# Organization and Mobility in Mobile Agent Computing

Ichiro Satoh

National Institute of Informatics,
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
ichiro@nii.ac.jp

**Abstract.** A mobile agent system for organizing multiple mobile agents is presented. It provides two unique two mechanisms for dynamically organizing mobile agents, which may be running on single or multiple computers. The first enables a mobile agent to contain other mobile agents inside it and migrate to another mobile agent or computer with its inner agents. It provides an approach to composing large-scale mobile software from a collection of mobile agents and using mobile agents as deployable software components. The second enables a mobile agent to be deployed at computers according to the movements of other mobile agents. It can move a federation of agents running on different computers, over a distributed system. It can build and aggregate distributed applications from one or more mobile components that can be dynamically deployed at mobile or stationary computers during the execution of the application. This paper also presents a prototype implementation of the system and its application.

## 1 Introduction

Distributed computing systems are composed of a number of software components running on different computers and interacting with one another via a network. The complexity of modern distributed systems impairs our ability to deploy components at appropriate computers using traditional approaches, such as those that are centralized and top-down Moreover, the requirements of applications in a distributed system tend to vary and change dynamically. Applications must adapt to such changes. Software components, which an application consists of, need to be adapted and deployed at computers in a distributed system according to changes in the requirements of the applications and the structure and computational resources of the system. Mobile agents can provide a solution to this problem, because they are autotomous programs that can travel from computer to computer in a network, at times and to places of their own choosing. Unfortunately, existing mobile agent systems lack the mechanisms for structurally assembling and relocating multiple mobile agents, which may run on different computers.

To solve this problem, a few attempts to organize mobile agents have been proposed, e.g., MobileSpaces [11], CLAIM [5], and FarGo [7]. MobileSpaces and CLAIM enable each mobile agent to be organized within a tree structure and to migrate to other mobile agents, which may run on different computers, with its inner agents. FarGo [7] introduces the notion of a dynamic layout for distributed applications. It explicitly binds more than one mobile agent to a single mobile agent and, when the latter migrates to another location, it relocates the latter agent at the same destination to follow the former agent. This paper proposes a framework for structurally and dynamically federating multiple mobile agent-based components running on either the same computer or different computers. The framework makes two contributions to distributed systems. The first enables large-scale mobile software to be composed from a collection of mobile agents and the second enables a mobile agent to be deployed at computers according to the movements of other mobile agents in a self-organizing manner. The system provides a general test-bed for bio-inspired approaches over real distributed systems.

In this paper, we describe our design goals (Section 2), the design of our framework, and a prototype implementation (Section 3). We outline programs in the system and applications running on it (Section 4), and explain the current status of the implementation (Section 5). We also describe our experience with the framework (Section 6). We then briefly review related work (Section 7), provide a summary, and discuss some future issues (Section 8).

## 2 Approach

Mobile agents within this framework are computational entities like other mobile agents. When an agent migrates, not only its code but also its state can be transferred to the destination.[1]

### 2.1 Mobile Agent Composition

Our framework enabled us to construct a distributed computing system as a federation of mobile agent-based software components running on the same or different computers. It provides two approaches for composing mobile agents.

**Strong Composition.** The framework enables a large-scale mobile agent to be organized within a tree structure according to the following notions.

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

When an agent contains other agents, we call the former agent a *parent* and the latter agents *children*. Agents that nested by an agent are called *descendent* agents of the agent, and conversely agents that are nesting an agent are called *ancestral* agents of the agent. Parent agents are responsible for providing their own services and resources to their children, and can directly access the services and resources offered by their children. These concepts themselves were discussed in our previous paper [11].

**Weak Composition.** The framework builds partitioned applications as mobile agent-based software components, enabling them to run on different computers and move to other computers while running. The movement of one agent may affect other agents. For example, two components may be required to be on the same computer when the first is a program that controls the keyboard and the second is a program that displays

[1] The framework treats the mobile code approach as a subset of the mobile agent approach.

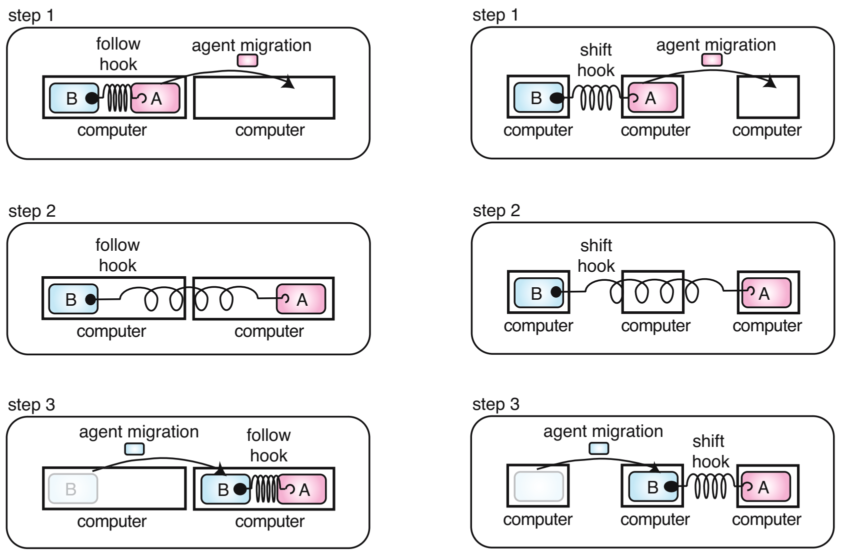


**Fig. 1.** Component migration with relocation policy

content on the screen. The framework enables each agent to explicitly specify a policy, called a *hook*, for agent migration. The current implementation provides two types of hooks, as shown in Fig. 1. The first enables an agent to follow another component and the second enables an agent to migrate to the source location of another agent.

## 2.2 Prototype-Based Agent-Creation

Object-oriented languages, which most existing mobile agents are defined in, are classified into two types: class-based and prototype-based. About twenty years ago, researchers discussed the advantages and disadvantages of these two languages.[2] Although the former has swept over almost the entire-field of object orientation, the latter still has several distinct advantages. Existing mobile agents are defined with class-based oriented languages, e.g., Java. Nevertheless, mobile agents can also be viewed as prototype-based objects. When a mobile agent migrates to another computer, the state of the agent's running program is marshaled into data and is then transmitted to the destination as passive data with its program code. Mobile agents can easily and naturally make replicas of themselves by duplicating their marshaled agents. As a result, mobile agents can be created by cloning existing agents as well as instantiating them from classes to define their behaviors. Mobile agents, on the other hand, cannot control the process of cloning themselves at program-level, because their cloning mechanisms are supported by their runtime systems or libraries, instead of their programs. Cloning

[2] There have been numerous discussions on the notions of prototype-based paradigms and delegation. However, we do not intend to discuss the definitions of these notions again. We will only introduce the notions as an approach to programming mobile agents.

facilities for object-creation provided by prototype-based languages are useful in enabling mobile agents to customize their cloning.

Class-based languages provide the notion of inheritance as a mechanism for sharing the behavior of objects, whereas prototype-based languages provide the notion of delegation-sharing both the behavior and state of objects. Although existing mobile agents have no mechanisms corresponding to the notion of delegation, it makes agents extensible. For example, each mobile agent is defined by classes that already know everything about the agent so that it cannot adapt its behavior to changes in its requirements or its execution environment. This problem is solved by allowing agents to be created by extending or sharing other agents.

## 3 Design and Implementation

This framework consists of two parts: runtime systems and mobile agents. It was implemented with Java language and operated on the Java virtual machine. We tried to contain the implementation within the framework as much as possible.

### 3.1 Runtime Systems for Hierarchical Mobile Agents

Each runtime system runs on a computer and executes and migrates mobile agents. Each also establishes at most one TCP connection with each of its neighboring systems and exchanges control messages, agents, and inter-component communications with these through the connection. Fig. 2 outlines the basic structure of a runtime system, which is similar to the micro-kernel architecture in several operating systems. That is, the system itself only offers minimal functions and other functions are implemented in mobile agents running on the system.

**Agent Hierarchy Management.** Each runtime system manages an agent hierarchy as a tree structure in which each node contains a mobile agent and its attributes. Also, each runtime system corresponds to the root node in its own tree structure. This framework

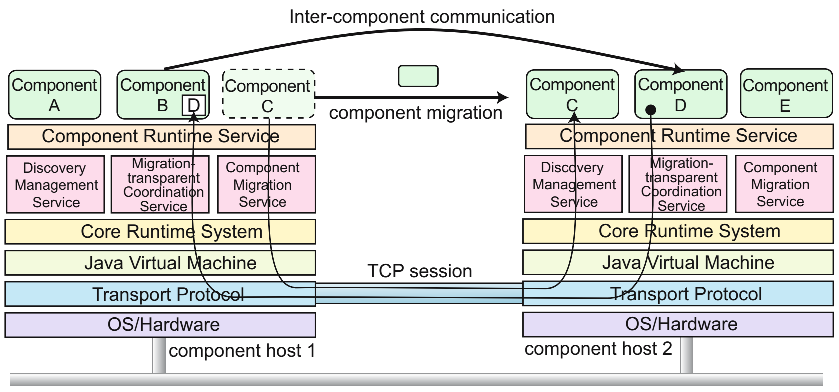


**Fig. 2.** Architecture for runtime system

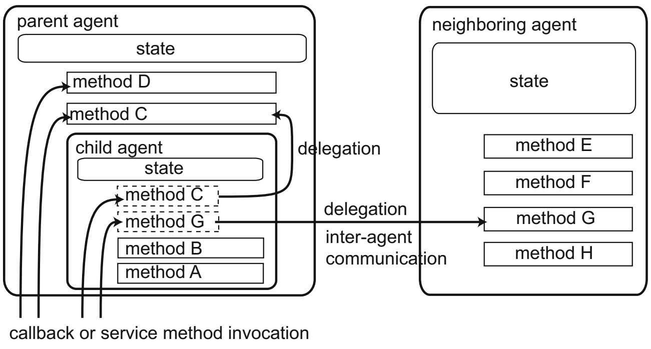


**Fig. 3.** Delegation between mobile agents

assumes that each agent is active but subordinate to its container agent. Therefore, each agent has direct control of its descendent agents. That is, an agent can instruct its descendent agents to move to other agents, and serialize and destroy them. No agent has direct control over its ancestral agents.

**Agent Execution Management.** The runtime system can control all agents in its agent hierarchy, under the protection of Java's security manager. Each agent can have one or more activities, which are implemented by using the Java thread library. Furthermore, the runtime system maintains the life-cycle of agents: initialization, execution, suspension, and termination. When the life-cycle state of an agent is changed, the runtime system issues certain events to the agent and its descendent agents. The system can impose specified time constraints on all method invocations between agents to avoid being blocked forever.

**Agent Delegation Management.** Agent hierarchy not only defines the structure of mobile agents but also their functions. Each agent can explicitly provide a set of service methods, which can be accessed by its children, instead of descendent agents. That is, a child agent can share the behavior and state of its parent agent like the notion of delegation in prototype-based languages. Fig. 3 has an example of delegation between mobile agents. The parent agent provides a method, called `writeOnDisk`, to save data in secondary storage but the child agent has no methods of saving its state. However, the child can access the parent's method to save its state in storage. As a result, the semantics and properties of an agent are partially provided by its parent agent. It is worth mentioning why we imposed the restriction that a mobile agent could not access any services supported by ancestral agents other than their parent and stationary agents. This restriction is the key idea in 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 delegate services upon which they relied.

**Agent Migration.** When an agent is moved inside a computer, the agent and its inner agents can still be running. When an agent is transferred over a network, the runtime system stores the state and the codes of the agent, including the agents embedded in it, into a bit-stream formed in Java's JAR file format, which can support digital signatures for authentication. The system provides a built-in mechanism for transmitting the bit-stream over the network by using an extension of the HTTP protocol.[3] The current system basically uses the Java object serialization package for marshaling agents. The package does not support the capturing of stack frames of threads. Instead, when an agent is saved or migrated, the runtime system issues events to it and all its descendent agents to invoke their specified methods, which should be executed before their migration, and then suspends their active threads and migrates them to the destination.

**Agent Cloning.** Mobile agents are often created as self-contained entities in other existing mobile agent systems, whereas this framework allows each mobile agent to share the behaviors of its ancestral agents. Therefore, if a clone had been created from an agent, which relies on the services of its parent, it could no longer access these services. Object-oriented languages, on the other hand, provide two mechanisms for cloning objects: shallow-copy and deep-copy. The former creates a clone of the state of an agent and shares the behaviors with the agent from which it was cloned by means of delegation. The latter creates a clone of both the state and behavior of the agent. Each runtime system provides two approaches corresponding to the two mechanisms.

- The first creates a clone of only the target agent and provides the forwarder agents of its ancestral agents to the clone so that the clone agent can access the services provided by the ancestral agents.
- The second not only creates a clone of the agent but also the clones of ancestral agents whose services the agent may access.

The current implementation does not support a mechanism for analyzing which ancestral agents the clone shares services with. Therefore, when an agent creates a clone, it specifies which ancestral agents should be cloned with it. Since the framework assumes that a component and its clone are independent, it does not support any mechanism for sharing their updating states with them.

## 3.2 Mobile Agent Model

Each agent in the current implementation of the framework is a collection of Java objects in the standard JAR file format.

**Interagent Communication.** Each agent can offer a meeting place for its inner agents. It initially supports basic types of inter-agent communication, e.g., asynchronous one-way message passing, synchronous method call, and future communication. However, runtime systems do not offer mechanisms for communicating between agents, which may be contained in different parents, in a computer. Instead, the system provides two agents, called forwarder agents, to support inter-agent communications between agents contained in different agents. Each agent has its own proxy agent, called a *forwarder*

[3] Section 5 describes how the system enables agent migration protocols to be implemented in mobile agents.

agent. When it receives messages, it automatically redirects these to its or their specified destinations. An agent permits other agents to communicate with it and it thus deploys its forwarder agents at their parent agents. As a result, the agents can send messages to the agent via its forwarder agent. When they want replies from the agent, they deploy their forwarder agents at it via its forwarder agent.

Forwarder agents are used for tracking the current locations of moving agents. When an agent wants to interact with another agent, it must know where the target agent is currently located. Immediately before an agent moves into another agent, it creates and leaves a forwarder agent behind. The forwarder agent inherits the name of the moving agent and transfers the visiting agent to the new location of the moving agent. Therefore, when an agent wants to migrate itself to or send a message to another agent that has moved elsewhere, it can migrate into the forwarder agent instead of the target agent. The forwarder agent then automatically transfers the visiting agent or message to the current location of the target agent.

Several schemes for efficiently forwarding messages or agents to and locating moving agents have been explored in the field of process/object migration in distributed operating systems. Such forwarder agents can easily support most of these schemes because they are programmable entities that can flexibly negotiate with one another. Moreover, since forwarder agents are still mobile agents, they can be dynamically deployed at remote computers.

**Agent Relocation.** When multiple mobile agents coordinate with one another, if one of them migrates to another computer, the others may be required to migrate to other computers. This framework provides a mechanism for enabling mobile agents to be dynamically deployed at computers according to the movement of other agents. The mechanism itself provides *carrier* agents, which convey their inner agents over a network. It enables each carrier agent to specify at most one target container agent. The former agent also carries its inner agents to a suitable computer according to its own policy, when the latter agent carries its inner agents to another computer or agent. We assumed that a carrier agent would have a policy for another carrier agent. That is, when the former agent migrated to another agent or computer, the framework would provide carriers agents based on several basic policies. For example, the latter (or its clone) migrates to the former's destination through the *follow* policy (or *dispatch* policy), and the latter (or its clone) would migrate to the former's source through the *shift* policy (or *fill* policy).

Each carrier agent can contain at most one mobile agent. It can inherit its inner agent and forward its received messages and visiting agent to this inner agent. Therefore, it can be viewed as its inner agent by external agents, which interact with this inner agent, and it can explicitly restrict the mobility of the inner agent. The carrier agent carries the inner agent according to its own policy. Although each carrier agent can have at most one policy, agents can be contained in one or more carrier agents.[4] We can easily define more advanced or complicated policies by combining these policies.

Since carrier agents are just programmable entities, we can easily customize their policies. The current implementation assumes that the carrier agents comprising a group

[4] When carrier agents are nested, a parent carrier agent's policy proceeds to its descendent carrier agents' policy.

will be deployed to computers within a localized space smaller than the domain of a sub-network for UDP multicasting. Therefore, the deployment of carrier agents is managed by exchanging control messages through UDP-multicasting. When a carrier agent migrates to another computer, the destination computers ask the source computer (or the previous source computer) to multicast a query message about carrier agents whose policies contain the moving carrier agent.

## 3.3 Mobile Agent Programming Model

Each agent is defined as a collection of Java objects. It has its own name based on the agent hierarchy and a message queue for incoming messages. It has to be an instance of the `Agent` interface, the `ContainerAgent` interface, the `DuplicatableAgent` interface, and/or the `MobileAgent` interface. The first defines the callback methods of a stationary agent, the second defines the callback methods of a container agent, the third defines the callback methods of a duplicatable agent, and the fourth defines the callback methods of a mobile agent. The callback methods are invoked by the runtime system when the life-cycle of a mobile agent changes. Parts of these interfaces are as follows:

```
public interface Agent {
  public void create(AgentEvent evt, Context ctxt);
  public void destroy(AgentEvent evt, Context ctxt);
}
```

where `create()` and `destroy()` are invoked after the agent is created and before it is terminated. This framework uses interfaces for agents as declarations about their basic functions, e.g., mobility and duplicatability.

```
public interface ContainerAgent extends Agent {
  public void add(AgentEvent evt, Context ctxt);
  public void remove(AgentEvent evt, Context ctxt);
}
```

The above program is the definition of the `ContainerAgent` interface, where `add()` is invoked after the agent has received another agent and `remove()` is invoked after it has sent the visiting agent.

```
public interface DuplicatableAgent extends Serializable, Agent {
  public void duplicate(AgentEvent evt, Context ctxt);
  public void parent(AgentEvent evt, Context ctxt);
  public void child(AgentEvent evt, Context ctxt);
}
```

Each agent must implement the above interface so that it can be cloned. `duplicate()` is invoked before the agent is duplicated. `parent()` is invoked at the original agent and `child()` is invoked at a clone of the agent after it is duplicated.

```
public interface MobileAgent extends Serializable, Agent {
  public void arrive(AgentEvent evt, Context ctxt);
  public void leave(AgentEvent evt, Context ctxt);
}
```

Each mobile agent must be an instance of the above interface. `arrive()` is invoked before the agent has migrated to another location `leave()` is invoked after it has migrated. The `AgentEvent` class in these programs defines information about the agent, e.g., its current location, source, and destination. The `Context` class defines service methods for agents as follows:

```
class Context {
  void go(URL url) throws NoSuchHostException { ... }
  void go(URL url, String methodName) throws NoSuchHostException,
    throws NoSuchMethodException { ... }
  AgentID shallowCopy() throws IllegalAccessException { ... }
  AgentID deepCopy(AgentID aid) throws IllegalAccessException { ... }
  ServiceID getService(Message msg)} throws NoServiceException { ... }
  Object execService(ServiceID sid)} throws IllegalAcceessException { ... }
  setPolicy(AgentProfile cref, MigrationPolicy mpolicy) { ... }
  setTTL(int lifespan) { ... }
  .....
}
```

We will now explain the main methods defined in the `Context` class.

– When an agent performs `go(url, methodName`, it migrates itself to the destination agent specified as the `url` and executes the method specified in the second argument. This url specifies the destination agent for agent migration based on the containment relationships of an agent hierarchy on a local or remote computer as follows:

  ```
  MATP://some.where.com/application-name/function-name
  ```

  where `MATP` specifies the protocol for agent migration.
– By invoking `shallowCopy()`, an agent creates a clone of itself, including its code and instance variables and its inner agents. When an agent invokes `deepCopy()` with the identifier of its ancestral agent, it creates a clone of its code and instance variables and its inner agents, and clones of the specified ancestral agent and the agents that are contained in the ancestral agent.
– An agent can access service methods provided by its parent agent by invoking `getService()` with an instance of the `Message` class, which can specify the kind of message, arbitrary objects as arguments, and the deadline for timeout exceptions.
– The framework provides APIs for invoking the methods of other agents. Our programming interface for method invocation is similar to CORBA's dynamic invocation interface and does not have to statically define any stub or skeleton interfaces through a precompiler approach because distributed computing environments are dynamic.
– The `setTTL()` specifies the life span, called time-to-live (TTL), of the agent. The span decrements TTL over time. When the TTL of an agent reaches zero, the agent automatically removes itself.

While each agent is running, it can declare at most one deployment policy and one or more message policies by invoking `setPolicy` of the `Context` class. Although policies are open for developers to define their own policies, the current implementation provides the following deployment policies.

- If an agent declares a *follow* policy for another agent, when the latter migrates to another computer, the former migrates to the latter's destination computer.
- If an agent declares a *dispatch* policy for another agent, when the latter migrates to another computer, a copy of the former is created and deployed at the latter's destination computer.
- If an agent declares a *shift* policy for another agent, when the latter migrates to another computer, the former migrates to the latter's source computer.
- If an agent declares a *fill* policy for another agent, when the latter migrates to another computer, a copy of the former is created and deployed at the latter's source computer.

When an agent is created, the dispatch and fill policies can explicitly control whether the newly created agent can inherit the state of its original agent. The following message policies forward messages to agents when messages are specified in the policies.

- If an agent declares a *forward* policy for another agent, when specified messages are sent to other agents, the messages are forwarded to the latter as well as the former.
- If an agent declares a *delegate* policy for another agent, when specified messages are send to the former, the messages are forwarded to the latter but not to the former.

Fig. 4 outlines four deployment policies, which are related to phenomena in biological processes. For example, a `follow` policy enables an agent to approach another agent. For example, when multiple agents declare a policy for a leader agent, they can swarm around it. A *shift* policy enables an agent to follow the movement of another agent. The former agent can track the latter as it moves. The policy thus corresponds to the

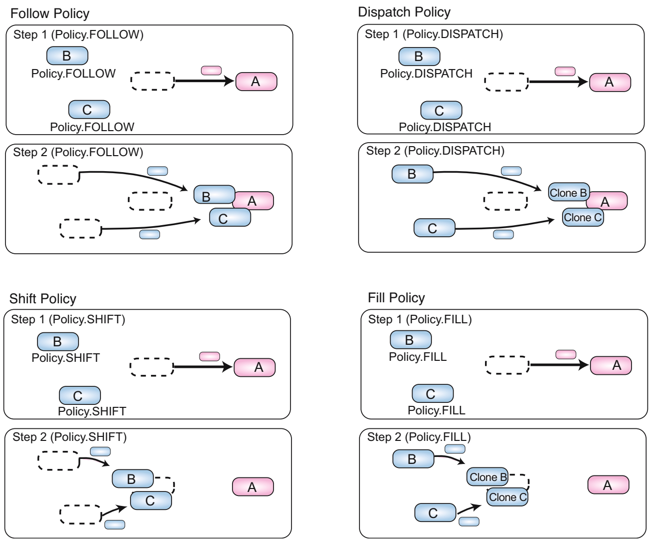
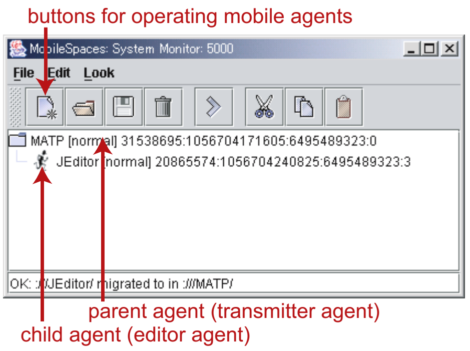


**Fig. 4.** Basic migration policies


buttons for operating mobile agents
MobileSpaces: System Monitor: 5000
File
Edit
Look
parent agent (transmitter agent)
child agent (editor agent)

```
MATP [normal] 31538695:1056704171605:6495489323:0
  JEditor [normal] 20865574:1056704240825:6495489323:3

OK: ://JEditor/ migrated to in :///MATP/
```

**Fig. 5.** Control window for runtime system

phenomenon of cytoplasmic streaming. A `dispatch)` policy enables an agent to stay in the current location and then deploy its clone at the destination of another moving agent. It can model the footprint of a motile cell. We have assumed that an agent can declare the policy for another agent and specify the TTLs of its clones as their life-spans. As the latter agent moves, cloned former agents are deployed at its footmark and these clones are automatically volatilized after their life-spans are over. Therefore, the clone agents can be viewed as a pheromone that is left behind after the latter agent has moved on. A *fill* policy corresponds to the phenomenon of cell division. The framework is open to define policies as long as they are subclasses of the `MigrationPolicy` so that we can easily define new policies, including bio-inspired ones. A *forward* policy is useful when two agents share the same information and a *delegate* policy provides a master-slave relation between agents.

## 4 Current Status

A prototype implementation of this framework was constructed with Sun's Java Developer Kit version 1.4. The implementation provided graphical user interfaces for operating the mobile agents shown in Fig. 5. These interfaces allowed us to easily load and migrate mobile agents via full drag-and-drop operations.

**Basic Performance.** Although the current implementation was not constructed for performance, we evaluated that of several basic operations in a distributed system where eight computers (Pentium-M 1.4 MHz with Windows XP Professional and J2SE 1.4.2) were connected through a fast ethernet. The cost of agent migration in an agent hierarchy was measured to be 4 ms, including the cost of checking whether the visiting agent was permitted to enter the destination agent or not. The cost of agent migration between agents allocated on two computers was measured to be 30 ms. The moving agent was simple and consisted of basic callback methods and contained two child agents. Its data capacity was about 7 Kbytes (zip-compressed). The cost of agent migration included that of opening TCP-transmission, marshaling the agents, migrating the agents from their source hosts to their destination hosts, unmarshaling the agents, and verifying security.

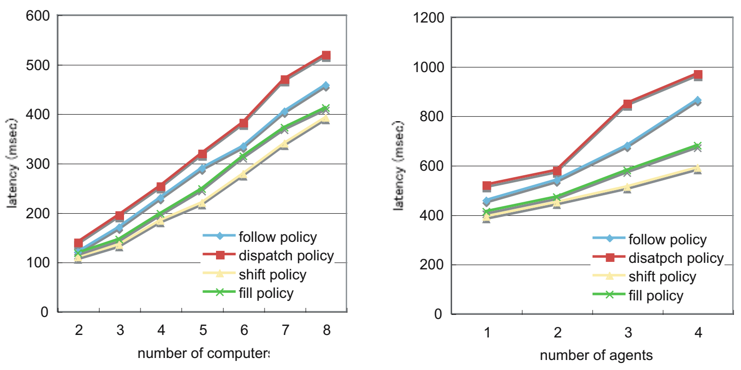


**Fig. 6.** Cost of multiple-hops for two agents between two to eight computers (left) and cost of multiple-hops for multiple agents between eight computers (right)

The left of Fig. 6 illustrates the cost of multiple-hops for two agents between two to eight computers, where the first agent declares a follow, dispatch, shift, or fill policy for the second and the second migrates between these computers sequentially without synchronizing the migration of the first. The latency between two computers is measured as the half-time of the round-trip time between the source and destination computers. To accurately measure the latency between more than three computers, these computers were connected through a ring topology. That is, the start and and goal of the second agent are assigned to the same computer and we measured the difference between the timing for the first agent to start and the second to arrive at the computer. Each cost at the left of Fig. 6 is the latency for the first agent arriving after the second has begun to migrate to another computer. The cost of agent migration according the dispatch (or fill) policy is larger than that of the the follow (or shift) policy, because the former needs to create a copy of the first agent that has the policy. The cost of agent migration according to follow (or dispatch) is larger than that for dispatch (or shift), because the former and latter agents are deployed at different computers.

The right of Fig. 6 shows the cost of multiple-hops for multiple agents between eight computers, when agents (from one to four) have follow, dispatch, shift, or fill policies for a moving agent. Unfortunately, the cost with many hops is large because the follow and dispatch policies vary due to congestion at several computers. That is, two or more agents may attempt to have their own active threads in a single processor and to simultaneously transmit themselves to the destinations of their target agent in a TCP network connection. Once agents experience congestion at a computer, they tend to migrate as a chunk to further destinations rather than as individuals and this often engulfs other newly arrival agents. Congestion does not always reappear, since computers are not synchronized and congestion often causes more congestion in agent routes. We expect that there will be large fluctuations in the cost of agent migration in large-scale, heterogenous, distributed systems.

**Security.** The current implementation can encrypt agents before migrating them over the network and can then decrypt them after they arrive at their destinations. Moreover, since each agent is simply a programmable entity, it can explicitly encrypt its particular fields and migrate itself with these fields and its own cryptographic procedure. The Java virtual machine could explicitly restrict agents so that they can only access specified resources to protect hosts from malicious agents. Although the current implementation cannot protect agents from malicious hosts, the runtime system supports authentication mechanisms for agent migration so that each agent host can only send agents to, and only receive from, trusted hosts.

## 5 Initial Experience

This section presents several example applications that illustrate how the framework works.

### 5.1 Point-to-Point Channels for Agent Migration

The first example is mobile agent-based active networking for mobile agents. It enables point-to-point agent migration to be provided by mobile agents, called *transmitters*. Transmitter agents correspond to a data-link layer or a network layer and are responsible for establishing point-to-point channels for agent migration between the source host and destination host through a (single-hop or multiple-hops) data transmission infrastructure, such as TCP/IP, as shown in Fig. 7. They abstract away the variety in the underlying network infrastructure and exchange their inner agents with coexisting agents running at remote computers through their favorite communication protocols. Furthermore, transmitter agents are implemented as mobile agents so that they can be dynamically added to and removed from the system by migrating and replacing corresponding agents, enabling them to keep up with changes in the network environment. After an agent arrives at a transmitter agent from the upper layer, the arriving agent indicates its final destination. The transmitter suspends the arriving agent (including its inner agents), then requests the core system to serialize the state and code of the arriving agent. It next sends the serialized agent to a coexisting transmitter agent located at the destination. The transmitter agent at the destination receives the data and then reconstructs the agent (including its inner agents) and migrates it to the destination or to specified agents that offer upper-layer protocols.

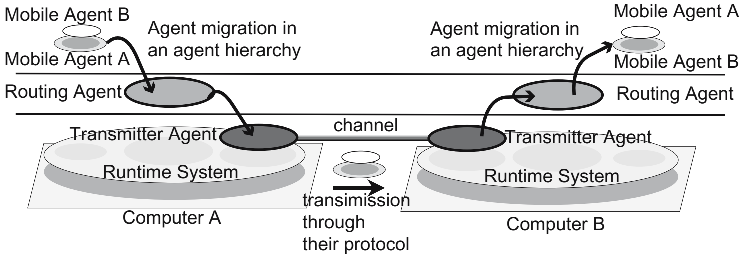
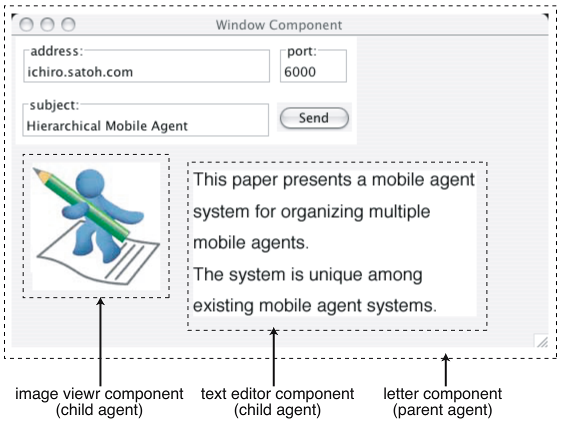


**Fig. 7.** Transmitter mobile agents for establishing channels between nodes

Several transmitter agents have already been implemented based on data communication protocols widely used on the Internet, such as TCP, HTTP, and SMTP. The authentication services normally available in a secure communications infrastructure include this functionality. Therefore, our secure transmitter agents, which can exchange agents, are implemented with a secure socket layer (SSL), which is one of the most popular secure communication protocols on the Internet. A virtual class is provided in Java that can be specialized to create transmitter agents for various protocols. Therefore, point-to-point channels can easily be implemented based on other secure communication protocols for data transmission.

## 5.2 Autonomic Electronic-Mail System

The second example is an electronic mail system based on the framework, consisting of two main components: an inbox document and letter documents (Fig. 8). The inbox document provides a window component that can contain two components. The first of these is the history of received mail and the second offers a visual space for displaying content selected from the history. A letter document corresponds to a letter. Since it is implemented as a compound document, it can contain various components for accessing text, graphics, and animation, in addition to a mobility-control component that defines an itinerary for more than one destination. It also has a window for displaying its content. It can migrate itself to its destination, but it is not a complete GUI application because it cannot display its content without the collaboration of its container, i.e., the inbox document. For example, to edit the text in a letter component, one simply clicks on it, and an editor program is invoked by the in-place editing mechanism of the framework. The component can deliver itself and its inner components to an inbox document at the receiver. After a moving letter has been accepted by the inbox document, if a letter in the list of received mail is clicked, the selected letter creates a frame object of itself and requests the document to display the frame object within its frame. Since the inbox document is the root of the letter component, when the document is stored


Window Component
Send
image viewr component (child agent)
text editor component (child agent)
letter component (parent agent)
address: ichiro.satoh.com
port: 6000
subject: Hierarchical Mobile Agent

This paper presents a mobile agent system for organizing multiple mobile agents.
The system is unique among existing mobile agent systems.

**Fig. 8.** Window for Compound Letter Agent

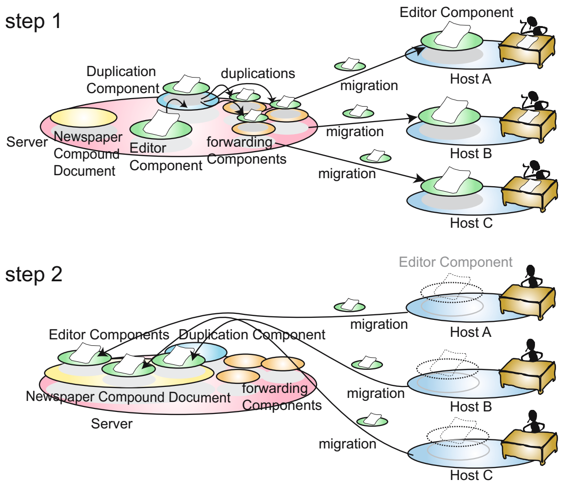


**Fig. 9.** Newsletter editing system

and moved, all the components embedded in the document are stored and moved with the document.

## 5.3 Application-Specific Document Distribution

One of the most illustrative examples of the framework presented in this paper is in providing documents to workflow management systems. A second example is an editing system for an in-house newsletter. Each newsletter is edited by automatically compiling one or more text parts, which are written by different people, as can be seen in Fig. 9. A newsletter is implemented as a compound document that can contain the text component inside it and each text part is a mobile agent including a viewer/editor program and its own text data. When the newsletter is being edited, each text part moves from the document to the computer on which it was written, and displays a window for its editor program on the computer's desktop to assist the writer, as shown in Fig. 9. Each editor goes back to the original document after the writer has finished writing it and then the document arranges the arriving components as a bound set. The document is still a mobile agent and can thus be easily duplicated and distributed to multiple locations.

## 5.4 Ant-Based Routing Mechanisms

Ants are able to locate a path to a food source using trails of chemical substances called pheromones deposited by other ants. Several researchers have attempted to use the notion of ant pheromones for network-routing mechanisms [1,4]. Our framework allows moving components to leave traces on trails that are automatically volatilized after their life-spans are over. A mobile agent corresponding to an ant corresponding to a pheromone is attached to another mobile agent corresponding to another ant based on the fill policy. When the latter agent randomly selects its destination and migrates there, the former agent creates a clone and migrates to the source host of the latter. Since each of the cloned agents defines its life-span by invoking `setTTL()`, they are active for a specified duration after being created. If there are other agents corresponding to pheromones in the host, the visiting agent adds their time spans to its own time span. When another agent corresponding to another ant migrates over the network, it can select a host that has agents corresponding to pheromones whose time-spans are the longest from the neighboring hosts. We experimented on ant-based routing for mobile agents using this prototype implementation and eight hosts. However, we knew that it would be difficult to quickly converge a short-path to the destination in real systems, because the routing mechanisms tend to diverge.[5]

## 5.5 Component Diffusion in Sensor Networks

The last example is the speculative deployment of components based on changes in the physical world. A mechanism is provided that dynamically and speculatively deploys components at sensor nodes when there are environmental changes. It was assumed that the sensor field was a two-dimensional surface composed of sensor nodes that monitor environmental changes, such as motion in objects and variations in temperature. It is well known that after a sensor node detects environmental changes within its coverage area, geographically neighboring nodes tend to detect similar changes after a short time. This diffusion occurs as follows in our framework. When a component on a sensor node detects changes in its environment, the component duplicates itself and deploys the clone at neighboring nodes as long as the nodes have the same kinds of components (Fig. 10). Each component is associated with a resource limit that functions as a generalized time-to-live field. Although a node can monitor changes in environments, it sets the TTLs of its components to their own initial value. It otherwise decrements TTLs over time. When the TTL of a component reaches zero, the component automatically removes itself. This example is still in the early stages of experimentation but we have developed mobile agent-based middleware for sensor networks [17] and plan to extend this framework to the middleware.

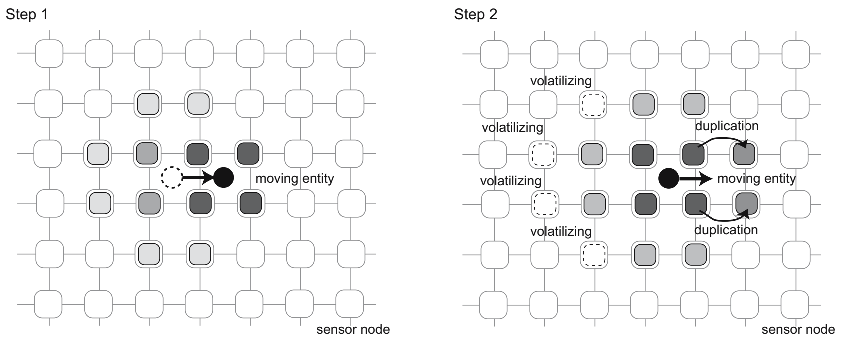


**Fig. 10.** Component diffusion with moving entities

[5] This problem is common in Ant-based routing mechanisms.

## 6 Related Work

Numerous mobile agent systems have been released, e.g., Aglets [9], Mole [16], Telescript [18], and Voyager [10]. Mole introduces the notion of agent groups to encourage coordination among mobile agents [2]. Its agent groups can consist of agents working together on a common task, but they are not mobile. The FarGo system introduces the notion of a dynamic layout for distributed applications [7] in a decentralized manner. This is similar to our relocation policy in the sense that it allows each agent to have its own policy, but it is aimed at allowing one or more agents to control a single agent, whereas ours aims at allowing one agent to describe its own migration. This is because our framework treats agents as autonomous entities that travel from computer to computer under their own control. This difference is important, because FarGo's policies may conflict if two agents can declare different relocation policies for one single agent. Our framework is free of conflict because each agent can only declare a policy to relocate itself but not for other agents.

There have been a few agent systems based on the concept of agent hierarchy. To our knowledge, the first attempt at introducing hierarchically mobile agents was the MobileSpaces mobile agent system. It proposed two concepts, agent hierarchy and inter-agent migration like the framework presented in this paper, and allowed more than one mobile agent to be dynamically assembled into a single mobile agent. It could provide a practical framework for mobile agent-based applications that were large and complex. Although the framework presented with this paper is based on experience with the MobileSpaces system, it not only offers hierarchical agent compositions but also horizontal agent compositions in the sense that agents can define their relocations according to the locations of other agents.

The notion of agent hierarchy presented in this paper is similar to a process calculus for modeling process migration, called *mobile ambients* [3]. 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 entirely change its semantics. El Fallah-Seghrouchni and Suna. proposed the CLAIM system [5] that provides hierarchical mobile agents based on the concept of mobile ambients. The system aimed at implementing basic operations of mobile ambients to support intelligent agents, whereas the system presented in this paper uses agent hierarchy as a (meta) mechanism for providing agents with various services, including agent organization over a distributed system.

Several mobile agent systems, e.g., Telescript, have introduced the concept of places in addition to mobile agents. Places are agents that 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 complex and large in scale, can be easily constructed by combining more than one agent.

There have been several attempts to construct an application from software components running on different computers. Most of these have aimed at dynamically configuring interactions between components or objects running on different computers (e.g., see [6,8]), whereas the framework presented in the paper aims at dynamically deploying components to different computers. Since it supports the typical communication primitives that existing approaches to configuring interactions between components need for coordinating and configuring distributed components, it can naturally use these approaches as configuration mechanisms for deployable components. That is, it can complement existing dynamic configuration approaches to distributed objects or components.

There have been several attempts to develop infrastructures to dynamically deploy components between computers in large-scale computing environments, e.g., workstation-clusters and grid computing. Most of these have aimed at dynamically deploying partitioned applications to different computers in distributed systems to balance the computational load or network traffic. However, they have explicitly or implicitly assumed centralized management approaches to deploy partitioned applications to different computers, so that they have not allowed each partitioned application to have its own deployment approach.

## 7 Conclusion

This paper described a framework for dynamically organizing multiple mobile agents in distributed computing environments. It is unique to existing systems because it provides two mechanisms for organizing multiple mobile agents. The first enables a mobile agent to contain other mobile agent inside it and migrate to another mobile agent or computer with its inner agents. It is useful in developing large-scale mobile software from a collection of mobile agents. The second enables a mobile agent to be deployed at computers according to the movements of other mobile agents. It can move a federation of agents, running on different computers, over a distributed system in a self-organizing manner. We designed and implemented a prototype system for the framework and demonstrated its effectiveness in several practical applications. We believe that the framework provides a general and practical infrastructure for building deployable applications over a distributed system.

In concluding, we would like to identify further issues that need to be resolved. We are interested in security mechanisms that would enable interactions between people and agents. We developed an approach to test context-aware applications on mobile computers [13], but need to develop a methodology for it. We are further interested in developing a methodology for testing distributed applications that are based on this new framework by using the approach. We also proposed a specification language for the itinerary of mobile software for hierarchical mobile agents [12,14,15]. The language enables more flexible and varied policies to be defined for deploying agents.

## References

1. O. Babaoglu and H. Meling and A. Montresor, Anthill: A Framework for the Development of Agent-Based Peer-to-Peer Systems, Proceeding of 22th IEEE International Conference on Distributed Computing Systems, July 2002.
2. J. Baumann and N. Radounklis, Agent Groups in Mobile Agent Systems, Proceedings of Conference on Distributed Applications and Interoperable Systems, 1997.

3. L. Cardelli and A. D. Gordon, Mobile Ambients, Proceedings on Foundations of Software Science and Computational Structures, LNCS, vol. 1378, pp. 140–155, Springer 1998.
4. G. Di Caro and M. Dorigo, AntNet: A Mobile Agents Approach to Adaptive Routing, Proceedings of Hawaii International Conference on Systems, pp.74-83, Computer Society Press, January 1998.
5. A. El Fallah-Seghrouchni, A. Suna CLAIM: A Computational Language for Autonomous, Intelligent and Mobile Agents, Proceedings of ProMAS'03, 2003.
6. K. J. Goldman, B. Swaminathan, T. P. McCartney, M. D. Anderson and R. Sethuraman, The Programmers Playground: I/O Abstractions for User-Configurable Distributed Applications, IEEE Transactions on Software Engineering, Vol.21, No.9, pp. 735-746, 1995.
7. O. Holder, I. Ben-Shaul, and H. Gazit, System Support for Dynamic Layout of Distributed Applications, Proceedings of International Conference on Distributed Computing Systems (ICDCS'99), pp 403-411, IEEE Computer Soceity, 1999.
8. Jeff Kramer and Jeff Magee, Dynamic configuration for distributed systems, IEEE Transactions on Software Engineering, Vol. 11, No. 4, pp.424-436, April 1985.
9. B. D. Lange and M. Oshima, Programming and Deploying Java Mobile Agents with Aglets, Addison-Wesley, 1998.
10. ObjectSpace Inc., ObjectSpace Voyager Technical Overview, ObjectSpace, Inc. 1997.
11. I. Satoh, MobileSpaces: A Framework for Building Adaptive Distributed Applications Using a Hierarchical Mobile Agent System, Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS'2000), pp.161-168, April 2000.
12. I. Satoh, Building Reusable Mobile Agents for Network Management, IEEE Transactions on Systems, Man and Cybernetics, vol.33, no. 3, part-C, pp.350-357, August 2003.
13. I. Satoh, A Testing Framework for Mobile Computing Software, IEEE Transactions on Software Engineering, vol. 29, no. 12, pp.1112-1121, December 2003.
14. I. Satoh, Configurable Network Processing for Mobile Agents on the Internet, Cluster Computing (The Journal of Networks, Software Tools and Applications), vol. 7, no.1, pp.73-83, Kluwer, January 2004.
15. I. Satoh, Selection of Mobile Agents, Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS'2004), pp.484-493, IEEE Computer Society, March 2004.
16. M. Strasser and J. Baumann, and F. Hole, Mole: A Java Based Mobile Agent System, Proceeding of ECOOP Workshop on Mobile Objects (MOS'96), 1996.
17. T. Umezawa, I. Satoh, and Y. Anzai, A Mobile Agent-based Framework for Configurable Sensor Networks, Proceedings of International Workshop on Mobile Agents for Telecommunication Applications (MATA'2002), LNCS, Vol. 2521, pp.128-140, Springer, 2002.
18. J. E. White, Telescript Technology: Mobile Agents, General Magic, 1995.