Self-organizing Software Components in Distributed Systems

Ichiro Satoh

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

Abstract. This paper presents a framework for deploying software components over a distributed system by using the notion of dynamics between components. It enables an application to be composed of one or more mobile components that can be deployed to different computers when the application is being executed. The key idea behind the framework is to provide components with deployment policies corresponding to gravitational and repulsive forces. The polices control the relocation relation between two components. As a result, a federation of distributed components can be moved and changed over a distributed system in a self-organizing manner. This paper also presents a prototype implementation of the approach and its applications.

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 scale and complexity of modern distributed systems impair our ability to deploy components to appropriate computers using traditional approaches, such as those that are centralized and top-down. The structure of a distributed system may also be frequently changed by adding or removing components and changing the network topology. Applications, which consist of components running on different computers, must adapt to such changes. When computers are about to shut down, for example, the components running on them must be deployed elsewhere. Moreover, the requirements of the applications tend to vary and changed dynamically. For example, users in a ubiquitous computing setting may also want to constantly interact with their applications running on nearby stationary computers. When they move from location to location, the components that the application consists of should be dynamically deployed at computers that are near their current position and can offer the computational resources required by the components.

To solve these problems, we have developed a framework for dynamically dispersing software components over a distributed system. It provides components with their own relocation policies without the need of any global policies. As a result, it enables individual components or a group of components to migrate over a network in a self-organizing manner without losing their previous coordination. We have presented earlier versions of the framework in this paper in our previous papers [13,15]. These previous versions supported the attachment of components to other components, but not the detachment of components, unlike this version. This problem is serious in implementing load-balancing and in fault-tolerant systems. The framework supports a mechanism for distributing components in addition to that for organizing a moving mass of components.

This paper describes our design goals (Section 2), the design of the framework, and a prototype implementation (Section 3). We also describe our experience with it (Section 4). We then briefly review related work (Section 5), provide a summary, and discuss some future issues (Section 6).

2 Basic Approach

Most modern large-scale systems consist of software components, which may run on different computers over a distributed system. The deployment of software components which a system consists of, seriously affects what a system can achieve and how efficiently it can achieve this. These components also need to be dynamically deployed and replaced at computers without them losing any previous coordination according to changes in the structure of the distributed system and the requirements of the system's applications. However, it is almost impossible for any centralized management systems to deploy components at appropriate computers because the systems have no global view of the distributed system as a whole. To solve this problem, our framework introduces two metaphors, i.e., *gravitational* and *repulsive* forces between components (Fig. 1). The former deploys components that coordinate with one another at the same computers or those nearby even when they move to other locations. The latter prevents specified components from being at the same or nearby computers. These component-deployment approaches are specified and managed as a relocation relationship between two components. That is, the framework enables each component to explicitly specify a deployment policy for its own migration as relocation between its current location and another component's location. An aggregation of components, each with its own deployment policies, can change its structure and move over a distributed system in response to changes in the underlying system and the requirements of the system's applications. All the deployment policies presented in this paper are managed in a non-centralized manner to maintain scalability and reliability.

Most interactions between components in object-oriented systems within a computer can be covered by three primitives: event passing, method invocation, and stream communication. Our framework enables these primitives to be available in partitioned systems on different computers. Achieving syntactic and (partial) semantic transparency for remote interactions requires the use of proxy objects that have the same interfaces as the remote components. The framework introduces such objects, called *references*, to track possibly moving targets and to interact with the these through the three primitives.

Remark

This framework was inspired by our earlier versions presented in previous papers [13,15]. The previous papers aimed at presenting the middleware for building and operating a large-scale system as a federation of one or more mobile components like the framework presented here, but they addressed ubiquitous computing environments

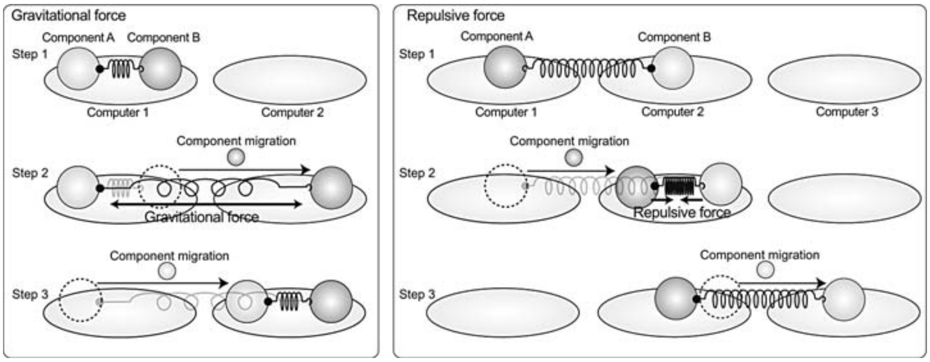

Fig. 1. Gravitational and repulsive policies

whose computers are heterogenous rather than large-scale distributed systems. The previous versions offered some of the gravitational relocation policies supported by this framework, but lacked any of the repulsive policies, which are essential in supporting load-balancing and fault-tolerant mechanisms. In fact, when many components are organized and deployed over a distributed system by only using the gravitational relocation policies, they tend to gather at several computers.

3 Design and Implementation

This framework consists of two parts: runtime systems and components. Each component in the current implementation is a collection of Java objects.

3.1 Component Runtime System

Each runtime system is running on a computer and is responsible for executing and migrating components to other computers. It establishes at most one TCP connection with each of its neighboring computers and exchanges control messages, components, and inter-component communications with these through the connection. Fig. 2 outlines the basic structure of a runtime system. Each component in the current implementation is a collection of Java objects in the standard JAR file format and can migrate from computer to computer and duplicate itself by using mobile agent technology [9].[1] When a component is transferred over the network, the component runtime system on the sending side marshals the code of the component and its state, e.g., instance variables in Java objects, into a bit-stream and then transfers them to the destination. The component runtime system on the receiving side receives and unmarshals the bit-stream so that the component can continue to be executed at the destination.

3.2 Component Programming Model

Each component runtime system governs all the components inside it and maintains their life-cycle states. When the life-cycle state of a component changes, e.g., when it is

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

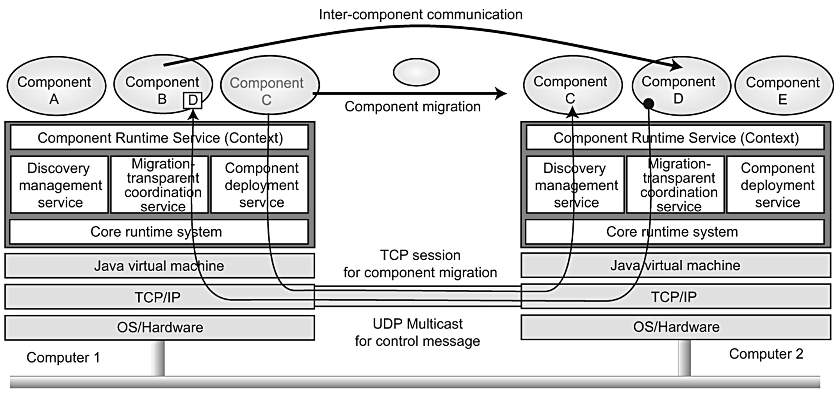
Inter-component communication
Component A
Component B
D
Component C
Component migration
Component C
Component D
Component E
Component Runtime Service (Context)
Discovery management service
Migration-transparent coordination service
Component deployment service
Core runtime system
Java virtual machine
TCP/IP
OS/Hardware
TCP session for component migration
UDP Multicast for control message
Computer 1
Computer 2

Fig. 2. Component runtime system

created, terminates, or migrates to another computer, the runtime system issues specific events to the component. This is because the component may have to acquire various resources, e.g., files, windows, or sockets, or release ones it had previously acquired. The current implementation uses Java's object serialization package for marshaling components. This package can save the content of instance variables in a component program but does not enable the stack frames of threads to be captured. Consequently, runtime systems cannot serialize the execution states of any thread objects. Instead, when a component is marshaled or unmarshaled, the runtime system propagates certain events to its components instructing them to stop their active threads and it then automatically stops and marshals them after a given period of time. Each component must be an instance of a subclass of the `MComponent` class. Here, we will explain the programming interface characterizing the framework.

```
class MComponent extends MobileAgent implements Serializable {
  void go(URL url) throws NoSuchHostException { ... }
  void duplicate() throws IllegalAccessException { ... }
  setPolicy(ComponnetProfile cref, MigrationPolicy mpolicy) { ... }
  setTTL(int lifespan) { ... }
  void setComponentProfile(ComponentProfile cpf) { ... }
  boolean isConformableHost(HostProfile hfs) { ... }
  void send(URL url, ComponentID id, Message msg) throws
    NoSuchHostException, NoSuchComponentException, ... { .... }
  Object call(URL url, ComponentID id, Message msg) throws
    NoSuchHostException, NoSuchComponentException, ... { .... }
  ....
}
```

A component executes `go(URL url)` to move to the destination host specified as a `url` by its runtime system, and `duplicate()` creates a copy of the component, including its code and instance variables. The `setTTL()` specifies the life span, called time-to-live (TTL), of the component. The life span decrements TTL over time. When the TTL of a component reaches zero, the component automatically removes itself.

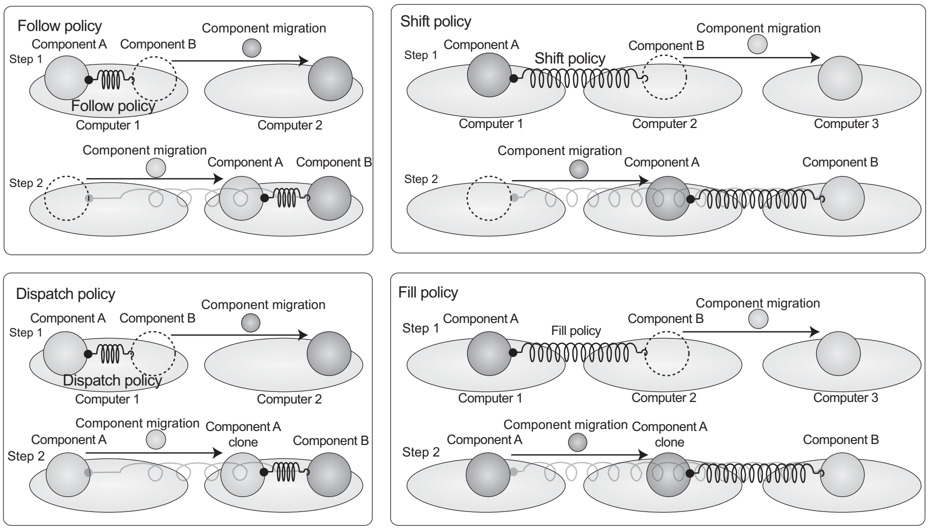

Fig. 3. Gravitational policies

Each component can have more than one listener object that implements a specific listener interface to hook certain events issued before or after changes are made in its life-cycle state. That is, each component host invokes the specified callback methods of its components when the components are created, destroyed, or migrate to another host.

3.3 Component Deployment Policy

A component can declare its own deployment policy by invoking the `setPolicy` method of the `MComponent` class while a component is running .

Let us now explain four *gravitational* policies (Fig. 3).

- If one component declares a *follow* policy for another, when the latter exists or migrates to a host, the former migrates to the latter's current or destination host.
- If a component declares a *dispatch* policy for another, when the latter migrates to another host, a copy of the former is created and deployed at the latter's destination host.
- If a component declares a *shift* policy for another, when the latter migrates to another host, the former migrates to the latter's source host.
- If a component declares a *fill* policy for another, when the latter migrates to another host, a copy of the former is created and deployed at the latter's source host.

The framework allows each component to have at most one gravitational policy for at most one component to reduce conflicts in individual or multiple policies. The *follow* policy is useful when relationships between components comprising an application need to be retained, and the *fill* policy is useful when components are distributed to hosts along the tracks of moving components. The deployment of one component depends on the location of another but the deployment of the latter does not need to depend on the location of the former. Instead, two components can explicitly declare policies for each other. When a component is created, the dispatch and fill policies can explicitly control whether the newly created component can inherit the state of its original.

We will next describe two *repulsive* policies. Each component can have more than one repulsive policy in addition to either the *shift* or *fill* policy.

- If a component declares an *exclusive* policy for one or more components, when the former and one of the latter are running on the same host, the former migrates to another host on which the latter components are not running.
- If a component declares an *extinct* policy for one or more components, when the former and one of the latter are running on the same host, the former terminates.

Fig. 4 illustrates these policies. If a component declares two or more polices, these policies must have different targets. The first corresponds to repulsive force and the second is used to eliminate components.

Components duplicated by the dispatch or fill policy have this policy for their original components. Each component can specify a requirement that its destination host must satisfy by invoking `setComponentProfile()`, with the requirement specified as `cpf`, where it is defined in CC/PP (composite capability/preference profiles) form [17], which describes the capabilities of the component host and the components' requirements. The class has a service method called `isConformableHost()`, which the component uses to determine whether the capabilities of the component host specified as an instance of the `HostProfile` class satisfy it requirements. Runtime systems transform the profiles into their corresponding LISP-like expressions and then evaluate them by using a LISP-based interpreter. When a component migrates to the destination according to its policy, if the destination cannot satisfy the requirements of the component, the runtime system recommends candidates that are hosts in the same network domain to the component. If a component declares repulsive policies in addition to a gravitational policy, the runtime system detects the candidates using the latter's policy and then recommends final candidates to the component using the former policy, assuming that the component is in each of the detected candidates.

3.4 Component Deployment Management

The policy-based deployment of components is managed by each component host without a centralized management server. Each component host periodically advertises its address to the others through UDP multicasting, and these hosts then return their addresses and capabilities to the host through a TCP channel.[2] (1) When a component migrates to another component host, each component automatically registers its deployment policy with the destination host. (2) The destination host sends a query message to the source host of the visiting component. There are two possible scenarios: the visiting component has a policy for another component or it is specified in another component's policies. (3-a) Since the source host in the first scenario knows the host running the target component specified in the visiting component's policy, it asks the

[2] We assumed that the components comprising an application would initially be deployed at hosts within a localized space smaller than the domain of a sub-network.

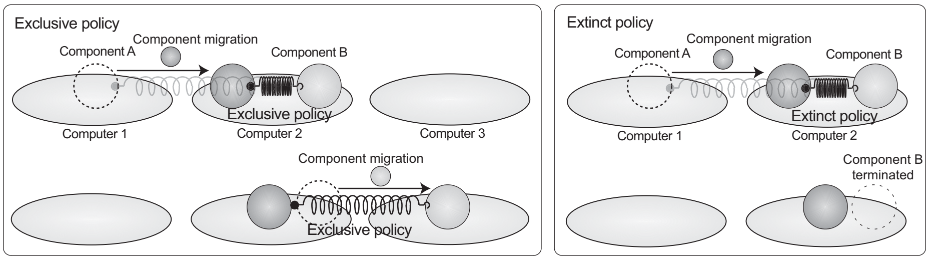

Fig. 4. Repulsive policies

host to send the destination host information about itself and about neighboring hosts that it knows, e.g., network addresses and capabilities. If the target host has retained the proxy of a target component that has migrated to another location, it forwards the message to the destination of the component via the proxy. (3-b) In the second scenario, the source host multicasts a query message within current or neighboring sub-networks. If a host has a component whose policy specifies the visiting component, it sends the destination host information about itself and its neighboring hosts. (4) The destination host next instructs the visiting component or its clone to migrate to one of the candidate destinations recommended by the target host, because this framework treats every component as an autonomous entity. Moreover, when the capabilities of a candidate destination do not satisfy all the requirements of the component, the component itself decides, on the basis of its own configuration policy, whether it will migrate itself to the destination and adapt itself to the destination's capabilities. The destination of the component may go into divergence or vibration mode due to conflicts between some of a component's policies, when it has multiple deployment policies. However, the current implementation does not exclude such divergence or vibration.[3]

3.5 Intercomponent Communication

The current implementation offers two communication policies for intercomponent interactions as follows:

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

The former policy is useful when two components share the same information and the latter policy provides a master-slave relation between them. The framework provides three interactions: publish/subscribe for asynchronous event passing, remote method invocation, and stream-based communication as well as message *forward* and *delegate*

[3] From our experience with several applications, most components in a system have at most a gravitational or a repulsive policy. Therefore, we do not always feel the needs to resolve such conflicts.

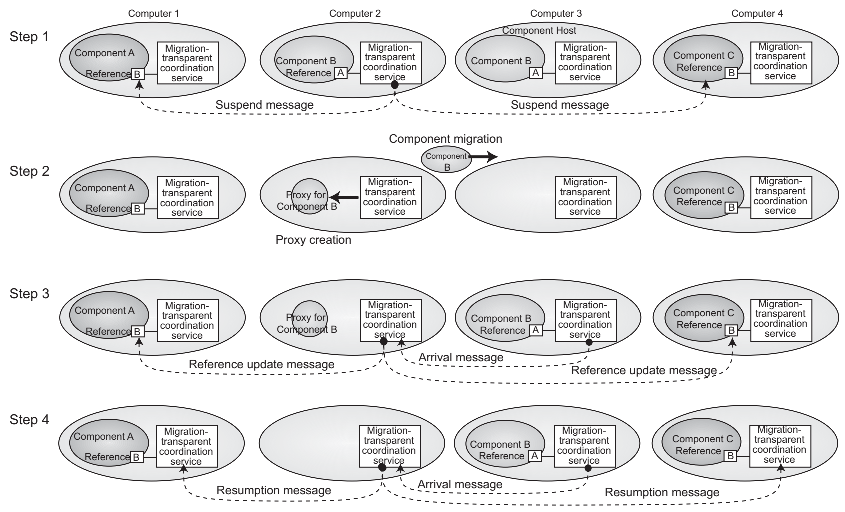

Fig. 5. Forwarding messages to migrated component

policies. Each runtime system offers a remote method invocation (RMI) mechanism through a TCP connection. It is implemented independent of Java's RMI because this has no mechanisms for updating references for migrating components. Each runtime system can maintain a database that stores pairs of identifiers of its connected components and the network addresses of their current runtime systems. It also provides components with references to the other components of the application federation to which it belongs. Each reference enables the component to interact with the component that it specifies, even if the components are on different hosts or move to other hosts.

Fig. 5 shows an approach enabling communication between a component moving from computer 2 to 3 and two components at computers 1 and 3. When a component, i.e., component B, requests the current runtime system to migrate to another computer, the system searches its database for the network addresses of runtime systems with components, i.e., computer 1 and 4. 1) It sends *suspend* messages to these systems to block any new uplinks from them to the migrating component with the destination's address. If the moving component contains references, the current runtime system sends the destination's address to the runtime systems that are running the components specified in the references so that they can update their databases. 2) It creates its own proxy at its current location and It migrates to its destination. 3) After the component arrives at its destination, it sends an *arrival* message with the network address of the destination to the departure runtime system and then *update* messages to the systems. 4) When the departure system receives the arrival message, it sends *resumption* messages with the address of the destination to runtime systems that may hold references to the moved component and then remove the proxy.

When a component begins to interact with another that is moving, the former can send messages to the source of the one that is moving before the basic algorithm above is completed. To solve this, a migrating component creates and leaves a proxy at the departure runtime system for the duration it takes the algorithm to finish. The proxy component receives uplinks from other runtime systems and forwards them to the moved component. Since not all components have to be tracked for other components to communicate with them, components can leave proxy components along their trail under their own control. Proxy components are also programmable entities, like components, so they can be modified based on application requirements.

3.6 Security

The current implementation is a prototype system to dynamically deploy the components presented in this paper. Nevertheless, it has several security mechanisms. For example, it can encrypt components before migrating them over the network and it can then decrypt them after they arrive at their destinations. Moreover, since each component is simply a programmable entity, it can explicitly encrypt its individual fields and migrate itself with these and its own cryptographic procedure. The Java virtual machine could explicitly restrict components so that they could only access specified resources to protect computers from malicious components. Although the current implementation cannot protect components from malicious computers, the runtime system supports authentication mechanisms to migrate components so that all runtime systems can only send components to, and only receive from, trusted runtime systems.

3.7 Current Status

A prototype implementation of this framework was constructed with Sun's Java Developer Kit, version 1.4 or later version.[4] Although the current implementation was not constructed for performance, we evaluated the migration of two components based on deployment policies. When a component declares a follow, dispatch, shift, or fill policy for another, the cost of migrating the former or its clone to the destination or the source of the latter after the latter begins to migrate is 92 ms, 116 ms, 89 ms, 118 ms, or 136 ms if the policy is follow, dispatch, shift, fill, or exclusive, where the cost of component migration between two computers over a TCP connection is 35 ms and the cost of duplicating a component in a computer was less than 7 ms.[5] This experiment was done with three computers (Pentium M-1.8 GHz with Windows XP and JDK ver.5) connected through a Fast Ethernet network. Migrating components included the cost of opening a TCP-transmission, marshaling the components, migrating them from their source computers to their destination computers, unmarshaling them, and verifying security.

4 Experience

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

[4] The functionalities of the framework, except for subscribe/publish-based remote event passing, can be implemented on Java Developer Kit version 1.1 or later, including Personal Java.

[5] The size of each of the three components was about 8 KB in size.

4.1 Dynamic Deployment for Duplicated Servers

We can easily implement distributed systems. Here, we present a fault-tolerant HTTP-based server to illustrate the use of these policies by combining gravitational policies, repulsive policies, and communication policies. Each component supports an HTTP server. It is a clonable component, where it and its clone declare forward policies for each other and its clone declares an exclusive policy for it. When a component is duplicated at a host, a clone is created at the host, but its exclusive policy deploys the clone at another host to distribute the original and cloned components at different computers to ensure tolerance against faults. When one of these receives messages from external systems, their forward policies send the messages to another so that they can share the same states. After the component duplicates itself, the cost of deploying its clone at another host is about 280 ms in the distribution system presented in the previous section.[6] This does not include the cost of terminating and restarting the HTTP server. The cost of forwarding a message is about 28 ms, where this is measured as the round-trip time and the message has no value. If the components declare delegate policies, they can support a master-slave instead of a duplication model.

4.2 Ant-Based Routing Mechanisms

Ants are able to locate a path to a food source using the 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 [1,2]. Our framework allows moving components to leave themselves on their trails and to become automatically volatile after their life-spans are over. A component corresponding to an ant, A, corresponding to a pheromone is attached to another component corresponding to an ant according to the *fill* policy. When the latter component randomly selects its destination and migrates to the selected destination, the former creates a clone and migrates to the source host of the latter. Since each of the cloned components defines its life-span by invoking `setTTL()`, they are active for a specified duration after being created. If there are other components corresponding to pheromones in the host, the visiting component adds their time spans to its own time span. When another component corresponding to another ant migrates over the network, it can select a host that has components corresponding to pheromones with the longest time-spans from neighboring hosts. We experimented with ant-based routing for components using this prototype implementation with more than eight hosts. However, we knew that it would be difficult to quickly converge a short-path to the destination in real distributed systems, because routing mechanisms tend to diverge.

4.3 Component Diffusion in Sensor Networks

The third example is the speculative deployment of components like 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

[6] This experiment assumes that the destination of the clone has been statically derived.

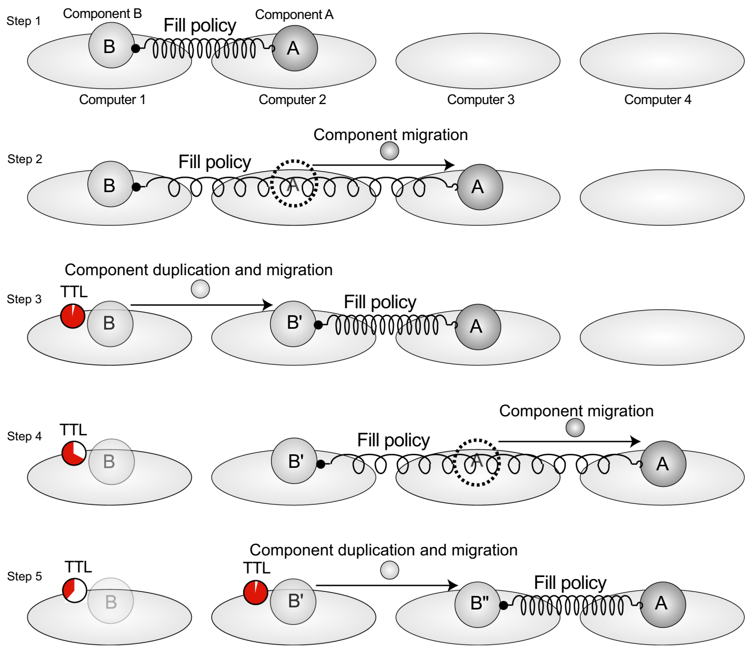

Fig. 6. Implementation of ant-based routing mechanism

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 period of time. It deploys monitoring components at sensor nodes, where each monitoring component can control and monitor its current sensor node and has its own TTL. Diffusion occurs as follows. When a component detects the presence of its target, it creates a specified number of its clones, e.g., two clones, where this number depends on the number of neighboring sensor nodes (Fig. 7). Each of the clones declares an exclusive policy for other monitoring components. It must migrate to a neighboring node according to the policy, because its original monitoring component is running on its current node. As a result, these clones are deployed at neighboring nodes around the target. When the target moves to another location, the monitoring components located at the nodes near the target detect the presence of the target and create their clones in the same way. We can provide application-specific components that declares a follow policy to monitor components. These components can be automatically deployed at nodes near the entity to annotate and assist the target. Each clone is associated with a resource limit that functions as a generalized TTL field. Although a node can monitor changes in interesting environments, it sets the TTLs of its components to their own initial values. It otherwise decrements TTLs as the passage of time. When the TTL of a component becomes zero, the component automatically removes itself.

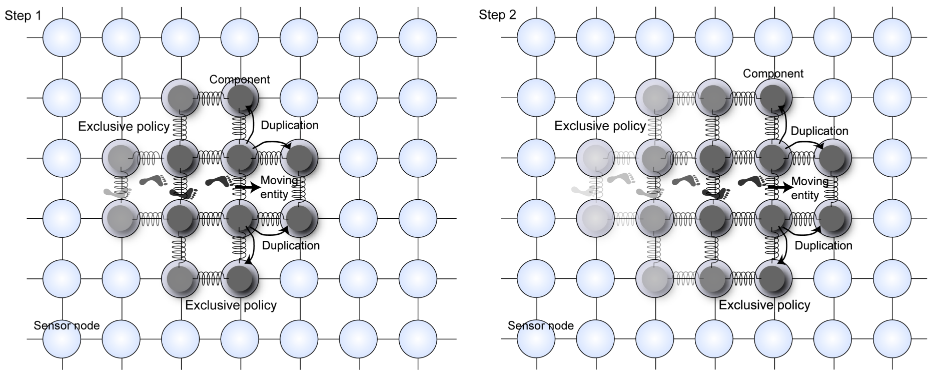

Fig. 7. Component diffusion for moving entity

5 Related Work

There have been several attempts to develop infrastructures for dynamically deploying components between computers in large-scale computing environments, e.g., workstation-clusters and grid computing. Most of them have aimed at dynamically deploying partitioned applications or systems to different computers in distributed systems to balance computational loads or network traffic. However, as they have explicitly or implicitly assumed centralized management approaches to deploying partitioned applications or systems to different computers, they have not allowed all partitioned applications or systems to have its own deployment approaches.

Of these, the FarGo system introduces a mechanism for distributed applications dynamically laid out in a decentralized manner [5]. This is similar to our relocation policy in the sense that it allows all components to have their own policies, but it is aimed at allowing one or more components to control a single component, whereas ours aims at allowing one component to describe its own migration. This is because our framework treats components 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 components can declare different relocation policies for one single component. Our framework is free of any conflict because each component can only declare a policy for its own relocation, and not for other components. Several researchers have introduced the dynamic deployment of partitioned applications as a technology that enables distributed computers to support various services, which they may not have initially been designed for, rather than to balance computational loads and traffic in a distributed system. For example, the Aura project [4] by CMU provides an infrastructure for binding tasks associated with users and migrating applications from computer to computer as users move about, like our framework does. Although Aura shares several common design goals with our framework, it focuses on providing contextual services to users rather than on integrating multiple computers to support functions and performance unattainable with a single computer. Like our framework, the Gaia project by the University of Illinois at Urbana-Champaign allows applications to be partitioned between different computers and move from computer to computer [8]. Gaia assumes that applications will be constructed based on a design pattern, called MPACC, which is is an extension of the MVC pattern [6], whereas our framework supports a variety of interactions between partitioned applications so that we do not have to assume any particular application model.

6 Conclusion

We described a framework for dynamically aggregating distributed applications in a distributed system based on physical dynamics. It was used to build an application from mobile software components, which can explicitly have policies for their own deployment. It enables a federation of components to be dynamically structured in a self-organized manner and to be deployed at computers as components that have gravitational and repulsive forces between them. 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 distributed and mobile applications.

In concluding, we would like to identify further issues that need to be resolved. The current implementation relies on Java's security manager. Nevertheless, we are interested in security mechanisms for components that have their own deployment policies and plan on introducing various such policies to support adaptive applications over a distributed system. We also proposed a specification language for the itinerary of mobile software [14]. The language enables more flexible and varied policies for deploying the components to be defined.

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. 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.
3. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns, Addison-Wesley, 1995.
4. D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste, Project Aura: Towards Distraction-Free Pervasive Computing, IEEE Pervasive Computing, vol. 1, pp. 22-31, 2002.
5. 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.
6. G. E. Krasner and S. T. Pope, A Cookbook for Using the Model-View-Controller User Interface Paradigma in Smalltalk-80, Journal of Object Oriented Programming, vol.1 No.3, pp. 26-49, 1988.
7. M. Román, C. K. Hess, R. Cerqueira, A. Ranganat R. H. Campbell, K. Nahrstedt K, Gaia: A Middleware Infrastructure to Enable Active Spaces, IEEE Pervasive Computing, vol. 1, pp.74-82, 2002.
8. M. Román, H. Ho, R. H. Campbell, Application Mobility in Active Spaces, Proceedings of International Conference on Mobile and Ubiquitous Multimedia, 2002.

9. 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.
10. 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.
11. I. Satoh, Configurable Network Processing for Mobile Agents on the Internet, Cluster Computing, vol. 7, no.1, pp.73-83, Kluwer, January 2004.
12. 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.
13. I. Satoh, Dynamic Federation of Partitioned Applications in Ubiquitous Computing Environments, Proceedings of 2nd International Conference on Pervasive Computing and Communications (PerCom'2004), pp.356-360, IEEE Computer Society, March 2004.
14. 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.
15. I. Satoh, Organization and Mobility in Mobile Agent Computing, Programming Multi-Agent Systems (Postproceedings of 3rd Workshop on ProMAS'05), Lecture Notes in Computer Science, vol. 3862, pp.187-205, April 2006.
16. C. Szyperski, Component Software, Addison-Wesley, 1998.
17. World Wide Web Consortium (W3C), Composite Capability/Preference Profiles (CC/PP), http://www.w3.org/TR/NOTE-CCPP, 1999.