one node from 9 nodes (10 - current node) and migrates there. We repeated the random migration 100 trials. In Figure 6, the migration times of *no transfer* and *without cache* are not so different in each trial. They are around about 105 msec in *without cache* and about 45 msec in *no transfer*. In contrary, in *with cache*, as the number of migration trial increases, the migration time decreases. At trial 5, *with cache* overtakes *without cache* and approaches to 5 msec (the result of *no transfer*).

C. Experience on Practical System

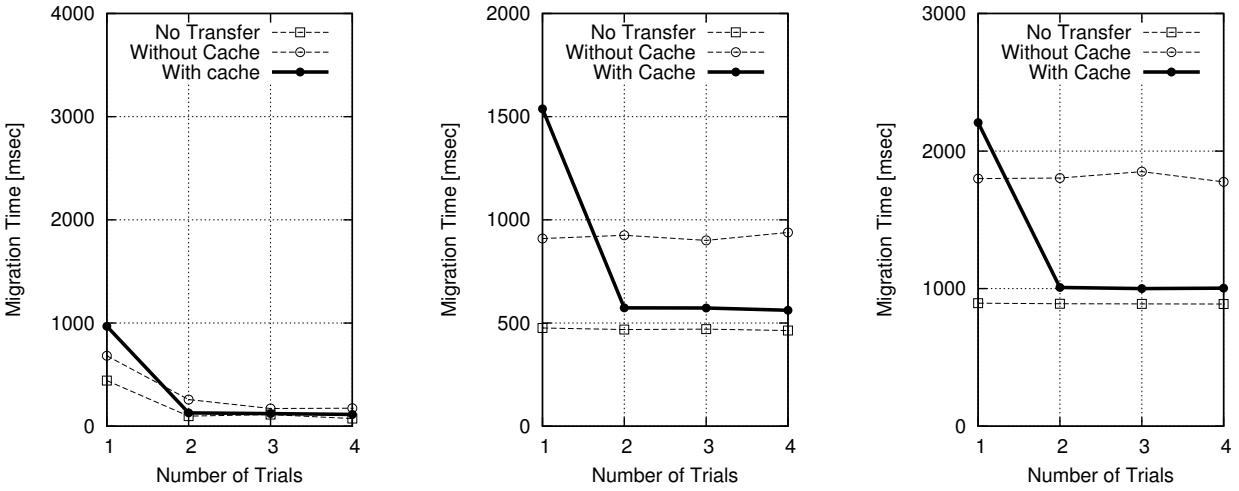
We conducted a experience on a meeting scheduling system [12], [13] to evaluate the effectiveness of the proposed method on practical systems.

This meeting scheduling system has been developed using Maglog. When a user intends to call a meeting, he only inputs information about the meeting. On behalf of the inviter, mobile agents move around each invited user's



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

Figure 4. Agent migration times on one-way pattern.



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

Figure 5. Results on each agent migration pattern.

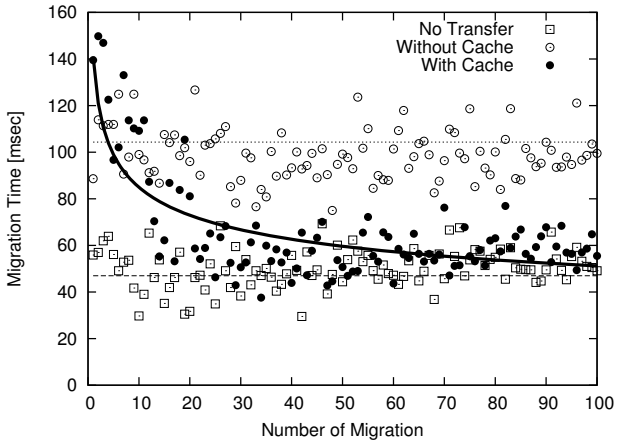


Figure 6. Results on random migration pattern.

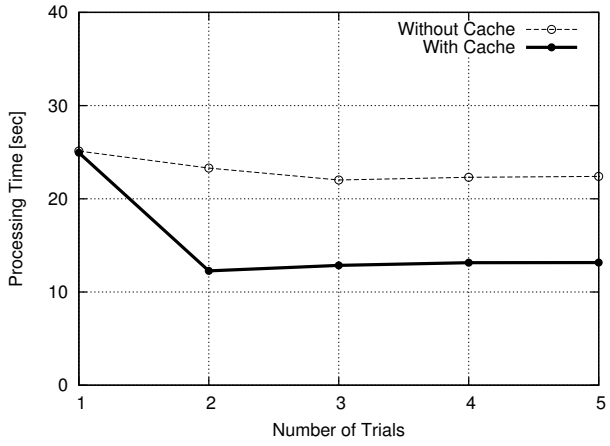


Figure 7. Results on a meeting scheduling system.

computer to ask whether he is able to join the meeting and negotiate with him if necessary.

In this system, a large number of mobile agents migrate over the network. Thus, if users want to call many meeting, then it causes increasing the number of mobile agents. As result, the system performance degradation occurs.

The results ware obtained using 11 computers of an Intel Pentium 4 processor 3.0 GHz and 1 GB RAM. These computes are connected via 1000 BASE-T Ethernet. An ARE of Maglog runs on JRE 1.5.0 on a Turbolinux 10 (Kernel 2.6.0). Additionally, in order to reduce the measurement error associated with user operations, we substitute dummy programs for the all of the user operations.

Figure 7 shows the result. Even in the first trial, the performance is almost same between *with cache* and *without cache*. After the second trial, the migration time of *with cache* is about half of *without cache*. To be precise, it is 52% *without cache*. This result shows that our proposed cache mechanism is really efficient in practical mobile agent applications.

V. CONCLUSION

In this paper, we proposed a cache mechanism of program codes for reducing data traffic on mobile agent migration. Since our method works on an agent runtime environment, it is independent from mobile agent migration strategies. Thus, it does not affect the implementation of each mobile agent.

We implemented our mechanism on a mobile agent framework, called Maglog, and conducted experimental results on a meeting scheduling system. In this experiment, our cache mechanism improved the performance of the system by 52%.

REFERENCES

[1] T. Chia and S. Kannapan, "Strategically mobile agents," in *Proceedings of the First International Workshop on Mobile Agents*, 1997, pp. 149–161.

[2] K. Jurasovic, G. Jezic, and M. Kusek, "A performance analysis of multi-agent systems," *International Transactions on Systems Science and Applications*, vol. 1, no. 4, pp. 335–342, 2006.

[3] T. Takahashi and H. Mizuta, "Efficient agent-based simulation framework for multi-node supercomputers," in *Proceedings of the 38th conference on Winter simulation*. Winter Simulation Conference, 2006, pp. 919–925.

[4] Y. Lee and K. Kim, "Optimal migration path searching using path adjustment and reassignment for mobile agent," in *Proceedings of the 2008 Fourth International Conference on Networked Computing and Advanced Information Management*, vol. 2, 2008, pp. 564–569.

[5] N. Miyata and T. Ishida, "Community-based load balancing for massively multi-agent systems," in *Massively Multi-Agent Technology*, ser. Lecture Notes in Computer Science. Springer, 2008, vol. 5043, pp. 28–42.

[6] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding code mobility," *IEEE Transactions on Software Engineering*, vol. 24, pp. 342–361, 1998.

[7] National Institute of Standards and Technology, *Secure Hash Standard*, ser. Federal Information Processing Standard 180-1. US Department Commerce, 1995.

[8] S. Motomura, T. Kawamura, and K. Sugahara, "Logic-based mobile agent framework with a concept of "field"," *Journal of Information Processing Society Japan*, vol. 47, no. 4, pp. 1230–1238, 2006.

[9] T. Kawamura, S. Motomura, and K. Sugahara, "Implementation of a logic-based multi agent framework on java environment," in *Proceedings of IEEE International Conference on Integration of Knowledge Intensive Multi-Agent Systems*, 2005, pp. 486–491.

[10] M. Banbara, N. Tamura, and K. Inoue, "Prolog cafe : A prolog to java translator system," in *Proceedings of the 16th International Conference on Applications of Declarative Programming and Knowledge Management*, 2005, pp. 1–11.

[11] Oracle Corporation, "Object Serialization," Web, 2011, http://docs.oracle.com/javase/6/docs/technotes/guides/serialization/.

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

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