A Policy-based Middleware for Self-Adaptive Distributed Systems

Jingtao Sun and Ichiro Satoh

Department of Informatics. School of Multidisciplinary Science
The Graduate University for Advanced Studies
National Institute of Informatics
Tokyo, Japan
Email: {sun, ichiro}@nii.ac.jp

***Abstract*—This paper presents a novel approach to dynamically adapting distributed applications to changes in environmental conditions. e.g., available network resources and users requirements. The key idea behind the approach is to introduce the relocation of software components to define functions between computers, as a basic mechanism for dynamic adaptation on distributed systems. It also introduces application-specific built-in policies for relocating software components to define high-level adaptation by human-readable declarative policy scripts. It is constructed as a middleware system for Java-based general-purposed software components. This paper describes the design and implementation of the approach with several applications, e.g., remote information retrieval and distributed media service.**

Keywords–software component; policy-based; self-adaptive; middleware; Distributed System; architecture-level.

I. INTRODUCTION

Distributed systems are essentially dynamic in the sense that computers and applications may be dynamically added to or removed from them or networks between computers may be disconnected or reconnected, dynamically. Therefore, the running of software components of which an application consists, should be depended by nature, so that the systems can adapt to various changes at component runtime systems. On the other hand, the running of software components on a distributed system should be adapted to and reuse them on different distributed systems.

Distributed applications are executed for multiple-purposes and multiple users whose requirements may change in various cases. However, on a variety of distributed systems, their structure may also changes. Adaptation must support the variety and change in the underlying systems and the requirements of applications should be separated from business logic. Therefore, we distinguish 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 mush 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 [3][7] 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 run on different computers through networks. Our proposed approach has two key points. 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 the changes have occurred, e.g., in the requirements of applications and the structures of system, its software components would automatically be relocated to different computers according to their policies to adapt to changes. We are constructing a middleware system for building and operating self-adaptive distributed systems.

The proposed approach is based on adaptive deployment of software components, but is different from other existing works [2][6][7], the functions of which are inside software component. If this components are adapted, other component may have serious problems. For example, they can not communication with the adapted ones. On the other hand, the relocation of software components do not lose potential functions of components. This problem 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 general-proposed approaches in distributed systems.

This paper focuses on the middleware we have developed to simplify the design and deployment of policy-based runtime system in real applications. Section 2 describes the requirements of distributed systems and gives readers the key idea behind the proposed approach. Section 3 gives an overview of how to design the policy-based middleware and what kind of adaptations are driven by declarative policy scripts. Following this, Section 4 describes how we implemented such runtime system support software components. Section 5 describes several applications of this middleware to demonstrate its strengths. Section 6 describes related work, with conclusions and planned future work described in Section 7.

II. APPROACH

In distributed systems, the requirements of applications or users and the environments of distributed systems often change, so they have to adapt to these changes inside of themselves. But in real environments, applications should not be necessary to recompile one more again in order to adapt to use a different network architecture or layout. This is because most of existing adaptation technology needs a large amount of resources to adapt the requirements of the applications or adaptation is limited or adaptive contents cannot be predicted. Therefore, we decided to propose a novel approach to relocate the running software components form one computer to another one, to adapt to changes for distributed systems,

 ISBN: 978-1-61208-378-0

e.g., adapting to distributed systems, networks architectures or available resources.

A. Requirements

Most existing distributed systems have been statically constructed based on several types of system architectures, e.g., client-server, peer-to-peer and master-slave according to their original requirements. However, with the development of in-depth in distributed systems, some of the requirements may be changed, wherefore the distributed systems need to dynamically adapting themselves. For example, computers and networks may have failures or some new computers may be added to or removed from the networks, or the requirements of applications, which may be running on different computers. Therefore, distributed systems need some abilities to adapt to such changes. Furthermore, to support drastic changes in system structures, distributed system architectures themselves require to change.

In this section, we describe the requirements of our policy-based middleware as follows:

- 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.
- Separation of concerns: software components of which an application consist should be defined independently in our adaptation mechanism. On the other hand, these software components will deploy themselves to destination computers, according to the predefined policies or user-defined policies, which can be developed by system operators or be automatically executed by themselves.
- General-purpose: Various of applications are running on distributed systems. Therefore, our adaptation mechanism should be implemented as a practical middleware to support general-purpose applications.
- Reduce Input/Output cost: The costs of Input/Output handling are huge in distributed systems, e.g., when users send reading requests to front-ends servers, they have to find out the requested files in the first, and then reading the content of the files line by line, until the ends of files. For this reason, the cost of Input/Output requires to be reduced.
- Dependability: In order to improve the dependability of distributed systems, middleware systems do not require to support a centralized management for software components and make sure that data keep their consistency.

Computers on distributed systems may have limited resources, e.g., processing, storage resources, and network architectures. Therefore, our approach should be available with such limited resources, whereas many existing adaptation approaches explicitly or implicitly assume that their targets of distributed systems have enriched resources.

B. Adaptation

Our approach separates software components from their policies for adaptation. This is because the user-defined policies can be reused to reduce the cost of compiling themselves once more.

1) Deployable software components: Generally, an application consists of one or more software components, which may be running on different computers. Therefore, in our approach, these components can be deployed at other computers, according to its deployment policies. It is defined as a collection of Java objects in the current implementation. It also has an interface, which called references. By executing them, soft components can migrate to destination computers, and then communicate with the destination components through dynamic methods invocation.

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 have to discuss policy scripts for adaptation on distributed systems. We describe these concepts as follows:

- This approach does not support any adaptation inside software components. Because software components should be general-purposed and adaptation-independent.
- Each component has one or more policies, where policy specifies the relocation of their components and instructs them to migrate to destination computers, according to specified conditions.
- Each policy specifies as a pair of a condition part and at most one destination part. The former is written in a first-order predicate logic-like notation, where predicates reflect information about the system and application. The latter refers to another component instead of itself. This is because such policies should be abstracted away from the underlying systems.

C. Destination of policies

Under the user-defined destination of policies, as Figure 1 shows, software components can be dynamically deployed at destination computers and the destinations of policies can be easily changed for reuse by other distributed systems. The policies are described as follows:

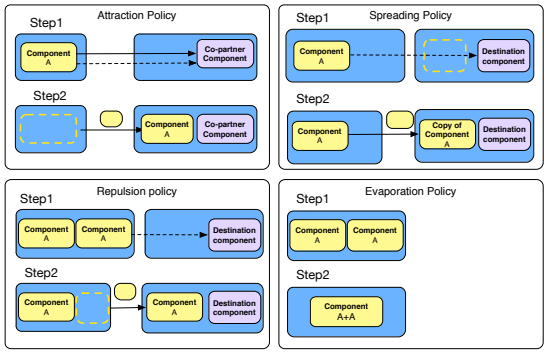

Figure 1. Destination of policies.

- Attraction: Frequent communication between two components yields stronger force. The both or one of the components are dynamically deployed at computers that other computers are located at.

 ISBN: 978-1-61208-378-0

- 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 dynamically adds to the lack of the functions.
- Repulsion: Computers are deployed at computers in a decentralized manner to avoid collisions among them. This policy migrates software components from regions with high concentrations of components to 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. The policy consists in locally applying to synthesize multiple components or periodically to reduce the relevance of functions.

This approach assumes that an application consists of one or more software components, which provide their own functions and may be running on different computers. It introduces the adaptive deployment of software components but not of adaptive functions inside software components. If functions inside software components are adapted, other components, which communicate with the adapted ones, may have serious problems. However, the relocation of software components does not lose the potential functions of components. This may seem to be simple but it makes their applications resilient without losing availability, dependability, or reliability. It can also separate adaptation from components, which define application logic, because components themselves are defined and executed independently of any adaptation.

III. 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 on distributed systems.

A. Dynamically adaptive system architecture

Our middleware architecture consists of three parts (Figure 2). The first part is the adaptation manager, the second part is runtime system and the third part is distributed applications. From this architecture, we can notice that:

- The first part is a component runtime system, which is responsible for executing software components, migrating software components and enabling them to invoke methods at other software components. However, by using these methods, the software components need to be serialized in the first, then migrate themselves from one computer to others. When these software components arrived at destination computers, they can communicate with the components of destination computers, according to naming inspection.
- The second part is adaptation manager, which is responsible for our runtime system. However, it can control the behaviour of components, select policies from destination database and fetch them and determine themselves where the software components should be moved. The policies are written by an Event-Condition-Action format scripting language.

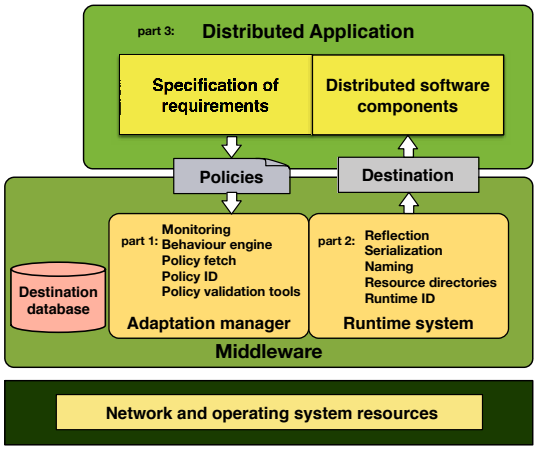

Figure 2. Middleware architecture.

- The third part consists in distributed applications, which can be designed by any general-purposes.

B. Component runtime system

Every runtime system allows each component to have at most one activity through the Java thread library. When the life-cycle state of a component is changed, e.g., when it create, terminates, duplicate, or migrates to another computer. The runtime system issues spcific events to the software 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. Through this method, we can easily hide the differences between the interfaces of objects at the original and other computers. Each runtime system can exchange components with other runtime system through TCP channels by using Object Input/Output Stream. When a component is transferred over networks, not only the code of the components, but also their state can be transmitted into a bit stream by using Java's object serialization package, and then the bit stream is transferred to the destination computers. The runtime system which is run on the receiving side receives and unmarshals the bit stream. When components have been deployed at destination computers, their methods should still be able to be invoked from other computers, which are running at local or remote computers.

C. Adaptation mechanism

The policy-based deployment of components is managed by adaptation managers, where they are running with software component runtime systems and they have not any centralized management servers. Each component runtime system periodically advertises its address to other runtime systems through UDP multicasting, and then these computers can return their addresses and capabilities to the destination computers through TCP channels. 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., network connections and application-specific conditions and the utility rates and processing capabilities of processors. The latter is specified as a relocation of

 ISBN: 978-1-61208-378-0

components. Our adaptation was intended to be specified in a rule-style notation.

1) Adaptation policy specification format: The adaptation manager offers an interpreter to execute the specification format. Since we need to predict conflicts and divergences that result from adaptations, we need to design a format for specifying adaptations policies, which are given as the relocation of components according to changes in their systems and the requirements of their applications. The format consists of conditions at destination parts. The two parts are defined based on a theoretical foundation to verify the validation of adaptations. The former is written in a first-order predicate logic-like notation, where predicates reflect information about the system and applications. e.g., the utility rates and processing capabilities of processors, network connections, and application-specific conditions. The latter represents the deployment and duplication of components in our adaptation instead of any application-specific behaviors, including communications and state transition, of the components. It is formulated as a process calculus.

Since policies are written in a XML format (as Figure 3 shows), it can be defined outside components. In addition, these user-defined policies can be reused for other components, and the components can be reused with other policies. For now, our adaptation manager provides four built-in policies. Each policy contains [Event-Condition-Action] three main tags. We can define the name of this policy in *Event* tag. The *Condition* tag shows when the software components should be migrated or not. It can be freely defined by the users or developers of distributed systems. Our interpreter is different from the existing researches [13][15], because our approach is focused on components migration, so in *Action* tag, we can judge the software components where to relocate or whether to migrate or not.

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy id = >
   <!-- Event of Policy  -->
        <Event>
              <Name>  arir1  </Name>
        </Event>
        <!-- Condition of Policy.e.g., predicate1,...predicaten  -->
        <Condition>
           adaptive_file_size >= max_file_size ll
                 adaptive_bandwidth >= fixed_value
        </Condition>
        <!-- The destination of relocation  -->
        <Action>
              <destination>
              If (condition == true){
                    migrate (component 1,...,component N,
                         destination_IP, destination_port )
                    recall (component 1,...,component N,
                         method_name )
                 } else {
                    unMigrate ()
                     }
              </destination>
        </Action>
</</Policy>
```

Figure 3. Policy format.

For example, to adapt remote information retrieval. In this sense, we can define an attraction policy and execute this format in adaptation manager as follows:

- When a component has an attraction policy for another component, if the condition specified in the policy is satisfied, the policy instructs the the former to migrate to the current computer of the latter.
- When a component has a spreading policy, the policy will make a copy of the component and instructs the copy to migrate to the current computer.
- When a component has a repulsion policy for another component, if this computer have the same or compatible components, this policy will migrate this component which communication with another component to the current computer.
- When a component has an evaporation policy, if the condition specified in the policy is satisfied, it terminates.

When the external system detects changes in environmental conditions, the runtime system can self-adaptive to migrate the search component to remote computer. If this remote computer is failure or waiting processing in threads, the search components can relocate to other computers, according to the user-defined policies. Then, the search component can fetch files inside of itself. Once the retrieval completed, the search component will return back, according to the attraction policy. However, the details will be described in Section 5.

IV. IMPLEMENTATION

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

A. Component runtime system

Each component is a general-purpose and programmable entity, which defined as a collection of Java objects and packaged in the standard JAR file format. It can migrate and duplicate themselves between computers. Our runtime system is similar to a mobile agent platform, 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. Our middleware system is built on the Java Virtual Machine (JVM), so it can abstract away between different operating systems.

The current implementation basically uses the Java object serialization package to marshal or duplicate components. The package dose 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. Furthermore, this system suspends their active threads. We also implement this system by using our original remote method invocation between computers instead of Java Remote Method Invocation (RMI); this is because Java RMI dose not support object migration.

B. Adaptation manager

The adaptation manager is running on each computer and consists of two parts: a database of policies and an event manager. The former will compile and execute user-defined policies and the latter will receive events from the external systems and notify changes in the underlying systems and applications.

 ISBN: 978-1-61208-378-0

We describe a process of the relocation of a software component, according to user-defined policies.

- When a component creates and arrives at a computer, it automatically registers its deployment policies with the database of the current adaptation manager.
- The manager periodically evaluates the conditions of the policies maintained in its database.
- When it detects the policies whose conditions are satisfied, it deploys software components at destination computer, according to the selected policies migrate component to the destination computer and dynamically invoke the methods of destination software components.

Two or more policies may specify on different destination computers, under the same condition that drive them. The current implementation provides no mechanism to solve conflict between policies. So, we assume that policies would be defined without any conflicts right now. The destination components may enter divergence or vibration models due to conflicts between component policies. In addition, they may have multiple deployment policies. However, the current implementation dose not exclude such divergence or vibration.

V. APPLICATIONS

This section describes two applications of the proposed approach.

A. Adaptive remote information retrieval

Suppose users who are using search engines to find 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 or the size of a component for searching patterns from data is bigger than the volume of target data, the approach is efficient. In this case, the components should be executed at remote computers that maintain the target data rather than at local computers. However, it is difficult to select where the components is to be executed because the volume of the result may be not known before. The proposed approach can solve this problem by relocating such components from local computers to remote computers while they are running. Figure 4 shows our system for adaptive remote information retrieval, which consists of client, search, and data access manager components. The client component and the data access manager component are stationary components. The search component supports finding text lines that match certain patterns provided from the client component in text files that it accesses via the data access manager component. It has an attraction policy that relocates itself from local to remote computers when the volume of its middle result is larger than the size of component; otherwise, it relocates from remote to local computers.

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 application to be available with limited resources and networks. Our approach is self-adaptive in the sense that it enables the search component to have its own adaptation

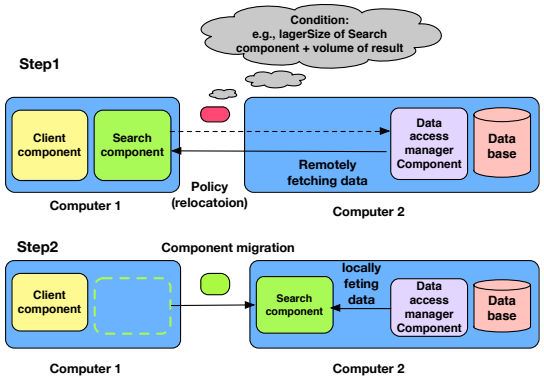

Figure 4. Adaptive remote information retrieval.

policy and manages itself according to the policy independently of these components themselves. It is independent of its underlying systems because the destination of our component relocation is specified as components, instead of computers themselves and they can be reused also. Furthermore, when the remote or local computer is failure, our system can migrate the retrieval programs to another computer, according to the user-defined policies for go on progressing without termination processes. This is the difference between traditional approaches and ours.

B. Adaptive distributed media service

Suppose a media distribution service, including video streaming [17]. Because of the sizes of media contents tend to be large recently, so that the cost of transmitting such contents from back-end servers to front-end servers and from front-end servers to clients becomes huge. Therefore, it is necessary in the vicinity of the data processing to save the high costs of data transmission.

We assume that users send requests to front-end servers, and convert video data via front-end servers before fetching them. Figure 5 shows our system for adapting distributed media service, which consists of client, search, Drawing UI component, and data access manager components. The client component and the data access manager component are stationary components. The search component supports to response the requests from users. When the tasks becomes excessive in a short time, the front-end server will become a bottleneck. The Drawing UI component supports to adapt to the size of the screens.

In this case, clients may fail to connect to or have to wait while data have processed by front-end servers. Therefore, the front-end servers components should be dynamically deployed at the back-end servers and processed at the back-end ones on behalf of the front-end ones. If the back-end servers have enough resources for processing. The contents at back-end servers do not need to be relocated and copied. This approach can reduce the cost of transmitting the content from the back-end servers to the front-end servers so that it is useful to avoid heavy traffic between the servers.

On the other hand, we assume that users try to select media contents from front-end servers, because they are responsible for managing the selections of contents and drawing UIs for

 ISBN: 978-1-61208-378-0

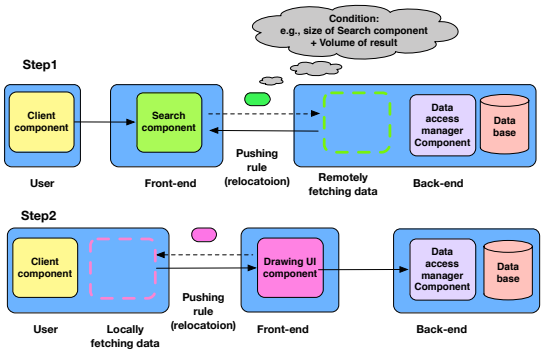

Figure 5. Adaptive distributed media service.

users to do. By using our approach, we can relocate the running components for selecting media from front-end servers to clients when clients have much capability to manage and draw UIs.

These deployments of software components can be specified in our policy-based middleware for adaptations and automatically invoked methods of destination components when conditions of policies are satisfied.

VI. Experiment

In this paper, we present the implementation of the proposed system on OS X, which has Intel Core i7 2GHz as CPU and 8GB memory and the download/upload speed of internet is 8.586KB/s and 15.83KB/s. The implementation uses raw socket to obtain all packages and is described in Java programming language. For experiment, this paper prepares a network environment where to adapt remote information retrieval. The experiment in this paper compares the cases use and non-use the proposed adaptation middleware system. In the experiment, we assume that one user retrieves a keyword named JAVA and we searched the keyword in three types of files.

A. Result of Experiment

The first one type of file has the data size of 17KB. The second one that the size of data is 1.1MB. The data size of the third one is 104MB. In addition, we compare the speed of search the same keyword in our system. Figure 6 shows the result of experiment for adapting remote information retrieval.

Keyword	File size	With adaptation	Without adaptation
JAVA	17KB	2.11sec	1.39sec
JAVA	1.1MB	4.62sec	4.97sec
JAVA	104MB	6.17sec	13.12sec

Figure 6. Results of experiment.

By the three sets of data, we know that if the file becomes larger, our self-adaptive middleware system can significantly reduce the time of remote information retrieval. From the first set of data, we know that when the size of retrieved file is almost as large as the retrieval programs, the processing time of our self-adaptive middleware system is longer than non-adaptation system. This is because, the deploy objects of software components in our middleware require serialization/de-serialization between local and remote sides. However, the non-adaptation system does not require its. From the second set of data, we know that when the size of retrieved file is almost 10 times larger than the retrieval programs, our system is slightly stronger than without adapted remote information retrieval. However, by the last set of data, we clearly know that when the size of the file becomes 100 times, the search time of our middleware system will spend half time of non-adaptation information retrieval. Furthermore, we can not only reduce remote information retrieval time through our approach, we can also enhance distributed systems reliability, dependability ,and availability. For example, when the remote computer fails, our system will temporarily freeze the retrieval programs, and migrate its to another computer. Then these programs will be thawed and resumed in remote side.

VII. Related Work

This section describes a selection of related research in the fields of distributed systems. It compares our approach with several existing adaptation approaches for distributed systems.

Many researchers have explored adaptation mechanisms for distributed systems [4][5][18]. They can be classified into three types. The first is to dynamically change coordination between programs, which are running on different computers for their adaptation, e.g., CORBA-based middleware [6][7][8][10]. This is one of the most typical adaptive coordination that enables client-side objects to automatically select and invoke server-side objects according to changes in their requirements of applications or system architectures. However, this type is limited. Because it only modifies the relationships between distributed programs instead of the computers, which are executing them. The second is to change programs for defining functions of which an applications consists, e.g., genetic programming [11]. It needs more resources to select generations of programs. On the other hand, it is difficult to predict their adaptation. Distributed systems should be predictable because they are often used for mission-critical applications. The third type is policy-based middleware on distributed systems [2][3][12][16]. By using policies to define the conditions of software components, these approaches can migrate the components to specified computers by a specified adaptation language, seems like [1][13][15]. However, the specified computers may be not a good choose. Because the specified computers may be have not enough available resources or the processing of threads are waiting several task or the connection has been broken. Conversely, our proposed approach can autonomously select the destination of deployed software components. Therefore, they do not care the computer is a specified one or not. It is a more general-purpose.

On the other hand, our approach can change the computers that execute programs for self-adaptation. Therefore, it enables the programs to escape from computers, which may have system failures or be shutdown.

The relocation of software components have been studied in the literature on mobile agents [9][14]. By using the technology, we may be able to dynamically relocate the executing programs of components. Furthermore, like ours, several

 ISBN: 978-1-61208-378-0

mobile agent platforms support mechanisms for mobility-transparence, where the mechanisms enables programs, which may be migrated to remote computers to continue to work on other computers. However, the technology itself does not intend to support adaptation so that it cannot abstract away adaptation from application-developers. Furthermore, our approach is inherently designed for dynamic adaptation on distributed systems.

VIII. CONCLUSION

This paper proposed an approach to adapt distributed applications by predefined or user-defined policies. It introduced the relocation of software components between computers as a basic mechanism for adaptation. It separated software components from their adaptations in addition to underlying systems by specifying policies outside the components. It is simple, but it 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 may have security problems, e.g., executions malicious programs, but it is can be solved by receiving only the programs that are transmitted from secure and reliable computers with authentication techniques. 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. We described several approaches with practical applications.

REFERENCES

[1] J.-X. Li, B. Li, L. Li and T.-S. Che, "A policy language for adaptive web services security framework." In: Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. Eighth ACIS International Conference on. IEEE, 2007. pp. 261-266.

[2] J. Keeney and V. Cahill, "Chisel: A policy-driven, context-aware, dynamic adaptation framework." Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on. IEEE, 2003. pp. 3-14.

[3] K. Yang, G. Alex and T. Chris, "Policy-based active grid management architecture." Networks, 2002. ICON 2002. 10th IEEE International Conference on. IEEE, 2002. pp. 243-248.

[4] A. Tripathi, "Challenges designing next-generation middleware systems." Communications of the ACM 45.6 ,2002, pp. 39-42.

[5] V. Issarny, C. Mauro and G. Nikolaos, "A perspective on the future of middleware-based software engineering." 2007 Future of Software Engineering. IEEE Computer Society, 2007. pp. 244-258.

[6] B. Gordon S, B. Lynne, I. Valerie, T. Petr and Z. Apostolos, "The role of software architecture in constraining adaptation in component-based middleware platforms." Middleware 2000. Springer Berlin Heidelberg, 2000. pp. 164-184.

[7] D. Kulkarni and T. Anand, "A framework for programming robust context-aware applications." Software Engineering, IEEE Transactions on 36.2. 2010. pp. 184-197.

[8] R. Olejnik, B. Amer and T. Bernard, "An object observation for a Java adaptative distributed application platform." Parallel Computing in Electrical Engineering, 2002. PARELEC'02. Proceedings. International Conference on. IEEE, 2002. pp. 171-176.

[9] I. Satoh, "Mobile agents." Handbook of Ambient Intelligence and Smart Environments. Springer US, 2010. pp.771-791.

[10] J. Zhang and B. H. Cheng, "Model-based development of dynamically adaptive software." Proceedings of the 28th international conference on Software engineering. ACM, 2006. pp. 371-380.

[11] Koza and R. John, "Genetic programming: on the programming of computers by means of natural selection." Vol. 1. MIT press, 1992.

[12] M. Luckey and E. Gregor, "High-quality specification of self-adaptive software systems." Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. IEEE Press, 2013. pp. 143-152.

[13] Anthony and J. Richard, "A policy-definition language and prototype implementation library for policy-based autonomic systems." Autonomic Computing, 2006. ICAC'06. IEEE International Conference on. IEEE, 2006. pp. 265-276.

[14] R. Montanari, T. Gianluca and S. Cesare, "A policy-based mobile agent infrastructure." Applications and the Internet, 2003. Proceedings. 2003 Symposium on. IEEE, 2003. pp. 370-379.

[15] N. Damianou, D. Naranker, L. Emil and S. Morris, "The ponder policy specification language." Policies for Distributed Systems and Networks. Springer Berlin Heidelberg, 2001. pp. 18-38.

[16] D. Ferraiolo and G. Serban, "The Policy Machine: A novel architecture and framework for access control policy specification and enforcement." Journal of Systems Architecture 57.4. 2011. pp. 412-424.

[17] Z.-J. Lei and D. G. Nicolas, "Context-based media adaptation in pervasive computing." Electrical and Computer Engineering, 2001. Canadian Conference on. Vol. 2. IEEE, 2001. pp. 913-918.

[18] A. Uribarren, J. Parra1, R. Iglesias1, J. P. Uribe1 and D. Lopez-de-Ipina, "A middleware platform for application configuration, adaptation and interoperability." Self-Adaptive and Self-Organizing Systems Workshops, 2008. SASOW 2008. Second IEEE International Conference on. IEEE, 2008. pp. 162-167.

 ISBN: 978-1-61208-378-0