Persisting Autonomous Workflow for Mobile Agents Using a Mobile Thread Programming Model

Minjie Zhang

Dept. of Computer Science and Software Engineering
The University of Newcastle, NSW 2305, Australia
Minjie@cs.newcastle.edu.au

Wei Li
Department of Information Management
Capital University of Economics and Business
Beijing 100026, China
Weili7@hotmail.com

Abstract

In this paper, we present a Mobile Thread Programming Model (MTPM), a model to simulate the persistence of a migratory thread, to overcome the problem of coexistence of mobility, persistence and autonomy for mobile agents. An advantage of MTPM over other code mobility paradigms is that the model simulates strong mobility at the application-level rather than at the system-level as used in many strong mobility-supporting systems. It is runtime dependent to migrate threads at system-level. However, MTPM is constructed on Java Virtual Machine (JVM) by using Serialization and Remote Method Invocation (RMI), thus it is suitable to heterogeneous environments without introducing new spatial and time complexities in the implementation. Distributed Task Plan (DTP), which is detailed in this paper, is a flexible implementation model of MTPM used to simulate the persistence of an agent thread. Also, a DTP is embedded with navigational and computational autonomies, so that a mobile agent can obtain a continuous and autonomous workflow only by executing a DTP.

1 Introduction

The mobile agent is one of the promising technologies used to deal with the application challenges raised with the increasing growth and diffusion of network systems, especially the Internet. Different systems [5][6][9][12] have been proposed to implement mobile agents, but few systems support autonomy of mobile agents that many WWW applications, such as mobile computing [12, 16], depend on. In the application of mobile computing, a user launches a mobile agent from a laptop that is connected to the Internet, then the user disconnects the laptop from the Internet. The mobile agent travels in the Internet autonomously, retrieving and updating information locally on behalf of its owner. Later, the mobile agent will return to the user's laptop and report the results when the user's laptop is reconnected to the

Internet. Mobile agents should have "intelligence" of self-contained navigation and computation, which give mobile agents the adaptation powers to the dynamic and heterogeneous networks, because in most cases mobile agents can not interact with their owners.

There are two kinds of features that must be satisfied by mobile agents in the context of autonomy. They are the persistence of an agent thread and the self-containment in navigation and computation. Unfortunately, the elaborated coexistence of mobility, persistence and autonomy are difficult and not adequately modeled and supported by most existing mobile agent systems. This paper proposes a Mobile Thread Programming Model (MTPM) with its implementation model, Distributed Task Plan (DTP) [11]. MTPM is an application-level model to simulate the persistence of threads after an agent migration. MTPM deals with heterogeneity of agents' execution environments by JVM without introducing any new spatial and time complexities in the implementation. DTP is a flexible implementation model of MTPM. DTP complies with the MTPM's programming paradigm and is embedded with navigational and computational autonomies. An agent plans its DTP when the agent is generated. When a DTP is executed by a mobile agent, the DTP generates continuous and autonomous workflows for the agent.

This paper is organized as follows. In Section 2, we analyze features of agent mobility, and sum up the limitations of widely studied technologies, which are unsuitable for generating persistent and autonomous workflows for mobile agents. In Section 3, we propose a new model, MTPM, for agent migration and prove its correctness. In Section 4, we outline the foundation of the MTPM implementation by using technologies of Object Serialization [15] and Remote Method Invocation (RMI) [16]. In Section 5, we describe an implementation model, DTP, of MTPM. A DTP plans distributed tasks for mobile agents. The execution of a DTP generates continuous workflows with navigational and computational autonomies for mobile agents. In Section 6, we compare the effectiveness of MTPM to typically related works by analyzing many factors. Finally we present our conclusions and directions of future researches in Section 7.

2 Problem Description on Agent Migration Mechanisms

Generally speaking, there are two kinds of agent migration mechanisms to be distinguished. They are often called weak and strong mobility [3]. Weak mobility permits an agent to migrate only with its codes and values of variables. After migration, the agent is restarted and values of its variables are restored. But the agents' execution starts from the beginning or from a special method rather than the stop point before agent migration. Weak mobility does not support the persistence of agent threads. Many mobile agent systems only support weak mobility of agents. They are Odyssey [5], Voyager [12], Java-To-Go [11], Aglets [9], Facile [17], Tocoma [8], Mole and Grasshopper [7] etc. Strong mobility permits the agent to migrate not only with codes but also with the whole state of thread execution. After migration an agent is restarted its execution exactly from the point where it was suspended before migration, so strong mobility supports the persistence of agent threads. Some mobile agent systems support strong mobility of agents. They are Telescript, Agent Tcl [6], Ara [13] and Sumatra [1] etc.

In many weak mobility supporting systems, the mechanism behind the weak mobility is to program a mobile agent kernel with many different methods that will be executed by the agent at different network nodes. When an agent executes the mobile primitives for migration, the agent must explicitly provide a destination address and a method to be executed at that destination. On the other hand, strong mobility requires that the mobile agent server transparently and randomly captures the thread's execution mapping of any agent, transports the captured mapping and restores the transported mapping after agent migration.

The state of the art is that mobile agent systems with weak mobility have wide platform acceptances because they are often constructed by popular languages such as Java, but they suffer from the following limitations for programming autonomous mobile agents.

1. Few procedures or primitives are provided for supporting agents' autonomies in the mobility and the computation. Although it is possible, it is difficult to program a mobile agent with desirable autonomies.

2. Their programming paradigms are not for workflow models [2], so they provide no inherent supporting for designing an autonomous agent. It is difficult for them to generate continuous workflows.

3. A mobile agent and its distributed tasks are programmed in the same program unit (or class), so both reusability and flexibility are lost. A mobile agent can only execute a distributed task without revising its codes.

Persistence is fundamental for the next-generation of agent-based applications [14]. Although current mobile agent systems with strong mobility are easy to expend for supporting autonomies for mobile agents, they are often constructed with special languages or they modify popular language's specification such as JVM for facilitating the capture of an agent's execution state. These prevent them from being widely accepted and used to build agent-based applications in multiple platforms. An evident example is that General Magic rewrites its mobile agent system, Telescript, into Odyssey by using Java, in order to be widely accepted. In addition, because threads are strongly bound to the runtime system, it is difficult for strong mobility at system-level implementation to deal with heterogeneous environments in which mobile agents roam. Also it is inefficient to implement persistence at system-level by capturing, transporting and restoring the execution state of the agent thread because an agent thread has huge information of execution stacks and heaps. It is reported in [4] that strong mobility is implemented at language-level, but [4] introduces extra time and space overhead at the same time.

In the context of autonomy, agents must have two features, the persistence of agent thread and the self-containment in navigation and computation. Limitations of current agent migration technologies have made us design a new mobile agent system Mobile Agent Template (MAT) [10] by Java for supporting autonomous mobile agents. In MAT, we program mobile agents with MTPM paradigm. We pursue coexistence of persistence, mobility and autonomy with MTPM. Fully transparent migration is not a necessity. MTPM simulates strong mobility at an application-level using a lightweight implementation on JVM, so it is suitable for programming mobile agents to heterogeneous environments without introducing new spatial and time complexities in the implementation. When it is generated, a mobile agent plans its DTP that is the implementation model of MTPM. The execution of a DTP generates

continuous and autonomous workflows for mobile agents. MTPM does not need any modification of JVM, and it uses two new mechanisms, Serialization and RMI, provided by JVM.

3 Mobile Thread Programming Model

3.1 Persistence simulation of a mobile agent's thread

In this section, we introduce MTPM, which is a model to simulate the persistence of migratory threads for mobile agents at application-level. This model depends on Serialization and RMI mechanisms of JVM. Writing an object state into a serialized form is sufficient to reconstruct the object as it is read. Thus, writing and reading objects are called object serialization and deserialization. A thread is ultimately code and data, and we suspect that state can always be represented by data. Object serialization and deserialization are essential and enough to simulate a persistent state of an agent thread based on the following proposed theory.

Definition 1: The serialization operation on an object *Obj* is denoted as *Ser(Obj)*, and the deserialization operation on a serialized object *Ser(Obj)* is denoted as *Deser(Ser(Obj))*.

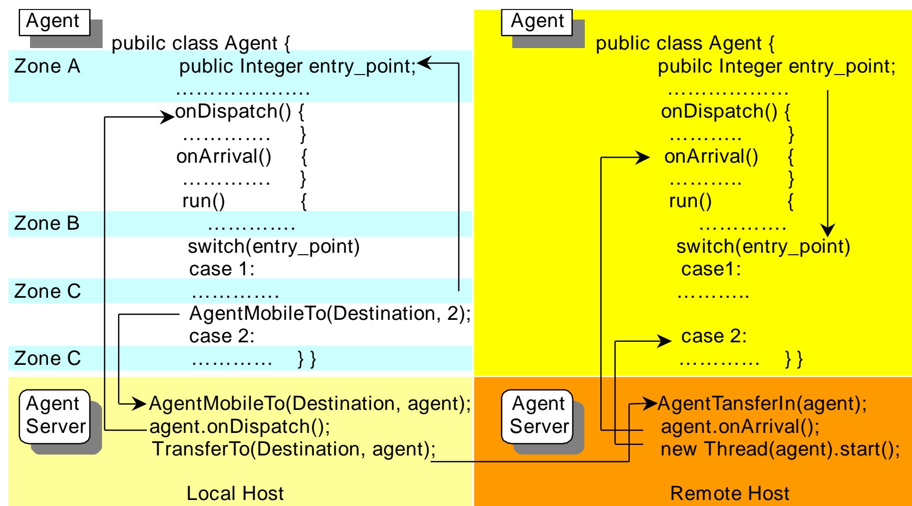

Fig.1 The migration simulation of agent's thread by MTPM

In order to simulate the state persistence of an agent's thread, the following three methods must be provided to a mobile agent.

onDispatch(): This method is called just before an agent migration. It performs serialization operations on every non-transient object *Obj* in Zone A (see Fig. 1) of an agent, i.e. for each *Obj* in Zone A of an agent, perform *Ser(Obj)*.

onArrival(): This method is called just after a serialized agent object is transported to the destination by RMI. Contrary to *onDispatch()*, it performs deserialization operations on every serialized object *Ser(Obj)* of an agent, i.e. for each serialized object *Ser(Obj)* of an agent, perform *Deser(Ser(Obj))*.

run(): This method is the running method of an agent thread. This method will be called when an agent thread is generated at the home machine or restored at a remote network node. In order to support the simulation of the state persistence of an agent's thread, the *run()* method should use the *switch entry_point* paradigm as shown in Fig.1. The *run()* method consists of *switch-case* statements. Every mobile primitive is the last statement in a case branch, and a mobile primitive sets a new entry of the *run()* method that will be recalled at the next destination from the new entry.

Having defined the above three methods, the migration simulation of an agent thread is also graphically illustrated in Fig.1. When an agent executes the mobile primitive such as *AgnetMobileTo(Destination, 2)*, the mobile primitive sets a new entry *entry_point* as 2 and calls the method *AgentMobileTo(Destination, this)* of current Agent Server. The Agent Server calls back the agent's method *onDispatch()* so as to give an opportunity to the agent to serialize its objects. Then, current Agent Server calls its method *TransferTo(Destination, agent)* to transport the agent object to the Agent Server at the destination. In fact, current Agent Server calls the remote method *TransferIn(agent)* of the Agent Server at the destination by RMI. RMI permits to transfer an object reference graph as a parameter to a remote method, so in fact, current Agent Server transports the agent in the form of a serialized object to the destination. The first thing of *TransferIn* is to call back the agent's method *onArrival()* to deserialize the agent's serialized objects, then generate a new thread to execute the agent. The agent will be executed from the statement case 2 when its method *run()* is recalled.

3.2 Proof of MTPM's correctness

Generally, objects that are generated by an agent are in Zone A, Zone B or Zone C. Objects in Zone A can be persisted by Java Serialization because they are class-level variables. However, objects in Zone B or Zone C can not be persisted by Java Serialization because local variables of a method are located in the method call stack of JVM and can not be reached by Java Serialization. But according to Object-oriented paradigm, any object that is generated in a method is local and transient, so any persistent information of an agent does not depend on objects in Zone B or Zone C. In addition, any object in Zone B, which may be used in following case branches, is regenerated when the method *run()* is recalled; Objects in Zone C do not depend on each other if they are in different case branches. Summarizing the above features in the proposed paradigm of agent design, we have the following theorem, which proves that the persistence of agent thread can be simulated by MTPM.

Axiom 1: For any object *Obj*, *Obj* is equal to *Deser(Ser(Obj)).*

Axiom 2: The execution state of an agent's thread is only determined by both the states of the agent's objects in Zone A and the execution point of the method *run()* in MTPM paradigm.

Theorem 1: The persistence of an agent thread can be simulated by MTPM during the agent migration.

Proof: We must define *entry_point* as a member variable of a mobile agent class because Serialization can not capture any local variable of a method. Suppose an agent executes a mobile primitive *AgentMobileTo(Destination, k)*, which is the number *k-1* statement of the agent's method *run()*, then the execution stop-point *k* is

stored in the object *entry_point* of the agent. Also suppose the agent has valid objects Obj_1, Obj_2,……, Obj_n (of course including the object *entry_point*) in Zone A, then the method *onDispatch()* of the agent will serialize all the objects, i.e. performs $Ser(Obj_1)$, $Ser(Obj_2)$,……, $Ser(Obj_n)$ when the method is called just after *AgentMobileTo*. When the agent object is transported to the destination, its method *onArrival()* is called. The method *onArrival()* deserializes all the serialized objects, i.e. performs $Deser(Ser(Obj_1))$, $Deser(Ser(Obj_2))$,……, $Deser(Ser(Obj_n))$. From *Axiom 1*, Obj_i is equal to $Deser(Ser(Obj_i)$, where i belongs to $\{1, 2,……n\}$, so states of all the objects in Zone A of the agent (of course including object *entry_point*) are persistent. …..…(a)

When the agent is restarted at the destination, its thread's execution method, *run()*, is called. All the objects in Zone B will be regenerated and the method *run()* will execute from the case statement that is determined by object *entry_point*. Because the object *entry_piont* is *k*, the stopped execution point is restored from the statement *k* after the agent migration has been completed. ………(b)

From (a) and (b), the execution state of the agent's thread is persistent after the agent migration according to *Axiom 2*.

4 Foundation of MTPM Implementation

Our agents need persistence, which is the ability of an object to record its execution state so the state can be reproduced in other environments. With the release of Java1.1, the Java community has gained access to a wide variety of features. Important features, which contribute to the implementation of MTPM, are object Serialization and RMI. Combinations of these make it possible to simulate persistence of an agent's thread with object persistence.

Object Serialization provides a program with the ability to read or write a whole object to and from a raw byte stream. It allows objects and primitives to be encoded into a byte stream suitable for streaming to some type of network or to a file-system, or more generally, to a transmission medium or storage facility. The real power of object Serialization is the ability of programs to easily read and write entire objects and primitive data types, without converting to/from raw bytes or parsing clumsy text data. Object Serialization has taken a step in the direction of being able to store objects instead of reading and writing their state in some foreign and possibly unfriendly format. In order to be persistent, the class definition of a mobile agent should implement the Serializable interface. We can customize serialization for an agent by rewriting and providing two methods writeObject and readObject to the agent. The two methods in agent implementation are functional equivalents to *onDispatch()* and *onArrival()* in MTPM. The process of serializing an object involves traversing the graph created by each object's references to other objects and primitives. So all the objects including agent object and objects that can be reachable by references of the agent are preserved during agent Serialization.

RMI enables a program running on a client computer to make method calls on an object located on a remote server machine. Object-oriented design requires that every task be executed by the object most appropriate to that task. RMI takes this concept one step further by allowing a task to be performed on the machine most appropriate to the task. A client can invoke the methods of a remote object with the same syntax that it uses to invoke methods on a local object. RMI has several advantages over traditional Remote Procedure Call (PRC). RMI can pass full objects as arguments and

return values. This means that we can pass complex types such as an agent object as a single argument without extra converting codes. Passing objects lets us use full power of object-oriented technology in agent migration. When passing an object as an argument, RMI moves class implementations of the object at the same time. At this point, RMI moves behavior from a client to a server or a server to a client, so we can benefit from fully object-oriented patterns for agent design. In addition, RMI uses built-in Java security mechanisms that allow the agent system to be safe when moving agents. Customized security mechanisms are easily integrated into agent system with RMI security Model such as specifying Security Manager in Java 1.1 or Policy File in Java 1.2. With RMI we can write a mobile agent system in the simplest form like this:

```
import java.rmi.*;
public interface AgentServer extends Remote {
    void        TransferTo (String Destination, Agent agent)
                                throws RemoteException, InvalidAgentException;
    void        TransferIn(Agent agent)
                                throws RemoteException, InvalidAgentException;
    AgentServer   getRemoteAgentServer(String Destination)
                                throws RemoteException, AgentServerNotFoundException;
    void        registerAgentServer(String agentServerName)
                                throws RemoteException, AgentNameInvalidException;
    void        unregisterAgentServer(String agentServerName)
                                throws RemoteException, EntryNotFoundException;
    void        createAgent(String agentClassName, Class parameterTypes[], Object initargs[])
                                throws RemoteException, InvalidAgentException;
     }

import java.io.Serializable;
pulblic interface Agent extends Serializable {
    void onDispatch();
    void onArrival();
    void run();    }
```

In an agent system, an Agent Server is a remote object to which other Agent Servers in the system have references. Transporting an agent would be a matter of creating a class that implemented the *Agent* interface, finding a server, and invoking its *TransferIn* method with the agent object as an argument. The implementation for the agent would be transported to the server and run there. We don not have to write the two methods of *onDispatch()* and *onArrival()* if we would like to perform default serialization for a mobile agent by RMI. After deserializing the agent, the *TransferIn()* method will start up a new thread for the agent and invoke its *run()* method.

5 DTP: a Flexible Implementation of MTPM

In fact, in the implementation of MTPM, we do not program all the tasks to be executed by an agent in its *run()* method because this kind of design can not support flexibility, reusability and workflow mode. Instead, we use a flexible implementation, Distributed Task Plan (DTP), to support continuous and autonomous workflows for mobile agents.

5.1 Architecture of DTP

In this section, we introduce Distributed Task Plan, which is a flexible implementation of MTPM for generating continuous and autonomous workflows for mobile agents. In MAT, we define two kinds of autonomies for mobile agents.

Definition2: Mobile autonomy is the capability of self-navigation of mobile agents through the underlying network.

Definition 3: Computational autonomy is the capability of self-containment of mobile agents in computational functions for the accomplishment of a distributed task.

In order to obtain desirable autonomies for mobile agents, firstly, we provide enough programming components called autonomous primitives, which are used to construct a DTP and further to provide mobile agents with autonomies in the navigation and the computation. Four kinds of autonomous primitives are defined in MAT for DTP designing.

1. Mobile primitives define the mobility of an agent. A mobile agent can merely transport itself to the next destination from the current network node by calling a simple migration primitive, or clone and transport each of its duplicates to different destinations by calling a multiple migration primitive.

2. Computational primitives define invocations of computational resources. A computational primitive specifies where to find the current computational procedure, how to load it and how to execute it. By using the computational primitives, a mobile agent realizes that (a) the current computational procedure is carried by the agent or is resident at the visiting node; (b) the procedure should be started in a different process or loaded into its own process; and (c) how to run the computational procedure such as synchronously or asynchronously.

3. Solution synthesis primitives define the combination of multiple solutions from different mobile agents. The solution synthesis is needed when a task is divided into several subtasks and executed by different mobile agents concurrently. It is highly efficient to divide a task into several subtasks and to assign these subtasks to different mobile agents for the executions when the task can be executed concurrently and multiple resources are available.

4. Control primitives define the execution flow of mobile primitives, computational primitives and solution synthesis primitives. Enough control structures in control primitives are needed to efficiently coordinate the executions of all the above three primitives.

Having defined primitives, we provide reasonable model to design DTP, which depicts distributed tasks for mobile agents by advantages of those pre-defined autonomous primitives. Normally, a DTP is composed of all the four kinds of autonomous primitives.

Definition 4: A Distributed Task Plan (DTP) is a static description of a distributed task, which is to be executed by a mobile agent.

A DTP consists of primitives, which are arranged into two lists. A list, which we prefer to call a control queue (*CQ*), only contains control primitives, and the other list, which we prefer to call a reusable primitive list (*RPL*), contains any primitives except

control primitives. The architecture of a DTP is graphically illustrated in Fig. 2 (concrete meanings of primitives of Fig. 2 are defined in [11]).

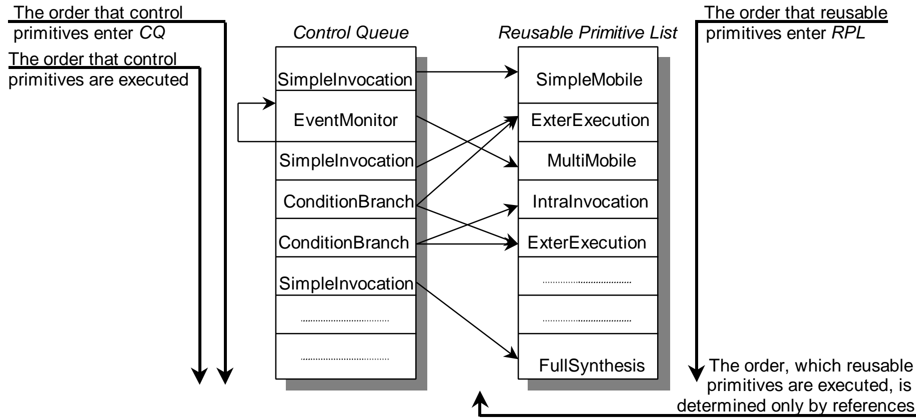

Fig.2 The architecture of DTP

5.2 Continuous and autonomous workflows of a DTP

When being generated, a mobile agent plans its own DTP for the execution of a distributed task satisfying a user's requirements. The planning includes *Objective Matching*, *Primitive Selection* and DTP *Generation* by using the user's requirements, network state information and task features. A mobile agent can also replan its DTP when current DTP fails during the execution. The planning of a DTP is detailed in [11].

Definition 5: An execution of a DTP by a mobile agent in a dynamic network environment is a continuous and autonomous workflow of the mobile agent.

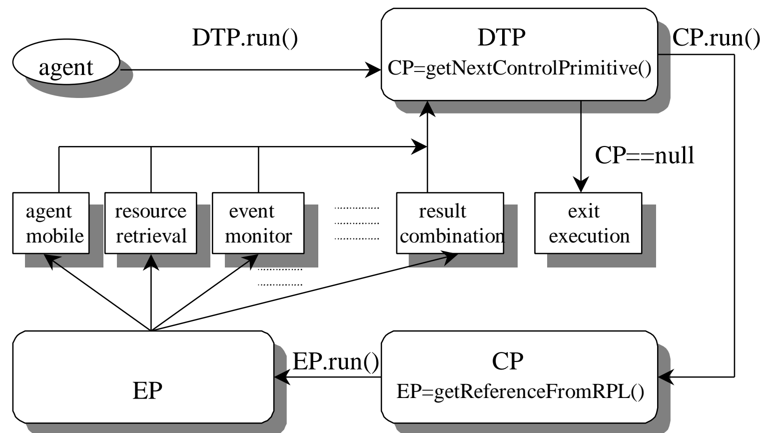

Fig. 3 The continuous & autonomous workflow generating by DTP

A mobile agent has a reference to its DTP, and a DTP has reference to a *CQ*. All the objects, from the agent itself, DTP to autonomous primitives, have a *run()* method. Every object's *run()* method just calls the *run()* method of another object to which the former has a reference. The autonomous workflow of a mobile agent is generated when the mobile agent executes its DTP by calling its *run()* method as shown in Fig.3.

The order of primitives in a *CQ* is important. The control primitives in a *CQ* are executed sequentially. A control primitive in a *CQ* has one or more references to primitives in a *RPL* corresponding to which type the control primitive is. A reference of a control primitive in a *CQ* depicts a possible invocation to a primitive in a *RPL*. The order of primitives in a *RPL* is not important because the invocations to them are determined only by references of control primitives in a *CQ*. A *RPL* is just a repository of autonomous primitives that a mobile agent may need to execute when transporting in underlying networks. So a mobile agent only executes control primitives in a *CQ* one by one, then further executes primitives in a *RPL*. An execution of a DTP is a continuous and autonomous workflow of a mobile agent. Constructing a complex workflow by using DTP provides a mobile agent with autonomy, flexibility and reusability in distributed applications.

6 Related Work

To our knowledge, providing transparent migration for agents at language-level is done in [4][18], and providing mobility, persistence and autonomy for agents at the same time is done in very few models besides our model. To capture the state of an agent for fully transparent migration, [4] has developed a preprocessor that instruments the programmers' Java codes by adding codes. Those added codes do the actual state capturing, and reestablish the state on restart at the target machine. [4] does this instrumentation by parsing the original program code using a Java based parser. In fact, what is done by [4] is a mechanical transformation of codes written for transparent migration into codes written for non-transparent migration. [4] has to deal with complex problems, such as saving and rebuilding local variables, objects and the method call stack, but to leave thread synchronization to programmers. In [18], a self-migration computation is separated into two layers. The computational layer consists of an arbitrary collection of functions distributed throughout the system, and the coordination layer deals primarily with the locations at which various functions are to be executed and the communication among functions. In [18], the programmers' original script must satisfy the following three conditions for facilitating the transformation of the original script into a pseudo code script that supports transparent migration: 1. The original script consists of only function calls; 2. All functions are numbered and each knows its possible successors; and 3. Any statement that may cause a context switch may execute only as the last statement of a function. Tab.1 compares some important features of [4], [18] and MTPM.

MTPM provides a lightweight mechanism that is functionally equivalent to transparent migration. MTPM does not introduce any extra time and space overhead as in [4], but only has the restriction as the third one of [18]. MTPM generates persistent and autonomous workflows at minimum time and space costs, and restrictions in programming paradigm.

Features	[4]	[18]	MTPM
Location of Mobile Instruction	anywhere	restricted(more)	restricted(less)
Transparent Migration	yes	yes	functionally equivalent
Preprocessing	yes	no	no
Extra Time Overhead	4%~19%	none	none
Blow-up Factor of Bytecode	3.4~4.7(times)	none	none
Autonomy	none	less	more
Platform Independence	yes	no	yes

Tab.1 Comparison of [4], [18] and MTPM

7 Conclusion

Many WWW applications such as mobile computing depend on autonomies of mobile agents. The threads of mobile agents should be continuous and autonomous workflows; i.e. the persistence and autonomy of thread of an agent are two basic features of an autonomous mobile agent. It is difficult, inefficient and runtime dependent to support thread persistence at system-level. In the context of autonomy and heterogeneity, the widely used code migration mechanisms provide no inherent support for the design of mobile agents. Thus, we have proposed and proved a model, MTPM that is suitable for designing a workflow of mobile agents. MTPM simulates the state persistence of thread of an agent by Serialization and RMI without introducing any new spatial complexity in the implementation.

DTP is a flexible implementation of MTPM. DTP complies with the programming paradigm defined by MTPM, so a DTP generates a continuous workflow when a mobile agent executes it. Because a DTP is composed of autonomous primitives, a DTP embeds some degree of autonomy or "intelligence" into mobile agents. Using a DTP, navigational and computational autonomies are carried by mobile agents as they transport through the underlying computational networks. A mobile agent can freely transport and use many different computational resources in a heterogeneous network by executing autonomous primitives in a DTP without interaction with its owner.

For supporting the coexistence of persistence, mobility and autonomy, we have presented a basic framework, MTPM, with its implementation model, DTP, in this paper. Our future work will focus on investigating the suitability of MTPM and DTP in WWW applications, such as Internet information retrieval, electronic commerce and Computer Support Cooperative Work (CSCW). From feedback of the investigations, we can find problems in MTPM and DTP, and make improvements in both the model and its implementation.

References

[1] A. Acharya, M. Ranganathan, and J. Saltz, Sumatra: A Language for resource-aware mobile Programs, In *Mobile Object System: Towards the Programmable Internet*, Lecture Notes in Computer Science, **No**. **1222**, Springer-Verlag, pp. 111-130, Linz, Austria, July 1996.

[2] Ting Cai, Peter Gloor, and Saurab Nog, Dartflow: A workflow management system on the Web using transportable agents, *Technical Report TR96-283*, Department of Computer Science, Dartmouth College, Hanover, N.H., 1996.
[3] A. Fuggetta, G. Picco, and G. Vigna, Understanding Code Mobility, *IEEE Transactions on Software Engineering*, **Vol. 24, No. 5**, pp. 342-361, May 1998
[4] Stefan Funfrocken, Transparent Migration of Java-Based Mobile Agents: Capturing and Reestablishing the State of Java Programs, In *Proceedings of the Second International Workshop on Mobile Agents*, Lecture Notes in Computer Science, **No. 1477**, Springer-Verlag, pp. 26-37, Stuttgart, September 1998.
[5] General Magic, Introduction to the Odyssey API, available at http://www. generalmagic.com /agents/ odysseyIntro.pdf, 1997-1998.
[6] R.Gray, Agent Tcl: A flexible and Secure mobile-agent system, In *Proceedings of Fourth Annual Tcl/Tk Workshop*, Monterey, California, July 1996.
[7] IKV, Grasshopper, available at http://www.ikv.de/products/grasshopper.html, 1999.
[8] D. Johansen, R. van Renesse, and F. B. Schneider, An introduction to the TACOMA Distributed system, Computer Science Technical Report 95-23, University of Tromso, Norway, 1995.
[9] D. B. Lange and M. Oshima, Programming and Developing Java Mobile Agents with Aglets, Forthcoming booking, Addsion-Wesley, 1998.
[10] Wei Li, and Minjie Zhang, Distributed Task Plan: A Model for Designing Autonomous Mobile Agents, in the Proceedings of International Conference on Artificial Intelligence, Las Vegas, pp. 336-342, 1999.
[11] William Li, and D. G. Messerschmitt, Java-to-go, Technical report, Dept. of EECS, university of California, Berkeley, available http://ptolemy.eecs.berkeley.edu /dgm/javatools/java-to-go/, 1996
[12] Object Space, Voyager Core Technology 2.0 User Guide, available at http://www.objectspace.com/ developers/voyager/white/voyager20.pdf, 1998.
[13] H. Prine, An introduction to mobile agent programming and the Ara system, *ZRI Technical Report 1/97*, Dept. of Computer Science, University of Kaiserslautern, available at http://www.uni-kl.de/AG-Nehmer/Ara.ara.html, January 1997.
[14] M. Mira da Silva, and A. Rodrigues da Silva, Insisting on Persistent Mobile Agent Systems, In *Proceedings of the First International Workshop on Mobile Agents*, Lecture Notes in Computer Science, **No. 1219**, Springer-Verlag, pp. 174-185, Berlin, April 1997.
[15] SUN, Object Serialization, available at http://java.sun.com/products/jdk/ 1.2/docs/guide/serialization, 1999.
[16] SUN, Java Remote Method Invocation Specification, available at http://java.sun.com/products/jdk/ 1.2/docs/guide/rmi/spec/rmi-title.doc.html, 1999
[17] B. Thomsen, L. Leth, and S.Prasad, Faclie Antigua Release Programming Guide, *Technical Report ECRC-93-20*, European Computer Industry Research Centre, Munich, Germany, Dec. 1993.
[18] Christian Wicke, Lubomir F. Bic, Michael B. Dillencourt, and Munehiro Fukuda, Automatic State Capture of Self-Migrating Computations in MESSENGERS, In *Proceedings of the Second International Workshop on Mobile Agents*, Lecture Notes in Computer Science, **No. 1477**, Springer-Verlag, pp. 68-79, 1998.