MobileSpaces: A Framework for Building Adaptive Distributed Applications Using a Hierarchical Mobile Agent System

Ichiro Satoh *

Department of Information Sciences, Ochanomizu University
2-1-1 Otsuka Bunkyo-ku Tokyo 112-8610, Japan
E-mail: ichiro@is.ocha.ac.jp

Abstract

This paper presents a new framework for constructing mobile agents. The framework introduces the notion of agent hierarchy and inter-agent migration and thus allows a group of mobile agents to be dynamically assembled into a single mobile agent. It provides a powerful method to construct a distributed application, in particular a large-scale mobile application. To demonstrate how to exploit our framework, we construct an extensible and portable mobile agent system based on the framework. The system is implemented as a collection of mobile agents and thus can dynamically change and evolve its functions by migrating agents that offer the functions. Also, mobile agent-based applications running on the system can naturally inherit the extensibility and adaptability of the system.

1 Introduction

Mobile agents are autonomous programs that can travel from computer to computer under their own control. They can provide a convenient, efficient, and robust framework for implementing distributed applications including mobile applications. On the other hand, component-based software development technology is being used widely [14] as a powerful approach for the development of distributed applications. The technology allows us to combine a collection of subcomponents into an application or a large-scale component. However, existing mobile agent systems unfortunately lack any mechanism for structurally assembling more than one mobile agent. This is a serious limitation in the development of a mobile agent-based application which is large in scale and complicated.

Moreover, a mobile agent system often needs to be used in heterogeneous environments, for example PDAs, embedded computers, and wireless networks, and is often required to provide visiting agents with the services that they need and may not have been initially supported. However, most existing mobile agents systems are inherently dependent on particular environments and cannot dynamically evolve and adapt themselves to the requirements of visiting agents and the execution environments.

To solve the above problems, this paper proposes two concepts, *agent hierarchy* and *inter-agent migration*. The former means that each mobile agent can be a container of other mobile agents inside itself, and the latter allows mobile agents to move inside other mobile agents as well as inside other computers. These concepts enable us to organize more than one mobile agent into a single mobile agent and introduce agent migration as a meta mechanism of dynamically changing and extending mobile agent-based applications. We try to construct a extensible and portable mobile agent system built on the Java language [2]. The system is characterized in that it allows a group of mobile agents to be composed hierarchically and its architecture itself is structured based on the concepts.

We should explain the reason our hierarchical mobile agent model is needed in the development of distributed applications. Although existing software development methodologies, including object orientation, construct large and complex mobile applications, such applications are essentially static and monolithic in the sense that they are not adaptable. Moreover, a large-scale application software program is often constructed as a collection of subcomponents. Consequently, a mobile application needs to be migrated as a whole with all its subcomponents. Our hierarchical model can naturally introduce mobile agents as mobile software components and can easily construct a large-scale and adaptable mobile application as a compound mobile agent.

This paper consists of the following sections. In Section 2, we present the basic ideas of the system presented in this paper. Section 3 presents a mobile agent system called *MobileSpaces*. In Section 4, we outline agent programs in the system and applications running on the system, and in Section 5 we describe the current implementation status and present some results of our basic evaluation. Section 6 sur-

*Also with PRESTO, Japan Science and Technology Corporation

veys related work and Section 7 gives some concluding remarks.

2 Basic Concepts

Our mobile agents are computational entities like other mobile agents. When each agent migrates, not only the code of the agent but also its state can be transferred to the destination. Furthermore, our mobile agent framework has the following unique concepts:

- **Agent Hierarchy:** Each mobile agent can be contained within one mobile agent.
- **Inter-agent Migration:** Each mobile agent can migrate between mobile agents as a whole with all its inner agents.

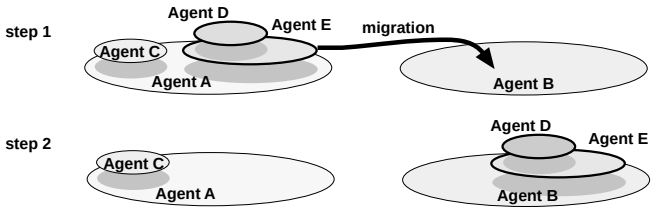

Figure 1: Agent Hierarchy and Inter-agent Migration

Mobile agents are organized in a tree structure and migrated as shown Figure 1. When an agent contains other agents, we call the former agent a *parent* the latter agents *children*. We call the agents which are nested by an agent, the *descendent* agents of the agent, and conversely we call the agents which are nesting an agent, the *ancestral* agents of the agent. Parent agents are responsible for providing their own services and resources to their children, and can directly access services and resources offered by their children.

Mobile Agents as Mobile Components

The first concept enables us to construct a mobile application by organizing more than one mobile agent, instead of constructing a large and monolithic mobile agent. The second concept allows a group of mobile agents to be treated as a single mobile agent. This is needed for the development of a mobile application, because a large-scale mobile application is often composed of a collection of subcomponents. Consequently, our mobile agents can be viewed as the mobile software components that have been studied in component-based software development technology [14].

Extensibility and Adaptability

Our concepts can make use of agent migration as a meta mechanism for changing and evolving a system consisting of one or more mobile agents. Each parent agent gives its own services and resources to its children. Therefore, when a mobile agent wants different services, the agent can acquire those services by migrating to the agent providing those services.

Also, our framework allows a system to be constructed as a collection of mobile agents. Such a system can customize its structure and its functions by migrating agents into it, while the system is running. In this paper, we try to construct a mobile agent system whose runtime system itself is implemented based on the framework. The system is extensible in the sense that it can dynamically change and adapt itself to its environment and the requirements of its executing mobile agents.

3 The MobileSpaces Mobile Agent System

This section presents a mobile agent system named *MobileSpaces*. The system can execute and migrate mobile agents that are incorporated with the framework presented in the previous section. Moreover, the architecture of the system is characterized in being based on the framework.

It is built on the Java virtual machine and mobile agents are given as Java objects [2]. The structure of the system is similar to a micro-kernel architecture as shown in several operating systems. That is, it consists of two parts: a core system and subcomponents as shown in Figure 2. The former offers only minimal and common functions independent of the underlying environment. The latter is introduced as a collection of subcomponents outside the core system and provides the other functions. All the subcomponents are implemented as mobile agents so that these subcomponents can be dynamically added to and removed from the system by migrating and replacing the corresponding agents.

3.1 The Core System

The core system is made as small as possible for the sake of portability and offers only the following minimal facilities: (1) agent hierarchy management, (2) agent execution management, and (3) serialization and deserialization of agents.

Agent Hierarchy Management

Each agent hierarchy is given as a tree structure in which each node contains a mobile agent and its attributes. Agent migration in an agent hierarchy is performed as merely a transformation of the tree structure of the hierarchy. Since each agent

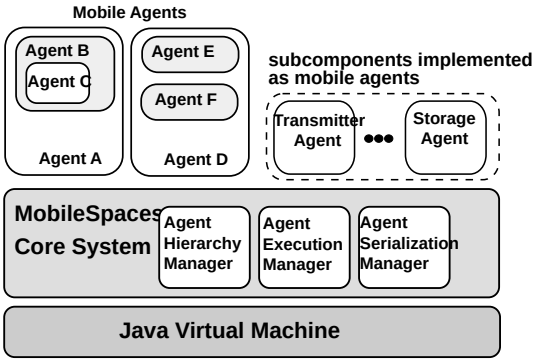

Figure 2: Architecture of MobileSpaces

hierarchy is basically maintained inside a computer, when an agent is moved in the same agent hierarchy, it and its descendent agents can still be running. Also, the core system corresponds to a stationary agent, called the *base agent*, at the root node of the tree structure. Consequently, agents can be viewed as the only constituent of our mobile agent system.

Each destination agent can judge whether it accepts a new visitor or not beforehand, whereas a visiting agent can know the available methods provided by the destination agent by using the class introspector mechanism of JDK 1.1. Also, an agent can be dynamically replaced by a new agent which is equipped with all the public methods supported by the original one, but the current implementation of our system does not allow the new agent to inherit any internal state of the old agent.

Agent Execution Management

The core system can control all the agents in its agent hierarchy, under the protection of the JDK1.1 security manager.

Each agent has direct control of its descendent agents. That is, an agent can instruct its descendent agents to move to other agents, serialize and destroy them. Moreover, each agent can directly invoke all the public methods of its descendent agents.[1]

In contrast, each agent has no direct control over its ancestral agents. Instead, each agent can have a collection of service methods which can be accessed by its children, instead of its descendant. A child agent can invoke the service methods provided by its parent under the control of the parent. In addition, each agent can access the service methods provided by its ancestral stationary agents, including the base agent.

Each agent can have one or more activities which are implemented by using the Java thread library. Furthermore, the core system maintains the life-cycle of agents: initialization, execution, suspension, and termination. When the life-cycle state of an agent is changed, the core system issues certain events to the agent and its descendent agents. The system can impose specified time constraints on all method invocations between agents in order to avoid being blocked forever.

It is worth mentioning why we have imposed the restriction that a mobile agent may not access any services supported by ancestral agents other than their parent and stationary agents. This restriction is a key idea for allowing successful migration to occur. If it were not imposed, then migrating an agent could mean that the descendants of that agent might suddenly find they could no longer access services upon which they relied.

Serialization and Deserialization of Agents

When an agent is transferred, it has to be marshaled into a bit-stream and then unmarshaled from it later. The core system provides a mechanism for marshaling and unmarshaling the states of agents. The reader may wonder why the mechanism is provided by the core system instead of subcomponents. This is because the core system has to check whether the serialized agent is valid or not in order to protect the whole system against invalid or malicious agents. The current implementation of our system uses the standard JAR file format for passing agents that can support digital signatures, allowing for authentication.

Our system uses the Java object serialization package for marshaling agents. The package does not support the capturing of stack frames of threads. Consequently, our system cannot serialize the execution states of any thread objects.[2] Instead, when an agent is serialized, the core system propagates certain events to its descendent agents in order to instruct the agent to stop its active threads, and then automatically stops and serializes them after a given time period.

3.2 Mobile Subcomponents

The core system supports only the functions independent of the underlying environment. Instead, the other functions are provided by subcomponents implemented as mobile agents outside of the core system. Also, agent migration is introduced as a basic mechanism for obtaining and changing functions provided by subcomponents. That is, when an agent wants to make use of a new function, the agent migrates into one of the agents which can offer the function.

Agent Migration between Computers

Agent migration between different computers is offered by subcomponents, called *transmitter* mobile agents, instead of

[1] The current implementation of MobileSpaces permits a parent agent to obtain references to the Java objects corresponding to its descendants.

[2] This limitation is not serious in the development of real mobile agent-based applications, as discussed in [13].

the core system. Transmitter agents are allocated on hosts. Each transmitter agent can exchange its inner agents with each other through its favorite communication protocol (as shown in Figure 3). When a mobile agent is preparing for a trip, the agent migrates itself into an appropriate transmitter agent.

The transmitter suspends the moving agent (including its nesting agents) and then serializes its state, classes, and destination address into a proper form for its communication protocol. Next, it transfers the serialized agent to a transmitter agent on the destination side. The transmitter agent receives the data and then reconstructs an agent (including its nesting agents) according to the data.

Each runtime system can be equipped with more than one transmitter agent in order to exchange agents through various communication protocols and networks. We have already implemented several transmitter agents which can transport their inner agents via several communication protocols such as TCP, UDP, and SMTP.

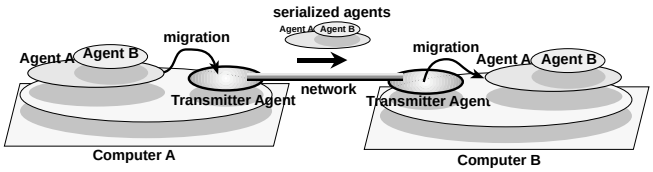

Figure 3: Transmitter Mobile Agents

Storage Service

Although the core system can serialize the states of agents into a bit-stream, the way to store and restore such a bit-stream in secondary storage is often dependent on the underlying system, such as operating system and hardware. Therefore, we introduce storage agents which can store their inner agents on secondary storages in their favorite ways. When an agent is to be stored onto a disk, the agent migrates into a storage agent corresponding to the disk. The storage agent serializes and stores the states and codes of its visiting agents as persistent data on the disk.

We have implemented a lot of mobile agent-based subcomponents for supporting various services for agents, such as agent termination, agent duplication, inter-agent communication, and resource management in addition to agent migration and storage.

3.3 Adaptability

Our system allows its functions to evolve and adapt to the execution environment and the requirements of visiting agents by migrating and changing mobile agent-based subcomponents for supporting the functions.

It is often argued that the advantage of agent migration lies in the reduction of communication costs in distributed computing settings. Although this argument is understandable, our system can make use of agent migration as a meta operation for mobile agents. When an agent wants a service, it can access the service by migrating itself to the agent which provides the service. The semantics and properties of an agent are partially provided by its parent agent and these can be changed by moving to other agents. In this sense, a parent agent can be viewed as a meta interpreter of its children. Agent migration in the agent hierarchy is accomplished with the `go()` command and each mobile is referenced by Uniform Resource Locators (URLs). The following statement requests an agent migration over the network.

```
go(new AgentURL("TCP-TRANSMITTER
    ://some.where.com/agent1/agent2"));
```

where `TCP-TRANSMITTER` denotes a transmitter agent which migrates its inner agents at TCP-based communication. When an agent performs the above command, the agent enters the transmitter agent to request that it is migrated into an agent which is a child agent, named `agent2`, of the `agent1` agent included in the base agent running on the host addressed as `some.where.com`.

In our framework, agent migration is introduced as a unified mechanism for operating mobile agents in addition to agent migration as follows:

```
go(new AgentURL("FILE:///agent.jar"));
```

where `FILE` denotes a storage agent which can store its visiting agents in a secondary storage. When an agent performs this command, the agent migrates into the storage agent in order to request for itself to be stored in a file, named `agent.jar`.

Our system allows a single service to be offered by more than one agent. Hence, each mobile agent can be provided its required service by one of the most suitable agents which can realize the service in the current execution environment.

To specify such suitable agents dynamically, our URLs can contain the form `$(variable)`, where `variable` denotes a variable name whose value is a string. These variables can be associated with environment variables provided by the shell program or the operating system as a dynamic URL (studied in [12, 15]). The current implementation of our system ensures that the underlying operating system can reflect changes in the values of its environment variables. We show an extended URL including variables as follows:

```
$(TRANSMITTER)://some.where.com/agent1/agent2
```

where `$(TRANSMITTER)` is a variable for specifying transmitter agents which can migrate their inner agents to remote computers. When the value of the `TRANSMITTER` environment variable is `TCP-TRANSMITTER`, which

specifies a transmitter agent, the above URL is interpreted as `TCP-TRANSMITTER://some.where.com/agent1/agent2`. When the network environment is changed, the operating system can detect the change and update the value of the `TRANSMITTER` environment variable so that the value refers to one of the most suitable agents in the current execution environment. Consequently, agent migration can be dynamically adapted to the current network environment.

4 Mobile Agents in the MobileSpaces System

In the MobileSpaces system, each agent consists of an agent program and an agent context implemented in Java language. The former defines the behavior of the agent and offers methods directly accessed by its parent agent. The latter defines methods indirectly accessed by its children, as shown in Figure 5. In addition, each agent has its own name based on the agent hierarchy. Every agent is equipped with a message queue for incoming messages and a message manager for communication among its child agents.

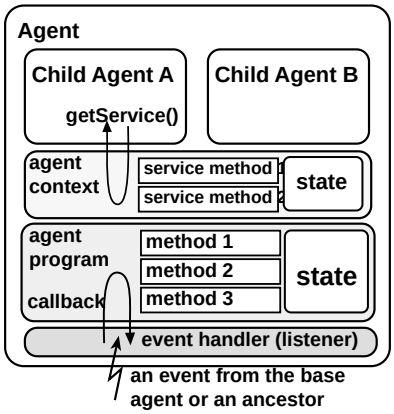

Figure 4: MobileSpaces Mobile Agent

4.1 Agent Program

Every agent program has to be an instance of a subclass of abstract class `Agent`. The `Agent` class consists of some fundamental methods used to control the mobility and the life-cycle of a mobile agent.

```
public class Agent extends MobileObject {
  // (un)registering a context for its children
  void addChildrenContext(Context context){ ... }
  void removeChildrenContext(Context con-
text){ ... }
  // registering listeners to hook certain events
  void addDefaultListener(
    DefaultEventListener listener){ ... }
  void removeDefaultListener(
    DefaultEventListener listener){ ... }
  // registering a name in an environment variable
  void register(String name, String value)
    throws ... { ... }
  // migrating the agent specified as url1 to
  // the target agent specified as url2
  void go(AgentURL url)
    throws NoSuchAgentException ... { ... }
  void go(AgentURL url1, AgentURL url2)
    throws NoSuchAgentException ... { ... }
  // asking its parent agent a message
  void getService(Message msg)
    throws NoSuchMethodException ... { ... }
  // issuing an event to an agent specified as url
  void dispatchEvent(AgentURL url, AgentEvent evt)
    throws ... { ... }
  ....
}
```

We explain some methods defined in the `Agent` class as follows:

- When an agent performs the `go(AgentURL url)` method, the agent migrates itself to the destination agent specified as `url`.
- A child agent cannot access any methods defined in its parent agent. Instead, each parent agent can be equipped with a context object which offers service methods in a subclass of the `Context` class, like the `AppletContext` of Java's Applet. These methods can be indirectly accessed by its children to get information about and interact with the environment such as their parent agent, their sibling agents, and the underlying computer system. Each child agent can invoke the public methods defined in the context of its parent agent by means of the `getService()` method defined in the `Agent` class. If any method corresponding to the called method is not in its parent agent, the presence of a corresponding method is tested at stationary agents, including the base agent, which are ancestral to the agent.
- The `dispatchEvent(AgentEvent evt)` method propagates an event specified as its argument to its descendants, whereas the `dispatchEvent(URL url, AgentEvent evt)` method issues an event to the agent specified as `url`.

Our system has an event mechanism based on the delegation-based event model introduced in the Abstract Window Toolkit of JDK 1.1 or later and thus each agent must be informed indications of such life-cycle state changes and can release various resources, such as files, windows, and sockets, which are captured by the agent.

To hook these events, each agent can have one or more listener objects. A listener object implements a specific listener

interface extended from the generic `AgentEventListener` interface. The `AgentEventListener` interface defines callback methods which should be invoked by the core system before or after the life-cycle state of the agent changes. One of the most basic listener interfaces, `DefaultEventListener` is shown as follows:

```
interface DefaultEventListener
  extends AgentEventListener {
  // invoked after creation at url
  void create(AgentURL url);
  // invoked before termination
  void destroy();
  // invoked after accepting a child
  void add(AgentURL child);
  // invoked before removing a child
  void remove(AgentURL child);
  // invoked after arriving at the destination
  void arrive(AgentURL dst);
  // invoked before moving to the destination
  void leave(AgentURL dst);
  ....
}
```

The above interface specifies fundamental methods invoked by the core system, when agents are created, destroyed, persisted, and migrated to another agent.

4.2 Examples

Transmitter Agents

Suppose an agent migrates between two computers by using transmitter agents as presented in the previous section. The following code fragment is the `SimpleTransmitter` class which defines a simple transmitter agent. Each `SimpleTransmitter` agent can exchange agents with each other via a given communication protocol.

```
public class SimpleTransmitter extends Agent
implements DefaultEventListener, Runnable {
  public SimpleTransmitter() {
    // registering itself as a listener
    addDefaultListener(this);
    // registrating a context for children
    addChildrenContext(new BaseContext);
    // naming itself as SIMPLE-TRANSMITTER in
    //   the TRANSMITTER environment variable
    register("SIMPLE-TRANSMITTER","TRANSMITTER");
  }
  public void create(AgentURL url) {
    // executing itself as an active program
    setAutonomous(true)
  }
  // invoked at having a visiting child agent
  public void add(AgentURL url) {
    // serializing the arrival agent
    Message msg = new Message("serialize");
    msg.setArg(url.getSource());
    byte[] data = (byte[])getService(msg);
    // sending the agent to its target host
    send_agent(data, url.getTarget();
  }
  public run() {
    while(true) {
      // receving the data
      receive_agent(byte[] data);
      // deserializing data as an agent
      Message msg = new Message("deserialize");
      msg.setArg(data);
      AgentURL url = (AgentURL)getService(msg);
      ...
    }
    ...
  }
  ...
}
```

When an agent in the agent hierarchy performs the following code:

```
go(new AgentURL("$(TRANSMITTER)
  ://some.where.com/agent1/agent2"));
```

where `$(TRANSMITTER)` is an environment variable for specifying transmitters. We now assume that the `SimpleTransmitter` agents are running in both computers and `$(TRANSMITTER)` variables are given as `TCP-TRANSMITTER` by the underlying system, such as operating system, which can monitor the execution environment.

The above code is interpreted as follows: after performing the `go()` method, the agent enters the transmitter agent specified as the value of the `$(TRANSMITTER)` variable, and asks the transmitter agent to migrate itself into the `/agent1/agent2` agent located at the computer addressed as `some.where.com`. Next, the `add(AgentURL url)` method defined in the `SimpleTransmitter` class is invoked with the identifier of the entering agent accompanied the original destination address. The `SimpleTransmitter` agent asks the base agent to serialize the migrating agent into a transmittable data and then it sends the data to the target computer `some.where.com`. On the receiver side, a `SimpleTransmitter` agent receives and then reconstructs the serialized agent according to the received data at the destination agent, i.e., `/agent1/agent2`.

Mobile Compound Documents

The above example is a just trivial program but there have been a lot of practical examples of the framework presented in this paper. One of the most illustrative examples of the framework is the implementation of the system itself, because its architecture and adaptability are based on the framework. Also, we have constructed various mobile agent-based applications running on the system, for example workflow management, CSCW, distributed information retrieval, active networks, and so on.

One of the most important examples is applications based on the concept of compound documents like OpenDoc developed by Apple Computer and IBM [1]. Our agent hierarchy allows compound documents given as mobile agents to

be dynamically composed into a compound document, while traditional mobile agents are isolated programs and thus cannot support any compound documents.

We have constructed an electronic mail system where each letter is a mobile agent incorporated with the framework presented in this paper. Therefore, each letter can contain more than one mobile agent-based component, i.e., text, graphics, and animation on the document of the letter as shown in Figure 5 and 6. Users can edit these inner components written in arbitrary data formats, because they are mobile agents and thus can include programs to edit their own contents. For example, to edit the text, simply click on it, and its editor program is invoked. The letter agent can autonomously deliver itself and its inner components to the destination. The receiver can read all the contents of the arriving letter, because the letter is a mobile agent that contains components for viewing the contents.

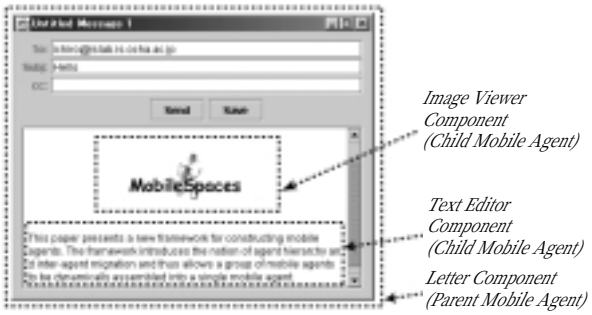

Figure 5: Window of the Compound Letter Agent

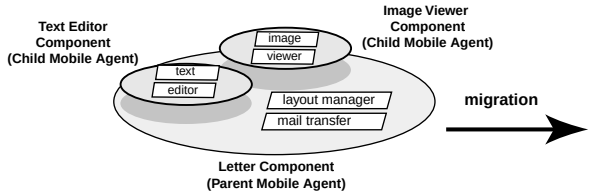

Figure 6: Structure of the Compound Letter Agent

5 Implementation and Performance

The MobileSpaces mobile agent system has been implemented in the Java language (JDK1.1 or later version). The core system is constructed independently of the underlying system and can run on any computer with a 1.1-compatible Java runtime. We have tried to keep the implementation within the framework as much as possible.[3]

[3]An implementation of the mobile agent system, including its examples is available from `http://islab.is.ocha.ac.jp/`.

Even though our implementation was not built for performance, we have performed a basic experiment of agent migration for two cases: agent migration in an agent hierarchy and agent migration between two computers (Pentium II-300 MHz with WindowsNT 4.0 and JDK 1.1.8) connected by 10BASE-T Ethernet. The moving agent is a simple implementation of the `DefaultEventListener` interface presented in the previous section.

Table 1: The time of agent migrations (msec)

	time
agent migration in an agent hierarchy	5
agent migration between two computers	30

The first result is the time of an agent migration in an agent hierarchy, and includes the cost to check whether the visiting agent is permitted to enter the destination agent or not. In the second experiment, agent migration is supported by transmitter agents allocated on two computers. Each transmitter agent can communicate with the other by using an application-level protocol for agent transmission whose mechanism is modeled on that of the HTTP protocol over TCP/IP communication. On the sender side, a transmitter agent serializes and transfers the codes and state of an agent (including its inner agents) to the transmitter on the receiver side and waits for an acknowledgment message. The marshaled agent consists of its serialized state, its codes, and its attributes such as name and capability, and is packed and compressed into a bit-stream which amounts to 1.5Kbytes. The second result is the sum of the marshaling, zip-based compression, opening TCP connection, transmission, security verifications, decompression, and unmarshaling.

6 Related Work

The notion of agent hierarchy presented in this paper is similar to a process calculus for modeling process migration called mobile ambients [5]. The calculus can formalize a mobile process including other mobile processes like ours, but it is just a theoretical framework. Therefore, to develop a practical implementation of the calculus, we must change its whole semantics.

Moreover, a lot of mobile agent systems have been released nowadays, for example see Aglets [9], MOA [10], Mole [13], Telescript [16], and Voyager [11]. To our knowledge, no existing mobile agent systems, including mobile object systems, are based on the concept of agent hierarchy proposed in this paper. Mole introduces the notion of agent groups in order to encourage coordination among mobile agents [3]. Mole's agent groups can consist of agents working together on a common task, but they are not mobile.

Also, Telescript and MOA introduce the concept of places in addition to mobile agents. Places are agents which can contain mobile agents and places inside them, but they are not mobile. Our mobile agent system, on the other hand, allows one or more mobile agents to be dynamically organized into a single mobile agent, and thus we do not have to distinguish between mobile agents and places. Therefore, a distributed application, in particular a mobile application, that is large in scale and complex can be easily constructed by combining more than one agent.

Our mobile agent system is characterized by its extensibility and adaptability in contrast to existing mobile agent systems that cannot extend and adapt their functions to their execution environments while they are running. In the literature on extensible operating systems and meta-level architecture, several researchers have explored frameworks to change the behavior of operating systems and applications according to their environments (for example, see [4, 6, 8, 17]). These systems can adapt themselves to their surrounding environments by means of special operations such as code migration or meta-level semantics. Most of them are not designed for constructing mobile applications and thus lack any mechanism for the migration of running applications. MROM [8] introduces the notion of meta-level semantics into mobile objects, but its mobile objects are essentially designed as isolated entities and thus cannot construct any large-scale mobile application as a compound mobile object. The Apertos operating system [17] introduces the notion of object migration as a meta mechanism like ours, but it needs to distinguish between meta-level semantics and base-level semantics. However, unlike the Apertos system, our system introduces agent migration as a single unified mechanism for processing agents and changing the system without explicitly introducing any meta-level semantics. As a result, it can naturally and easily adapt and extend its own functions by means of agent migration, which is one of the most essential mechanisms in mobile agent computing. Also, mobile agent-based applications running on the system can inherit the extensibility and adaptability of the system.

7 Conclusion

We presented a new framework for dynamically assembling a group of mobile agents into a single mobile agent. This framework makes two contributions. One of them is the providing of a powerful method to construct a mobile agent-based application that is large in scale and complicated. It is also useful in multi-agent technology because it can provide a general programmable framework for coordinating mobile agents. The other is the introduction of agent migration as a meta mechanism to dynamically evolve and extend a system consisting of one or more mobile agents. We have implemented a mobile agent system based on the framework. The system is unique in that it can dynamically change and evolve its functions by migrating agents that offer the functions, while other existing systems cannot do this. Also, mobile agent-based applications running on the system can enjoy the extensibility and adaptability of the system.

Finally, we would like to point out further issues to be resolved. Security is essential in mobile agent computing. The current implementation of the system relies on the JDK 1.1 security manager and provides a simple mechanism for authentication of agents. However, many security features are left open for the next release. Also, the programming interface of the current implementation is not yet satisfactory. We plan to design a more elegant and flexible interface.

References

[1] Apple Computer Inc., OpenDoc: White Paper, Apple Computer Inc., 1994.

[2] K. Arnold and J. Gosling, The Java Programming Language, Addison-Wesley, 1996.

[3] J. Baumann and N. Radounklis, Agent Groups in Mobile Agent Systems, Proceedings of Conference on Distributed Applications and Interoperable Systems, 1997.

[4] B. N. Bershad, et al, Extensibility, Safety and Performance in the SPIN Operating System, Proceedings of Symposium on Operating Systems Principles, 1995.

[5] L. Cardelli and A. D. Gordon, Mobile Ambients, Foundations of Software Science and Computational Structures, LNCS, Vol. 1378, pp. 140–155, 1998.

[6] D. R. Engler, M. F. Kaashoek, and J. O. Toole, Exokernel: An Operating System Architecture for Application-level Resource Management, Proceedings of Symposium on Operating Systems Principles, 1995.

[7] R. S. Gray, Agent Tcl: A Transportable Agent System, CIKM Workshop on Intelligent Information Agents, 1995.

[8] O. Holder and I. Ben Shaul, A Reflective Model for Mobile Software Objects, IEEE International Conference on Distributed Computing Systems, pp.339-346, 1997.

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

[10] D. S. Milojicic, W. LaForge, and D. Chauhan, Mobile Objects and Agents (MOA), Proceedings of USENIX Conference on Object Oriented Technologies and Systems, April 1998.

[11] ObjectSpace Inc, ObjectSpace Voyager Technical Overview, ObjectSpace, Inc. 1997.

[12] B. N. Schilit, and N. Adams, and R. Want, Customizing Mobile Application, Proceedings of Workshop on Mobile Computing Systems and Applications, IEEE Press, 1993.

[13] M. Strasser and J. Baumann, and F. Hole, Mole: A Java Based Mobile Agent System, Proceedings of ECOOP Workshop on Mobile Objects, 1996.

[14] C.Szyperski, Component Software, Addison-Wesley, 1998.

[15] G. Voelker, and B. Bershad, *Mobisaic: An Information System for a Mobile Wireless Computing Environment*, Proceedings of Workshop on Mobile Computing Systems and Applications, IEEE Press, 1994.

[16] J. E. White, Telescript Technology: Mobile Agents, General Magic, 1995.

[17] Y. Yokote, The Apertos Reflective Operating System: The Concept and its Implementation, Proceedings of OOPSLA'92, pp. 414–434, 1992.