Bio-inspired Deployment of Distributed Applications

Ichiro Satoh

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
Tel: +81-3-4212-2546 Fax: +81-3-3556-1916
E-mail: ichiro@nii.ac.jp

Abstract. This paper presents an approach to developing and managing self-organizing distributed computing systems. The approach is used to construct an application as a dynamic federation of mobile components that can migrate from computer to computer while the application is being executed. It also enables each component to explicitly define its own migration policy as the migration of other components. Therefore, a federation of components can be migrated and transformed according to its components' local policies, including bio-inspired deployment approaches. The approach was implemented as not only a test-bed system for the organization of multi-agents but also a middleware for real distributed systems. This paper describes a prototype implementation of the middleware built on a Java-based mobile agent system and its applications that illustrates the utility and effectiveness of the approach.

1 Introduction

Distributed computing systems are often composed of a number of software components, which run on different computers and interact with each other via a network. The complexity of modern distributed systems has already frustrated our ability to deploy components at appropriate computers through traditional approaches, such as centralized and top-down techniques. It is difficult to adapt such systems to changes in execution environments, such as adding or removing components and network topology, and to the requirements of users. This problem becomes more serious in ubiquitous computing as well as large-scale distributed systems, because ubiquitous computers are heterogeneous and their computational resources, such as processors, storage, and input and output devices, are limited so that they can only support their own initial applications. An application can execute on a group of one or more computers to satisfy its own requirements beyond the capabilities of individual computers. Moreover, such a group must be configurable in run-time because the goals and positions of users may change dynamically. We believe that the solutions to extreme dynamics and complexity in distributed systems, including ubiquitous computing environments, are based on metaphors drawn from biological processes.

Therefore, this paper presents a framework to adapt a federation of components, which may run on heterogeneous computers, to changes in user requirements and their associated contexts, such as locations and tasks. The framework is based on two key ideas. The first is to implement components as mobile agents that can travel from computer to computer under their own control. That is, each component can autonomously

migrate to another computer and duplicate itself. The second is to facilitate the dynamic federation of one or more components as a virtual computer over distributed systems. The framework enables such a federation to be transformed and made mobile through bio-inspired self-organization, such as that undertaken by cells in their transforming and crawling locomotion.

This paper continues with a description of the issues we consider are necessary for the framework (Section 2) and a description of the design goals for it (Section 3). We then describe its design (Section 4) and a prototype implementation (Section 5). We also discuss our experience with two applications, which we used the framework to develop (Section 6), and briefly review related work (Section 7). We present some future issues in brief (Section 8) and close with a summary (Section 9).

2 Approach

The goal of this framework is to provide a general infrastructure that enables applications on a distributed system to be deployed dynamically.

2.1 Distributed and Mobile Applications

The framework assumes that each application is composed of one or more software components, as we can see in Fig. 2. Each component corresponds to a motile unicellular structure since it is self-contained and self-mobile. An aggregation of components can also be treated as a pseudoplasmodium, because such an aggregation can change its structure and move over a distributed system according to changes in the underlying system and the requirements of the application (Fig. 1). The framework provides support for migration-transparent interactions between dynamically deployable components. It instructs components to migrate to computers that can satisfy their requirements. Where to deploy components is an application-dependent decision and a well-known practical policy is that any application that enables interaction with users should be executed at nearby computers to reduce network latency. For example, when the framework detects changes in a user's positions, it provides addresses of nearby computers by using the location information services we presented in our previous paper [17] and components then migrate to the closest of the computers.

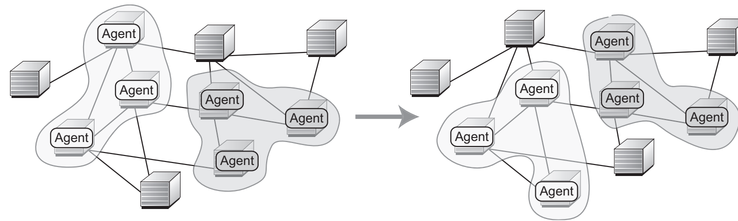

Fig. 1. Group migration

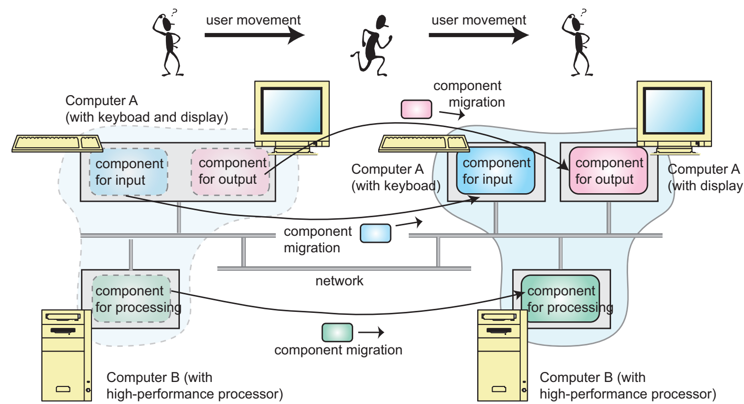

Fig. 2. Federation of Network-enabled Appliances

2.2 Drawing Inspiration from Cell Transformation and Locomotion

Applications must not be bound to heterogeneous computers, which may have limited computational resources for specifics applications, but should be able to run on any computer that can satisfy their requirements and those of their users. However, it is difficult to deploy components at appropriate computers on a distributed computing system where computers are dynamically added and removed. Furthermore, the requirements of users or applications may vary. For example, mobile users may also want to constantly change the computers with which they interact. Consequently, applications should be able to move from computer to computer to follow their users. Therefore, our framework should enable a federation of partitioned applications, i.e., components, to partially or entirely migrate to suitable computers according to changes in user conditions and their associated context, e.g., locations, current tasks, and the number of components.

Federation Mobility of Components as Cell-locomotion The framework is used to build an application as a set of mobile agent-based components and enables components to move to other computers while the application is running. As a result, the movement of one component may affect other components. For example, two components are required to remain at the same computer or nearby computers, when the first is a program that controls the keyboard and the second is a program that displays content on the screen. Since each component travels between computers under its own control, a federation of components tends to spread over a distributed system so that these distant components cannot efficiently coordinate with one another due to latency in communication. The framework therefore enables each component to explicitly specify its own constraints to migrate components. For example, if a component has a migration constraint dependent on another component, when the other component moves to another location, the former component decides its destination according to its own migration constraints, i.e., the source or destination of the other component. Such constraints are

defined as policies within components and allow us to specify physical structures and mechanisms in motile cells, such as membrane and cytoplasmic streaming, and gel-to-sol transitions.

Speculative Deployment of Components as Cell-lamellipodia Lamellipodia are flattened and protrusive projections that periodically expand from the surface of a cell. Effective movement requires a motile cell to be polarised, so that its protoplasm membrane is relatively quiescent everywhere else except its leading edge where lamellipodia periodically project outward in all directions. As they pull on one another they create intervening regions in which the cortex is stretched. This tug of war continues until one lamellipodium aligns in a dominant direction and becomes unipolar, then migrates in that direction. Lamellipodia can be viewed in terms of speculative migration or expansion. Each component, however, should migrate to one of the most eligible computers that can satisfy its requirements as long as its migration constraints are valid. However, it cannot always establish precisely which destination is the most suitable. This framework permits a component to speculatively deploy its clones at multiple computers and to select one of the most appropriate clones. This mechanism corresponds to the process lamellipodia go through in motile cells.

2.3 Architecture

Our framework should be used as a general test-bed for providing various bio-inspired approaches in distributed systems as well as a middleware for adaptive distributed systems. There may also be one or more approaches to deploying components in a distributed system, because these are often application-specific. Therefore, the framework itself should be as independent as possible of any component-deployment approach and of any particular phenomenon in biological processes. By separating component-deployment approaches from infrastructures, the framework provides a general middleware for exchanging components between computers and enables such approaches to be implemented within components instead of the middleware. That is, each component can have its own deployment policy for specifying spatial constraints between its location and the locations of other components at neighboring computers. As a result, a federation of components is managed by each of the components' policies instead of any global policy.

3 Design and Implementation

The framework presented in this paper was implemented in Sun's Java Developer Kit version 1.4 and uses a Java-based mobile agent system to provide mobile components. It consists of two parts: mobile components and component hosts. The first defines partitioned applications. The second is a middleware and enables components to migrate from computer to computer.

3.1 Mobile Component

It is almost impossible to automatically partition existing standalone applications across multiple computers. Instead, this framework relies on the concept of a component-based application construction [21]. That is, an application is loosely composed of software components, which may run on different computers. In the current implementation of the framework, each component is a collection of Java objects in the standard JAR file format that can migrate from computer to computer and duplicate itself through mobile agent technology.[1] After arriving at its destination or being duplicated, each component can continue working without losing the accumulated work, such as the content of instance variables in the component's program, at the source computers. It is also equipped with its own identifier and that of the federation that it should belong to. It can explicitly specify the computational capability that its destination hosts must offer in CC/PP form as we will discuss later. If a component is on a computer that cannot satisfy its requirements, its intent is to leave the computer.

As we will discuss in the following section, although the current implementation supports five several migration policies for the mobilities of two components, we will only present two typical policies as follows:

- When a component declares *follow* for another component, if the other component moves, the declarer or its clone migrates to the destination or a nearby proper host.
- When a component declares *fill* for another component, if the other component moves, the declarer or its clone migrates to the source of the latter component or a nearby proper host.

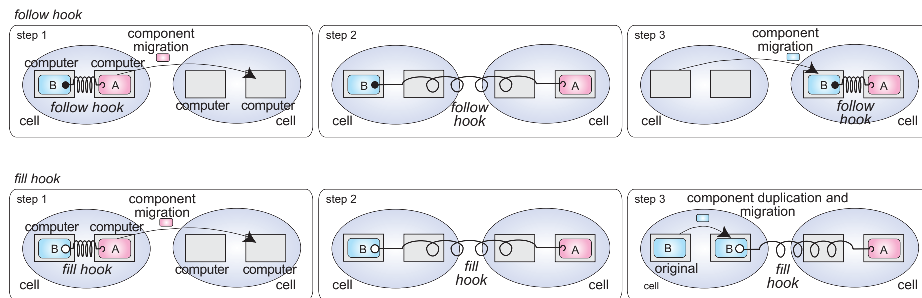

Fig. 3. Component migration with relocation policies

The first policy gathers components around specified components like aggregating dictyostelium and the second policy makes components track the footprints of other moving components like cytoplasmic streaming in cells. Fig. 3 and 4 have examples of the group migration of three components. When component B has a *follow* policy for

[1] JavaBeans can easily be translated into components in the framework.

component A and component C has a *dispatch* policy for component B, if component A moves, component B moves to component A's destination host because the host satisfies component B's requirements and a copy of component C moves to component B's source host. Each component can change its policy while it is running. When some components in a federation alternately become mobile or stationary, their irregular movements correspond to the gel-to-sol transitions in motile cells.

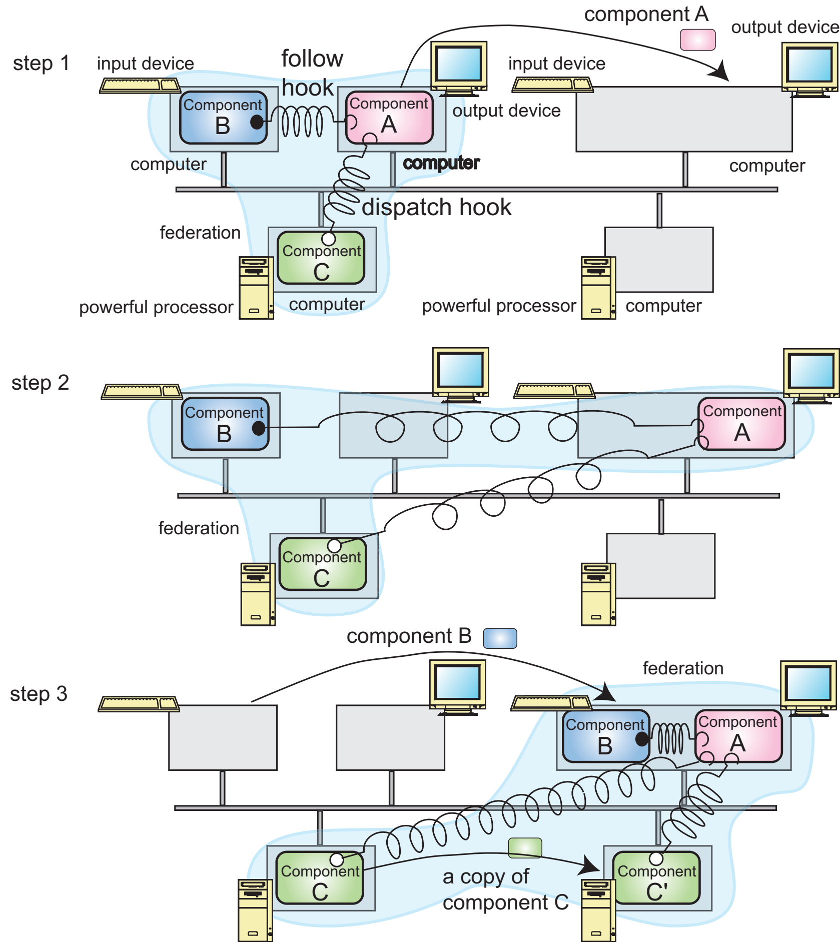

Fig. 4. Examples of component group migration with relocation policies

Such policies may be similar to the dynamic layout of distributed applications in the FarGo system [9]. However, FarGo's policies aim at allowing a component to control other components, whereas our policies aim at allowing a component to describe its own migration, because our framework always treats components as autonomous enti-

ties that travel from computer to computer under their own control. Note that policies may conflict in FarGo when two components can declare different relocation policies for a single component. However, our framework is free of any conflict because each component can only declare a policy to relocate itself instead of other components.

Each component can have references to other components within the application federation that it belongs to. Each reference allows a component to interact with the component that it specifies, even when the former and latter components reside at different computers or move to other computers. The current implementation of the references provides mobility-transparent remote method invocation.

3.2 Component Host

Each component host provides a runtime system for executing components and migrating them to another place. Fig. 5 outlines the basic structure of a runtime system. Each host establishes at most one TCP connection to each of its neighboring hosts and exchanges control messages, components, and inter-component communications with the other hosts through the connection. Since it is constructed on the Java virtual machine, it can conceal differences between the platform architecture of the source and destination hosts, such as the operating system and hardware.

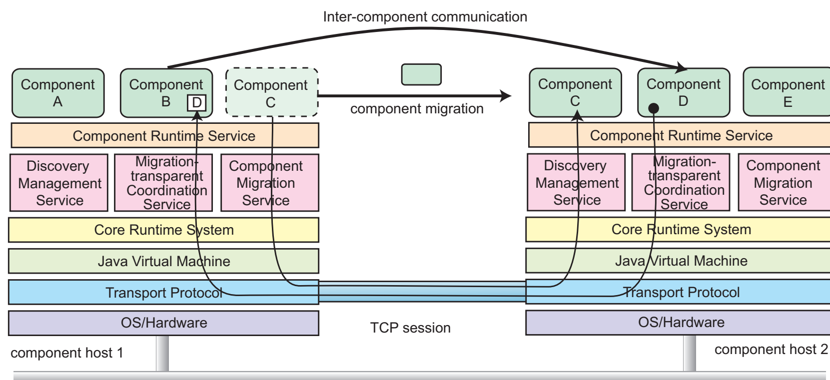

Fig. 5. Architecture of component host.

Component Runtime Service Each runtime system governs all the components inside it and maintains the life-cycle state of each component. When the life-cycle state of a component changes, e.g. when it is created, terminates, or migrates to another host, the runtime system issues specific events to the component. This is because the component may have to acquire various resources or release them such as files, windows, or sockets, which it had previously captured.

The framework offers a language-based on CC/PP (composite capability/preference profiles) [23] to describe the capabilities of component hosts and the requirements of components. For example, a description contains information on the properties of a computing device: the vendor and model class (PC, PDA, or phone), the screen size, the number of colors, CPU, memory, input devices, and secondary storage. Each host informs the components within it, or its neighboring hosts, about its profile specified with the language. Then, each of the components autonomously selects and migrates to one of the candidate destinations. Moreover, since each component can count coexisting components that are of the same type as it is, it can leave computers that have a high component density.

Component Migration Service All component hosts can exchange components with others through the use of mobile agent technology. When a component is transferred over a network, the component host on the sending side marshals the code of the component and its state into a bit-stream and then transfers these to the destination. Another component host on the receiving side receives and unmarshals the bit-stream. The current implementation uses the standard JAR file format for passing components that can support digital signatures, allowing for authentication. It also uses Java's object serialization package for marshaling components, which can save the content of instance variables in a component program but does not support the stack frames of threads being captured. Consequently, component hosts cannot serialize the execution states of any thread objects. Instead, when a component is marshaled and unmarshaled, the component host propagates certain events to its components to instruct the components to stop their active threads, and then automatically stops and marshals them after a given period. Moreover, each host has a database on the locations of components it has received to support migration-transparent inter-component interactions. When a component moves, the source host forwards messages to the moved component and the destination host updates the databases of other hosts by multicasting control messages.

4 Component Programming

In this framework, each component is implemented as a collection of Java objects that are defined as subclasses of the `Component` class as follows:

```
class Component extends MobileAgent implements Serializable {
  void go(URL url) throws NoSuchHostException { ... }
  setPolicy(ComponnetProfile cref,
    MigrationPolicy mpolicy, boolean coexist) { ... }
  setTTL(int lifespan) { ... }
  void setGroupIdentifier(GroupIdentifier gid) { ... }
  GroupIdentifier getGroupIdentifier() { ... }
  void setComponentProfile(ComponentProfile cpf) { ... }
  ComponentProfile getComponentProfile(ComponentRef ref) { ... }
  boolean isConformableHost(HostProfile hfs) { ... }
  ....
}
```

We will explain some of the methods defined in the `Component` class. A component executes the `go(URL url)` method to move to the destination host specified as the

url by its runtime system. The setTTL() specifies the life span, called Time-To-Live (TTL), of the component. The span decrements the TTL value as the passage of time. When the TTL of a component becomes zero, the component automatically removes itself. The setGroupIdentifier() method ties the component to the identity of the federation specified as gid. Each component can specify a requirement that its destination hosts must satisfy by invoking the setComponentProfile() method, with the requirement specified as cpf. The class has a service method called isConformableHost(), which the component uses to decide whether or not the capabilities of the component hosts specified as an instance of the HostProfile class can satisfy the requirements of the component. Each component can have more than one listener object that implements a specific listener interface to hook certain events issued by the runtime system before or after changes in its life-cycle state.

4.1 Migration Policy Programming

While each component is running, it can declare its own migration policy by invoking the setPolicy method of the Component class as follows:

```
setPolicy(cref, mp);
```

where the first argument is a reference to another component. The second argument is an instance of the MigrationPolicy class.

```
MigrationPolicy mp = new MigrationPolicy(int policy);
```

When a component specified as cref migrates from its source to its destination, the component creates an instance of the class with one of the following actions:

– If the second argument is new MigrationPolicy(Policy.FOLLOW), the component migrates to the same destination computer.
– If the second argument is new MigrationPolicy(Policy.DISPTACH), the component duplicates itself and migrates its clone to the same destination computer.
– If the second argument is new MigrationPolicy(Policy.SHIFT), the component migrates to the source computer.
– If the second argument is new MigrationPolicy(Policy.FILL), the component duplicates itself and migrates its clone to the source computer.
– If the second argument is new MigrationPolicy(Policy.STAY), the component stays at the current computer.

where each component can have at most one policy. Figure 6 outlines four basic policies, where two components, B and C, have policies for component A.

These policies are related to phenomena in biological processes. For example, Policy.FOLLOW enables a component to come near another component. When multiple components declares a policy for a leader component, they can swarm around the leader component. Policy.SHIFT enables a component to follow the movement of another component. The former component can track the latter component as it moves. The policy thus corresponds to the phenomenon of cytoplasmic streaming.

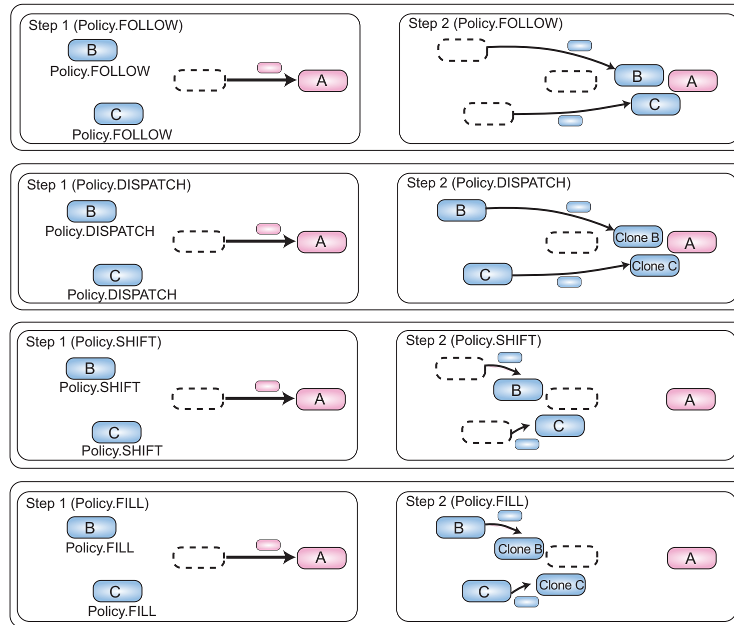

Fig. 6. Basic migration policies

`Policy.DISPATCH)` enables a component to stay in the current location and then deploys its clone at the destination of another moving component. `Policy.DISPATCH)` can model the footprint of a motile cell. We have assumed that a component can declare the policy for another component and specify the TTLs of its clones as their life-spans. As the latter component moves, cloned former component are deployed at the footmark of the latter component and these clones are automatically volatilized after their life-spans are over. Therefore, the clone components can be viewed as a pheromone that is left behind after the latter component has moved on. `Policy.FILL` 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.

4.2 Component Coordination Programming

Component references are responsible for tracking possibly moving targets and for invoking the targets' methods. This framework provides the APIs for invoking the methods of other components on local or other computers with copies of arguments. Our programming interface to invoke methods 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 our target is a dynamic computing system.

```
Message msg = new Message("print");
```

```
msg.setArg("hello world");
Object result = cref.invoke(msg);
```

The above code fragment is used to invoke a method of the component specified as the `cref` reference. Apart from this, the framework supports a generic remote publish/subscribe mechanism that enables subscribers to express their interest in an event so that they can be notified afterwards of any event fired by a publisher. This is implemented through Java's dynamic proxy mechanism, which has been a new feature of the Java 2 Platform since version 1.3.[2]

5 Current Status

A prototype implementation of this framework was constructed with Sun's Java Developer Kit version 1.4 and although it was not built for performance, we measured the cost of component migration. For example, the cost of migrating the federation of three components in Fig. 4 is 180 ms, where the cost of migrating a component between two hosts over a TCP connection is 42 msec. This experiment was done with five computers (1.2-GHz Pentium III, with Windows XP and JDK 1.4.2) connected through a Fast Ethernet network. The latency included the costs of the following processes: transmitting the component's requirements from the source host to the LIS through TCP, transmitting a candidate destination from the LIS to the source host through TCP, marshaling the component, migrating the component from the source host to the destination host through TCP, unmarshaling the agent, and verifying security.

The current implementation can encrypt components before migrating them over a network and then decrypt them after they arrive at their destination. Moreover, since each component is just 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 can explicitly restrict components to only access specified resources to protect hosts from malicious components. Although the current implementation cannot protect components from malicious hosts, the runtime system supports some authentication mechanisms to migrate components through mobile agent technology so that each component host can only send agents to and only receive from trusted hosts.

6 Initial Experience

This section presents three examples that illustrate how the framework works.

6.1 Desktop Teleporting in Ubiquitous Computing Environments

The first example is a mobile editor and is composed of three partitioned components. The first, called *application logic*, manages and stores text data and should be executed on a host equipped with a powerful processor with much amount memory. The second, called a *viewer*, displays text data on the screen of its current host and should

[2] As the dynamic creation mechanism is beyond the scope of the papers. we have left it for future publications.

be deployed at hosts equipped with large screens. The third is called a *controller* and forwards texts from its current host's keyboard to the first component. They have the following relocation policies. The application logic and control components have *follow* hook policies for the viewer component to deploy itself at the current host of the viewer component or nearby hosts. As we can see from Fig. 7, we assumed that the three components had been initially stored in two hosts.

The system can track the movement of the user in physical space through RFID-tag technology.[3] It also introduces a component, called a *user-counterpart*, since the component works as a virtual counterpart in cyberspace. The component can automatically move to hosts near the current location of the user, even while the user is moving. That is, a user-counterpart is always at a host near the user. Because the viewer component has a *follow* hook policy to move the user-counterpart component, it moves to a host that has a user-counterpart or nearby hosts. When a user moves to another location, the components can be dynamically allocated at suitable hosts without the loss of any coordination as we can see from Fig. 7. When application-specific components are animal cells, the counter component can be treated as a bait for those cells.

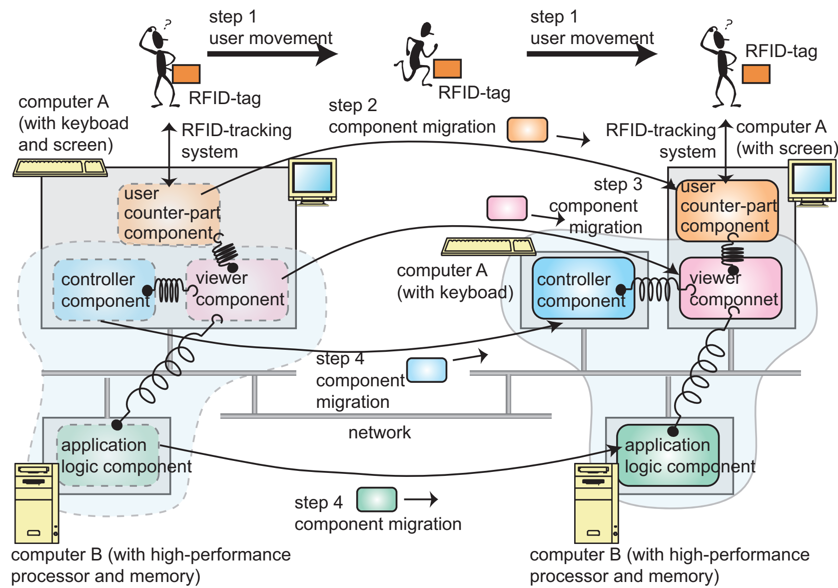

Fig. 7. Initial allocation of components for editor-application.

[3] An RFID-based location-dependent deployment of component was presented in our previous paper [17].

6.2 Ants-based routing mechanisms

Ants are able to locate a path to a food source using trails of chemical substances called pheromones that are deposited by other ants. Several researchers have attempted to use the notion of ant pheromones for network-routing mechanisms [3, 20]. Our framework allows moving components to leave themselves on their trails and to become automatically volatilized after their life-spans are over. A mobile agent corresponding to an ant A corresponding to a pheromone is attached to another mobile agent corresponding to an ant according to the *fill* policy. When the latter agent randomly selects its destination and migrates to the selected destination, 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 the `setTTL` method, 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 the 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 routing mechanisms tend to be diverging.

6.3 Component Diffusion in Sensor Networks

The second example is the speculative deployment of components as is done with cell-lamellipodia. This provides a mechanism that dynamically and speculatively deploys components at sensor nodes when there are environmental changes. This mechanism was inspired by lamellipodia in cells. It assumes that the sensor field is a two-dimensional surface composed of sensor nodes and it monitors environmental changes, such as motion in objects and variations in temperature. It is a well known fact that after a sensor node detects environmental changes in its area of coverage, some of its geographically neighboring nodes tend to detect similar changes after a short time. Diffusion occurs as follows. When a component on a sensor node finds changes in its environment, the component duplicates itself and deploys the copy at neighboring nodes as long as the nodes have the same kinds of components (Fig. 8). Each component is associated with a resource limit that functions as a generalized Time-To-Live field. Although a node can monitor changes in interesting environments, it sets the TTLs of its components as their own initial value. It otherwise decrements TTLs as the passage of time. When the TTL of a component becomes zero, the component automatically removes itself. This example is still in the early stages of experimentation but we have developed a mobile agent-based middleware for sensor networks [22] and plan to extend this framework to the middleware.

7 Related Work

The section discusses several bio-inspired approaches to distributed and multi-agents systems. Most of the work has been based on simulators. For example, Swarm [7] and

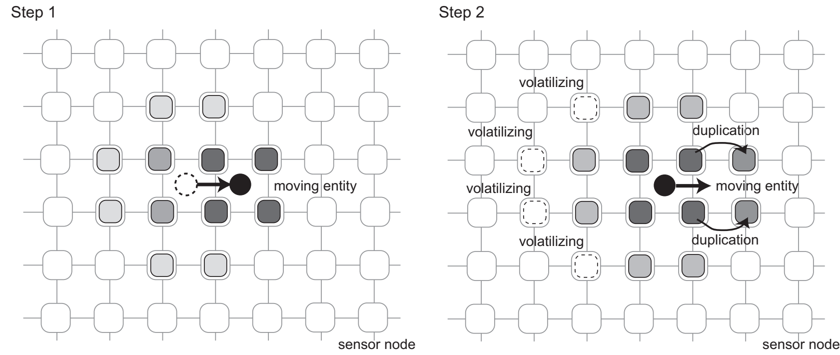

Fig. 8. Component diffusion in moving entity

MASS [6] are general simulators for multi-agent models. However, real systems are complex and varied. Our goal was also to provide a practical middleware for adaptive distributed systems. Unfortunately, we could not gain a rich experience with bio-inspired approaches in real systems because there have been few real systems based on approaches in the real world.[4] We still lack a lot of data that are essential to simulating the approaches accurately. Therefore, real experiments in a real distributed system must have priority over simulation-based experiments for actual experience to accumulate.

A few attempts have provided infrastructures for real distributed systems, like ours. The Anthill project [1] by University of Bologna developed a bio-inspired middleware for peer-to-peer systems, which is composed of a collection of interconnected nests. Autonomous agents, called ants can travel across the network trying to satisfy user requests, like ours. The project provided bio-inspired frameworks, called Messor [11] and Bison [12]. Messor is a load-balancing application of Anthill and Bison is a conceptual bio-inspired framework based on Anthill. The main difference between Anthill, including its applications, and our framework is that it introduces agents as independent entities and ours permits components to be organized in a self-organized manner. The Co-Field project [10] by University di Modena e Reggio Emilia proposed the notion of a computational force-field model for coordinating the movements of a group of agents, including mobile devices, mobile robots, and sensors. However, the model only seems to be available within the limits of simulation and not within a real distributed system. Hive [8] is a distributed agent middleware for building decentralized applications and it can deploy agents at devices in ubiquitous computing environments and organize these devices as groups of agents. Although it introduced metaphors drawn from ecology, it cannot change the structure of agents dynamically whereas ours can.

We described an infrastructure for location-aware mobile agents in a previous paper [17]. Like the framework presented in this paper, this infrastructure provides tagged entities, including people and things, with application-level software to support and an-

[4] In fact, several existing simulation-based results seem to be based on arbitrary hypotheses in the sense that various parameters in their experiments lack any technical grounds.

notate them. However, since it cannot partition an application into one or more components, it must deploy and run an application within single instead of multiple computers. We presented an early prototype implementation of the federation mechanism presented in this paper in another previous paper [18]

8 Future Work

There are still further issues that need to be resolved. The final goal of this middleware is to provide a general test-bed for various bio-inspired approaches for adaptive distributed systems. Although the current implementation focuses on the deployment of components, we plan to extend it so that it can be used to modify the behavior of each component, while they are running. Also, as its performance is not yet entirely satisfactory, further measurements and optimizations will be needed. The current migration policy for partitioned applications may still be naive. We have studied some higher-level routings for mobile agents in previous papers [14, 16, 19] and are interested in applying routing approaches to partitioned applications. We plan to develop a monitoring and testing system for components by using an approach where we test context-aware applications on mobile computers [15].

9 Conclusion

This paper presented a middleware system for providing a dynamic federation of components on a distributed system. Since the middleware enabled each component to migrate over a distributed system under its own policy, the federation was mobile and able to be transformed in a self-organized manner. For example, it permitted components to follow other moving components and deployed their clones at different computers similar to what happens in the locomotion of motile cells. We designed and implemented a prototype middleware system and demonstrated its effectiveness in several applications.

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. B. L. Brumitt, B. Meyers, J. Krumm, A. Kern, S. Shafer, EasyLiving: Technologies for Intelligent Environments, Proceedings of International Symposium on Handheld and Ubiquitous Computing (HUC'00), pp. 12-27, September, 2000.
3. G. Di Caro and M. Dorigo, AntNet: Distributed Stigmergetic Control for Communications Networks, Journal of Artificial Intelligence Research, vol.9, pp. 317-365, 1998.
4. K. J. Goldman, B. Swaminathan, T. P. McCartney, M. D. Anderson, R. Sethuraman The Programmers' Playground: I/O Abstraction for User-Configurable Distributed Applications, IEEE Transactions on Software Engineering, vol. 21, no. 9, pp.735-746, September 1995.
5. A. Harter, A. Hopper, P. Steggeles, A. Ward, P. Webster, The Anatomy of a Context-Aware Application, Proceedings of Conference on Mobile Computing and Networking (MOBICOM'99), pp. 59-68, ACM Press, 1999.

6. B. Horling, and V. Lesser, and R. Vincent, Multi-Agent System Simulation Framework Proceeding of IMACS World Congress 2000 on Scientific Computation, Applied Mathematics and Simulation, August 2000.
7. N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The Swarm Simulation System, A Toolkit for Building Multi-Agent Simulations, Technical report, Swarm Development Group, June 1996.
8. N. Minar, M. Gray, O. Roup, R. Krikorian, P. Maes, Hive: Distributed Agents for Networking Things, International Symposium on Agent Systems and Applications / International Symposium on Mobile Agents (ASA/MA'99), 1999.
9. 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.
10. M. Mamei, L. Leonardi, F. Zambonelli, Co-Fields: A Unifying Approach to Swarm Intelligence, International Workshop on Engineering Societies in the Agents World (ESAW 2002), Lecture Notes in Computer Science, vol. 2577, Springer Verlag 2003.
11. A. Montresor, H. Meling, and O. Babaoglu, Messor: Load-Balancing through a Swarm of Autonomous Agents, Proceedings of International Workshop on Agents and Peer-to-Peer Computing, July 2002.
12. A. Montresor and O. Babaoglu, Biology-Inspired Approaches to Peer-to-Peer Computing in BISON Proceedings of International Conference on Intelligent System Design and Applications, Oklahoma, August 2003.
13. M. Román, H. Ho, R. H. Campbell, Application Mobility in Active Spaces, Proceedings of International Conference on Mobile and Ubiquitous Multimedia, 2002.
14. 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.
15. I. Satoh, A Testing Framework for Mobile Computing Software, IEEE Transactions on Software Engineering, vol. 29, no. 12, pp.1112-1121, December 2003.
16. 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.
17. I. Satoh, Linking Phyical Worlds to Logical Worlds with Mobile Agents, Proceedings of IEEE International Conference on Mobile Data Management (MDM'2004), pp. 332-343, IEEE Computer Society, January 2004.
18. I. Satoh, Dynamic Federation of Partitioned Applications in Ubiquitous Computing Environments, Proceedings of IEEE International Conference on Pervasive Computing and Communications (PerCom'2004), pp.356-360, IEEE Computer Society, March 2004.
19. 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.
20. R. Schoonderwoerd, O. Holland, and J. Bruten, Ant-like agents for load balancing in telecommunications networks, Proceedings of Conference on Autonomous Agents, pages 209-216. ACM Press, 1997.
21. C. Szyperski, Component Software, Addison-Wesley, 1998.
22. Umezawa T, Satoh I, Anzai Y. A Mobile Agent-based Framework for Configurable Sensor Networks. Proceedings of International Workshop on Mobile Agents for Telecommunication Applications (MATA'2002); Lecture Notes in Computer Science 2002; Springer; Vol. 2521: 128-140.
23. World Wide Web Consortium (W3C), Composite Capability/Preference Profiles (CC/PP), http://www.w3.org/TR/NOTE-CCPP, 1999.