Electronic Preprint for Journal of Information Processing Vol.24 No.6

Regular Paper

Theory and Implementation of an Adaptive Middleware for Ubiquitous Computing Systems

Jingtao Sun[1,a)] Ichiro Satoh[1,2,b)]

Received: December 21, 2015, Accepted: July 5, 2016

Abstract: An adaptive middleware system for ubiquitous computing environments, which are dynamic by nature, is proposed. The system introduces the relocation of software components to define functions between computers as a basic mechanism for adaptation on ubiquitous computing. It also defines a language for specifying adaptation policies. Since the language is defined on a theoretical foundation, it enables us to reason about and predict adaptations beforehand. It is also useful to detect conflicts or divergences that may be caused by the adaptations. It supports general-purpose software components and the relocation of the components according to policies described in the language. We describe the design and implementation of the system and present two practical applications.

Keywords: adaption, disaggregated computing, software component, migration

1. Introduction

Ubiquitous computing systems are used as infrastructures in the real world and provide a variety of applications for multiple users who have their unique requirements. However, ubiquitous computers tend to have limited computational resources and they cannot always maintain and execute all the required applications for all the users. The system should be adapted only to the requirements of running applications and present users and not to idle applications and absent users in order to save computational resources. These systems are dynamic by nature compared to distributed systems on clustering servers, including cloud computing systems. This is because computers may be added to or removed from them and networks between computers may often be connected or disconnected, even while the computers are running. Ubiquitous computing systems should therefore be adapted to such changes.

Several researchers have attempted to introduce adaptations into ubiquitous computing systems. Ubiquitous computing systems need to support availability, dependability, and reliability because they are often used for mission-critical purposes. However, the adaptations themselves may result in failure or uncertainty in the sense that their effects may seriously affect the systems. The systems need to reduce the degree of uncertainty resulting from their adaptations. Also when multiple adaptations are activated, there may be conflicts between them. Adaptations may need to be activated infinitely, if their conditions are satisfied after they have been performed. Applications consisting of software components that may be running on different computers should be resilient so that the systems can adapt themselves to various changes at runtime.

This paper presents an adaptive middleware system for ubiquitous computing. There are two key ideas behind the proposed system. The first is to introduce the relocation of software components as a basic adaptation mechanism. Our adaptation does not change application-specific software components so that it reduces the degree of uncertainty associated with it; rather it duplicates and migrates software components, which may be running, to remote computers in accordance with changes in system environments and application requirements. When changes in a distributed system occur, e.g., in the requirements of the applications and the structures of the system, the software components that it consists of are automatically relocated to different computers according to policies defining their adaptations. The second idea is to formalize our adaptations on a theoretical foundation defined as a process calculus. This would enable us to specify the adaptations and analyze their effects, e.g., where and what functions are provided after adaptation. Our middleware system can specify and interpret policies for adaptations on the basis of the foundation outside software components.

2. Related Work

This section outlines related work. There have been many attempts to introduce the notion of adaptation into ubiquitous computing systems [5], [11], [25]. Existing adaptation approaches for distributed systems can be classified into three types, discovery-based, connection-based, and software-level approaches.

The first approach has been explored as adaptive discovery mechanisms for services and resources. For example, Jaeger et al. [9] introduced the notion of self-adaptation to an object request broker such as CORBA. When clients want services, their broker selects services in accordance with contexts. Cheng et al. [4] presented an adaptive selection mechanism for servers by enabling

[1] Department of Informatics, The Graduate University for Advanced Studies, Chiyoda, Tokyo 101–8430, Japan
[2] National Institute of Informatics, Chiyoda, Tokyo 101–8430, Japan
[a)] sun@nii.ac.jp
[b)] ichiro@nii.ac.jp

selection policies. EASY [18] enabled services to be discovered according to the specifications of functional and non-functional properties of the services in ubiquitous/pervasive computing environments. This approach cannot adapt coordinations between multiple services.

The second approach allows communication between software components with connectors or unified interfaces to be reconnected/disconnected in accordance with changes. Garlan et al. [7] presented a framework called Rainbow that provided a language for specifying self-adaptation. Although the framework was not solely aimed at distributed and ubiquitous computing systems, it supported adaptive connections between operators of components that might be running on different computers. Lymberopoulos et al. [14] proposed specifications for adaptations based on their policy specification language, called *Ponder* [6], but these were aimed at specifying management and security policies rather than application-specific processing and therefore did not support the mobility of components. These approaches assume that certain software components are statically provided at computers beforehand. Therefore, they cannot deploy components at computers that do not have components, when the computers are newly connected to a distributed system, or when other computers become busy.

The third approach enables software to be dynamically modified in accordance with changes. Genetic programming enables software to be evolved by randomly mutating and crossing over programs under specified evaluation metrics. Ubiquitous computing systems have no resources to execute and evaluate large numbers of generated programs. Computational reflection aspect-oriented programming (AOP) enables software to be open to dynamically defining itself without compromising portability or exposing parts unnecessarily, where the software 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. Many researchers have introduced AOP into adaptive distributed systems. For example, McKinley et al. [15] proposed a middleware system with compositional adaptation by using AOP. They can modify parts of programs running on single computers but do not support distributed systems themselves. Satoh et al. [22] proposed a bio-inspired adaptation by introducing the notion of cellular-differentiation into distributed systems to change available functions in accordance with the frequency of invoking the functions from the external system; however the adaptation cannot migrate any functions between computers.

A few researchers have attempted to adapt the relocations of program codes or components. EXEHDA [12] and SATIN [26] dynamically deployed application codes at computers in accordance with the movement of users, but they lack any mechanisms for specifying user-defined adaptations and cannot migrate components that have already been executed. Suda et al. [23] proposed a bio-inspired middleware system for disseminating network services in dynamic and large-scale networks where there were large numbers of decentralized data and services. The system enabled agents to be replicated, moved, and deleted like ours, but their purpose was to introduce biological metaphor into distributed systems rather than adaptive approaches. The FarGo system introduced a mechanism for distributed applications dynamically laid out in a decentralized manner [8]. The system could not specify any conditions for their policies, unlike ours. Satoh [20] proposed other relocation policies for relocating components based on policies that other components moved to. These existing attempts at relocating program codes or components have no mechanism to specify policies for their relocations outside components.

The language proposed in this paper is defined on the basis of process calculi for distributed systems [17]. Several researchers have attempted to extend the notion of adaptation into process calculi. For example, Bravetti et al. [1] proposed a theory for adaptive processes based on higher-order process calculi. It treated an adaptation process as the replacement/substitution of processes by other processes. Brogi et al. proposed a process calculus with the notion of dynamic adaptors between components [2]. However, existing process calculi for adaptations did not support the mobility of processes. Melliti el al. [16] proposed a labelled transition system for specifying autonomic service composition in a ubiquitous computing setting. These attempts did not support specifying the relocation of components and lack any implementation of their calculi. There have been several attempts to define formal models for the mobility of processes or agents, e.g., Ref. [19]. However, these attempts aimed at specifying application-specific behaviors of mobile processes or agents, whereas our approach focuses on adaptation policies.

3. Approach

This section outlines the requirements of the proposed system and the basic ideas underlying it.

3.1 Requirements

Adaptations in ubiquitous computing systems have unique requirements that those in distributed systems may not have. We assume that applications in such systems consist of one or more software components that may be running on different computers.

- *Self-adaptation:* Like distributed systems, ubiquitous computing systems essentially have no global view due to communication latency between computers. They need to adapt themselves to various changes in ubiquitous computing systems without any centralized management systems.
- *Separation of concerns:* Adaptations should be able to be reused independently of applications, and vice versa. Adaptations should be defined outside application-specific components or the underlying systems.
- *Predictability and availability:* Adaptations can dynamically modify ubiquitous computing systems but may result in uncertainty or failure, so their effects should be predictable beforehand. Also, our system should enable applications to remain available after their adaptations.
- *Detection of conflicts and divergences:* Although individual adaptations may be appropriate, they may cause serious problems, e.g., *conflict* and *divergence*, when they are activated simultaneously. For example, one adaptation may deny the effects of one or more the other activated adapta-

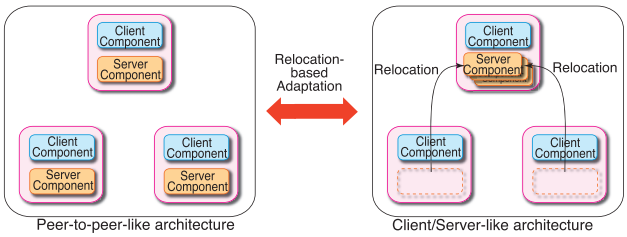

Fig. 1 Adaptation as relocation of software component.

tions. Adaptations may be activated infinitely if their conditions are satisfied after they have been performed. We should detect conflicts and divergences in multiple adaptations as much as possible.

- *Limited resources:* Ubiquitous computers often have only limited resources, e.g., processor power, amount of memories. Our adaptations should be able to save computational resources and be performed with few resources.

3.2 Adaptation for Ubiquitous Computing

To satisfy the above requirements, we introduce the relocation of software components as a basic mechanism for adapting ubiquitous computing systems (**Fig. 1**). The architectures of distributed systems, including ubiquitous computing systems, depend on the location of software for defining functions as well as the functions provided by the software. For example, client-server (C/S) and peer-to-peer (P2P) are often used for the same applications, e.g., file sharing. The function provided by each peer in a P2P architecture corresponds to the combination of functions provided on the client and server sides in a C/S architecture, although P2P may have a mechanism for discovering other peers. Therefore, our system changes the locations of software components for defining application-specific functions rather than coordinating between the components. Our adaptation can make applications resilient without losing availability because the relocations of software components do not lose the potential functions of components. This seems to be simple and conservative but even so would be enough to support most adaptive applications in ubiquitous computing systems.

There have been several techniques to relocate software components between computers, e.g, mobile agents, code mobility, and remote evaluation [21]. The first enables software components to continue to their processing at destinations after their migration, the second needs to restarts program codes after deploying the codes at the destinations, and the third can change the the location of software only during the execution of the software. Therefore, the proposed approach is based on the first.

To separate adaptations from the software components that define application logic, we introduce a language for specifying adaptation policies outside the components. The language is constructed on the basis of a theoretical foundation to predict the effects of adaptations. The language consists of condition and destination parts, with the former written in a first-order predicate logic-like notation and the latter in a process calculus and to specify the deployment and duplication of components. Although individual adaptations may be appropriate, they may cause serious problems, e.g., *conflict* and *divergence*, in ubiquitous computing systems. The language helps to detect conflicts and divergences in multiple adaptations.

4. Adaptation Policy Specification Language

This section defines our policy specification language for relocation-based adaptation based on a process calculus for communicating systems [17]. The operational semantics of the language is given in the Appendix. We first define several notations:

- Let $\mathcal{I}$ be a set of the identifiers of components with conditions, ranged over by $A, B, \ldots$.
- Let $\mathcal{N}$ be a set of location names, ranged over by $n, n_1, n_2, \ldots$.
- Let $\mathcal{X}$ be a set of location variable or function, ranged over by $x, \ldots$, and $Loc(A)$ returns the current location of component, called A.
- Let $\mathcal{L}$ be $\mathcal{N} \cup \mathcal{X}$, ranged over by $\ell, \ell_1, \ldots$.
- Let $\mathcal{M}$ be a set of the signatures of methods, which are defined in components and can be invoked from policies, ranged over by $m, m_1, \ldots$.

Our framework enables users to specify user-defined policies for adaptation by means of the expressions defined below.

Definition 4.1 Set $\mathcal{D}$ of located process expressions, ranged over by $D, D_1, D_2, \ldots$, is the smallest set containing the following expressions:

D, D_1, D_2	::=	$\ell[E \mid P]$	(*Located component*)
	|	$D_1 \parallel D_2$	(*Distributed components*)
E, E_1, E_2	::=	C `then` G	(*Conditional action*)
	|	$E_1 + E_2$	(*Alternative selection*)
	|	$E_1 ; E_2$	(*Sequential composition*)
	|	$\mathbf{0}$	(*Termination*)
G	::=	`moveTo`(ℓ)	(*Migration*)
	|	`copyAt`(ℓ)	(*Duplicatoin & migration*)
	|	`remove`	(*Elimination*)
	|	`callback`(m)	(*Action*)
P, P_1, P_2	::=	P_1 , P_2	(*Composition*)
	|	A	(*Component*)
	|	ϵ	(*No component*)

where C is a condition defined in Definition 4.2. $E ; 0$ is often abbreviated as E. □

We describe several constructors' intuitive meanings in the language. $\ell[E \mid A]$ means that a component A located at ℓ is executed as an expression specified as E. $D_1 \parallel D_2$ represents distributed components D_1 and D_2 executed in parallel. $E_1 + E_2$ may behave as E_1 or E_2 and $E_1 ; E_2$ is executed as E_2 after E_1. For example, $\ell_1[C$ `then moveTo`$(\ell_2) \mid A]$ means that if condition C is positive, a component A located at computer ℓ_1 is relocated to ℓ_2, where `moveTo`(ℓ_2) is a migration to ℓ_2. ℓ_1[`copyAt`$(\ell_2) \mid A$] means that a component A located at computer ℓ_1 make a copy of A at computer ℓ_2, ℓ_1[`remove` $\mid A$] means that a component A terminates, and ℓ_1[`callback`$(m) \mid A$[means a component A invokes a callback method m[*1]. $\ell[C_1$ `then copyAt`(ℓ_2) +

[*1] The `callback`(m) primitive may make polices dependent on components. Nevertheless, the language supports the primitive for practice.

C_2 then callback(m); remove | A, B] means that if condition C_1 is positive then two components A and B are the policy that makes copies of A and B and deploys the copies at ℓ_2. If condition C_2 is positive, the policy invokes a callback method named as m, in A and B and then terminates A and B. Otherwise, A and B stay at their current computer. The adaptions specified in the language can be predicted in the sense that the operational semantics of the language can emulate the effects of the adaptions.

Next, we define our conditional functions as propositional logic predicates [*2].

Definition 4.2 The set of conditions, C, is the smallest set containing the following expressions:

$$C, C_1, C_2 \quad ::= \quad \phi \mid \neg C \mid C_1 \wedge C_2 \mid C_1 \vee C_2 \mid true \mid false$$

where ϕ is a logical predicate symbol and returns either *true* or *false* with more than a zero parameter. □

The ϕ in Definition 4.2 represents a user-defined function in components or a system's built-in function. The former is provided as a method from the component that its policy is assigned to. It should be application-specific and defined by users, and the current implementation provided several built-in functions:

Definition 4.3 Each predicate can have zero or more parameters and return either *true* or *false*.

- exist(A, ℓ) (exist : $\mathcal{P} \times \mathcal{L} \rightarrow$ *true or false*) returns *true* if the same or compatible component(s) of component A exists at location ℓ, otherwise it returns *false* [*3].
- delay(*time*) (delay : $\mathcal{T} \rightarrow$ *true or false*) blocks the following execution for the *time* interval and then returns *true*, where $\mathcal{T}$ is an infinite set of relative time values.
- receive(m, ℓ, A) (receive : $\mathcal{M} \times \mathcal{L} \times \mathcal{P} \rightarrow$ *true or false*) returns *true* if the component that the policy is assigned to receives a message labelled as m from component A. □

User-defined functions are implemented inside components or the runtime system. User-defined functions defined in components can be accessed inside the components. We will now present typical adaptation policies. The destinations of component relocations may not be specified beforehand. $\mathcal{X}$ can contain a function, written as *destination*(*Spec*), that returns the name of a location that can satisfy the condition written as *Spec*.

Example 4.4 Basic adaptations illustrated in **Fig. 2**.

- *Attraction:* The following policy is assigned to component A at computer ℓ. If a computer, called @*another*, has the same or compatible component of component A, the policy instructs A to migrate to @*another*, and then follows E.

  $$\ell[\texttt{exist}(A, @another)\ \texttt{then}\ \texttt{moveTo}(@another); E \mid A]$$

- *Spreading:* The following policy is assigned to component A, at computer ℓ. If a computer, called @*another*, does not have the same component A, the policy makes a copy of component A and deploys the copy at the @*another* computer and then behaves as E, where A' is the copy.

  $$\ell[\neg\texttt{exist}(A, @another)\ \texttt{then}\ \texttt{copyAt}(@another); E \mid A]$$

[*2] The current implementation itself supports a first-order logic predicates, but the paper uses predicates as propositional logic ones.

[*3] In the current implementation of the proposed system, *compatible components* of a component are defined through the subclasses of the component by using Java's inheritance mechanism.

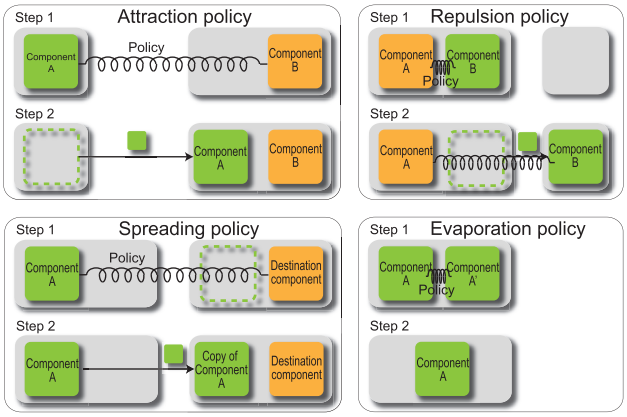

Fig. 2 Basic policies for adaptations.

- *Repulsion:* The policy is assigned to component A. If there are the same or compatible components of A at computer ℓ, the policy instructs component A to migrate to another computer, called @*another*, and then follows E.

  $$\ell[\texttt{exist}(A, \ell)\ \texttt{then}\ \texttt{moveTo}(@another); E \mid A]$$

- *Evaporation:* The policy is assigned to component A. If there are one or more components A at the current computer, the policy eliminates redundant component A at the computer specified at ℓ.

  $$\ell[\texttt{exist}(A, \ell)\ \texttt{then}\ \texttt{remove}; \mathbf{0} \mid A]$$

- *Time-To-Live:* After a certain time has passed, the policy terminates component A.

  $$\ell[\texttt{delay}(t)\ \texttt{then}\ \texttt{remove}; \mathbf{0} \mid A]$$

There is a variety of changes with which a ubiquitous computing system needs to cope. Such a system may have multiple adaptations to such changes. These adaptations may be in conflict with others, even when each of them is appropriate. For example, an adaptation invoked at a computer may seriously affect other systems. Although an adaptation itself is useful to improve its target system's performance, it may be counter to the reliability of the system. We can formally analyze several properties of adaptation through the components described in our language.

Property 4.5 The adaptation of a component described as $\ell[C_1\ \texttt{then}\ E_1 + \cdots + C_n\ \texttt{then}\ E_n \mid A]$, is *locally conflict*, if more than two of $C_1, \ldots C_n$ are positive at a location. This is *global conflict* if and only if more than two of $C_1, \ldots C_n$ can be positive at every location.

Local conflicts at certain locations can be detected by evaluating the overlaps of conditions of two or more policies at the locations, whereas global conflicts can be detected by the overlaps of conditions of two or more policies at arbitrary locations. When detecting local or global conflicts by using Property 4.5, we should modify adaptation policies. Conflicts in policies may cause uncertainty. For example, if the conditions of two policies are overlapped, we cannot expect which policies can be activated when the overlapped conditions are satisfied.

There may be no guarantee that activated adaptations can be stopped. Suppose an adaptation is invoked and executed at a

change because the change satisfies the condition of the adaptation. After the adaptation is executed, the condition is still satisfied so that the adaptation is invoked again. When the condition of the effectiveness of an adaptation is inappropriate, the adaptation may go into an infinite loop; it becomes divergent.

Property 4.6 A component described as $\ell[C \texttt{ then } E \mid A]$ is locally *divergent* if C is still positive after executing E.

If a component is divergent, its adaptation may be endless. Such an adaptation should be modified so that it is not divergent. We can confirm whether an adaptation is divergent or not by evaluating C condition at every step of E. We can also find divergent adaptation between more than two components by using the language because adaptations written in the language can be interpreted as possible itineraries.

5. Design and Implementation

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 (**Fig. 3**). The first part runs on each computer and is responsible for executing, duplicating, and exchanging components at computers and the second part is responsible for interpreting and analyzing adaptation policies written in our proposed language. The system and components are executed on the Java virtual machine (JVM), that can abstract away differences between operating systems and hardware. Components can be composed from Java objects such as JavaBean modules without their adaptations, where their adaptation policies are specified in the language outside them to support the separation of concerns.

5.1 Component Runtime System

A runtime system can load each component from a set of program codes with state from a local or remote storage and allows each component to have active threads. When the life-cycle state of a component changes, e.g., when it is created, terminates, duplicates, 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.

Each runtime system can exchange components with other runtime systems through a TCP channel using mobile-agent technology. When a 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. The bit stream is transferred to the destination and then received and transformed into components. After the components arrive at the destination, they can continue being processed. Even after components have been deployed at destinations, their methods should still be able to be invoked from other components that may be running at local or remote computers. The runtime systems exchange information about components that visit them to trace the locations of components in a peer-to-peer manner. The current implementation supports an original remote method invocation (RMI) mechanism through a TCP connection independently of Java's RMI. When migrating a component, each runtime system can forward method invocations to/from the destination by using the mechanism.

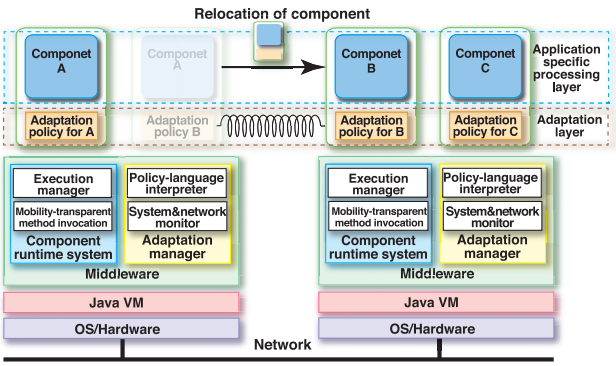

Fig. 3 System structure.

5.2 Adaptation Manager

Each adaptation manager periodically advertises its address to the others by using *system and network monitor* in Fig. 3, where the monitor is responsible for sending other managers, which may be running on different computers, information through UDP multicasting and receiving information from other computers and these computers then return their addresses and capabilities to other computers through a TCP channel [*4]. When the manager does not receive advertisement information from other computers for longer than a specified duration, it assumes the computers or networks to the computers have problems. The manager stores policies for components and provides a mechanism to select policies whose conditions are satisfied. It evaluates the conditions of its storing policies when the external system detects changes in environmental conditions, e.g., user requirements and resource availability.

Each policy is specified based on the language defined in Definition 4.1. The adaptation manager has a database to store policies and offers an interpreter consisting of two parts. The first is responsible for evaluating the conditions of adaptations C, C_1, C_2,... as first-order logic predicates with predicates that 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 responsible for evaluating the itinerary of components, described as E, E_1, E_2, ... on the basis of the operational semantics defined in Definitions A.2 and A.3. Policies for components can be added or removed after after creating the components except while the policies are activated. We will now describe a process of relocating a component according to one of its policies.

(1) When a component is created 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 in a peer-to-peer manner.
(2) The manager then analyzes the policies of its visiting com-

[*4] Each runtime system is assumed to have its own adaptation manager. In the current implementation, a group of runtime systems on different computers can share an adaptation manager to save computational and network resources.

ponents for Properties 4.5 and 4.6. If it detects conflict or divergence in any of them, it blocks them.

(3) It periodically evaluates the conditions of the policies maintained in its database.

(4) When it detects 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 from Property 4.5 under the same condition that drive them. The manager analyzes whether there are conflicts in the policies of its visiting components by using a technique to find conflicts between multiple predicates studied in existing propositional or first-class logic systems. The destination of the component may enter divergence modes, as in Property 4.6. The manager analyzes whether or not the effects of adaptations are repetitious by using a runtime system like that in existing runtime checking techniques [24] based on Property 4.6. The current implementation does not exclude any conflicts or divergences but can predict the presence of typical conflicts or divergences. As mentioned in the previous section, the language permits us to specify the requirements of computers at which components are deployed instead of the addresses of the computers, where the requirements are defined as a set of the constraints or limitations that destination computers must satisfy. In the current implementation constraints are evaluated as a constraint satisfaction problem (CSP) by using an existing tool for symbolic CSP, named JaCop [10].

5.3 Current Status

Our runtime system was implemented as a mobile agent platform, but it has been constructed independently of any existing middleware systems because existing middleware systems, including mobile agents and distributed objects, do not support the policy-based relocation of software components. The current implementation basically uses the Java object serialization package to marshal or duplicate components. This package does not support the capture of stack frames of threads; rather 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 then suspends their active threads. The system 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 can explicitly restrict components so that they can 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 components from, trusted runtime systems. Each component is a general-purpose programmable entity defined as a collection of Java objects such as JavaBeans and packaged in the standard JAR file format. Each can explicitly provide user functions in C when it is evaluated by the adaptation manager. It has no specifications for adaptation inside it but can be migrated to and duplicated in a remote computer by the current adaptation manager.

Table 1 Basic performance.

Adaptation	Time	Remark
Component duplication	15 ms	
Component migration	35 ms	between two computers
Attraction policy	85 ms	in Example 4.4
Spreading policy	140 ms	in Example 4.4
Repulsion policy	115 ms	in Example 4.4
Evaporation policy	14 ms	in Example 4.4

Our implementation was not built for performance, but even so we have performed several experiments on a distributed system of sixteens computers (Intel Core i7 2 GHz) with MacOS X 10.9 and JDK 7 connected through 1-Gbps ethernet networks. **Table 1** shows the costs of executing four policies: *attraction*, *spreading*, *repulsion*, and *evaporation*, described in Example 4.4. Each of the costs was the sum of interpreting the policies, marshaling, authentication, transmission, security verification, decompression, and unmarshaling after the target component or its clone after the adaptation manager detected the changes that should have activated the polices. Distributed systems potentially have several semantics of failures. Nevertheless, the current implementation supports the detection of crash failures at other runtime systems and disconnections in networks, where crash failures means that when a computer has troubles, it stops functioning properly. Each runtime system and adaptation manager consume smaller than 80 MB memory. It can transfer components to other runtime systems through TCP/IP. so that it is independent of any physical networks that support TCP/IP. Our approach is available in limited resources.

6. Applications

This section presents two applications to demonstrate the utility of the proposed approach.

6.1 Supporting Communities of Ubiquitous Computers

Ubiquitous computing systems are often managed in an ad-hoc manner because computers and services may be dynamically added to or removed from the systems. Service discovery mechanisms, e.g., Universal Plag-in-Play (UPnP) and Service Location Protocol (SLP), are used for automatic detection of computers and services. Such mechanisms depend on the scales of ubiquitous computing systems. When there are only a few computers and services, there is no server to support such mechanisms, so each computer or service needs to detect other existing computers and services in a peer-to-peer (P2P) manner. In contrast, when there are many computers and services in a network, P2P-based service discovery mechanisms result in congestion in the network and expansion of information about other computers in each computer. One or a small number of computers should be responsible for managing other computers in a client-server (C/S) manner. Therefore, service discovery mechanisms should adapt their architecture to the number of computers in a network, e.g., from P2P to C/S, and vice versa.

We constructed an adaptive service discovery system as a subset-UPnP to evaluate an adaptation between P2P-based and

C/S-based service discovery mechanisms [*5]. We assumed that each *control-point* component periodically multicasts an *advertisement* message with its address to other *control-point* components in a network through UDP multicasting in a UPnP-like format. We introduced two types of components: *control-point* and *device* components. The former maintained a database for information about computers, e.g., network addresses, names, and their preferences, while the latter had the following policy with a function `receivedFrom(`*n*`,`*t*`)` that returned `true` if it received messages from more than a number of computers, specified as `n`, within a certain duration specified as *t*.

```
@peer[(receivedFrom(n,t) then
  moveTo(computerSending());
    exist(Control_proint, @another) then remove) +
      (¬receivedFrom(n,t) then copyAt(@another))
        | Control_proint]
```

The above policy is for each *control-point* component and means that when a *control-point* component located at a node specified as @*peer* receives messages from more computers than *n* within *t* duration, the `receivedForm(`*n, t*`)` function returns `true`. Next, it relocates the component to the computer specified as a function `computerSending()`, where the function returns the address of the computer from which the component receives *advertisement* messages. When a component receives a lesser number of *advertisement* messages than *n*, it makes a copy of the component at another computer specified as @*another*, which has a *device* component, but not any *control-point* components. This example does not assume which computers should become @*another* because it is just an example of our adaptation. Nevertheless, in real systems a destination specified @*another* should have the capability to work as a server for discovering and managing services.

Each *device* component behaved as: when a *device* component received an *advertisement* message from an unknown *control-point* component at another computer, it returned a *advertisement* message to the *control-point* component. Next, the *control-point* component asked the *device* about the latter's preference. In the second phase, more computers are connected to the network. When a *control-point* component received *advertisement* messages from more than the specified number of other *control-point* components at other computers within the certain duration, the former relocated to one of the computers running the latter and then the former gave its information stored in its database to the latter. Therefore, the number of *control-point* components was reduced so that network traffic could be reduced and coordinations among *control-point* components could change from P2P architecture to C/S one. When a component received a lesser number of *advertisement* messages than the specified number, it distributes its clone at other computers. As a result, their coordinations change from C/S architecture to P2P one. Note that the *device* and *control-point* components were defined independently of the policy.

[*5] There have been many attempts to apply mobile objects or agents to ubiquitous computing, e.g., Refs. [3], [21]. Unlike these attempts, our approach is to use the mobility of components as a mechanism for adaptations in ubiquitous computing.

6.2 Spreading Software for Sensor Nodes

The second example is the adaptive deployment of components over a sensor network. It is a well known that after a sensor node detects environmental changes, e.g., the presence or movement of people, in its area of coverage, some of its geographically neighboring nodes tend to detect similar changes after a period of time. 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 only at nodes around such changes.

We assumed that the sensor field was a two-dimensional surface composed of sensor nodes and that it monitored environmental changes, such as motion in objects and variations in temperature [*6]. Each software component had *spreading* and *time-to-live* policies described in Example 4.4 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 each adaptation manager received an event to notify changes from sensors, it evaluated the policies of the component and 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., when people were walking, the components located at the nodes near the change could detect the change in the same way because clones of the components had been deployed at the nodes. The policy for each of the *Sensing* components is described as:

```
@current[G | Sensing]
+ delay(time) then remove + detect(target) then G
    where G is ¬exist(Sensing, @neighbor)
      then copyAt(@neighbor)
```

where @*current* is an element of $\mathcal{X}$ and specifies the current computer of the component and `detect(`*target*`)` is a user-defined function that returns *true* when the movement of a target is discovered. We can select the destinations by using the *destination*(*Spec*) according to the specification of neighboring nodes. Each clone was associated with a time-to-live (TTL) limit by using a `delay` function and @*neighbor* is an element of $\mathcal{X}$ and specifies the spatially neighboring nodes around the current node. Although a node could monitor changes in interesting environments, it sets the TTLs of its components to their own initial

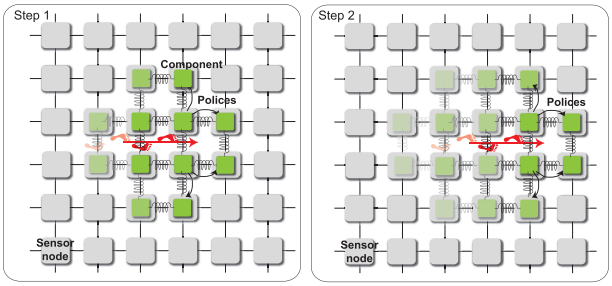

Fig. 4 Component diffusion for moving entity.

[*6] This experiment was evaluated with 256 virtual servers on Amazon EC2.

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 the policy to save computational resources and batteries at the node. The TTL limit of each *Sensing* component was reset when the component detected the target because, when `detect`(*target*) became positive, the policy initially executed G again.

7. Conclusion

In this paper, we presented an adaptive middleware system for ubiquitous computing environments. It introduced the relocation of software components to define functions between computers as a basic mechanism for adaptation on ubiquitous computing systems. Although simple, this provided various adaptations to support the adaptations required in such systems. It also introduced a language for specifying adaptation policies. Since the language was defined on a theoretical foundation, it enabled us to analyze adaptations. It was constructed as a general-purpose middleware system on distributed systems, including ubiquitous computing systems, instead of any simulation-based systems. Two practical applications demonstrated that the system was effective to construct ubiquitous computing systems. In future we will identify further issues that need to be resolved. We need to improve the implementation of the approach. For example, the language should be refined with experiences in specifying a variety of adaptations for ubiquitous computing systems. We also want to develop more applications with the approach to evaluate its utility. As other existing adaptation approaches for distributed systems, our approach cannot guarantee consistency at adaptations. We are interested in proposing adaptations with more strict consistency.

References

[1] Bravetti, M., Di Giusto, C., Perez, J.A. and Zavattaro, G.: Adaptable Processes, *LNCS*, Vol.8, No.4, pp.1–71 (2012).

[2] Brogi, A., Canal, C. and Pimentel, E.: Soft Component Adaptation, *Elec. Notes in Theoretical Computer Science*, Vol.85, No.3, pp.1–16, Elsevier (2003).

[3] Cardoso, R.S. and Kon, F.: Mobile agents: A key for effective pervasive computing, *ACM OOPSLA 2002 Workshop on Pervasive Computing* (2002).

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

[5] Da, K., Dalmau, M. and Roose, P.: A Survey of adaptation systems, *International Journal on Internet and Distributed Computing Systems*, Vol.2, No.1, pp.1–18 (2011).

[6] Damianou, N., Dulay, N., Lupu, E. and Sloman, M.: The Ponder Policy Specification Language, *Proc. Workshop on Policies for Distributed Systems and Networks* (*POLICY'95*), pp.18–39 (1995).

[7] Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B.R. and Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure, *IEEE Computer*, Vol.37, No.10, pp.46–54 (2004).

[8] Holder, O., Ben-Shaul, I. and Gazit, H.: System Support for Dynamic Layout of Distributed Applications, *Proc. International Conference on Distributed Computing Systems* (*ICDCS'99*), pp.403–411 (1999).

[9] Jaeger, M.A., Parzyjegla, H., Muhl, G. and Herrmann, K.: Self-organizing broker topologies for publish/subscribe systems, *Proc. ACM Symposium on Applied Computing* (*SAC 2007*), pp.543–550 (2007).

[10] Kuchcinski, K. and Szymanek, R.: JaCoP Library (2008), available from ⟨http://jacopguide.osolpro.com/guideJaCoP.html⟩

[11] Kakousis, K., Paspallis, N. and Papadopoulos, G.A.: A survey of software adaptation in mobile and ubiquitous computing, *Journal Enterprise Information Systems*, Vol.4, No.4, pp.355–389 (2010).

[12] Lopes, J., Souza, R. and Geyer, C.: A Middleware Architecture for Dynamic Adaptation in Ubiquitous Computing, *Journal of Universal Computer Science*, Vol.20, No.9, pp.1327–1351 (2014).

[13] Luckey, M. and Engels, G.: High-Quality Specification of Self-Adaptive Software Systems, *Proc. Workshop on Software Engineering for Adaptive and Self-Managing Systems* (*SEAMS 2013*), pp.143–152 (2013).

[14] Lymberopoulos, L., Lupu, E. and Sloman, M.: An Adaptive Policy Based Management Framework for Differentiated Services Networks, *Proc. 3rd International Workshop on Policies for Distributed Systems and Networks* (*POLICY 2002*), pp.147–158, IEEE Computer Society (2002).

[15] McKinley, P.K., Sadjadi, S.M., Kasten, E.P. and Cheng, B.H.C.: Composing Adaptive Software, *IEEE Computer*, Vol.37, No.7, pp.56–64 (2004).

[16] Melliti, T., Poizat, P. and Mokhtar, S.B.: Distributed Behavioural Adaptation for the Automatic Composition of Semantic Services, *Proc. International Conference on Fundamental Approaches to Software Engineering, LNCS*, Vol.4961, pp.146–162 (2008).

[17] Milner, R.: Functions as Processes, *Mathematical Structures in Computer Science*, Vol.2, No.2, pp.119–141 (1992).

[18] Mokhtar, S.B., Preuveneers, D., Georgantas, N., Issarny, V. and Berbers, Y.: EASY: Efficient semAntic Service discoverY in pervasive computing environments with QoS and context support, *Journal of Systems and Software*, Vol.81, No.5, pp.785–808 (2008).

[19] Paulino, H. and Lopes, L.: A Mobile Agent Service-Oriented Scripting Language Encoded on a Process Calculus, *Proc. 7th Joint Modular Languages Conference*, *LNCS*, Vol.4228, pp.383–402 (2006).

[20] Satoh, I.: Self-organizing Software Components in Distributed Systems, *Proc. 20th International Conference on Architecture of Computing Systems System Aspects in Pervasive and Organic Computing* (*ARCS'07*), *LNCS*, Vol.4415, pp.185–198 (2007).

[21] Satoh, I.: Mobile Agents, *Handbook of Ambient Intelligence and Smart Environments*, pp.771–791, Springer (2010).

[22] Satoh, I.: Evolutionary Mechanism for Disaggregated Computing, *Proc. 6th International Conference on Complex, Intelligent, and Software Intensive Systems* (*CISIS 2012*), pp.343–350 (2012).

[23] Suda, T. and Suzuki, J.: 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).

[24] Tamura, G.: Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems, *Software Engineering for Self-Adaptive Systems II, LNCS*, Vol.7475, pp.108–132 (2013).

[25] Weyns, D., Iftikhar, M.U., Iglesia, D.G. and Ahmad, T.: A Survey of Formal Methods in Self-Adaptive Systems, *5th International Conference on Computer Science and Software Engineering*, pp.67–79 (2012).

[26] Zachariadis, S. and Mascolo, C.: The SATIN component system - a metamodel for engineering adaptable mobile systems, *IEEE Trans. Softw. Eng.*, Vol.32, No.11, pp.910–927 (2006).

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

Appendix

We give the operational semantics of the language defined in Definition 4.1 based on the structural congruence between expressions $\equiv$ and a reduction relation $\longrightarrow$. The semantics is defined in the style of Milner's transition/reaction relation for π-calculus [17].

Definition A.1 $\equiv$ is structural congruence.

$D \equiv D \qquad D_1 \equiv D_2 \quad then \quad D_2 \equiv D_1$

$D_1 \equiv D_2 \; and \; D_2 \equiv D_3 \quad then \quad D_1 \equiv D_3$

$D_1 \| D_2 \equiv D_2 \| D_1 \qquad D_1 \| (D_2 \| D_3) \equiv (D_1 \| D_2) \| D_3$

$\ell[E \mid P_1] \| \ell[E \mid P_2] \equiv \ell[E \mid P_1 \, , \, P_2]$

$E_1 \equiv E_2 \quad then \quad \ell[E_1 \mid P] \equiv \ell[E_2 \mid P]$

$E \, ; \mathbf{0} \equiv E \quad \mathbf{0} \, ; E \equiv E \quad E_1 \, ; (E_2 \, ; E_3) \equiv (E_1 \, ; E_2) \, ; E_3$

$E + \mathbf{0} \equiv E \quad E_1 + E_2 \equiv E_2 + E_1$

$E_1 + (E_2 + E_3) \equiv (E_1 + E_2) + E_3$

$E_1 \equiv E_2 \quad then \quad C \, \mathtt{then} \, E_1 \equiv C \, \mathtt{then} \, E_2$

$P \equiv P$ $\quad P_1 \equiv P_2$ *then* $P_2 \equiv P_1$

$P_1 \equiv P_2$ *and* $P_2 \equiv P_3$ *then* $P_1 \equiv P_3$

$P_1 \equiv P_2$ *then* $\ell[E \mid P_1] \equiv \ell[E \mid P_2]$ □

We provide no structural congruence over C, but two logic predicates are the same if they are equivalent in first-order logic. The operational semantics of the language is given by two kinds of transition relations: $\longrightarrow \subseteq \mathcal{E}\times\mathcal{A}\times\mathcal{E}$ and $\longrightarrow \subseteq \mathcal{D}\times\mathcal{A}\times\mathcal{A}\times\mathcal{D}$, where $\mathcal{E}$ is the smallest set of expressions E, E_1, E_2 in Definition 4.1 and $\mathcal{A}$ is the union of the following sets: $\cup_{\ell\in\mathcal{L}}\texttt{moveTo}(\ell)$, $\cup_{\ell\in\mathcal{L}}\texttt{copyAt}(\ell)$, $\{\texttt{remove}\}$, and $\cup_{m\in\mathcal{M}}\texttt{callback}(m)$ and $\mathcal{A}$ is ranged over $\alpha, \ldots$.

Definition A.2 $\longrightarrow$ is a labelled transition relation over $\mathcal{E}\times\mathcal{M}\times\mathcal{E}$.

$$\frac{-}{\texttt{moveTo}(\ell) \xrightarrow{\texttt{moveTo}(\ell)} \mathbf{0}} \qquad \frac{-}{\texttt{copyAt}(\ell) \xrightarrow{\texttt{copyAt}(\ell)} \mathbf{0}}$$

$$\frac{-}{\texttt{remove} \xrightarrow{\texttt{remove}} \mathbf{0}} \qquad \frac{-}{\texttt{callback}(m) \xrightarrow{\texttt{callback}(m)} \mathbf{0}}$$

$$\frac{E_1 \xrightarrow{\alpha} E_1'}{E_1 ; E_2 \xrightarrow{\alpha} E_1' ; E_2} \qquad \frac{E_1 \xrightarrow{\alpha} E_1'}{E_1 + E_2 \xrightarrow{\alpha} E_1'}$$

$$\frac{C \textit{ is true}, \quad E \xrightarrow{\alpha} E'}{C \,\texttt{then}\, E \xrightarrow{\alpha} E'}$$

$$\frac{E_1 \equiv E_2 \quad E_1 \xrightarrow{\alpha} E_1' \quad E_1' \equiv E_2'}{E_2 \xrightarrow{\alpha} E_2'}$$ □

Definition A.3 $\longrightarrow$ is a labelled transition relation over $\mathcal{D}\times\mathcal{M}\times\mathcal{L}\times\mathcal{D}$.

$$\frac{D_1 \xrightarrow{\alpha}_\ell D_1'}{D_1 \parallel D_2 \xrightarrow{\alpha}_\ell D_1' \parallel D_2} \qquad \frac{E \xrightarrow{\texttt{moveTo}(\ell')} E'}{\ell[E \mid P] \xrightarrow{\texttt{moveTo}(\ell')}_\ell \ell'[E' \mid P]}$$

$$\frac{E \xrightarrow{\texttt{copyAt}(\ell')} E'}{\ell[E \mid P] \xrightarrow{\texttt{copyAt}(\ell')}_\ell \ell[E' \mid P] \parallel \ell'[E' \mid P]}$$

$$\frac{E \xrightarrow{\texttt{remove}} E'}{\ell[E \mid P] \xrightarrow{\texttt{remove}}_\ell \ell[E' \mid \epsilon]} \qquad \frac{E \xrightarrow{\texttt{callback}(m)} E'}{\ell[E \mid P] \xrightarrow{\texttt{callback}(m)}_\ell \ell[E' \mid P]}$$

$$\frac{D_1 \equiv D_2 \quad D_1 \xrightarrow{\alpha}_\ell D_1' \quad D_1' \equiv D_2'}{D_2 \xrightarrow{\alpha}_\ell D_2'}$$ □

We illustrate transitions of some policies described in Example 4.4.

Example A.4

- *Attraction:* of Example 4.4 has the following transition if a computer, named *@another*, has the same or compatible component of A.

$$\ell[\texttt{exist}(A, @another)\,\texttt{then}\,\texttt{moveTo}(@another)\,;E. \mid A] \xrightarrow{\texttt{moveTo}(@another)}_\ell @another[E \mid A]$$

- *Spreading:* of Example 4.4 has the following transition if a computer, named *@another*, does not have the same or compatible component of A.

$$@current[\neg\texttt{exist}(A, @another)\,\texttt{then}\,\texttt{copyAt}(@another)\,;E \mid A] \xrightarrow{\texttt{copyAt}(@another)}_{current} @current[E \mid A] \parallel @another[E \mid A]$$

Jingtao Sun was born in 1987. He is a Ph.D. candidate in the Graduate University for Advanced Studies and has been associated with the Information Processing Society of Japan since 2011. His research interest is adaptation and distributed systems. He is a student member of IPSJ, CCF, IEEE and ACM.

Ichiro Satoh received his B.E., M.E, and Ph.D. degrees in Computer Science from Keio University, Japan in 1996. From 1996 to 1997. Since 2001, he became an associate professor in the National Institute of Informatics (NII), Japan. Since 2006, he has been a professor of NII. Also, he was a visiting researcher of Rank Xerox Laboratory from 1994 to 1995 and a PRESTO (SAKIGAKE) researcher of Japan Science and Technology Corporation from 1999 to 2002. His current research interests include distributed and ubiquitous computing. He received the IPSJ Paper Award, IPSJ Yamashita SIG Research Award. He is a member of six learned societies, including ACM and IEEE.