Test-bed Platform for Bio-inspired 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 general-purpose test-bed platform for implementing and evaluating bio-inspired approaches over real distributed systems. It enables each software agent to be dynamically organized with other agents and deployed at computers according to its own organization and deployment policies. We developed several bio-inspired approaches with the platform to evaluate them over real distributed systems instead of using simulation-based evaluations. In fact, our experiment on an ant-based routing mechanism, which is one of the most typical bio-inspired approaches for distributed systems obtained different results from the results of existing simulation-based experiments. We believe that our platform is useful for bridging the gap between simulation-based approaches and real systems.

1. INTRODUCTION

The scale and complexity of modern distributed systems is beyond our ability to manage these using traditional approaches, such as those that are centralized and top-down. Furthermore, the structure of a distributed system may also be changed frequently by software agents (or components) being added or removed and the network topology being changed. To solve this problem, many researchers have explored bio-inspired approaches for distributed systems. However, most existing approaches are still at the concept level or have been evaluated based on simulations. Real distributed systems, on the other hand, are complex and varied. Nevertheless, most existing simulation-based results seem to have been based on arbitrary hypotheses in the sense that various parameters in their simulations have lacked any technical grounds. Unfortunately, such unrealistic simulations have often only provided non-sensing or impractical results. We still lack a great deal of data that are essential to simulating the approaches accurately. Therefore, real experiments in distributed systems must have priority over simulation-based experiments for us to be able to accumulate actual meaningful experience.

To solve this problem in bio-inspired approaches, we constructed a middleware system for dynamically federating and deploying software agents, which are autonomous and programmable entities, at distributed computers as a general-purpose test-bed for existing bio-inspired approaches over real distributed systems. The system deploys and federates multiple agents in a self-organizing manner in the sense that it enables all agents to define their own individual deployment and coordination policies. In fact, it implemented several bio-inspired approaches for distributed systems and yielded various results, which were different from those obtained from existing simulation-based work.

In this paper, we describe the design goals (Section 2), the design of our agents (Section 3), and a prototype implementation (Section 4). We present a basic evaluation of the platform (Section 5) and also describe our experience with the framework (Section 6). We briefly review related work (Section 7), provide a summary, and discuss some future issues (Section 8)

Bionetics '08, November 25-28, 2008, Hyogo, Japan

2. BASIC APPROACH

Most bio-inspired approaches for distributed systems can support non-centralized management. This paper presents a general test-bed platform that enables agents to define its own individual dynamic deployment and organization policies and to be executed over real distributed systems.

To support the dynamic deployment of agents, the proposed platform introduces two metaphors, i.e., *gravitational* and *repulsive* forces between agents (Fig. 1). The former deploys agents that coordinate with one another at the same computers or those nearby even when they move to other locations. The latter prevents specified agents from being at the same or nearby computers. These agent-deployment approaches are specified and managed as a relocation relationship between two agents. That is, the platform enables each agent to explicitly specify a deployment policy for its own migration as a relocation between its current location and another agent's location. An aggregation of agents, each with its own individual 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.

To support the self-organization of agents, the proposed platform enables agents to discover other agents and interact with co-agents. Most interactions between agents in object-oriented systems within a computer can be covered by three primitives: event passing, method invocation, and stream communication. Our platform 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 agents. The platform introduces such objects, called *references*, to track possibly moving targets and to interact with the these through the three

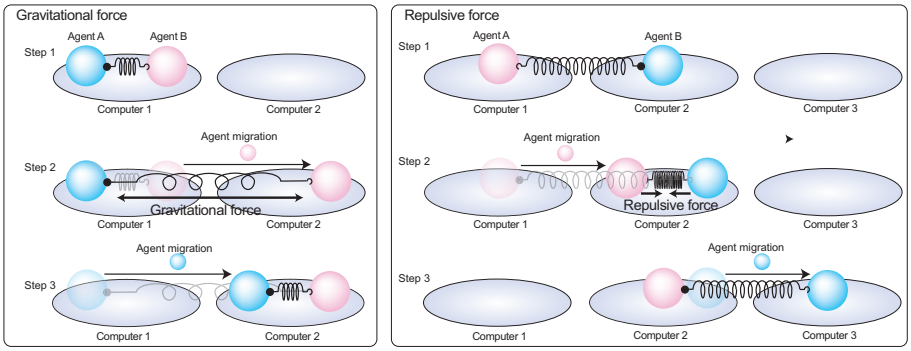

Figure 1: Gravitational and repulsive policies

primitives.

3. AGENT PROGRAMMING MODEL

To support a test-bed platform for bio-inspired approaches over distributed systems, the platform was designed and implemented as a general-purpose runtime system for agents, which are running for self-organizing or bio-inspired approaches. Figure 2 outlines the basic structure of the platform. Agents are autonomous and self-contained programmable entities, which can be dynamically deployed at different computers and organized with agents, which may run on different computers, according to their own deployment and organization policies.

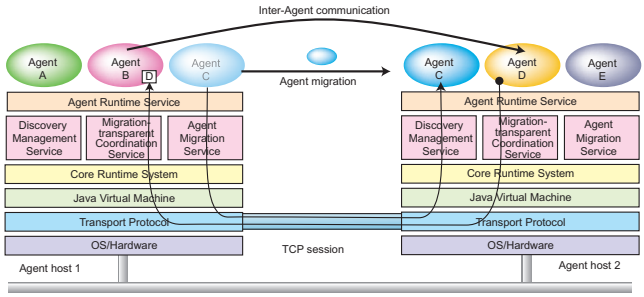

Figure 2: Agent runtime system.

Each agent 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.[1] Each agent must be an instance of a subclass of the `MAgent` class.

```
class MAgent extends MobileAgent implements
  Serializable {
  void go(URL url) throws
    NoSuchHostException { ... }
  void duplicate() throws
    IllegalAccessException { ... }
  setPolicy(ComponnetProfile cref,
    MigrationPolicy mpolicy) { ... }
  setTTL(int lifespan) { ... }
  void setAgentProfile(
    AgentProfile cpf) { ... }
  boolean isConformableHost(
    HostProfile hfs) { ... }
  void send(URL url, AgentID id, Message msg)
    throws NoSuchHostException,
      NoSuchAgentException, ... { .... }
  Object call(URL url, AgentID id,
    Message msg) throws NoSuchHostException,
      NoSuchAgentException, ... { .... }
  ....
}
```

[1]JavaBeans can easily be translated into agents in this platform.

Each agent can execute `go(URL url)` to move to the destination host specified as a `url` by its current platform, and `duplicate()` creates a copy of the agent, including its code and instance variables. The `setTTL()` specifies the life span, called the time-to-live (TTL), of the agent. The lifespan decrements TTL over time. When the TTL of an agent reaches zero, the agent automatically removes itself.

Each agent platform governs all the agents inside it and maintains their life-cycle states. When the life-cycle state of a agent changes, e.g., when it is created, terminates, or migrates to another computer, the platform issues specific events to the agent. This is because the agent 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 agents. This package can save the content of instance variables in an agent program but does not enable the stack frames of threads to be captured. Consequently, the platform cannot serialize the execution states of any thread objects. Instead, when an agent is marshaled or unmarshaled, the platform propagates certain events to its agents instructing them to stop their active threads and it then automatically stops and marshals them after a given period of time. To capture such events, each agent 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 agent host invokes the specified callback methods of its agents when the agents are created, destroyed, or migrate to another host.

3.1 Agent deployment policy

Let us explain deployment policies of agents. The current implementation has the following four *gravitational* policies (Fig. 3).

- If one agent 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 an agent 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.

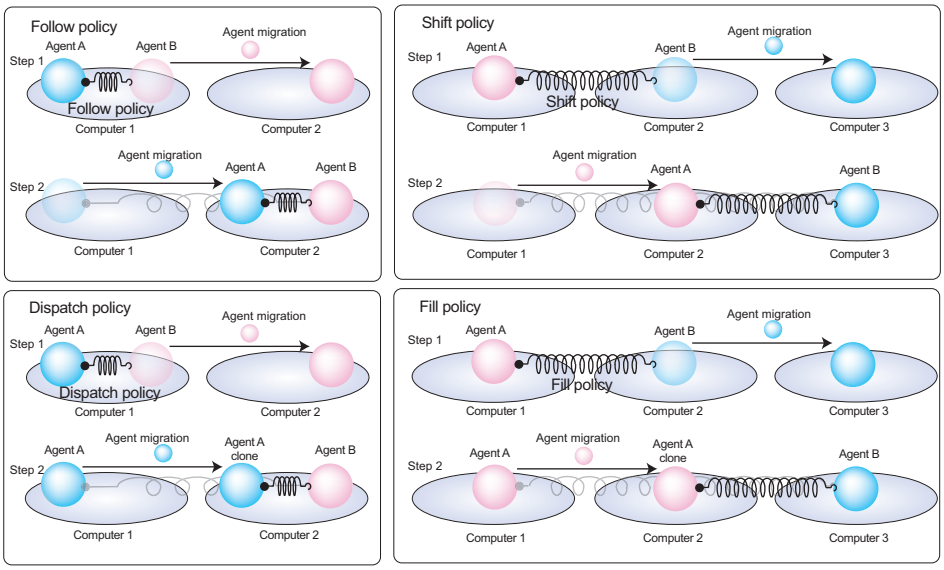

Figure 3: Gravitational policies

- If an agent declares a *shift* policy for another, when the latter migrates to another host, the former migrates to the latter's source host.
- If an agent 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 platform system allows each agent to have at most one gravitational policy for at most one agent to reduce conflicts in individual or multiple policies. These deployment policies can be related to phenomena in biological processes. For example, a *follow* policy enables one agent to approach another. 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 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 that is moving. It can model the footprint of a motile cell. We have assumed that one agent can declare the policy for another and specify the TTLs of its clones as their lifespans. As the latter agent moves, cloned former agents are deployed at its footprint and these clones are automatically volatilized after their lifespans are over. Therefore, the cloned 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 platform 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.

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

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

Figure 4 illustrates these policies. If an agent declares two or more polices, these policies must have different targets. The first corresponds to repulsive force and the second is used to eliminate agents.

Agents duplicated by the dispatch or fill policy have this policy for their original agents. Each agent can specify a requirement that its destination host must satisfy by invoking `setAgentProfile()`, with the requirement specified as `cpf`, where it is defined in composite capability/preference profiles (CC/PP) form [14], which describes the capabilities of the agent host and the agents' requirements. The class has a service method called `isConformableHost()`, which the agent uses to determine whether the capabilities of the agent host specified as an instance of the `HostProfile` class satisfy it requirements.

3.2 Inter-agent communication

The current implementation offers two communication policies for inter-agent interactions as follows:

- If an agent declares a *forward* policy for another, 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, 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 agents share the same information and the latter policy provides a master-slave relation between them. The platform provides three interactions: publish/ subscribe for asynchronous event passing, remote method invocation, and stream-based communication as well as message *forward* and *delegate* policies.

4. AGENT PLATFORM

This section presents an implementation of our agent platform. The platform was constructed as a general-purpose middleware for executing and organizing software agents, which may run on different computers.

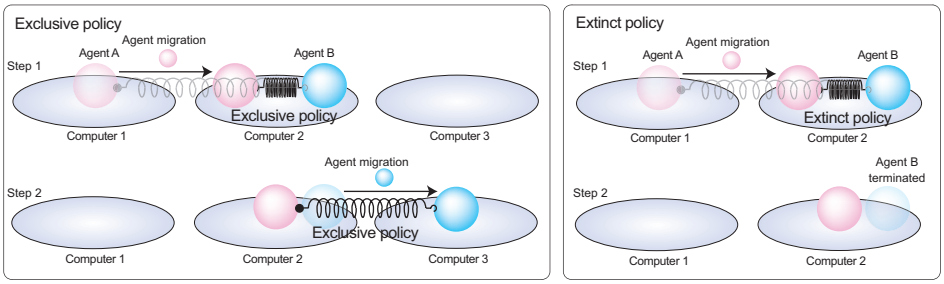

Figure 4: Repulsive policies

Each platform is running on a computer and is responsible for executing and migrating agents to other computers. It establishes at most one TCP connection with each of its neighboring computers and exchanges control messages, agents, and inter-agent communications with these through the connection. When an agent is transferred over the network, the agent platform on the sending side marshals the code of the agent and its state, e.g., instance variables in Java objects, into a bit-stream and then transfers them to the destination. The agent platform on the receiving side receives and unmarshals the bit-stream so that the agent can continue to be executed at the destination.

4.1 Agent deployment management

The policy-based deployment of agents is managed by each agent host without a centralized management server. Each agent 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] The procedure involves four steps. 1) When an agent migrates to another agent host, each agent automatically registers its deployment policy with the destination host. 2) The destination host sends a query message to the source host of the visiting agent. There are two possible scenarios: the visiting agent has a policy for another agent or it is specified in another agent's policies. 3-a) Since the source host in the first scenario knows the host running the target agent specified in the visiting agent's policy , it asks the 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 agent that has migrated to another location, it forwards the message to the destination of the agent via the proxy. 3-b) In the second scenario, the source host multicasts a query message within current or neighboring subnetworks. If a host has an agent whose policy specifies the visiting agent, it sends the destination host information about itself and its neighboring hosts. 4) The destination host next instructs the visiting agent or its clone to migrate to one of the candidate destinations recommended by the target host, because this platform treats every agent as an autonomous entity. Moreover, when the capabilities of a candidate destination do not satisfy all the requirements of the agent, the agent 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 agent may go into divergence or vibration mode due to conflicts between some of an agent's policies, when it has multiple deployment policies. However, the current implementation does not exclude such divergence or vibration.[3]

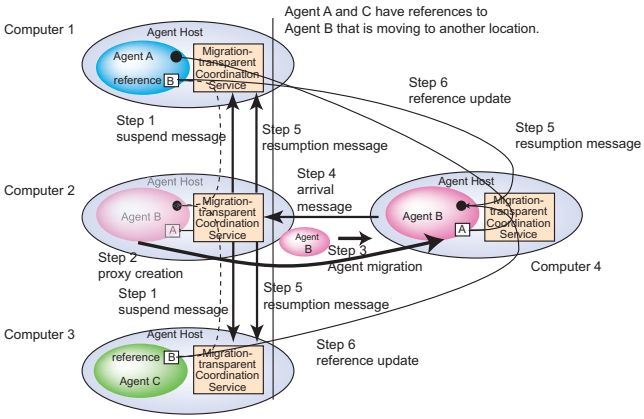

Figure 5: Forwarding messages to migrated agent

4.2 Resource management

Our agent platform enables agents to define the computational resources they require, which their destination must/should satisfy in CC/PP form. The platform systems transform the forms into corresponding LISP-like expressions and then evaluate them by using a LISP-based interpreter. When an agent migrates to the destination according to its policy, if the destination cannot satisfy the requirements of the agent, the platform system recommends candidates that are hosts in the same network domain to the agent. If an agent declares repulsive policies in addition to a gravitational policy, the platform system detects the candidates using the latter's policy and then recommends final candidates to the agent using the former policy, assuming that the agent is in each of the detected candidates.

4.3 Inter-agent communication management

Each platform 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 agents. Each platform system can maintain a database that stores pairs of identifiers of its connected agents and the network addresses of their current platform systems. It also provides agents with references to the other agents of the application federation to which it belongs. Each reference enables the

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

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

agent to interact with the agent that it specifies, even if the agents are on different hosts or move to other hosts.

Figure 5 shows an approach enabling communication between an agent moving from computer 2 to 3 and two agents at computers 1 and 3. When an agent, i.e., agent-B, requests the current platform system to migrate to another computer, the system searches its database for the network addresses of platform systems with agents, i.e., computers 1 and 4. 1) It sends *suspend* messages to these systems to block any new uplinks from them to the migrating agent with the destination's address. If the moving agent contains references, the current platform system sends the destination's address to the platform systems that are running the agents 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 agent arrives at its destination, it sends an *arrival* message with the network address of the destination to the departure platform system and then sends *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 platform systems that may hold references to the moved agent and then removes the proxy.

When an agent 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 agent creates and leaves a proxy at the departure platform system for the duration it takes the algorithm to finish. The proxy agent receives uplinks from other platform systems and forwards them to the moved agent. Since not all agents have to be tracked for other agents to communicate with them, agents can leave proxy agents along that are not just traces under their own control. Proxy agents are also programmable entities, like agents, so they can be modified based on application requirements.

4.4 Security management

The current implementation is a prototype system to dynamically deploy the agents presented in this paper. Nevertheless, it has several security mechanisms. For example, it can encrypt agents before migrating them over the network and it can then decrypt them after they arrive at their destinations. Moreover, since each agent 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 agents so that they could only access specified resources to protect computers from malicious agents. Although the current implementation cannot protect agents from malicious computers, the platform system supports authentication mechanisms to migrate agents so that all platform systems can only send agents to, and only receive from, trusted platform systems.

4.5 Remarks

The proposed framework itself does not support clock synchronization between the runtime systems running on different computers, because it is designed as just an infrastructure for implementing distributed algorithms, including bio-inspired approaches and a synchronization mechanism. The framework enables multiple runtime systems to run and interact with one another in a single computer by using existing virtual machine (VM) technologies.

5. EARLY EVALUATION

A prototype implementation of this framework was constructed with Sun's Java Developer Kit version 1.5 or later version. Although the current implementation was not constructed for performance, we evaluated the migration of two agents based on deployment policies. This experiment was done with three computers (Pentium M-1.8 GHz processor with Windows XP and JDK ver.5) connected through a Fast Ethernet network. The cost of agent migration included the costs of opening a TCP-transmission, marshaling agents, migrating them from their source computers to their destination computers, unmarshaling them, and verifying security. To evaluate the cost of migrating agents organized as a motile cell, we defined two types of agents, a head agent and a body agent, where their individual sizes were about 8 KB. One head agent and more than one body agent were organized in a sequence, where the head agent traveled at the head and each body agent declared a deployment policy for the head agent or another body agent. Agents that declared a follow or dispatch policy were deployed at the same agents and the agents declared a shift or fill policy at different computers. We measured the time that the tails of the body agents arrived at their final destinations after the head agent had started to migrate to the next destination and then divided the time by the number of tail-agent's hops. This can be viewed as a *cytoplasmic streaming* in a cell. The costs presented in this paper are the averages for ten evaluations for each of the experiments.

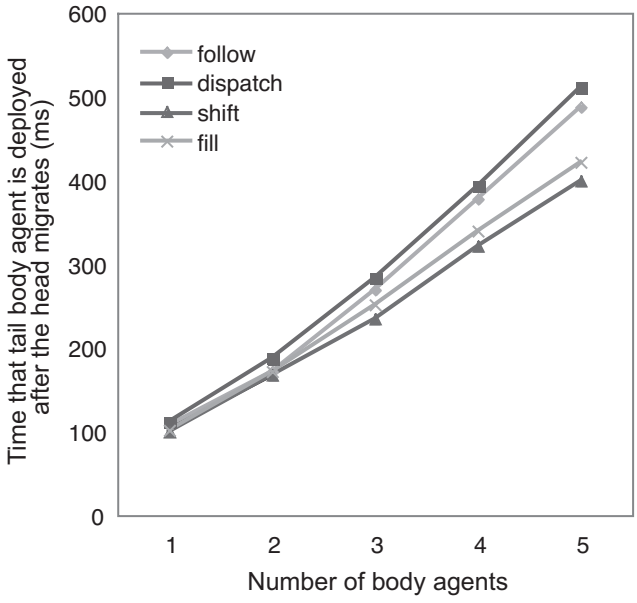

Figure 6: Costs of deploying more than one sequentially connected agent.

Figure 6 lists the costs of the follow, dispatch, shift, or fill policy where the horizontal axis is the number of body agents. The cost of agent deployment included the costs of opening a TCP-transmission, marshaling the agents, migrating the agents from their source computers to their destination computers, creating class loaders, and unmarshaling the agents in addition to the cost of managing the deployment policy. Figure 7 lists the costs of deploying more than one body agent that declares a follow, dispatch, shift, or fill policy for the head agent. That is, all the body agents have the head agent as their target.

As we can see from Fig. 6, the cost of deploying agents does not depend on the number of body agents. This is because the current implementation supports a synchronous approach, in the sense that the deployment of each agent that declares another agent is executed after the latter agent has arrived at its next destination. If the framework supported an asynchronous approach in the sense

that agents were deployed in parallel, there may be the possibility of reducing the cost of deploying agents. However, this approach often results in congestion at some computers so that it does not always reduce the cost of deploying agents.

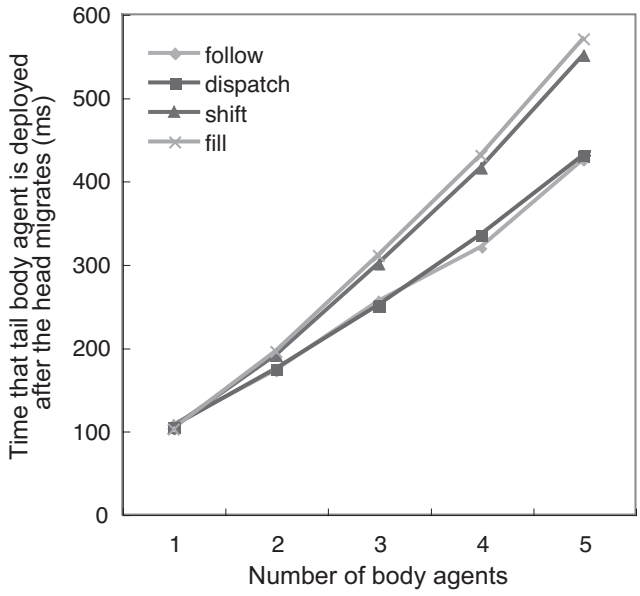

Figure 7: Cost of deploying more than one agent connected to head agent.

The cost of deploying agents in Fig. 7, on the other hand, tended to depend on the numbers of body agents, because one or more body agents were simultaneously deployed on the same computers. In fact, several body agents were often bunched at several computers, because the deployments of body agents were independent of one another after the head agent had arrived at the destination. The timing for the arrival of deploying agents tended to diverge. Since the head agent was designed to travel along its itinerary without waiting for body agents to be deployed, the head agent tended to advance separately from the body agents. Note that there are more efficient approaches, but these assume that the migrations of multiple agents are synchronized. However, such synchronization itself is a onerous task so that they are not always more efficient. We believe that the framework is highly scalable because it is executed without any centralized servers.[4]

These measured costs are more reasonable than the cost of migrating an agent between two computers. The time cost of the follow policy was less than that of the shift policy and the time cost of the dispatch policy was more than that of the fill policy. This was because the head agent and the body agents with the follow or dispatch policy were deployed at the same computers so that the cost of detecting agents that declared moving agents was small.

The platform enables us to implement dependable systems as well as bio-inspired approaches. For example, we constructed a fault-tolerant HTTP-based server to illustrate the use of these policies by combining deployment and communication policies. Each agent supports an HTTP server. Each is clonable, where the agent and its clones declare forward policies for each other and its clone declares an exclusive policy for it. When an agent 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 agents at different computers to ensure against faults. When one of these receives messages from external systems, their forward policies send the messages to the other so that they share the same states. After the agent duplicates itself, the cost of deploying its clone at another host is about 280 ms in the distribution system presented in the previous section.[5] This does not include the cost of terminating and restarting the HTTP server. The cost of forwarding a message is about 28 ms, where it is measured as the round-trip time and the message is of no value. If the agents declare delegate policies, they can support a master-slave model instead of a duplication model.

[4]We could not evaluate the framework on a large number of computers, unfortunately, due to limitations in our experiment environments.

6. EXPERIENCE

To illustrate the effectiveness of the proposed platform, we implemented and evaluated an ant-based routing mechanism, which was one of the most typical bio-inspired approaches for distributed computing, with the platform. The mechanism assumes that 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 [1, 2].

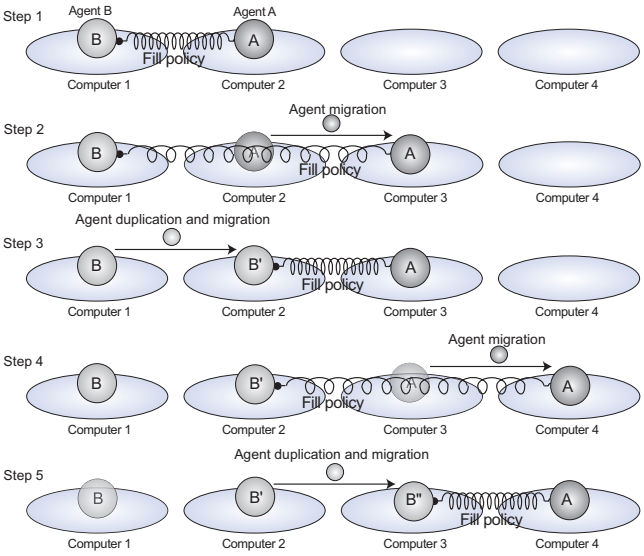

Figure 8: Agent diffusion for moving entity

The platform allows moving agents to actually leave themselves and not traces of themselves on their trails and to become automatically volatile after their lifespans are over. Agent-A corresponds to an ant and agent-B corresponding to a pheromone is attached to agent-A with the *fill* policy. As shown in Fig. 8, when agent-A randomly selects one of the current computer's neighboring computers as its destination and migrates to the selected destination, agent-B creates a clone and migrates to the source host of the latter. Since each of the cloned agents defines its lifespan by invoking `setTTL()`, they are active for a specified duration after being created. If there are other agents corresponding to pheromones in the computer, the visiting agent adds their lifespans to its own lifespan. When another agent corresponding to another ant migrates

[5]This experiment assumes that the destination of the clone is statically given.

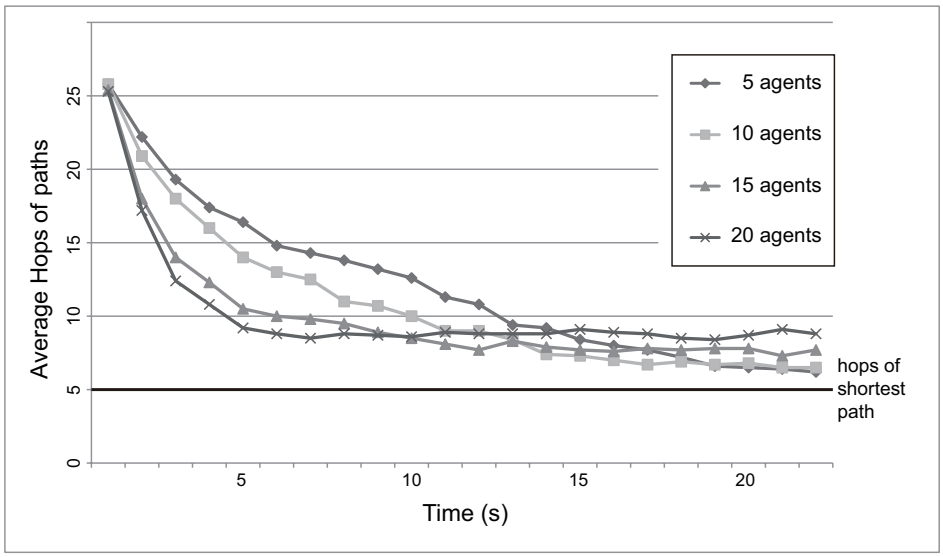

Figure 10: Convengent path with agents

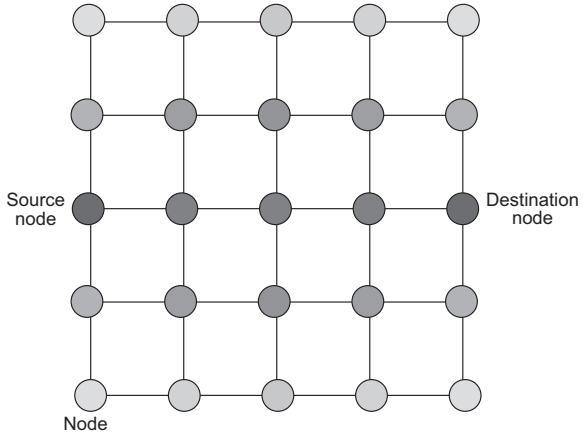

Figure 9: Experiment for ant-based routing

over the network, it can select a computer that has the agent-B with the longest lifespan from neighboring hosts. We experimented with ant-based routing for agents using this platform with more than twenty-five computers, where each had a Pentium M-1.8 GHz processor with Windows XP and JDK ver.5. These computers were connected with one another via a switching-hub for Giga-Ethernet network, but their platform systems were organized in five-by-five lattice structure and were connected to at most four of their neighboring computers (Fig. 9).

However, we knew that it would be difficult to converge a short path to the destination in real distributed systems, although existing simulation-based experiments had positive results. Figure 10 shows the convergence rate of the ant-based routing mechanism with five, ten, fifteen, and twenty A-agents. Existing simulation-based experiments have concluded that the convergence speed of the paths of ant agents to the shortest path have became faster as the number of ant agents increased. However, our experiment showed that as the number of ant agents increased, the convergence speed of the paths of ant agents were fast but the paths of ant agents were far from the shortest path. This is because the nodes that was on or closes to the shortest path tend to be congest. That is, many agents pass through such nodes. Therefore, the migration of agents were often blocked at their current nodes until the agents that previously arrived had left the nodes. The pheromones remaining at such nodes were volatilized because the number of agents that had passed through these nodes had decreased. This is an essential problem with ant-based routing mechanisms, but existing simulation-based approaches have unfortunately ignored the problem of congestion at nodes on or close to the shortest path.

7. RELATED WORK

This section discusses several bio-inspired approaches to distributed and multi-agents systems. A few attempts have provided infrastructures for real distributed systems, like ours. The Anthill project [1] by the 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 [7] and Bison [8]. Messor is a load-balancing application of Anthill and Bison is a conceptual bio-inspired platform based on Anthill. The main difference between Anthill, including its applications, and our platform is that it introduces agents as independent entities and ours permits agents to be organized in a self-organized manner. The Co-Field project [5] by the 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 usable within the limits of simulation and not within a real distributed system. Our deployment policies may be similar to the dynamic layout of distributed applications in the FarGo system [3]. However, FarGo's policies aim at allowing an agent to control others, whereas our policies aim at allowing an agent to describe its own individual migration, because our platform always treats agents as autonomous entities that travel from computer to computer under their own control. FarGo's policies may conflict

when two agents can declare different relocation policies for a single agent. However, our platform is free of any conflict because each agent can only declare a policy to relocate itself instead of other agents.

This platform was inspired by our earlier versions presented in previous papers [12, 13]. The previous papers aimed at presenting the middleware for building and operating a large-scale system as a federation of one or more mobile agents like the platform presented here, but they addressed ubiquitous computing environments whose computers were heterogenous rather than large-scale distributed systems. The previous versions offered some of the gravitational relocation policies supported by this platform, but lacked any of the repulsive policies, which are essential in supporting load-balancing and fault-tolerant mechanisms. In fact, when many agents are organized and deployed over a distributed system by only using gravitational relocation policies, they tend to gather at several computers. We believe that our platform can work in a large-scale infrastructure for overlay networking, e.g., PlanetLab. Nevertheless, the platform is designed for a large-scale infrastructure for distributed computing, i.e., grid computing and cloud computing, so that it can dynamically deploy and organize software components for computing rather than networking.

8. CONCLUSION

This paper presented a general-purpose test-bed platform for bio-inspired or self-organizing approaches to distributed systems. It was constructed as a real middleware system for dynamically deploying agents at different computers, instead of using simulation-based systems. We designed and implemented a prototype system for the middleware and demonstrated its effectiveness in several applications. Since the middleware enabled each agent to specify its own policy as a relocation between the agent and another, it could not only move individual agents but also a federation of agents over a distributed system in a self-organized manner. We evaluated one of the most typical bio-inspired approaches for distributed systems, i.e., an ant-based routing mechanism. The results from our experiment on a real distributed system were different from the result from existing simulation-based experiments, because the latter ignored many properties of real distributed systems. We believe that our platform is useful for bridging the gap between simulation-based approaches and real distributed systems.

In concluding, we would like to identify further issues that need to be resolved. We plan to evaluate existing bio-inspired approaches to distributed systems with the platform. We also proposed a specification language for the itinerary of mobile software [12]. The language enables more flexible and varied policies for deploying the agents to be defined. We plan to apply the platform to a mobile agent-based network-management system [9].

9. 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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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, August 2003.

[9] 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.

[10] I. Satoh, Configurable Network Processing for Mobile Agents on the Internet, Cluster Computing, vol. 7, no.1, pp.73-83, Kluwer, January 2004.

[11] I. Satoh, Bio-inspired Organization for Multi-agents on Distributed Systems, Proceedings of 2nd Workshop on Biologically Inspired Approaches to Advanced Information Technology (BioADIT'2006), Lecture Notes in Computer Science (LNCS), vol.3853, pp.355-362, Springer, January 2006.

[12] I. Satoh, Building and Selecting Mobile Agents for Network Management, Journal of Network and Systems Management, vol.14, no.1, pp.147-169, Springer, 2006.

[13] I. Satoh, Cell-locomotin-based Agent Migration over Distributed Systems, Proceedings of 1st International Conference on Complex, Intelligent and Software Intensive Systems (CISIS'2007), pp.74-81, IEEE Computer Society, April 2007.

[14] World Wide Web Consortium (W3C), Composite Capability/Preference Profiles (CC/PP), http://www.w3.org/TR/NOTE-CCPP, 1999.