International Journal of Computer Systems (ISSN: 2394-1065), Volume 02– Issue 11, November, 2015
Available at http://www.ijcsonline.com/

A Middleware-level Approach to Adaptive Distributed Systems

Jingtao Sun
Department of Informatics
The Graduate University for Advanced Studies (SOKENDAI)
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
sun@nii.ac.jp

Abstract

This paper proposes an approach to adapting distributed applications to changes in environmental conditions, e.g., user requirements and resource availability. The key idea behind the proposed approach is to introduce the relocation of software components to define functions between computers as a basic mechanism for adaptation on distributed systems. It also introduces application-specific built-in policies for relocating components to define higher-level adaptation. It is constructed as a middleware system for Java-based general-purposed software components. This paper describes the proposed approach and the design and implementation of the approach with three applications, e.g., dynamic deployment of software at nodes in a sensor network and adaptive selection between data replication between primary backup and chain replication approaches in a distributed system.

Keywords*: Distrubuted Systems, Middleware-level approach*

I. Introduction

The complexity and dynamism of distributed systems are beyond our ability to build and manage systems through conventional approaches, such as those that are centralized and top-down. This is because distributed systems are complicated and dynamic by nature. For example, computers and software components of which an application consists may be dynamically added to or removed from them, and networks between computers may be connected or disconnected, dynamically. Software running on distributed systems should be resilient so that the systems can adapt themselves to various changes at runtime. Software running on a distributed system should be adaptive to reuse it on different distributed systems.

Distributed applications are executed for multiple-purposes and multiple users whose requirements are various and change, on a variety of distributed systems whose structures may change. Adaptation to support variety and change in the underlying systems and applications' requirements should be separated from business logic. Therefore, we distinguish between adaptation concerns and business logic concerns by using the principle of separation of concerns so that developers for applications can concentrate their business logic rather than adaptation as much as possible. A solution to this is to introduce concern-specific languages for separating adaptive concerns from business logic concerns. There have been several attempts to support the separation of concerns on non-distributed systems, but adaptation mechanisms in distributed systems tend to be complicated so that it is difficult to define primitive adaptation.

This paper addresses the separation of adaptation concerns from application-specific logic concerns in distributed systems. We assumed that a distributed application would consist of one or more software components, which might have been running on different computers through a network. Our proposed approach has two key ideas. The first is to introduce policies for relocating software components as a basic adaptation mechanism. The second is to provide nature-inspired relocation policies for application-specific adaptations. When changes in a distributed system occurred, e.g., in the requirements of the application and the structures of the system, its software components would automatically be relocated to different computers according to their policies to adapt it to the changes. We are constructing a middleware system that will be used for building and operating adaptive distributed systems.

The proposed approach is based on adaptive deployment of software components but not on adaptive functions inside software components like other existing work. If functions inside software components are adapted, other components, which communicate with the adapted ones, may have serious problems. On the other hand, the relocation of software components does not lose potential functions of components. This may seem to be simple but it makes their applications resilient without losing availability, dependability, and reliability. In fact, our approach can provide adaptation

between practical approaches in distributed systems. For example, *primary-backup* and *chain replication*, which are widely used in distributed systems, including cloud computing, consistently support replication mechanisms with consistency on distributed systems. Nevertheless, the latter has been designed to improve throughput rather than latency in comparison with the former. They should be dynamically selected according to the requirements of applications, which may often change.

II. Related Work

This section outlines related work. The notion of adaptation is rapidly attracting attention in the area of distributed systems. There have been several attempts to develop adaptive distributed systems. Most of them have aimed at managing balance computational loads or network traffic.

Common approaches for self-organization have included genetic computation, genetic programming [9], and swarm intelligence [2], [4]. 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. Most existing approaches have only focused on their target problems or applications but they are not general purpose, whereas distributed systems are. 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, real distributed systems have no room to execute such large numbers of agents.

Computational reflection refers to the ability of a program to reason about and define its own behavior. Reflection enables a system to be open to dynamically define itself without compromising portability or revealing parts unnecessarily. In this approach, the program contains one or more meta levels, which enable reconfiguration of the underlying base-level code. Separation of concerns enables the separate development of an applications functional behavior and its adaptive behavior involving crosscutting concerns. A widely used technique is aspect-oriented programming (AOP), where the code implementing a crosscutting concern, called an *aspect*, is developed separately from other parts of the system and woven with the business logic at compile- or run-time. Reflective and AOP approaches are primitive so that they do not directly support adaptation for distributed systems.

Several researchers have explored software adaptation in the literature on adaptive computing and evolution computing. Jaeger et al. [8] introduced the notion of self-organization to an object request broker and a publish/subscribe system. Georgiadis et al. [5] 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 the internal behaviors inside them. Like ours, Cheng at al. [3] 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. Herrman et al. proposed the bio-inspired deployment of services on sensor networks [6]. Unlike ours, their work focused on the deployment and coordination of services, instead of the adaptation of software itself to provide services. Nakano and Suda [11], [16] proposed bio-inspired middleware, called Bio-Networking, for disseminating network services in dynamic and large-scale networks where there were large numbers of decentralized data and services. 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' behaviors 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 another system. Our approach was designed independently of the capabilities of distributed systems because adaptive policies should be able to be reused in other distributed systems.

There have been several attempts to design specification languages for self-adaptation. Several formal approaches supported specific aspects of self-adaptation exist. For example, Zhang [18] proposed an approach to formally modeling and specifying two aspects: adaptation behavior and non-adaptive behavior, separately in a graph-based notation and generate an implementation of the system. They extended Linear Temporal Logic (LTL) with an adaptive operator that allows the description of adaptive behavior. However, the approach was intended to adapt finite state machines to changes so that it could not support any distributed systems. FORMS [17] provides an encompassing formally founded vocabulary for describing and reasoning about different concerns of self-adaptive software architectures. It, however, does not define any notion of execution semantics, that is how the adaptation of the system takes place. ACML [10] allows the separated and explicit specification of self-adaptivity concerns by using LTL notations. Based on formal semantics we show how to apply quality assurance techniques to the modeled self-adaptive system, which enable the provisioning of hard guarantees concerning self-adaptivity characteristics such as adaptation rule set stability and deadlock freedom. Its specification was limited to the expressiveness of state machines so that did not support distributed systems.

The relocation of software components have been studied in the literature on mobile agents [14]. However, existing mobile agent platforms have been designed for solving problems in distributed systems, e.g., the reduction of network latency and fault tolerance, instead of adaptation. There have been a few attempts to introduce the policy-based relocation of software components or agents. The FarGo

system introduced a mechanism for distributed applications dynamically laid out in a decentralized manner [7]. This was similar to our relocation policy in the sense that it allowed all components to have their own policies, but it only supports a simple relocation unlike ours, and could not specify any conditions for their policies, unlike ours. Satoh [13] proposed other relocation policies for relocating components based on policies that other components moved to. However, these policies did not have the conditions that select and execute them unlike the approach proposed in this paper. One of the authors proposed an adaptation mechanism for distributed systems [15]. However, the mechanism was aimed at adapting functions of software components, which are statically located at computers, by using the notion of differentiation instead of their locations.

III. Approach

As the requirements of applications and the structures of systems may often change in distributed systems, the applications need to adapt themselves to such changes. Our approach introduces the relocation of software components to define functions at other computers as a basic adaptation mechanism.

A. Requirements

Distributed systems are used for multiple purposes and users and need abilities to adapt them to various changes results from their dynamic properties. Our adaptation has five requirements.

- *Self-adaptation:* 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.
- *On-demand deployment of software:* Computers in may have limited resources so that they cannot support software for various applications beforehand. To coordinate multiple computers for an applications, software for the application need to be dynamically deployed at appropriate computers.
- *Separation of concerns:* All software components should be defined independently of our adaptation mechanism as much as possible. As a result, developers should be able to concentrate on their own application-specific processing.
- *Reusability:* There have been many attempts to provide adaptive distributed systems. However, the approaches and parameters in most of them these strictly and statically depended on their target systems, so that they would need to be re-defined overall to be reused in other distributed systems. Our adaptation should be abstracted away from the underlying systems for reasons of reusability.
- *No-centralized management:* There is no central entity to control and coordinate computers. Our adaptation should be managed without any centralized management for reasons of avoiding any single points of failures and performance bottleneck for reliability and scalability.

There are various applications running on distributed systems are various. Therefore, the approach should be implemented as a practical middleware to support general-purpose applications. Computers on distributed systems may have limited resources, e.g., processing, storage resources, and networks. Our approach should be available with such limited resource, whereas many existing adaptation approaches explicitly or implicitly assume that their target distributed systems have enriched resources. The bandwidth of networks on several distributed systems tend to be narrow and their latency cannot be neglected. The approach should support such networks.

B. Adaptation

Our approach separates software components from their policies for adaptation, although components have their own policies.

1) Deployable software component:: This approach assumes that an application consists of one or more software components, which may be running on different computers. Each component is general-purpose and is a programmable entity. It can be deployed at another computer according to its deployment policy, while it have started to run. It is defined as a collection of Java objects like JavaBeans component in the current implementation. It also has an interface, called a *reference*, to communicate with other components through dynamic method invocation developed in common object request broker architecture (CORBA). The interface supports the notion of being *mobility-transparent* in addition to that of inter-component communication, in the sense that it can forward messages to co-partner components after it has migrated to another computer through a network.

2) Deployment policy for adaptation:: Each component can have one or more policies, where each policy is basically defined as a pair of information on where and when the component is deployed. Before explaining deployment policies in the proposed approach, we need to discuss policies for adaptation in distributed systems. Our approach introduces the these concepts:

- *Adaptation as relocation of component:* Our approach introduces the relocation and coordination of software components as a mechanism. Instead, it does not support any adaptation inside software components.
- Each application-specific component can have one or more policies, where each policy is basically defined as a pair of information on where and when the component is deployed.

- Policies can be defined outside components, so that they can be reused for other components and the components can be reused with other policies.
- The destinations of the relocation of software components are specified at the addresses of other components instead of the addresses of computers, because our adaptation should be independent of the underlying system, e.g., computers and networks.

Since our approach is designed for disaggregated computing rather than general distributed computing, we intend to provide only a set of essential and useful adaptation policies for disaggregated computing as follows:

- The approach supports relocation of software components but not any adaptation inside software components When more than one dimension must be considered for adaptation, representing the policies and choices between policies tends to be too complicated to define and select policies. Therefore, we intend to support at most one dimension, i.e., the dynamic deployment of components.
- Each component has one or more policies, where a policy specifies the relocation of its components and instructs them to migrate to the destination according to conditions specified in the policy. The validation of every policy can be explicitly configure to be one-time, within specified computers, or permanent within its component.
- Each policy is specified as a pair of a condition part and at the most one destination part. The former is written in a first-order predicate logic-like notation, where predicates reflect information about the system and applications. The destination part refers to another components instead of the computer itself. This is because such policies should be abstracted away from the underlying systems, e.g., network addresses, so that they can be reused on other distributed systems. The policy deploys its target component (or a copy of the component) at the current computer of the component specified as the destination, if the condition is satisfied.
- The approach also provides built-in policies for adaptation as extensions of the above primitive relocation policy. In fact, it is not easy to define relocation policies, because such policies tend to depend on the underlying systems.

Since components for which other components have policies can be statically or dynamically deployed at computers, the destinations of policies can easily be changed for reuse by other distributed systems. Next, we describe our built-in policies for adaptation in distributed systems (Fig. 1).

Our approach provides a set of essential and useful adaptation policies for distributed systems:

- *Attraction:* Frequent communication between two components yields stronger force. The both or one of the components are dynamically deployed at computers that other components are located at.
- *Spreading:* Copies of software components are dynamically deployed at neighboring and propagated from one computer to another over a distributed system. This policy progressively spreads components for defining functions over the system and reduces the lack of the functions.
- *Repulsion:* Components are deployed at computers in a decentralized manner to avoid collisions among them. This policy moves software components from regions with high concentrations of components to regions with low concentrations.
- *Evaporation:* Excess of components results in overloads. The same or compatible functions must be distributively processed to reduce the amount of load and information. This policy consists in locally applying to synthesize multiple components or periodically reduces the relevance of functions.

Each policy is activated only when its condition specified in it is satisfied, where the condition is written in a first-order predicate logic-like notation, where predicates reflect information about the system and applications.

IV. Design

The proposed approach dynamically deploys components to define application-specific functions at computers according to the policies of the components to adapt distributed applications to changes in distributed systems.

Our middleware system consists of two parts: a *component runtime system* and an *adaptation manager*, where each of the systems are coordinated with one another through a network. The first part is responsible for executing and duplicating components at computers and also exchanging components and messages in runtime systems on other computers through a network. The second part is responsible for managing policies for adaptation. It consists of an interpreter for policies written in our proposed language and a database system to maintain the policies.

A. Component runtime system

Each runtime system allows each component to have at most one activity through the Java thread library. When the life-cycle state of a component changes, e.g., when it is created, terminates, duplicate, or migrates to another computer, the runtime system issues specific events to the component. To capture such events, each component can have more than one listener object that implements a specific listener interface to hook certain events issued before or after changes have been made in its life-cycle state. The current implementation uses the notion of dynamic method invocation studied in CORBA so that it can easily hide differences between the interfaces of objects at the original and other computers.

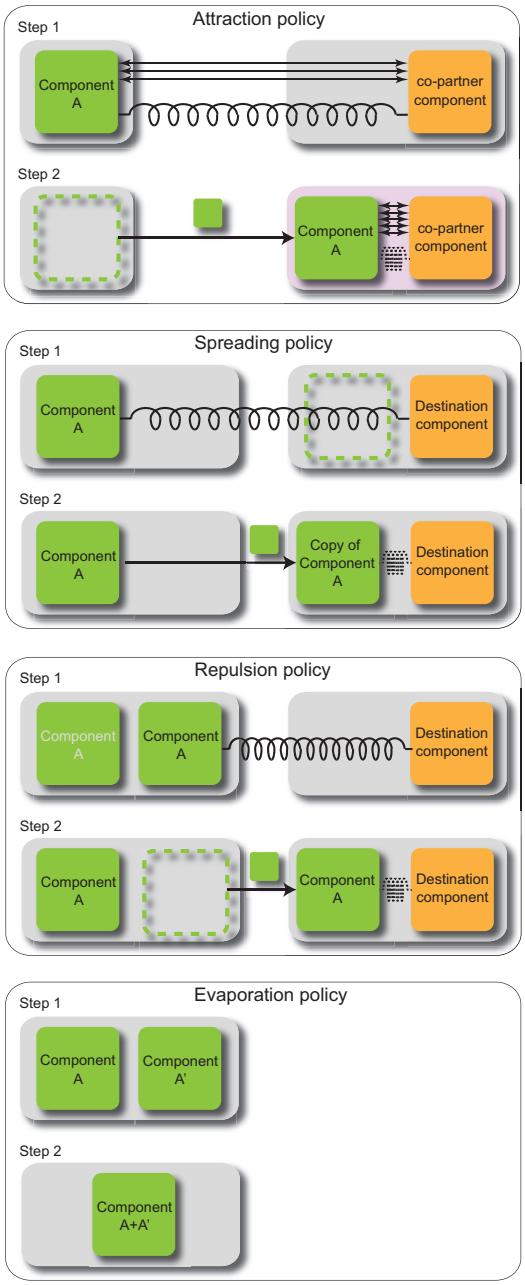

Figure 1. Built-in Relocation Policies for adaptation

Each runtime system can exchange components with other runtime systems through a TCP channel using mobile-agent technology. When an component is transferred over the network, not only the code of the component but also its state is transformed into a bitstream by using Java's object serialization package and then the bit stream is transferred to the destination. The component runtime system on the receiving side receives and unmarshals the bit stream.

Even after components have been deployed at destinations, their methods should still be able to be invoked from other components, which are running at local or remote computers. The runtime systems exchanges information about components that visit them with one another in a peer-to-peer manner to trace the locations of components.

B. Adaptation manager

Figure 2 shows the policy-based relocation of component. Each adaptation manager periodically advertises its address to the others through UDP multicasting, and these computers then return their addresses and capabilities to the computer through a TCP channel.[1] It evaluates the conditions of its storing policies, when the external system detects changes in environmental conditions, e.g., user requirements and resource availability.

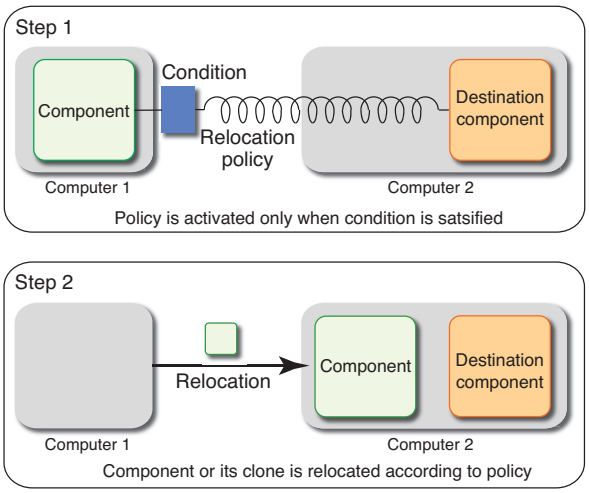

Figure 2. Attraction policy for relocation

Each policy is specified as a pair of conditions and actions. The former is written in a first-order predicate logic-like notation and its predicates reflect various system and network properties, e.g., the utility rates and processing capabilities of processors, network connections, and application-specific conditions. The latter is specified as a relocation of components. Our adaptation was intended to be specified in a rule-style notation. However, existing general-purpose rule-based systems tend to be unwieldy because they cannot express necessary adaptation expertise or subtleties of adaptation in distributed systems.

name {	(Name of policy)
$predicate_1, \cdots, predicate_n$	(Condition of policy)
relocation ($component_{id}$)	(The destination of relocation)
validation	(Validation of policy)
}	

where *relocation* in the syntax is provided with built-in or user-defined policies. *validation* specifies whether the policy

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

is one time or permanent. Our adaptation has four built-in policies:

- When a component has an *attraction* policy for another component, if communications between the former and latter becomes more than a specified number, the policy instructs the former to migrate to the current computer of the latter.
- When a component has a *spreadinG:* policy for another component, if the current computer of the latter does not have the same or compatible components, the policy makes a copy of the former copy of it and instructs the copy to migrate to the current computer of the latter.
- When a component has a *repulsion* policy for another component, if there are a specified number or more of the same or compatible components of the former at the current computer, the policy instructs the former to migrate to the current computer of the latter.
- When a component has an *evaporation* policy, if there are a specified number or more of the same or compatible components of the former at the current computer or its specified life span, called *time-to-live* (TTL) is over, it terminates.

For example, when the condition of the *attraction* policy is the movement of the co-partner component, the target component follows the movement of the co-partner. This is useful when the two components need to interact frequently and/or require heavy data-transfer on each interaction yet they cannot be programmed inside a single component. When the condition of the *evaporation* policy is that the target component and specified component are at the same computer, it reduces the number of components.

V. Implementation

This section describes the current implementation of a middleware system based on the proposed approach.

A. Deployable component

Each component is a general-purpose programmable entity defined as a collection of Java objects like JavaBeans and packaged in the standard JAR file format. It has no specification for adaptation inside it, but it can be migrated duplicated to a remote computer by the current adaptation manager. We introduce our original remote method invocation between computers instead of Java remote method invocation (RMI), because Java RMI does not support object migration. 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, as was discussed in Section III. 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. These *references* are managed by using our original protocol for locating components by using UDP multicasting.

B. Component runtime system

Our runtime system is similar to a mobile agent platform [14], but it has been constructed independently of any existing middleware systems. This is because existing middleware systems, including mobile agents and distributed objects, have not supported the policy-based relocation of software components. The system is built on the Java virtual machine (JVM), which can abstract away differences between operating systems.

The current implementation basically uses the Java object serialization package to marshal or duplicate components. The package does not support the capture of stack frames of threads. Instead, when a component is duplicated, the runtime system issues events to it to invoke their specified methods, which should be executed before the component is duplicated or migrated, and it then suspends their active threads.

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

C. Adaptation Manager

The adaptation manager is running on each computer and consists of three parts: an interpreter, a database for policies, and an event manager. The first is responsible for evaluating policies, the second maintains the policies that components are running on the computer, and the third receives events from the external systems to notify changes in the underlying system and applications and then forwards them to the first.

We describe a process of the relocation of a component according to one of its policies. (1) When a component creates or arrives at a computer, it automatically registers its deployment policies with the database of the current adaptation manager, where the database maintains the policies of components running on its runtime system. (2) The manager periodically evaluates the conditions of the policies maintained in its database. (3) When it detects the policies whose conditions are satisfied, it deploys components according to the selected policies at the computer that the destination component is running on.

Two or more policies may specify different destinations under the same condition that drive them. The current implementation provides no mechanism to solve conflict between policies. We assumed that policies would be defined without any conflicts between policies. The destination of the component may enter divergence or vibration modes due to conflicts between some of a component's policies, if it has multiple deployment policies. However, the current implementation does not exclude such divergence or vibration.

VI. Applications

This section describes two applications of the proposed approach.

A. Adaptive remote information retrieval

Suppose users try to search certain text patterns from data located at remote computers like Unix's `grep` command. A typical approach is to fetch files from remote computers and locally find the patterns from all the lines of the files. However, if the sum of the volume of its result and the size of a component for searching patterns from data is smaller than the volume of target data, the approach is not efficient. The component should be executed at remote computers that maintain the target data rather than at local computers. However, it is difficult to select where the component is to be executed because the volume of the result may not be known before.

The proposed approach can solve this problem by relocating such components from remote computers to local computers and vise versa while they are running. Figure 3 shows our system for adaptive remote information retrieval, which consists of *client*, *search*, and *data access manager* components. The first and the third are stationary components. The second supports finding text lines that match certain patterns provided from the first in text files that it accesses via the third. It has a *attraction* policy that relocates itself from local to remote computers when the volume of its middle result is larger than the size of the component, otherwise it relocates from remote to local computers[2]

Although the volume of the result depends on the content of the target files and the patterns, it is typically about one over hundred less than the volume of the target files. Therefore, the cost of our system is more efficient in comparison with Unix's `grep` command. This means that our approach enables distributed applications to be available with limited resources and networks, as was discussed in Section III. Our approach is self-adaptive in the sense that it enables the *search component* to have its own adaptation policy and manage itself according to the policy independently of these components themselves. It is independent of its underlying systems because the destinations of our component relocations are specified as components instead of computers themselves. Our adaptation can be reused by changing the locations of such destination components.

[2]The size of the component is about 20 KB.

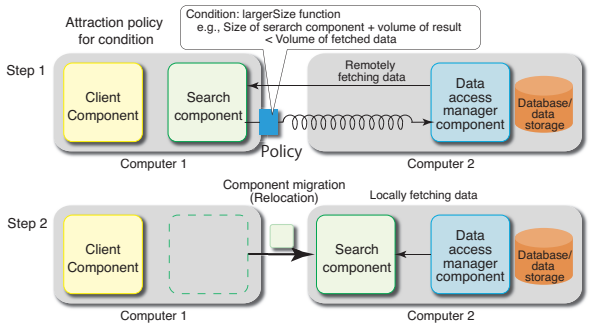

Figure 3. Adaptive remote information retrieval

B. Spreading software for sensor nodes

The second example is the dynamic deployment of software components over a sensor network in a self-adaptive manner. It is a well known fact that after a sensor node detects environmental changes, the presence of people, in its area of coverage, some of its geographically neighboring nodes tend to detect similar changes after a period of time. Software components should be deployed at nodes where and when environmental changes can be measured. The basic idea behind this example is to deploy software components at only nodes around such changes. Such deployment could be easily provided from our policy-based relocation.

We assumed that the sensor field was a two-dimensional surface composed of sensor nodes and it monitored environmental changes, such as motion in objects and variations in temperature. Each software component had *spreading* and *evaporation* policies in addition to its application-specific logic, i.e., monitoring environmental changes around its current node, where the destination components of the former were neighboring sensor nodes and the condition of the latter was the detection of changes within a specified time. We assumed that such a component was located at nodes close to the changes. When the event manager of the current node detected changes with sensors, the adaptation manager evaluated the policies of the component, if there were no components at neighboring nodes, it made clones of the component and deployed clones at the neighboring nodes. When the change moved to another location, e.g., walking people, the components located at the nodes near the change could detected the changed in the same way, because clones of the components had been deployed at the nodes.

Each clone was associated with a resource limit that functions as a generalized TTL field. Although a node could monitor changes in interesting environments, it set the TTLs of its components to their own initial values. It otherwise decremented TTLs as the passage of time. When the TTL

of a component became zero, the component automatically removed itself according to its *evaporation* policy to save computational resources and batteries at the node.

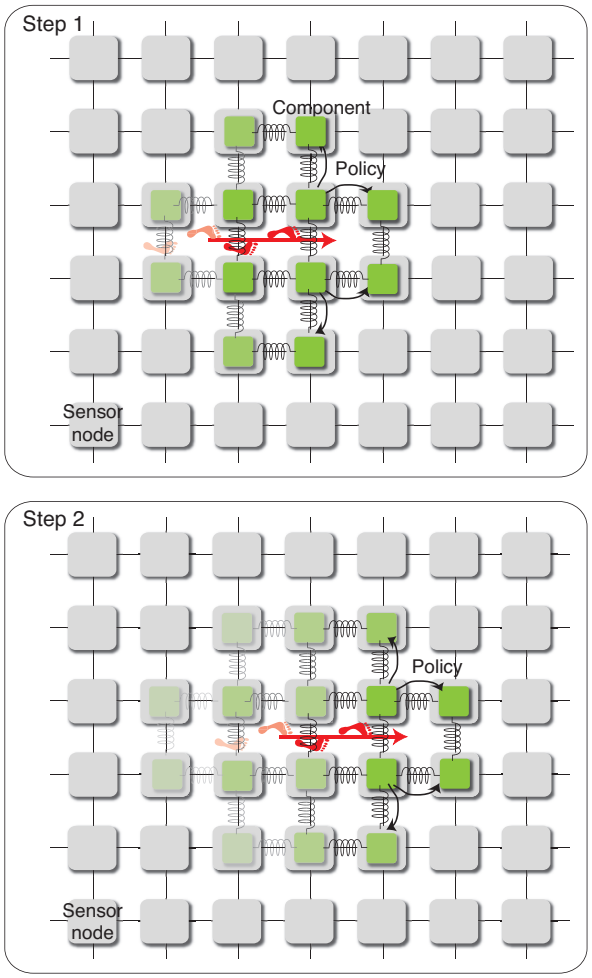

Figure 4. Component diffusion for moving entity

VII. Conclusion

This paper proposed an approach to adapting distributed applications for resilient distributed systems. It introduced the relocation of software components between computers as a basic mechanism for adaptation. It also provided four relocation policies, called *attraction*, *repulsion*, *spreading*, and *evaporation* to easily implement practical adaptations. It separated software components from their adaptations in addition to underlying systems by specifying policies outside the components. It was simple but provided various adaptations to support resilient distributed systems without any centralized management. It was available with limited resources because it had no speculative approaches, which tended to spend computational resources. The relocation of components between computers was useful to avoid network latency. It was constructed as a general-purpose middleware system on distributed systems instead of any simulation-based systems. Components could be composed from Java objects like JavaBean modules. The three practical applications proved that the proposed approach was useful to construct resilient distributed systems.

In concluding, we would like to identify further issues that need to be resolved. We need to improve the implementation of the approach. The policy specification language of the current implementation is still naive. We are interested in refining it. We also want to develop more applications with the approach to evaluate its utility.

References

[1] P.A. Alsberg and J.D. Day: A principle for resilient sharing of distributed resources, In Proceedings of 2nd International Conference on Software Engineering (ICSE'76), pp.627-644, 1976.
[2] E. Bonabeau, M. Dorigo, and G. Theraulaz: Swarm Intelligence: From Natural to Artificial Systems, Oxford University Press, 1999.
[3] 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), pp.2-8, ACM Press, 2006.
[4] M. Dorigo and T. Stutzle: Ant Colony Optimization, MIT Press, 2004.
[5] 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), pp.33-38, ACM Press, 2002.
[6] K. Herrman: Self-organizing Ambient Intelligence, VDM, 2008.
[7] 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.
[8] 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), pp.543-550, ACM, 2007.
[9] J.R. Koza: Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, 1992.
[10] M. Luckey and G. Engels: High-Quality Specification of Self-Adaptive Software Systems, in Proceedings of Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'2013), pp.143-152, 2013.
[11] T. Nakano and T. Suda: Self-Organizing Network Services With Evolutionary Adaptation, IEEE Transactions on Neural Networks, vol.16, no.5, pp.1269-1278, 2005.
[12] R. van Renesse and F. B. Schneider: Chain replication for supporting high throughput and availability, in Proceedings of 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI'2004), 2004.
[13] I. Satoh: Self-organizing Software Components in Distributed Systems, in Proceedings of 20th International Conference on Architecture of Computing Systems System Aspects in Pervasive and Organic Computing (ARCS'07), Lecture Notes in Computer Science (LNCS), vol.4415, pp.185-198, Springer, March 2007.
[14] I. Satoh: Mobile Agents, Handbook of Ambient Intelligence and Smart Environments, pp.771-791, Springer 2010.

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

[16] 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, pp.249-260, 2005.

[17] D. Weyns, S. Malek, J. Andersson: FORMS: Unifying Reference Model for Formal Specification of Distributed Self-Adaptive Systems, ACM Transactions on Autonomous and Adaptive Systems, Vol. 7, No. 1, 2012.

[18] J. Zhang and B.H.C. Cheng: Model-based development of dynamically adaptive software, Proceedings of 28th International Conference on Software Engineering (ICSE'2006), pp.371-380, ACM, 2006.