An Approach for Recovering Distributed Systems from Disasters

Ichiro Satoh(✉)

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

Abstract. This paper presents an approach to recovering distributed applications, which consist of software agents running on different computers from drastic damages by disasters. The approach is inspired from regeneration mechanisms in living things, e.g., tails of lizards. 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 like differentiation processes in cells. It can also initialize and restart differentiated software agents, when some agents cannot be delegated like regeneration processes. It is constructed as a general-purpose and practical middleware system for software agents on real distributed systems consisting of embedded computers or sensor nodes.

1 Introduction

Hundreds of natural disasters occur in many parts of the world every year, causing billions of dollars in damages. This fact may contrast with the availability of distributed systems. Distributed systems are often treated to be dependable against damages, because in distributed systems data can be stored and executed at multiple locations and processing must not be performed by only one computer. However, all existing distributed systems are not resilient to damages in the sense that if only one of the many computers fails, or if a single network link is down, the system as a whole may become unavailable. Furthermore, in distributed systems partially damaged by disasters surviving computers and networks have no ability to fill functions lost with damaged computers or networks.

On the other hand, several living things, including vertebrates, can *regenerate* their lost parts, where *regeneration* is one of developmental mechanisms observed in a number of animal species, e.g., lizard, earthworm, and hydra, because regeneration enables biological systems to recover themselves against their grave damages. For example, reptiles and amphibians can partially regenerate their tails, typically over a period of weeks after cutting the tails. *Regeneration* processes are provided by (de)differentiation mechanism by which cells in a multicellular organism become specialized to perform specific functions in a variety of tissues and organs. The key idea behind the approach proposed in this paper was inspired from *(de)differentiation* as a basic mechanism for regeneration like living things. The approach introduces a (de)differentiation mechanism

© Springer International Publishing AG, part of Springer Nature 2018
P. Korošec et al. (Eds.): BIOMA 2018, LNCS 10835, pp. 270–282, 2018.
https://doi.org/10.1007/978-3-319-91641-5_23

into middleware systems for distributed systems, instead of any simulation-based approaches.[1]

Our middleware system aims at building and operating distributed applications consisting of self-adapting/tuning software components, called agents, to regenerate/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 Example Scenario

Let us suppose a sensor network to observe a volcano. Its sensor nodes are located around the volcano. Each of the nodes have sensors to measure accelerations result from volcano tectonic earthquakes around it in addition to processors and wired or wireless network interfaces. The locations of sensor nodes tend to be irregular around the volcano.

A disaster may result in drastic damages in sensor networks. For example, there are several active or dormant volcanoes in Japan. Sensor networks to detect volcano ash and tremor are installed at several spots in volcanoes. Volcanic eruptions, including phreatic eruptions, seriously affect such sensor networks. More han half sensor nodes may be damaged by eruptions. Nevertheless, the sensor networks should continue to monitor volcano tectonic earthquakes with only their surviving nodes as much as possible.

Sensor nodes in a volcano are located irregularly, because it is difficult for people to place such nodes at certain positions in volcanoes, because there are many no-go zones and topographical constraints. Instead, they are distributed from manned airplanes or unmanned ones. Therefore, they tend to be overpopulated in several areas in the sense that the coverage areas of their sensors are overlap or contained. To avoid congestion in networks as well as to save energy consumption, redundant nodes should be inactivated.

3 Requirements

To support example scenarios discussed in the previous section, our approach needs to satisfy the following requirements: *Self-adaptation* is needed when environments and users' requirements change. To save computational resources and energy, distributed systems should adapt their own functions to changes in their systems and environments. *Saving resources* is important in distributed systems used in field, e.g., sensor networks, rather than data centers, including cloud

[1] There is often a gap between the real systems and simulations. We believe that adaptive distributed systems need more experiences in the real systems.

computing. Our approach should conserve limited computational resources, e.g., processing, storage resources, networks, and energy, at nodes as much as possible. *Non-centralized management* can support reliability and availability. Centralized management may be simple but can become a single point of failures. Therefore, our adaptation should be managed in a peer-to-peer manner. Distributed systems essentially lack no global view due to communication latency between computers. Software components, which may be running on different computers, need to coordinate them to support their applications with partial knowledge about other computers. Our approach should be practical so that it is implemented as a general-purpose middleware system. This is because applications running on distributed systems are various. Each of software components should be defined independently of our adaptation mechanism as much as possible. As a result, developers should be able to concentrate their application-specific processing.

4 Approach: Regeneration and Differentiation

The goal of the proposed approach is to introduce a *regeneration* mechanism into distributed systems like living things. Regenerations in living things need redundant information in the sense that each of their cells have genes as plans for other cells. When living things lose some parts of their bodies, they can regenerate such lost parts by encoding genes for building the parts with differentiation mechanisms. Differentiation mechanisms can be treated as selections of parts of genes to be encoded. Since a distributed application consists of software components, which may be running on different computers like cells, we assume that software components have program codes for functions, which they do not initially provide and our differentiation mechanisms can select which functions should be (in)activated or well/less-developed.

Each software component, called agent, has one or more functions with weights, where each weight indicates the superiority and development of its function in the sense that the function is assigned with more computational resources. Each agent initially intends to progress all its functions and periodically multicasts messages about its differentiation to other agents of which its distributed application consist. Such 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.

5 Design

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 regeneration/differentiation.

5.1 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.

5.2 Regeneration

We outline our differentiation processes for regeneration (Fig. 1). The Appendix describes the processes in more detail.

- *Invocation of behaviors:* Each agent periodically multicasts messages about the weights of its behaviors to other agents. When an agent wants to execute a behavior, even if it has the behavior, it compares the weights of the same or compatible behaviors provided in others and it. It select one of the behaviors, whose weights are the most among the weights of these behaviors. That is, the approach selects more developed behaviors than less developed behaviors.
- *Well/Less developing behaviors:* When a behavior is executed by other agents, the weight of the behavior increase and the weights of the same or behaviors provided from others decrease. That is, 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.
- *Removing redundant behaviors:* The agent only provides the former behaviors and delegates the latter behaviors to other agents. Finally, when the weights of behaviors are zero, the behaviors become dormant to save computational resources.
- *Increasing resources for busy behaviors:* Each agent can create a copy of itself when the total weights of functions 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.
- *Reactivating dormant behaviors:* When an agent does not receive messages about the weights of behaviors provided in agents, treats such behaviors to be lost. When it has the same or compatible behaviors, which are dormant, it resets the wights of the behaviors, to their initial values. Therefore, they are regenerated and differentiated according to the above process again.

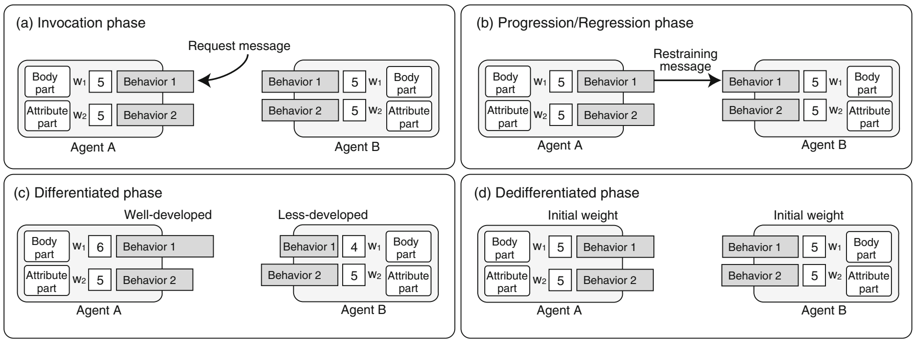

Fig. 1. Regeneration in agents

6 Implementation

To evaluate our proposed approach, we constructed it as a middleware system with Java (Fig. 2), which can directly runs on 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. 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. Messages for exchanging information about the weights of differentiation are transmitted as multicast UDP packets. Application-specific messages for invoking methods corresponding to behaviors in agents are implemented through TCP sessions.

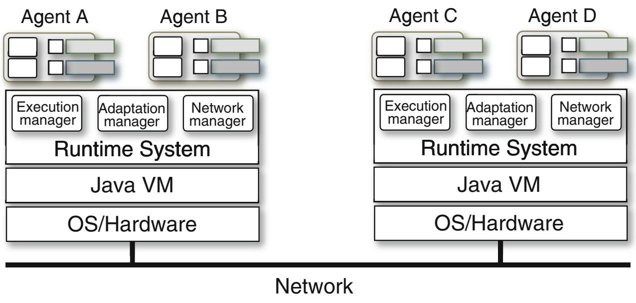

Fig. 2. Runtime system

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 disconnection 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.

7 Evaluation

This section describes the performance evaluation of our implementation.

7.1 Basic Performance

Although the current implementation was not constructed for performance, we evaluated several basic operations in distributed systems consisting of eights embedded computers, where each computer is a Raspberry Pi computer, which has been one of the most popular embedded computers (its processor was Broadloom BCM2835 (ARM v6-architecture core with floating point) running at 700 MHz and it has 1 GB memory and SD card storage (16 GB SDHC), with a Linux operating system optimized to Raspberry Pi, and OpenJDK. The cost of transmitting a message through UDP multicasting was 17 ms. The cost of transmitting a request message between two computers was 28 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 messages to other agents every 110 ms through UDP multicasting.

We evaluated the speed of convergence in our differentiation. Each computer had one agent having three functions, called behavior A, B and C, where behavior A invoked B and C behaviors every 200 ms and the B and C behaviors were null behaviors. We assigned at most one agent to each of the computers. 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 of weights would be ten.

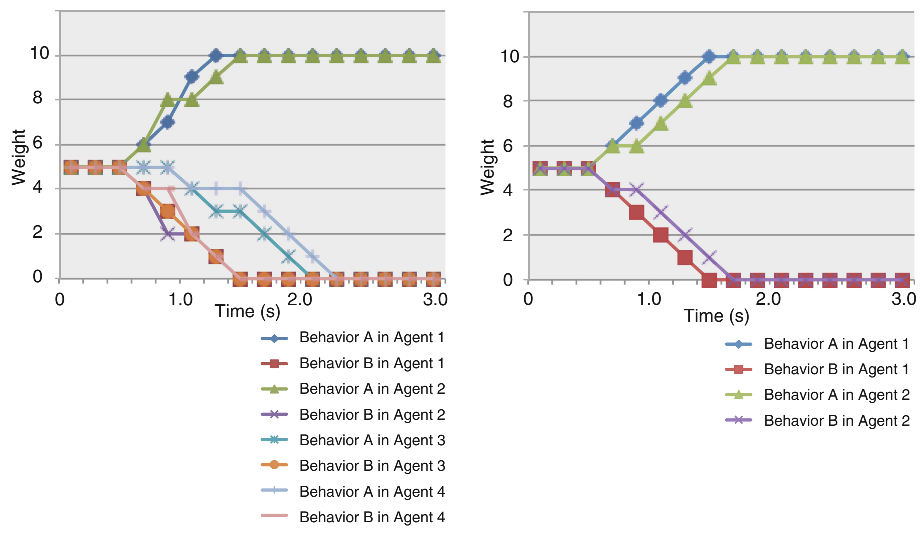

Fig. 3. Convergence in four agents with two behaviors (Left) and Convergence in eight agents with two behaviors (Right)

Differentiation started after 200 ms, because each agent knows the presence of other agents by receiving heartbeat messages from them. The right of Fig. 3 details the results obtained from our differentiation between four agents on four computers and The left of Fig. 3 between eight agents on eight computers. Finally, two agents provide behavior B and C respectively and the others delegate the two behaviors to the two agents in both the cases. 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.

7.2 Sensor Networks Recovering from Damaged by Disasters

Let us suppose a sensor-network system consisting of 15 × 15 nodes connected through a grid network, as shown in Fig. 4. The system was constructed on a commercial IaaS cloud infrastructure (225 instances of Amazon EC2 with Linux and JDK 1.7). This experiment permitted each node to communicate with its eights neighboring nodes and the diameter of a circle in each node represents the weight of a behavior. Nodes were connected according to the topology of the target grid network and could multicast to four neighboring runtime systems through the grid network. We assume that each agent monitors sensors in its current node and every node has one agent.

We put agents at all nodes and evaluated removing of redundant agents. Each agent has conflict with agents at its eights neighboring nodes, because it can delegate its function to them, vice versa. Figure 5(i) shows the initial weights of agents. (ii) and (iii) show the weights of behaviors in agents eight and

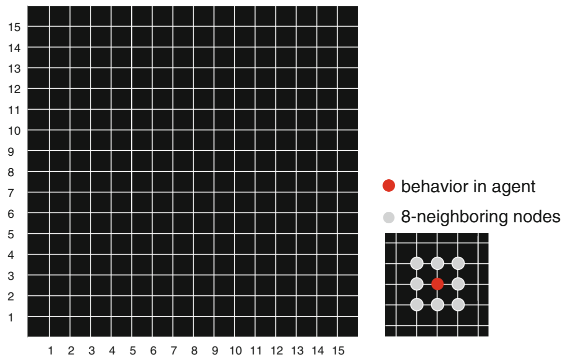

Fig. 4. 15 × 15-Grid network on cloud computing

sixteen seconds later. Even though differentiated behaviors were uneven, they could be placed within certain intervals, i.s., two edges on the grid network. This proved that our approach was useful in developing particular functions of software components at nodes.

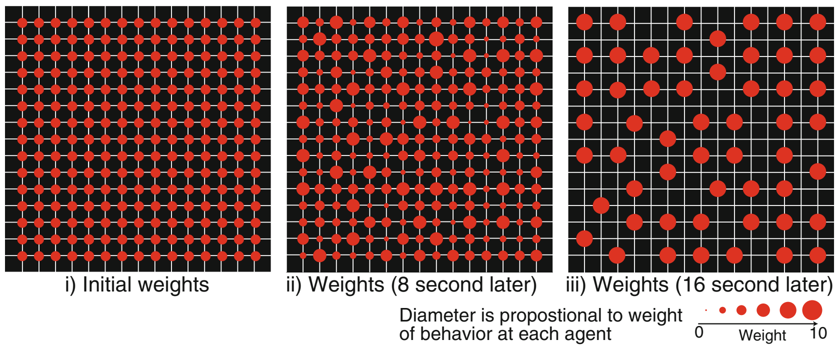

Fig. 5. Removing redundant agents

Figure 6(i) was the initial weights of agents on the network. We explicitly made a flawed part in the network (Fig. 6(ii)). Some agents dedifferentiate themselves in nodes when a flawed part made in the network. In the experiment agents around the hole started to activate themselves through dedifferentiation. The weights of their behaviors converged according to the weights of their behaviors to the behaviors of other newly activated agents in addition to existing agents. Finally, some agents around the hole could support the behaviors on behalf of the dismissed agents with the flawed part. This result prove that our approach could remedy such a damage appropriately in a self-organized manner. This is useful for sensing catastrophes, e.g., earthquakes and deluges.

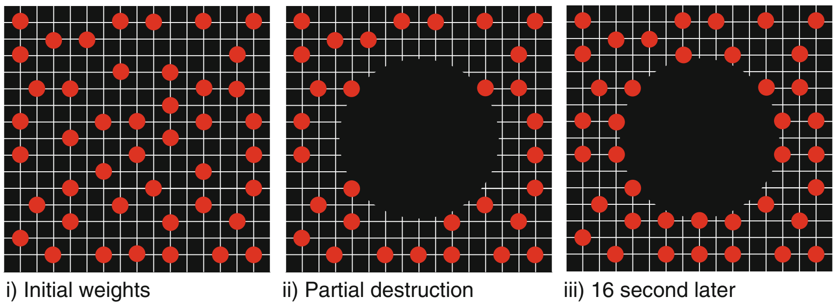

Fig. 6. Regeneration to recover damage

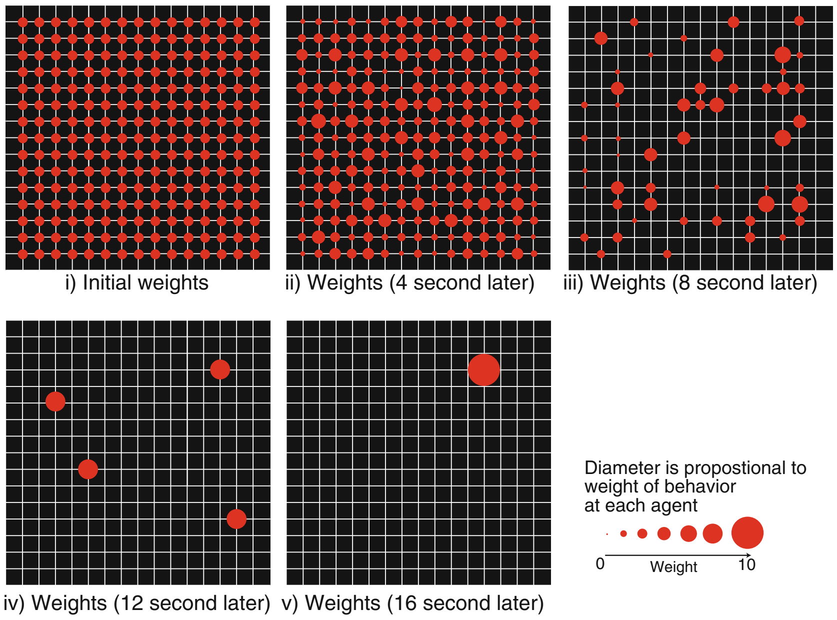

Fig. 7. Agents are differentiated in broadcasting to all agents

Next, we assume each node could multicast to all agents through the grid network. Figure 7 shows only one agent is activated and the others are inactivated after their differentiations, because the latter can delegate the function to the former. We partitioned the grid network as shown Fig. 8(ii). The above half has a well-developed behavior and the below half lacks such behavior. Therefore, all agents in the below half reset their weights as shown Fig. 8(iii) and they are differentiated. Finally, only one agent is activated on the below half part (Fig. 8(iv)).

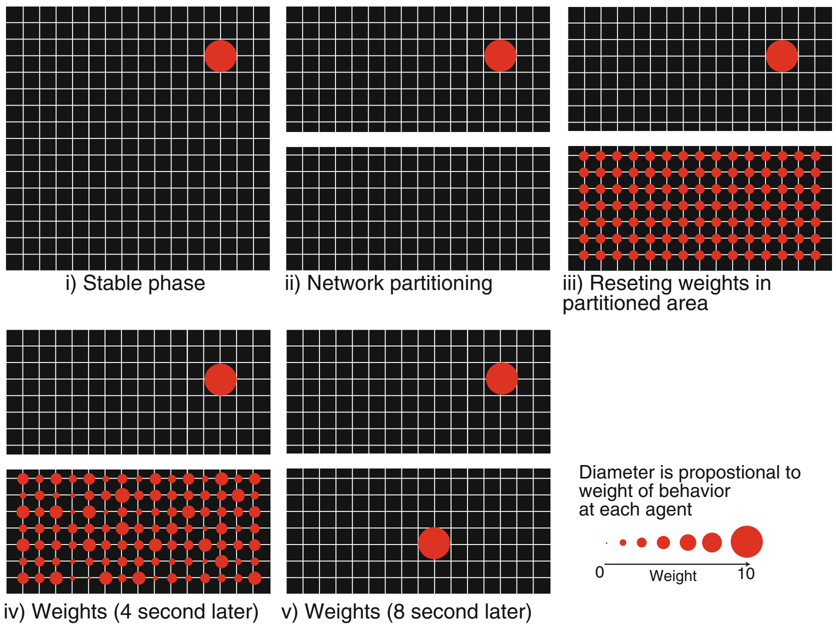
i) Stable phase
ii) Network partitioning
iii) Reseting weights in partitioned area
iv) Weights (4 second later)
v) Weights (8 second later)
Diameter is propostional to weight of behavior at each agent
0
Weight
10

Fig. 8. Agents are differentiated in network partitioning

8 Related Work

We compare between our approach and other existing bio-inspired approaches for distributed systems. 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 [2] and Bison [3]. Messor is a load-balancing application of Anthill and Bison is a conceptual bio-inspired framework based on Anthill. One of the most typical self-organization approaches to distributed systems is swarm intelligence [4,5]. 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. 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 [6,7]. 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. Our approach should be independent of the capabilities of distributed systems as much as possible.

Finally, we compare between our approach and our previous ones, because we constructed several frameworks for adaptive distributed systems. One of them enabled distributed components to be dynamically federated [8]. We also presented an early version of the proposed approach [9], but the version was designed for adaptive services over enrich distributed systems, e.g., cloud computing. They did not support any disaster management.

9 Conclusion

This paper proposed an approach to recovering distributed applications from violent damages, which might result from disasters. The approach is unique to other existing approaches for disaster-tolerant approaches for distributed systems. It was inspired from a bio-inspired mechanism, regeneration in living things. It was also available at the edge of networks, e.g., sensor networks and Internet-of-Thing (IoT). It enabled agents, which were implemented as software components, to be differentiated. When a component delegated a function to another component coordinating with it, if the former had the function, this function became less-developed and the latter's function became well-developed like differentiation processes in cells. It could also initialize and restart differentiated software components, when some components could not be delegated like regeneration processes in lizards. It was constructed as a general-purpose and practical middleware system for software components on real distributed systems consisting of embedded computers or sensor nodes.

Appendix

This appendix we describe our model for regenerating software components, called agents, by using a differentiation mechanism in detail. We specify from 1-th to n-th behaviors of k-th agent, as $b_1^k, \ldots, b_n^k$ and the weight of behavior b_i^k as w_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.

Invocation of Behaviors

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 (Fig. 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 (Fig. 1(b)).[2]

[2] Restraining messages correspond to cAMP in differentiation.

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.

Well/Less Developing Behaviors

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.

Removing Redundant Behaviors

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) (Fig. 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.

References

1. Babaoglu, O., Meling, H., Montresor, A.: Anthill: a framework for the development of agent-based peer-to-peer systems. In: Proceedings of 22nd International Conference on Distributed Computing Systems (ICDCS 2002), Washington, D.C., USA, pp. 15–22. IEEE Computer Society (2002)
2. 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). https://doi.org/10.1007/3-540-45074-2_12

3. Montresor, A., Babaoglu, O.: Biology-inspired approaches to peer-to-peer computing in BISON. In: Abraham, A., Franke, K., Köppen, M. (eds.) Intelligent Systems Design and Applications. ASC, vol. 23, pp. 515–522. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-44999-7_49
4. Bonabeau, E., Dorigo, M., Theraulaz, G.: Swarm Intelligence: From Natural to Artificial Systems. Oxford University Press, Oxford (1999)
5. Dorigo, M., Stützle, T.: Ant Colony Optimization. Bradford Company, Scituate (2004)
6. Nakano, T., Suda, T.: Self-organizing network services with evolutionary adaptation. IEEE Trans. Neural Netw. **16**(5), 1269–1278 (2005)
7. Suzuki, J., Suda, T.: A middleware platform for a biologically inspired network architecture supporting autonomous and adaptive applications. IEEE J. Sel. Areas Commun. **23**(2), 249–260 (2005)
8. Satoh, I.: Self-organizing software components in distributed systems. In: Lukowicz, P., Thiele, L., Tröster, G. (eds.) ARCS 2007. LNCS, vol. 4415, pp. 185–198. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71270-1_14
9. Satoh, I.: Resilient architecture for complex computing systems. In: 18th International Conference on Engineering of Complex Computer Systems, pp. 256–259, July 2013