Coordination-level Adaptation 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 proposes a framework for adapting the behaviors and coordinations of software agents to changes in distributed systems. It is unique to other existing approaches for self-adaptation because it introduces the notions of differentiation in cellular into real distributed systems. When an agent delegates a function to another agent coordinating with it, if the former has the function, this function becomes less-developed and the latter's function becomes well-developed. The framework was constructed as a middleware system and allowed us to define agents as Java objects written in JavaBean. We present several evaluations of the framework in a distributed system instead of any simulation-based systems.**

***Keywords*-Differentiation; Distributed System; Coordination.**

I. INTRODUCTION

Distributed systems are dynamic and complicated by nature and may partially have malfunctioned, e.g., network partitioning. Their scale and complexity are beyond the ability of traditional management approaches, e.g., centralized and top-down ones. Distributed systems should adapt themselves to changes in their system structures, including network topology, and the requirements of their applications. Many researchers have explored adaptations for distributed systems. The author proposed an approach to adapting behaviors in software components that a distributed application consisted of without any centralized and top-down management systems [9]. The approach was inspired by a natural adaptation, called *cellular differentiation*, which was a mechanism by which cells in a multicellular organism become specialized to perform specific functions in a variety of tissues and organs. The approach could differentiate their functions according to their roles in whole applications and resource availability, as just like cells. It involves function matching as differentiation factors in functions, where service matching is the process of comparing the function request against the available function advertisements and determining which function best satisfies the request. When a component delegates a function to another component, if the former has the function, its function becomes less-developed in the sense that it has less computational resources, e.g., active threads, and the latter's function becomes well-developed in the sense that it has more computational resources.

This paper presents the second step of our approach. The previous approach supports adaptations only at internal behaviors of software components [9], because it could not adapt coordinations between components. The proposed approach presented in this paper aimed at adaptations in coordinations between more than one component in addition to the internal behaviors of components. Although existing attempts for adaptive coordinations between computers have been explored, our approach has two notable advantages. The first is to manage adaptive coordinations without any centralized and top-down management systems. The second is keep consistency between adaptations in multiple computers, because adaptive coordinations need to be supported at more than one computer. Adaptations for distributed systems have several unique requirements that adaptaions for non-distributed systems do not have. Since a distributed system consists of multiple computers, adaptations tend to affect behaviors on more than one computer. However, adaptations may not be synchronized. Therefore, when some computers achieved their adaptations, others may not yet. In particular, adaptations for coordinations between multiple computers needs their adaptations need to be synchronized.

The remainder of this paper is structured as follows. In Section II, we present the related work. Section III discusses the basic approach and Section IV presents the design and implementation of our proposal. Section V evaluates the implementation and Section VI describes applications. Section VII gives some concluding remarks.

II. RELATED WORK

This section discusses several related studies on software adaptation in distributed systems. Existing can be classified into four types, parameter-level, software-level, location-level, and coordination-level adaptations. Although the first and second approaches are common in non-distributed systems, the third and fourth are available only in distributed systems.

The first is to adapt system parameters, e.g., durations of timeouts and the amount of available resources to changes in distributed systems. Adaptations in the type tend to be limited. The second is to adapt software for defining components running on computers. It enables behaviors of software in distributed systems to be adapted to changes in the systems. As mentioned previously, our previous approach is in the type. One of the most typical approaches of the type

 ISBN: 978-1-61208-391-9

is genetic programming [6]. The fitness of every individual program in the population need to be evaluated in each generation and multiple individuals are stochastically selected from the current population based on their fitness. However, since distributed systems may have an effect on the real world and be used for mission-critical processing, there is no chance of ascertaining the fitness of randomly generated programs. Blair et al. [1] tried to introduce self-awareness and self-healing into a CORBA-compatible Object Request Broker (ORB). Their system was a meta-level architecture with the ability of dynamically binding CORBA objects. The aim of resource management strategy is to maximize the profits of both customer agents and resource agents in large datacenters by balancing demand and supply in the market. Several researchers have addressed resource allocation for clouds by using an auction mechanism. For example, Lin et al [7] proposed a mechanism based on a sealed-bid auction. The cloud service provider collected all the users' bids and determined the price. Zhang et al. [12] introduced the notion of spot markets and proposed market analysis to forecast the demand for each spot market.

The third is location-level adaptations. Suda et al. proposed bio-inspired middleware, called Bio-Networking, for disseminating network services in dynamic and large-scale networks where there were a large number of decentralized data and services [8], [10]. Although they introduced the notion of energy into distributed systems and enabled agents to be replicated at and moved to suitable computers according to the number of service requests from clients, where the selection of computers depends on distances between agents and clients. As most of their parameters, e.g., energy, tended to depend on a particular distributed system, so that they may not have been available in other systems. Our approach should be independent of the capabilities of distributed systems as much as possible.

The fourth is coordination-level adaptations. Jaeger et al. [4] introduced the notion of self-organization to ORB and a publish/subscribe system. Georgiadis et al. [3] presented connection-based architecture for self-organizing software components on a distributed system. Like other software component architectures, they intended to customize their systems by changing the connections between components instead of internal behaviors inside the components. Cheng at al. [2] presented an adaptive selection mechanism for servers by enabling selection policies, but they did not customize the servers themselves. They also needed to execute different servers simultaneously. There have been many attempts to apply self-organization into distributed systems, e.g., a myconet model for peer-to-peer network [11]. There has been no attempts in the third and fourth types that keep consistency between adaptions at multiple computers.

III. Basic approach

This paper introduces the notion of (de)differentiation into a distributed system as a mechanism for adapting coordinations between software components, which may be running on different computers connected through a network.

Differentiation-inspired coordination-level adaptation: This mechanism was inspired by differentiation in dictyostelium discoideum. When dictyostelium discoideum cells aggregate, they can be differentiated into two types: prespore cells and prestalk cells. Each cell tries to become a prespore cell and periodically secretes chemical substance, called cAMP, to other cells. If a cell can receive more than a specified amount of the substance from other cells, it can become a prespore cell. There are three rules. 1) the substance chemotaxically leads other cells to prestalk cells. 2) A cell that is becoming a prespore cell can secrete a large amount of the substance to other cells. 3) When a cell receives more substance from other cells, it can secrete less substance to other cells.

Each agent has one or more functions with weights, where each weight corresponds to the amount of the substance and indicates the superiority of its function. Each agent initially intends to progress all its functions and periodically multicasts *restraining* messages to other agents federated with it within the domain of current networks. Restraining messages lead other agents to degenerate their functions specified in the messages and to decrease the superiority of the functions. As a result, agents complement other agents in the sense that each agent can provide some functions to other agents and delegate other functions to other agents that can provide the functions. Finally, functions that are often delegated to other agents, and then become inactive in the sense that they lose their computational resources.

Consistency in distributed adaptations: Coordination-based adaptations often need to modify protocols and application-logics in multiple computers. Such modifications are often required to be synchronized. Suppose an adaptation for coordination between two computers. While the first computer achieved its modification for the adaptation and another does not yet, if coordination between them happen, their coordination may be inconsistent, because the protocol or application-logic of one computer does not match with that of the another. To solve this problem, we introduce a synchronization mechanism for blocking coordinations among computers until the computers that do the coordinations complete their adaptations.

IV. Design and Implementation

Our approach is maintained through two parts: runtime systems and agents. The former is a middleware system for running on computers and the latter is a self-contained and autonomous software entity. It has three protocols for (de)differentiation and delegation.

 ISBN: 978-1-61208-391-9

A. Adaptive agent

Each agent is an autonomous programmable entity and consists of one or more functions, called the *behavior* parts, and its state, called the *body* part, with information for (de)differentiation, called the *attribute* part. These parts are implemented as a set of Java objects. We can define each agent as a single JavaBean, where each method in JavaBean needs to access the database maintained in the body parts.

- The body part maintains program variables shared by its behaviors parts like instance variables in object orientation. When it receives a request message from an external system or other agents, it dispatches the message to the behavior part that can handle the message.
- The behavior part defines more than one application-specific behavior. It corresponds to a method in object orientation. As in behavior invocation, when a message is received from the body part, the behavior is executed and returns the result via the body part.
- The attribute part maintains descriptive information with regard to the agent, including its own identifier. The attributes contains a database for maintaining the weights of its own behaviors and for recording information on the behaviors that other agents can provide.

The agent has behaviors $b_1^k, \ldots, b_n^k$ and w_i^k is the weight of behavior b_i^k. Each agent (k-th) assigns its own maximum to the total of the weights of all its behaviors. The W_i^k is the maximum of the weight of behavior b_i^k in k-th agent. The maximum total of the weights of its behaviors in the k-th agent must be less than W^k. ($W^k \geq \sum_{i=1}^{n} w_i^k$), where $w_j^k - 1$ is 0 if w_j^k is 0. The W^k may depend on agents. In fact, W^k corresponds to the upper limit of the ability of each agent and may depend on the performance of the underlying system, including the processor. Note that we never expect that the latter will be complete, since agents periodically exchange their information with neighboring agents. Furthermore, when agents receive no retraining messages from others for longer than a specified duration, they remove information about them.

The approach offers two communication policies for intercomponent interactions.

- 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* policies.

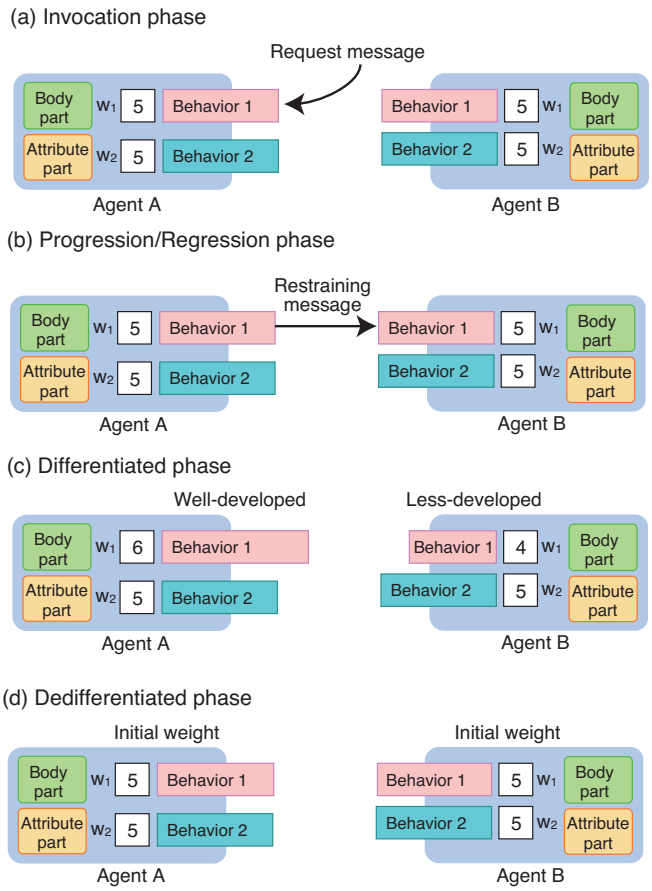

Figure 1. Differentiation mechanism for agent

B. Adaptive coordination

Next, we describe our differentiation-inspired adaptation mechanism.

1) Removing redundant functions: Behaviors in an agent, which are delegated from other agents more times, are well developed, whereas other behaviors, which are delegated from other agents fewer times, in a cell are less developed. Finally, the agent only provides the former behaviors and delegates the latter behaviors to other agents.

1: When an agent (k-th agent) receives a request message from another agent, it selects the behavior (b_i^k) that can handle the message from its behavior part and dispatches the message to the selected behavior (Figure 1 (a)).
2: It executes the behavior (b_i^k) and returns the result.
3: It increases the weight of the behavior, w_i^k.
4: It multicasts a restraining message with the signature of the behavior, its identifier (k), and the behavior's weight (w_i^k) to other agents (Figure 1 (b)).

The key idea behind this approach is to distinguish between internal and external requests. When behaviors are invoked

by their agents, their weights are not increased. If the total weights of the agent's behaviors, $\sum w_i^k$, is equal to their maximal total weight W^k, it decreases the minimal (and positive) weights (w_j^k is replaced by $w_j^k - 1$ where $w_j^k = \mathbf{min}(w_1^k, \ldots, w_n^k)$ and $w_j^k \geq 0$). The above phase corresponds to the degeneration of agents.

1: When an agent (k-th agent) receives a restraining message with regard to b_i^j from another agent (j-th) , it looks for the behaviors ($b_m^k, \ldots b_l^k$) that can satisfy the signature specified in the receiving message.
2: If it has such behaviors, it decreases their weights ($w_m^k, \ldots w_l^k$) and updates the weight (w_i^j) (Figure 1 (c)).
3: If the weights ($w_m^k, \ldots, w_l^k$) are under a specified value, e.g., 0, the behaviors ($b_m^k, \ldots b_l^k$) are inactivated.

C. Invocation of functions

When an agent wants to execute a behavior, even if it has the behavior, it needs to select one of the behaviors, which may be provided by itself or others, according to the values of their weights.

1: When an agent (k-th agent) wants to execute a behavior, b_i , it looks up the weight (w_i^k) of the same or compatible behavior and the weights ($w_i^j, \ldots, w_i^m$) of such behaviors ($b_i^j, \ldots, b_i^m$).
2: If multiple agents, including itself, can provide the wanted behavior, it selects one of the agents according to selection function ϕ^k, which maps from w_i^k and $w_i^j, \ldots, w_i^m$ to b_i^l, where l is k or $j, \ldots, m$.
3: It delegates the selected agent to execute the behavior and waits for the result from the agent.

The approach permits each agent to use its own evaluation function, ϕ, because the selection of behaviors often depends on its application and coordination. Although there is no universal selection function for mapping from behaviors' weights to at most one appropriate behavior like a variety of creatures, we can provide several functions.

D. Increasing resources for busy functions

The approach also provides a mechanism for duplicating agents, including their states, e.g., instance variables, as well as their program codes and deploying a clone at a runtime system. It permits each agent (k-th agent) to create a copy of itself when the total weights ($\sum_{i=1}^{n} w_i^k$) of functions ($b_1^k, \ldots, b_n^k$) provided in itself is the same or more than a specified value. The sum of the total weights of the original agent and those of the clone agent is equal to the total weights of the original agent before the agent is duplicated. The current implementation supports two conditions. The first permits each agent (k-th) to create a clone of it when the total of its weights ($\sum_{i=1}^{n} w_i^k$) is more than its maximal total weight W^k and the second condition is twice that of the total initial weights of the functions. When a busy agent running as a user program has no access resources, it allocates resources to the clone agent via the external control system.

E. Releasing resources for redundant functions

Each agent (j-th) periodically multicasts messages, called *heartbeat messages*, for a behavior (b_i^j), which is still activated with its identifier (j) via the runtime system. When an agent (k-th) does not receive any heartbeat messages with regard to a behavior (b_i^j) from another agent (j-th) for a specified time, it automatically decreases the weight (w_i^j) of the behavior (b_i^j), and resets the weight (w_i^k) of the behavior (b_i^k) to be the initial value or increases the weight (w_i^k) (Figure 1 (d)). The weights of behaviors provided by other agents are automatically decreased without any heartbeat messages from the agents. Therefore, when an agent terminates or fails, other agents decrease the weights of the behaviors provided by the agent and if they then have the same or compatible behaviors, they can activate the behaviors, which may be inactivated.

F. Consistent Adaptation based on Primary-Backup Protocol

Our framework uses a primary-backup scheme to maintain consistent states between one primary server and replica servers for adaptation. A primary server receives all incoming client requests, executes them, and propagates the resulting to the backup replica severs. To detect failures in the primary and replica servers, they periodically sends heartbeat messages to one another.

- When the primary server crashes, some of the replica servers detects the inactivation of the primary because they cannot receive any heartbeat messages from the primary. They execute a recovery protocol both to agree upon a common consistent state before resuming regular operation and to establish a new primary to broadcast state changes. To exercise the primary role, a replica server must have the support of a quorum of processes. As replica servers can crash and recover, there can be over time multiple primaries and in fact the same replica server may exercise the primary role multiple times.
- When replica servers crash, the primary detects the inactivation of the servers because it cannot receive any heartbeat messages from them. It removes them from its list of replica servers. When they become activated, it sends the latest updates of the state that were adapted after they crash.

We associate an instance value with each established primary. A given instance value maps to at most one replica server.

Each runtime system is constructed as a middleware system with Java (Figure 2). It is responsible for executing agents and for exchanging messages in runtime systems on other computers through a network. When a runtime system

is (re)connected to a network, it multicasts heartbeat messages to other runtime systems to advertise itself, including its network address in a plug-and-play protocol manner.

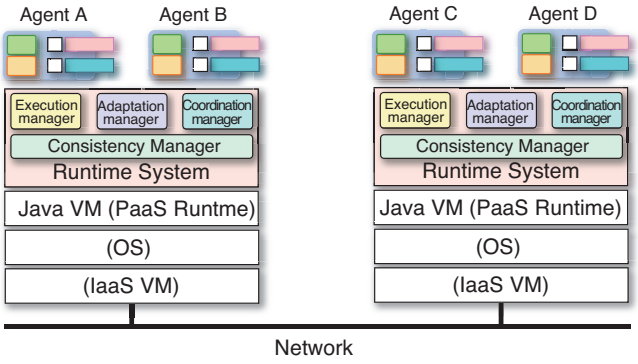

Figure 2. Runtime system

Adaptation messages, i.e., restraining and heartbeat messages, are transmitted as multicast UDP packets, which are unreliable. When the runtime system multicasts information about the signature of a behavior in restraining messages, the signature is encoded into a hash code by using Java's serial versioning mechanism and is transmitted as code. Restraining messages for behaviors that do not arrive at agents are seriously affected, because other agents automatically treat the behaviors provided by the senders to be inactive when they do not receive such messages for certain durations. Since our mechanism does not assume that each agent has complete information about all agents, it is available even when some heartbeat messages are lost.

Application-specific messages, i.e., request and reply, are implemented through TCP sessions as reliable communications. When typical network problems occur, e.g., network partitioning and node failure during communication, the TCP session itself can detect such problems and it notifies runtime systems on both sides to execute the exception handling defined in runtime systems or agents. The current implementation supports a multiplexing mechanism to minimize communication channels between agents running on two computers on at most TCP session. To avoid conflicts between UDP packets, it can explicitly change the periods of heartbeat messages issued from agents. 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.

V. Evaluation

Although the current implementation was not constructed for performance, we evaluated that of several basic operations in a distributed system where eight computers (Intel Core 7i Duo 2.8 GHz with MacOS X 10.9 and J2SE version 7) were connected through a giga-ethernet. The cost of transmitting a heartbeat or restraining message through UDP multicasting was 11 ms. The cost of transmitting a request message between two computers was 21 ms through TCP. These costs were estimated from the measurements of round-trip times between computers. We assumed in the following experiments that each agent issued heartbeat messages to other agents every 100 ms through UDP multicasting.

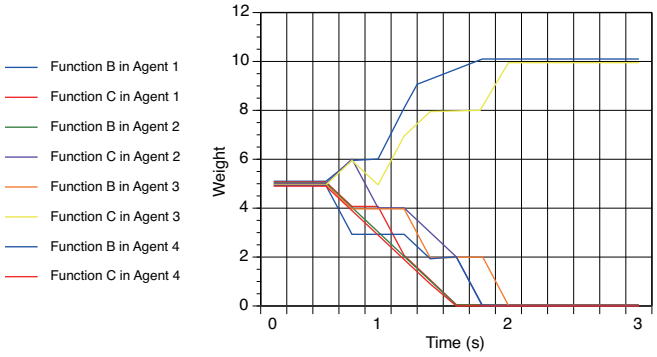

Figure 3. Degree of progress in differentiation-based adaptation

The experiment was carried out to evaluate the basic ability of agents to differentiate themselves through interactions in a reliable network. Each agent had three behaviors, called A, B, and C. The A behavior periodically issued messages to invoke its B and C behaviors or those of other agents every 200 ms and the B and C behaviors were null behaviors. Each agent that wanted to execute a behavior, i.e., B or C, selected a behavior whose weight had the highest value if its database recognized one or more agents that provided the same or compatible behavior, including itself. When it invokes behavior B or C and the weights of its and others behaviors were the same, it randomly selected one of the behaviors. We assumed in this experiment that the weights of the B and C behaviors of each agent would initially be five and the maximum of the weight of each behavior and the total maximum W^k of weights would be ten.

Figure 3 presents the results we obtained from the experiment. Both diagrams have a timeline in minutes on the x-axis and the weights of behavior B in each agent on the y-axis. Differentiation started after 200 ms, because each agent knows the presence of other agents by receiving heartbeat messages from them. Figure 3 shows the detailed results of our differentiation between four agents, where their weights were not initially varied and then they forked into progression and regression sides.

 ISBN: 978-1-61208-391-9

VI. APPLICATION

We describe a practical application with this approach to illustrate the utility of our (de)differentiation for service deployment and composition in a disaggregated computing setting. Our application is initially a single drawing editor service at a server and then automatically duplicates, deploys, and differentiates the service and its clones at different computers. Each service is defined based on a model-view-control (MVC) pattern as an an agent consisting of *model*, *view*, and *control* behaviors. The first manages and stores drawing data and should be executed on a computer equipped with a powerful processor and a lot of memory. The second part displays drawing data on the screen of its current host and should be deployed at computers equipped with large screens. The third part forwards drawing data from the pointing device, e.g., mouse, of its current computer to the first behavior.

When the server is connected to a network, the agent automatically introspects the capabilities of computing devices connected to the network via the current runtime system. When it discovers a computer equipped with a pointing device and a large display, e.g., a smart TV, the agent makes a clone of it with its behaviors and deploys the clone agent at the smart TV. The original agent, which is running on the server, decreases the weights of its behaviors corresponding to the view and control parts and the clone agent, which is running on the smart TV, decreases the weight of its behavior corresponding to the model part. This is because the server has no display or pointing device and the smart TV had no storage device. Therefore, each of the agents delegates the behaviors that its computer does not support to another agent. As a result, their weights for behaviors monotonously increases or decreases and they are then successfully differentiated according to the capabilities of their current computers. A user could view pictures stored in the server on the screen of a smart TV.

When a user disconnects the server from the network, the agent running on the server dedifferentiates itself, because it lacks its co-partners, to which it delegates the behaviors corresponding the view and control parts. When it connects to another network with another smart TV, it can clone itself and differentiate itself and the clone. However, since the agent running on the smart TV has no data, it does not invoke inactive behavior, which corresponds to the model part, and is then terminated.

Although this may be carried out by using non-differentiation approaches, this approach had several advantages. For example, it has no central management system so that it can avoid single points in performance bottlenecks and failures. It make our management tasks easier. That is, after we just deploy only one agent at a computer, the approach enables the agent to automatically duplicate, deploy, and adapt itself and its clones according to the capabilities of computers and the demands of its applications.

VII. CONCLUSION

This paper proposed a framework for adapting software agents, which coordinate with one another, on distributed systems. It is unique to other existing software adaptations in introducing the notions of differentiation and cellular division in cellular slime molds, e.g., dictyostelium discoideum, into software agents. When an agent delegates a function to another agent, if the former has the function, its function becomes less-developed and the latter's function becomes well-developed. When agents have many requests from other agents, they create their clone agents. The framework was constructed as a middleware system on real distributed systems instead of any simulation-based systems. Agents can be composed of Java objects.

REFERENCES

[1] G. S. Blair, at al., Reflection, self-awareness and self-healing in OpenORB, in Proceedings of 1st Workshop on Self-healing systems (WOSS'2002), ACM Press, 2002, pp.9–14.

[2] S. Cheng, D. Garlan, B. Schmerl, Architecture-based self-adaptation in the presence of multiple objectives, in Proceedings of International Workshop on Self-adaptation and Self-managing Systems (SEAMS'2006), ACM Press, 2006, pp.2–8.

[3] I. Georgiadis, J. Magee, and J. Kramer, Self-Organising Software Architectures for Distributed Systems in Proceedings of 1st Workshop on Self-healing systems (WOSS'2002), ACM Press, 2002, pp.33–38.

[4] M. A. Jaeger, H. Parzyjegla, G. Muhl, K. Herrmann, Self-organizing broker topologies for publish/subscribe systems, in Proceedings of ACM symposium on Applied Computing (SAC'2007), ACM, 2007, pp.543–550.

[5] J.R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, 1992.

[6] W. Lin, G. Lin, and H. Wei, Dynamic Auction Mechanism for Cloud Resource Allocation In Proceedings of 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid'2010), 2010, pp.591–592.

[7] T. Nakano and T. Suda, Self-Organizing Network Services With Evolutionary Adaptation, IEEE Transactions on Neural Networks, vol.16, no.5, 2005, pp.1269–1278.

[8] I. Satoh, Evolutionary Mechanism for Disaggregated Computing, In Proceedings of 6th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS'2012), IEEE Computer Society 2012, pp.343–350.

[9] T. Suda and J. Suzuki: A Middleware Platform for a Biologically-inspired Network Architecture Supporting Autonomous and Adaptive Applications. IEEE Journal on Selected Areas in Communications, vol.23, no.2, 2005, pp.249–260.

[10] P. L. Snyder, R. Greenstadt, G. Valetto, Myconet: A Fungi-Inspired Model for Superpeer-Based Peer-to-Peer Overlay Topologies, in Proceedings of 3rd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO'2009), 2009, pp.40–50.

[11] Q. Zhang, E. Gurses, R. Boutaba, and J. Xiao, Dynamic resource allocation for spot markets in clouds, in Proceedings of 11th USENIX Conference on Hot topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE'2011), USENIX Association, 2011.

 ISBN: 978-1-61208-391-9