Adaptive Distributed Systems with Cellular Differentiation Mechanisms

Ichiro Satoh(✉)

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

Abstract. This paper proposes a bio-inspired middleware for self-adaptive software agents on distributed systems. It is unique to other existing approaches for software adaptation because it introduces the notions of differentiation, dedifferentiation, and cellular division in cellular slime molds, e.g., dictyostelium discoideum, 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.

1 Introduction

Self-adaptiveness is useful in distributed systems, because 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. This paper presents a bio-inspired self-tuning approach for adapting software components that a distributed application consists of without any centralized and top-down management systems. It is characterized in introducing *cellular differentiation* into distributed systems. It is the mechanism by which cells in a multicellular organism become specialized to perform specific functions in a variety of tissues and organs. It is impossible for us to expect what functions software components should have and how computational resources should be assigned to software components. This is because distributed systems are dynamic and may partially have malfunctioned, e.g., network partitioning. Our middleware system aims at building and operating distributed applications consisting of self-adapting/tuning software components, called agents, to differentiate their functions according to their roles in whole applications and resource availability, as just like cells. It involves treating the undertaking/delegation of functions in agents from/to other agents as their differentiation factors. When an agent delegates a function to another agent, 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.

2 Related Work

This section discusses several related studies on software adaptation in distributed systems. One of the most typical self-organization approaches to distributed systems is

P.C. Vinh et al. (Eds.): ICTCC 2014, LNICST 144, pp. 171–180, 2015.
DOI: 10.1007/978-3-319-15392-6_17

swarm intelligence [2,3]. Although there is no centralized control structure dictating how individual agents should behave, interactions between simple agents with static rules often lead to the emergence of intelligent global behavior. There have been many attempts to apply self-organization into distributed systems, e.g., a myconet model for peer-to-peer network [10], and a cost-sensitive graph structure for coordinated replica placement [4]. Most existing approaches only focus on their target problems or applications but are not general purpose, whereas distributed systems have a general-purpose infrastructure. Our software adaptation approach should be independent of applications. Furthermore, most existing self-organization approaches explicitly or implicitly assume a large population of agents or boids. However, since the size and structure of real distributed systems have been designed and optimized to the needs of their applications, the systems have no room to execute such large numbers of agents.

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 [5] 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.

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 [8,11]. Although they introduced the notion of energy into distributed systems and enabled agents to be replicated, moved, and deleted according to the number of service requests, they had no mechanism to adapt agents' behavior unlike ours. 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.[1] Our approach should be independent of the capabilities of distributed systems as much as possible.

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. The project provided bio-inspired frameworks, called Messor [6] and Bison [7]. Messor is a load-balancing application of Anthill and Bison is a conceptual bio-inspired framework based on Anthill.

3 Basic Approach

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

Differentiation: When dictyostelium discoideum cells aggregate, they can be differentiated into two types: prespore cells and prestalk cells. Each cell tries to become a

[1] For example, they implicitly assumed a quantitative relation between the costs of agent processing and migration, but such a relation depends on individual distributed systems.

prespore cell and periodically secretes cAMP to other cells. If a cell can receive more than a specified amount of cAMP from other cells, it can become a prespore cell. There are three rules. 1) cAMP chemotaxically leads other cells to prestalk cells. 2) A cell that is becoming a prespore cell can secrete a large amount of cAMP to other cells. 3) When a cell receives more cAMP from other cells, it can secrete less cAMP to other cells.

Each agent has one or more functions with weights, where each weight corresponds to the amount of cAMP 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. 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.

Dedifferentiation: Agents may lose their functions due to differentiation as well as be busy or failed. The approach also offers a mechanism to recover from such problems based on dedifferentiation, which a mechanism for regressing specialized cells to simpler, more embryonic, unspecialized forms. As in the dedifferentiation process, if there are no other agents that are sending restraining messages to an agent, the agent can perform its dedifferentiation process and strengthen their less-developed or inactive functions again.

4 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.

4.1 Agent

Each agent 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.

- 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 is returned 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. 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 certain duration, they remove information about them.

4.2 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 2 (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 2 (b)). [2]

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 one of 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 2 (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.

[2] Restraining messages correspond to cAMP in differentiation.

4.3 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 agents to use their own evaluation functions, ϕ, because the selection of behaviors often depends on their applications. 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.

4.4 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 2 (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.

4.5 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 different VM in IaaS or a runtime system in PaaS. 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 mother agent and those of the daughter agent is equal to the total weights of the mother 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 in PaaS has no access resources, it allocates resources to the daughter agent via the external control system.

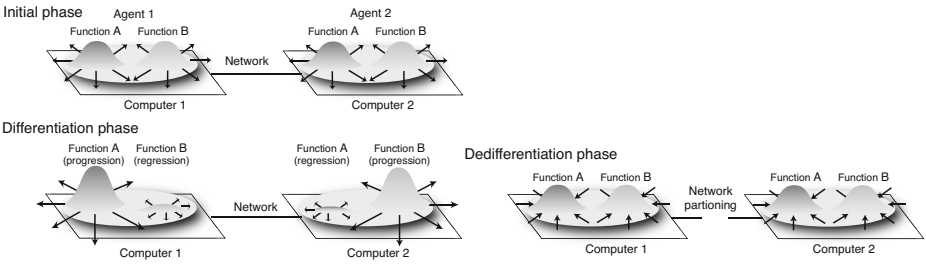

Fig. 1. Differentiation mechanism for software configuration

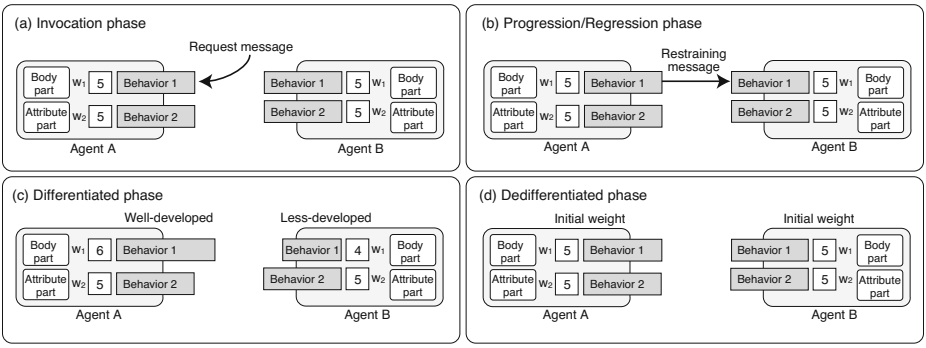

Fig. 2. Differentiation mechanism for agent

5 Experiment

To evaluate our proposed approach, we constructed it as a middleware system with Java (Figure 3), which can directly runs on Java-based PaaS runtime systems or Java VM running on VMs in IaaS, e.g., Amazon EC2. It is responsible for executing duplicating, and deploying agents based on several technologies for mobile agent platforms [9]. It is also responsible for executing agents and for exchanging messages in runtime systems on other IaaS VMs or PaaS runtime systems through TCP and UDP protocols. Each runtime system multicasts heartbeat messages to other runtime systems to advertise itself, including its network address through UDP multicasts.

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 even available when some heartbeat messages are lost.

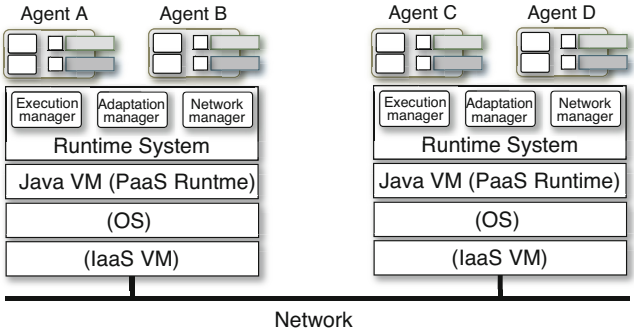

Fig. 3. Runtime system

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 a TCP session. To avoid conflicts between UDP packets, it can explicitly change the periods of heartbeat messages issued by agents.

Each agent is an autonomous programmable entity. The body part maintains a key-value store database, which is implemented as a hashtable, shared by its behaviors. We can define each agent as a single JavaBean, where each method in JavaBean needs to access the database maintained in the body parts. Each method in such a JavaBean-based agent is transformed into a Java class, which is called by another method via the body part, by using a bytecode-level modification technique before the agent is executed. Each body part is invoked from agents running on different computers via our original remote method invocation (RMI) mechanism, which can be automatically handled in network disconnections unlike Java's RMI library. The mechanism is managed by runtime systems and provided to agents to support additional interactions, e.g., one-way message transmission, publish-subscription events, and stream communications.

Since each agent records the time the behaviors are invoked and the results are received, it selects behaviors provided in other agents according to the average or worst response time in the previous processing. When a result is received from another agent, the approach permits the former to modify the value of the behavior of the latter under its own control. For example, agents that want to execute a behavior quickly may increase the weight of the behavior by an extra amount, when the behavior returns the result too soon.

6 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 2 Duo 1.83 GHz with MacOS X 10.6 and J2SE version 6) 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 22 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.

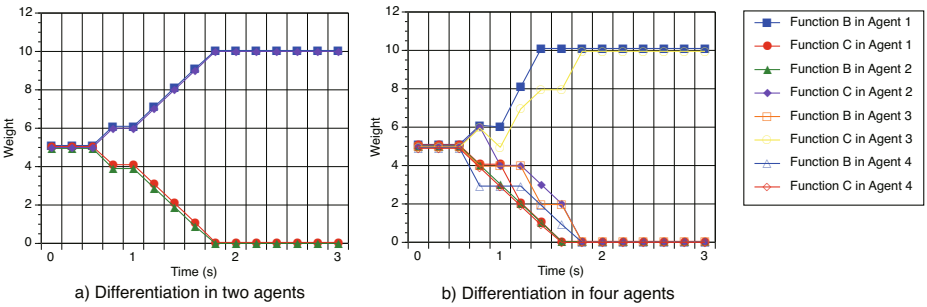

Fig. 4. Degree of progress in differentiation-based adaptation

The first 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 4 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 4 (a) details the results obtained from our differentiation between two agents. Their weights were not initially varied and then they forked into progression and regression sides. Figure 4 (b) shows the detailed results of our differentiation between four agents and Figure 4 (c) shows those of that between eight agents. The results in (b) and (c) fluctuated more and then converged faster than those in (a), because the weights of behaviors in four are increased or decreased more than those in two agents. Although the time of differentiation depended on the period of invoking behaviors, it was independent of the number of agents. This is important to prove that this approach is scalable.

Our parameters for (de)differentiation were basically independent of the performance and capabilities of the underlying systems. For example, the weights of behaviors are used for relatively specifying the progression/repression of these behaviors.

The second experiment was carried out to evaluate the ability of the agents to adapt to two types of failures in a distributed system (5). The first corresponded to the termination of an agent and the second to the partition of a network. We assumed in the following experiment that three differentiated agents would be running on different computers and each agent had four behaviors, called A, B, C, and D, where the A behavior invokes other behaviors every 200 ms. The maximum of each behavior was ten and the agents' total maximum of weights was twenty. The initial weights of their behaviors (w_B^i, w_C^i, w_D^i) in i-th agent were $(10,0,0)$ in the first, $(0,10,0)$ in the second, and $(0,0,10)$ in the third.

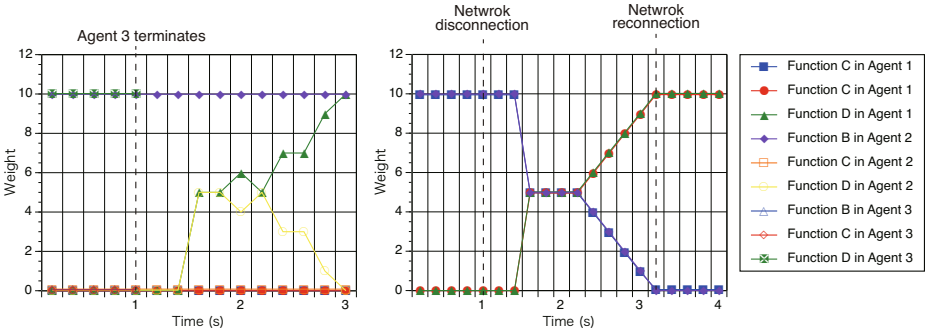

Fig. 5. Degree of progress in adaptation to failed agent

7 Conclusion

This paper proposed a framework for adapting software agents on distributed systems. It is unique to other existing software adaptations in introducing the notions of (de)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 daughter agents. The framework was constructed as a middleware system on real distributed systems instead of any simulation-based systems. Agents can be composed from Java objects. We are still interesting in reducing the number of messages for adaptation like quorum sensing in cells.

References

1. Babaoglu, O., Meling, H., Montresor, A.: Anthill: a framework for the development of agent-based peer-to-peer systems. In: Proceeding of 22th IEEE International Conference on Distributed Computing Systems (July 2002)
2. Bonabeau, E., Dorigo, M., Theraulaz, G.: Swarm Intelligence: From Natural to Artificial Systems. Oxford University Press (1999)
3. Dorigo, M., Stutzle, T.: Ant Colony Optimization. MIT Press (2004)
4. Herrman, K.: Self-organizing replica placement - a case study on emergence. In: Proceedings of 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007), pp. 13–22. IEEE Computer Society (2007)
5. Lin, W., Lin, G., Wei, H.: Dynamic auction mechanism for cloud resource allocation. In: Proceedings of 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid 2010), pp. 591–592 (2010)
6. Montresor, A., Meling, H., Babaoğlu, Ö.: Messor: load-balancing through a swarm of autonomous agents. In: Moro, G., Koubarakis, M. (eds.) AP2PC 2002. LNCS (LNAI), vol. 2530, pp. 125–137. Springer, Heidelberg (2003)
7. Montresor, A., Babaoglu, O.: Biology-inspired approaches to peer-to-peer computing in BISON. In: Proceedings of International Conference on Intelligent System Design and Applications, Oklahoma (August 2003)
8. Nakano, T., Suda, T.: Self-Organizing Network Services With Evolutionary Adaptation. IEEE Transactions on Neural Networks **16**(5), 1269–1278 (2005)
9. Satoh, I.: Mobile Agents. In: Handbook of Ambient Intelligence and Smart Environments, pp. 771–791. Springer (2010)
10. Snyder, P. L., Greenstadt, R., Valetto, G.: 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), pp. 40–50 (2009)
11. Suda, T., Suzuki, J.: A Middleware Platform for a Biologically-inspired Network Architecture Supporting Autonomous and Adaptive Applications. IEEE Journal on Selected Areas in Communications **23**(2), 249–260 (2005)
12. Zhang, Q., Gurses, E., Boutaba, R., Xiao, J.: 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)