SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2005; **35**:1349–1391
Published online 27 June 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.674

Automatic adaptation of mobile applications to different user devices using modular mobile agents

Tzu-Han Kao*,† and Shyan-Ming Yuan

Department of Computer and Information Science, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu 300, Taiwan

SUMMARY

Wearable, handheld, and embedded or standalone intelligent devices are becoming quite common and can support a diverse range of applications. In order to simplify development of applications which can adapt to a variety of mobile devices, we propose an adaptation framework which includes three techniques: follow-me, context-aware adaptation, and remote control scheme. For the first, we construct a personal agent capable of carrying its owner's applications. Second, we design a personal agent capable of carrying applications with an adaptable hierarchical structure. Then, applications can be adapted approximately to the context of devices by using an attribute-based component decision algorithm. Finally, to achieve a remote control scheme, we distribute the computational load of applications on the resource-restricted mobile devices. An application is divided into two parts that can be executed on a user device and a server separately. In short, this framework facilitates the development of widespread applications for ubiquitous computing environments. Furthermore, it enables the applications to follow their owners and automatically adapt to different devices. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: adaptation; mobile applications; mobile agents; context-aware; mobile execution environment; ubiquitous computing environment

1. INTRODUCTION

1.1. Motivation

With the progress of mobile technology, embedded systems and information appliances have been developed, and various kinds of handsets, networked facilities, and personal mobile devices enrich our lives. These technologies have been applied to many fields. For example, there are networked

*Correspondence to: Tzu-Han Kao, Department of Computer and Information Science, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu 300, Taiwan.
†E-mail: gis89539@cis.nctu.edu.tw

Contract/grant sponsor: National Science Council; contract/grant number: NSC93-2752-E-009-006-PAE

Received 29 May 2003
Revised 10 December 2004
Accepted 10 December 2004

TVs and home entertainment facilities in home appliances; Internet-capable PDAs, mobile phones, wearable computers in personal mobile applications; and embedded servers in business applications. Accordingly, context-aware applications, which adapt their behaviors to a changing environment [1,2] according to the context, such as indoor position, time of the day, nearby equipment, and user activities [3], can be developed. Context-aware mobile tourist guides [4] and location-aware shopping assistants [5] are two examples.

We can now foresee an ubiquitous computing environment [6] where a user can retrieve his personal information through any nearby computing facility, such as mobile and embedded computing devices, desktop computers, etc. In such an environment, information presented on the devices can be adjusted according to the context of these devices. One of the applications, called ImageGathering, where a multimedia campus guidance system is built on a campus, can be taken as an example. Wherever they are on campus, students can always ask this system for the location of a building by using a Java phone, PDA, or a laptop. Depending on the context of the student's device, a formatted image suitable for the student's device can be delivered to the student. When a visitor would like to enter some building on the campus, he can use his Java phone for more information on that building, and then a PNG image of 64×54 pixels will be sent to him. A notebook user can get a JPEG image of 340×256 pixels.

The delivery of the required image, depending on the context of device capabilities and user preferences, dominates the functions of this ImageGathering system. In addition, whatever device is used, users' applications will still continue. A user, for example, can use a desktop computer to check his daily report. When he moves from room to room, information about his report can still be acquired by a handheld PDA. In brief, we aim at providing a context-aware adaptive framework that can not only adapt functions of applications which personally rely on the context of the devices used, but also keep the executing states of applications even by using different devices.

1.2. Objectives and methodologies

Several problems obstruct the development of applications on small and handheld devices.

- The resources of the small and mobile devices are restricted, in terms of three aspects: memory, power, and networking. First, the size of the needed memory can be a problem. For example, a Java KVM already requires between 160 and 512 kB. Second, the needed battery power over time has to be available in the device. Finally, the communication need has to cope with the limited bandwidth in wireless (and sometimes low-power) networks.
- The computational capability of small and mobile devices is limited. Unlike stationary personal computers at 2 GHz or more CPU speed, those of the small and mobile devices are lower, like a Microsoft smart phone at 200 MHz CPU speed.
- The characteristics of the devices, such as screen size, color number, and resolution, are diverse. For instance, the resolution of Nokia 6600 is 176×208 (pixels), while that of Nokia 7210 is 128×128.

Therefore, we concentrate on two objectives to solve these problems. The first is to distribute computational load of programs running on small and handheld devices. To expand the computational capabilities of programs, some components of programs are designed for servers. The limited resources of these devices, therefore, will not restrict the capabilities of the programs. The second objective is to accomplish the adaptation of the functions of applications. To reach this goal, we structure an

application composed of its functions, each of which can be implemented by one or more candidate components[‡]. Furthermore, for each function, a proper component among the candidate components capable of implementing the function can be appropriately chosen to substantiate this function (said corresponding function) in accordance with context profiles of user devices.

The methods proposed in this paper contain three primary mechanisms: *remote control scheme*, *follow-me*, and *context-aware adaptation*. The remote control scheme divides the program of an application into two parts: *front-end* and *back-end* modules. The front-end module runs on the client device. The back-end module can be executed on a server and communicate with the front-end module through *remote dynamic invocation*, an invocation mechanism between the two modules (detailed in Section 6). As a result, certain components of an application with heavy computing load can be enveloped into the back-end module and run at the server side. For follow-me, a *personal agent* can not only be anchored at a certain server to serve its owner, but can also carry back-end modules of applications as migrating among servers with its owner. This provides the flexibility for application developments.

To adapt functions of applications to the context of user devices, we use Composite Capabilities/Preference Profiles (CC/PP) [7–11] and Wap User Agent Profile (WAP UAProf) [11,12] context modeling frameworks. CC/PP as defined by the W3C CC/PP working group, can describe device capabilities and user preferences in RDF/XML format. Generally, it is used for a device's delivery context and for the adaptation of content presented to devices. WAP UAProf was developed by the WAP research groups for the same purpose. Furthermore, structuring applications helps to adapt them, so an application is organized into front-end and back-end modules that contain numerous components, each of which can be declared to implement a function comprising the application. Additionally, we design *Application Structure and Component Constraints* (ASCC), an XML-based description that describes the requirements of components of an application by using constraint sets. Integrating the use of the ASCC and CC/PP profiles can enable the proposed system to be aware of structures and constraints of applications for adapting applications further.

This paper is organized as follows. After an overall introduction to the context-aware adaptation framework in Section 2, we describe the essential aspects of personal agents and applications in Section 3, and the mechanisms for agent migration and application reconstruction in Section 4. Sections 5 and 6 present an attribute-based component decision algorithm, performance evaluation, application interfaces, and remote dynamic invocation. Finally, we discuss related work in Section 7 and present conclusions in Section 8.

2. SYSTEM OVERVIEW

2.1. Example scenario

The last section has mentioned our main objectives and focuses. In this section, we explain the architecture and components of our framework. Figure 1 illustrates a scenario for an application which

[‡]A component is an object in object-oriented programming.

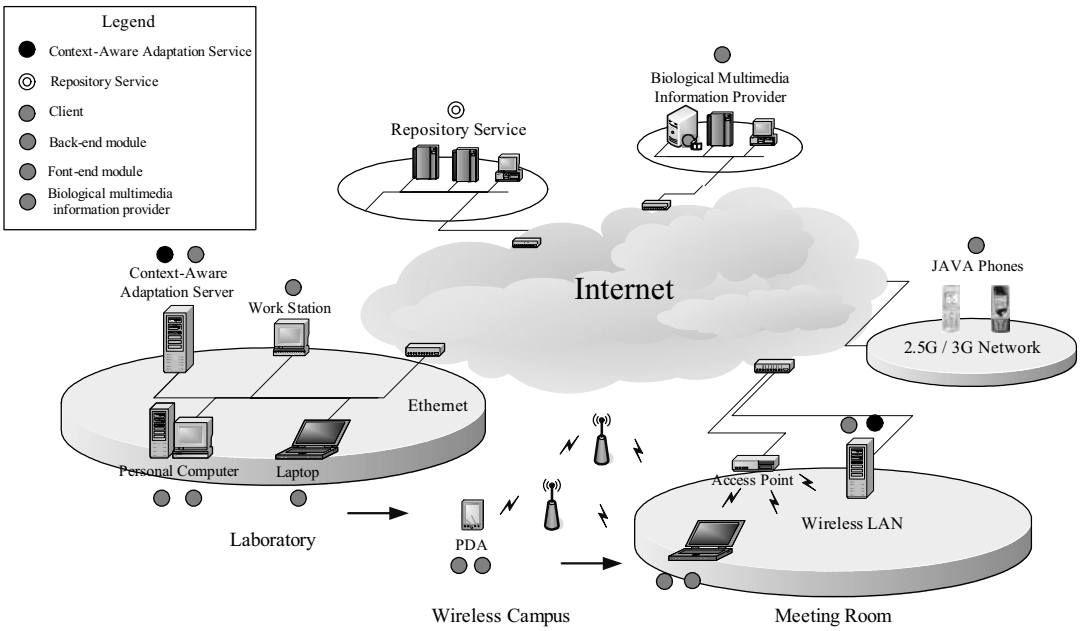

Figure 1. An overview of the system infrastructure.

can be developed with our framework. Assume that on the campus there is a wired Ethernet and IEEE 802.11. Students can then surf the Internet via the networks. Let us assume that a student wants to gather certain butterfly images. When in his laboratory, he can collect the information via his personal computer. Then, when he goes out for a meeting (the arrow indicates the direction of his movement), he can use a PDA to continue collecting the images. After arriving at the meeting room, he can go on working if there is another laptop available there. In so doing, his work will continue whether or not he carries his device. This scenario includes two critical techniques. First, the image can be suitably resized according to the context of the device. Second, the collection status is continuously executed without interruption while moving or after changing devices. In order to approach this, we attempt to design our framework to provide the following functionalities.

- It can divide a program of the application into two modules: one is the back-end module running at the server side for retrieving the images and transmitting the images to users' devices. The other is the front-end modules executed on users' devices for representing the gathered images, as shown in Figure 1.
- Numerous computing transformation and adaptation algorithms, which need heavy computational resources, will be enveloped in the back module to be executed at the server side instead of running the whole program on the devices. Thus, it will weaken the restriction of application development by this limitation of resources on devices.

- The system can change the transformation and adaptive component of the application to others, according to what devices users use.
- The back-end module can move with the user.

2.2. System infrastructure overview

Figure 1 exhibits the infrastructure of our system, in which there are three main components: *context-aware adaptation*, *repository service*, and *client*. The client devices can be mobile or stationary and may include computing devices such as PCs, PDAs, laptops, smart phones, Java Phones, etc. A front-end module for each user can be executed on its owner's device. The access networks of the devices include wireless IEEE 802.11x networks, and 2.5/3G telecommunication networks. On the server side, there is a context-aware adaptation server (CAAS server) in each local area network (LAN). In this design, we do not assert that each LAN must have a CAAS server. But, if we assume there is a server in a LAN, back-end modules of applications can be carried by agents, and moved to the server in the LAN where their owners move.

2.3. System architecture

Figure 1 demonstrates the infrastructure of our system. Internally, there are three primary constituents, client, context-aware service, and repository service, which correspond to *client tier*, *context-aware service tier*, and *repository service tier*, respectively, as presented in Figure 2.

2.3.1. Client tier

This tier contains various mobile, handheld, or stationary computing appliances. As in Figure 2, the devices cover J2ME-capable phones, PJava-capable PDAs, and J2SE-runable laptops or personal computers. PJava, J2ME [13,14], and J2SE are the Java Virtual Machines for different operating systems and hardware environments. Each of these has some particular configuration profiles. In J2ME, the profiles of KVM, J2ME Configuration, and MIDP are involved. In addition to having the functions of J2ME, PJava also includes Configuration, Foundation Profile, Personal RMI, CDC Profiles, whereas the J2SE environment comprises JVM and optional Java API.

The front-end module on the device can send a registering message with its context profile to inform CAAS servers of its capabilities. After that, while the remote invocation interface of the front-end module is invoked, the user agent of the front-end module will serialize the invocation into messages and send the messages to the back-end module. Then, the corresponding method of the back-end module will be invoked. We call this scheme remote dynamic invocation (explained in Section 6) in our system. To implement this scheme, we design two components. One is *User Agent*, which provides an interface. Programmers can use the interface to invoke the methods of back-end modules. The other is *CC/PP Negotiator*, which can send CC/PP profiles [8,10,11] to inform CAAS servers of devices' capabilities when the user's device initially connects to this server. Figure 3 shows the RDF/XML [15–17] serialization of a context profile, which is wrapped in transmitted messages. It illustrates 16 (bits per pixel) in the hardware component of a user device.

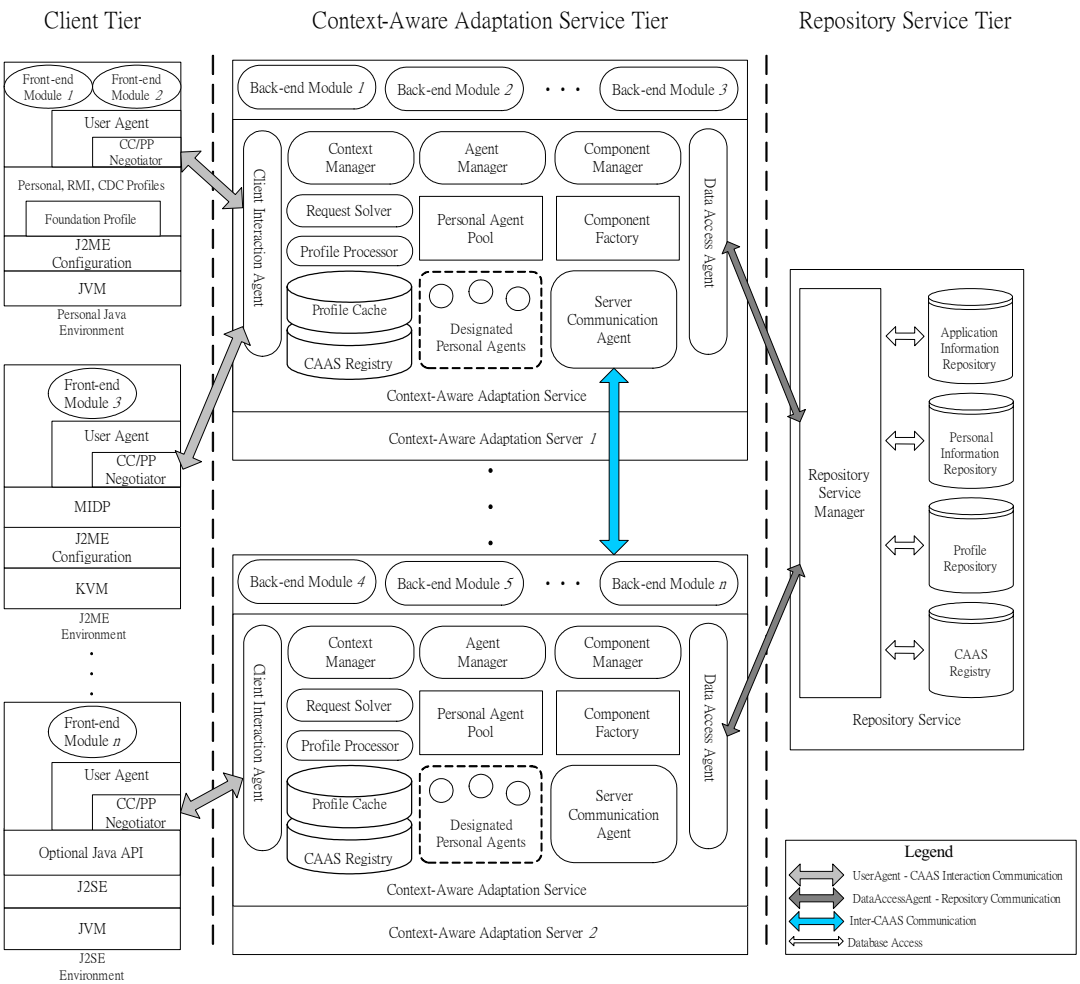

Figure 2. The inner architecture of the context-aware adaptation framework.

2.3.2. *Context-aware adaptation service tier*

This tier includes at least one CAAS server that can support the migration of agents, execution of back-end modules, and application adaptation. The servers connect with each other via Remote Method Invocation (Java RMI) [18] and IP multicasting. The RMI connection is capable of serializing objects and transmitting continuously the serialized results over networks to support agent migration. Through the connection, personal agents can carry their owners' back-end modules and migrate


```
GET /ccpp/html/ HTTP/1.1
...
<?xml version="1.0"?>
    <rdf:RDF ...>
    <rdf:Description rdf:ID="MyDeviceProfile">
      <prf:component>
        <rdf:Description rdf:ID="HardwarePlatform">
          <rdf:type rdf:resource="http://www.wapforum.org/profiles/UAPROF/ccppschema-
            20010426#HardwarePlatform"/>
          <prf:BitsPerPixel>16</prf:BitsPerPixel>
        </rdf:Description>
      </prf:component>
    </rdf:Description>
</rdf:RDF>
```

Figure 3. A CC/PP profile.

between CAAS servers. In this system, there are two modes (synchronous/asynchronous), which can be set here. The synchronous mode allows the personal agent to migrate with its owner, while the asynchronous mode requires the agent to stay at its resident server.

We take advantage of IP multicast to transmit the notification of application deployment and RMI connection setup. Each server listens to the same IP multicast address. If messages are transmitted to the address, the servers which listen to this address can receive the messages. For example, if an application is deployed into the repository service, the notification of the deployment will be sent through the channel. The servers, listening to the multicast IP address, can receive the notification. Data needed to be sent in the application deployment are transmitted via the RMI connections between CAAS servers and the repository server. In implementation, the multicast connection is constructed in the beginning of this system. Namely, CAAS servers and the repository server listen to a multicast address before constructing RMI connections. A server thus can build an RMI channel connecting with another by broadcasting a joining message to the multicast address. The servers which receive the broadcast messages will reply to those servers sending the joining messages.

CAAS servers are capable of adapting applications[§] carried by the agents migrating from other servers. An image transformation function, for instance, can be implemented by two candidate components: BMP-to-PNG, and BMP-to-JPEG. The former will be applied when the requesting client device is a J2ME-capable phone. While the user uses a desktop computer instead, the latter component can be chosen to implement this function.

A CAAS server principally includes four constituents: *client interaction agent*, *context manager*, *agent manager*, and *component manager*. The *client interaction agent* serves as an interactor, which communicates with CC/PP negotiators on user devices. It can transmit the messages used in the CC/PP negotiation protocol and remote dynamic invocation.

[§] A process is used to decide on proper components for the carried application.

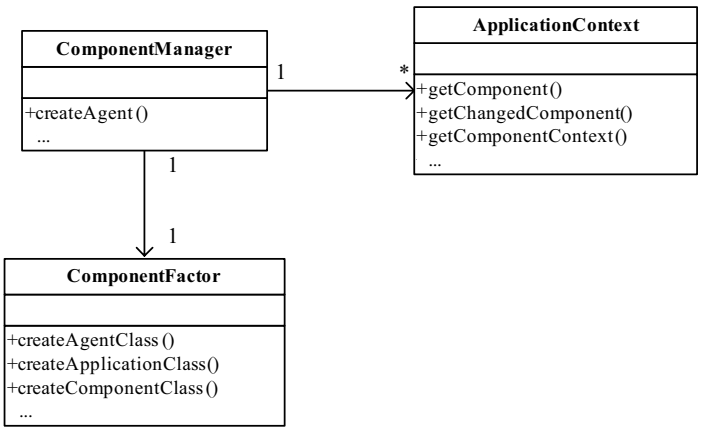

Figure 4. Classes of component manager, component factory, and application context.

For communicating with user agents on client devices and adapting applications, the *context manager* can parse the CC/PP profiles embedded in the registration messages in the initial negotiation process of the CC/PP protocol. To handle these profiles, we apply DELI service [19] and Jena API [20] to a context manager. One is *request solver*, which is capable of unpacking HTTP1.1 request messages to retrieve the CC/PP profiles. The other is *profile processor*, which can parse the CC/PP profile. In addition, all of the parsed profiles will be collected into *profile cache*. Thus, devices can transmit the changed part to the service instead.

The *agent manager* is capable of constructing and maintaining personal agents. Additionally, it controls an agent's lifecycle, and invokes the corresponding method related to the state change of the lifecycle. In order to save the cost of constructing agents, and to immediately respond to user devices when connecting to the back-end module, the *agent pool* is built. By using this pool, agents can be constructed beforehand and recycled after completing their tasks.

The *component manager* supplies the agent manager with the classes needed for constructing an application adaptation[¶]. Figure 4 presents the internal classes for creating the object instances to construct agents, applications, and components by `createAgent()`, `createApplication()`, and `createComponent()`, respectively (explained in Section 3.3). Also, an `ApplicationContext` class, which contains an *application structure table*, a *component decision tree*, and a *changed component table*, etc., can support the application adaptation and the component construction.

Server communication agents can be used to construct RMI and IP multicast links between CAAS servers. The *data access agent* has the function of requesting and receiving data from

[¶]The reasons for constructing classes and the procedure will be explained in Section 4.4.

the repository service. Another connection is built by using `URLClassLoader`[‖] between a CAAS server and the repository service.

2.3.3. Repository service tier

This tier acts as the database and directory service in our system. Three categories of information are stored, including application information, personal information and context profiles, as shown in Figure 2. This tier also stores the classes and ASCC files of the applications deployed. Note that an *application structure and component constraints* (ASCC) profile, designed in this framework, is an XML-based profile. It describes application structures and component information. When a programmer has finished developing an application, he or she can pack the code of the application into a Java Jar file (a compressed file containing the class files), and then store this packed file and the ASCC description of the application into the repository service.

In order to deliver the information concerning a deployed application to CAAS servers, two mechanisms are designed. One is *application preloading*, in which the repository service notifies the CAAS servers once an application is deployed into the repository. The other is *application remedy*, which can be applied if the required classes of the applications carried by the agent are not found. It is implemented when a CAAS server accepts a migrated agent, as discussed in Section 4.4.

3. PERSONAL AGENT

3.1. The state transfer of the agent

A personal agent, which is an active object with a state, is assigned to serve a user. The term 'active' means that the agent has a thread to perform a certain method invocation requested by the front-end module. An agent will invoke the requested method of the back-end side when it receives a request. Consequently, the result is sent back to the front-end modules. The state transition of an agent is presented in Figure 5.

In the `Ready` state of an agent, the agent is activated to be ready for receiving invocation requests from the front-end module. When receiving a request, it invokes the corresponding method of the appointed application. Then, the state will transit to the `Execution` state. When invoking is completed, the agent will send the result of the execution to the front-end module, and its state will change back to the `Ready` state. Moreover, the carried applications can be adapted by the agent manager only in this state. If this occurs, it will change to the `Stop` state and to the `Adaptation` state soon afterwards. In the `Stop` state, the agent is deactivated and does not receive any requesting invocation. Thereafter, components of each function of the application can be switched appropriately. The situations that cause the state to transit to the `Stop` state are (i) a logout message received from the user agent; (ii) no messages received from the user agent for a period of time; (iii) the agent is instructed to migrate to another server. Moreover, conditions that make the state transit back to the

[‖] `URLClassLoader` is a Java class capable of loading classes from remote computers. It will be used if the required classes do not exist in CAAS servers.

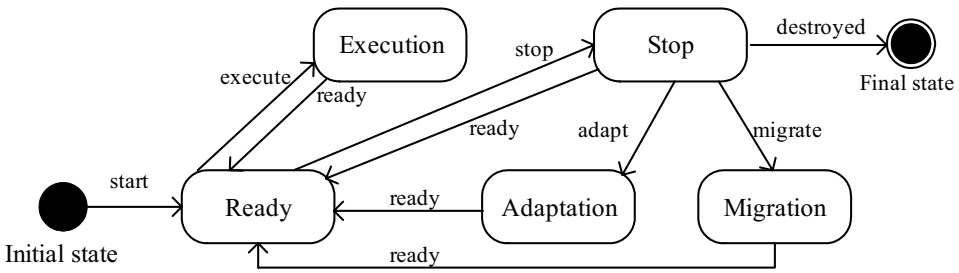

Figure 5. State transition diagram of the personal agent.

`Ready` state are (i) an agent manager has got the agent from the agent pool and then assigned it to its user for recycling; (ii) application adaptation has been performed; (iii) agent migration has been completed.

3.2. The structure of an agent

In this section, we discuss the inner structure of an agent. Figure 6 indicates an agent, which is composed of a state and a body. The state records the information related to the agent's user and the applications carried by the agent.

3.2.1. Agent state

An agent state (Figure 6) is composed of *agent ID* (the identifications of the agent), *User ID* (its owner), *Device ID* (the owner's device), and *Application IDs* (the applications carried). In addition, the agent state records the states of applications. Each of the states includes *absent component IDs* and *an event queue*. Absent component IDs are the identifications of the component objects withdrawn from the agent body. An event queue is responsible for queuing the requested events in the execution state. The queue is used to keep events, so it stops the execution of processes from being interrupted by incoming events. Specifically, an event queue retains the events of the state transition and notifications of the invocations from the front-end module. State transition and invocations will be scheduled in the order of FIFO, so if a new event arrives, it will be put at the rear of the queue. Then, to process these events, the main thread of this agent obtains an event from the front of this queue. Taking the scenario in Section 1.1 for example, the user agent, in the front-end module of a user device, requests the user's personal agent for a picture. When the personal agent receives this invocation, events concerning the invocation will be generated and put in the queue. In this example, the events corresponding to 2, 3, 4, 5, 6, 12 (in Figure 7) are put into the event queue. Next, if any request arrives or the state transition is triggered, the notification relevant to these events will follow the previous requested notifications.

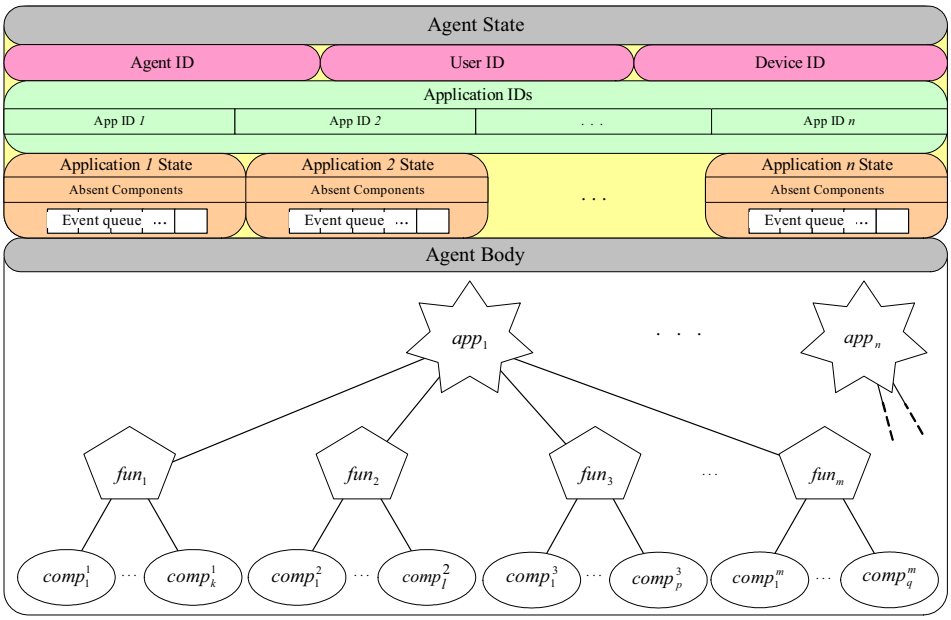

Figure 6. An internal view of a personal agent and an application structure.

3.2.2. *Agent body*

Figure 7 shows the detail of the implementation of the example ImageGathering. In this example, there are two basic systems: one is the application developed through the API of our system; the other is a biological multimedia information provider on the Internet. This provider has three types (image, video, and text) of data provided, shown on the right-hand side of the figure.

We divide the application into two parts: the front-end module and the back-end module. The front-end module contains two main constituents: an image display that can show images, and a requester that can send requests and receive the replied images. This module executes on the client device. The back-end module consists of four constituents: Image Transmitter, Cache, Image Retriever, Transcoder, and Data Access. In this module, Image Transmitter is responsible for receiving clients' requests and replying the required images. Image Retriever can obtain the requesting images from the biological multimedia information provider through Data Access. Next, it will pass the images to Transcoder for transforming images, such as image resizing, or to the cache for subsequent retrieving of images immediately. The following lists the possible procedures used to retrieve the required images that the student wants to collect:

- $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 12 \rightarrow 13$: this procedure means that when Image Transmitter receives the user's request, the required images are obtained from the information provider.

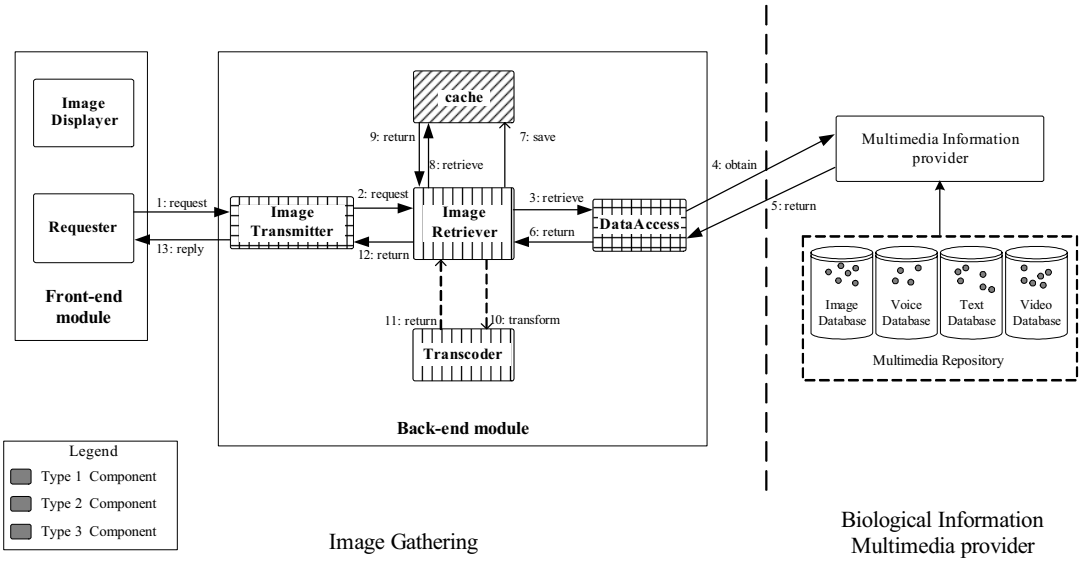

Figure 7. The back-end module of the application Image Gathering.

- $1 \rightarrow 2 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 12 \rightarrow 13$: this process refers to obtaining the required image from Cache.
- $1 \rightarrow 2 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 13$: this indicates that the images are passed to Transcoder for resizing after Image Receiver gets the images in the second bullet, and then the results are transmitted to the requester.

Figure 6 shows an application, which is structured hierarchically. This application is composed of more than one function that could be implemented by at least one component. The application structure exhibited in Figure 7 can also be expressed in the form of the two-level hierarchy demonstrated in Figure 8. As we can see in Figure 8, each function links to candidate components. The function Image Retriever, for example, links to three components: JPEG Retriever, GIF Retriever, or PNG Retriever. This indicates that the function Image Retriever can be implemented by the three components.

Components, in our system, are classified into three categories: Type 1, Type 2, and Type 3. Type 1 components have the characteristics inclusive of *stateful*, *relative*, and *immoveable*. The stateful property means that the component records some particular data. For instance, Image Cache for the cache function belongs to this category. In contrast, specifying the stateful property `No` means that the components do not keep track of any particular data. If we declare a component as relative, it is associated with certain resources, and these components have database or TCP/IP connections. For example, the components of the function Data Access need to be declared as this type, since it connects to the database of the multimedia provider over the networks. Another moveable property is used to modify the components that fail to be carried in agent migration.

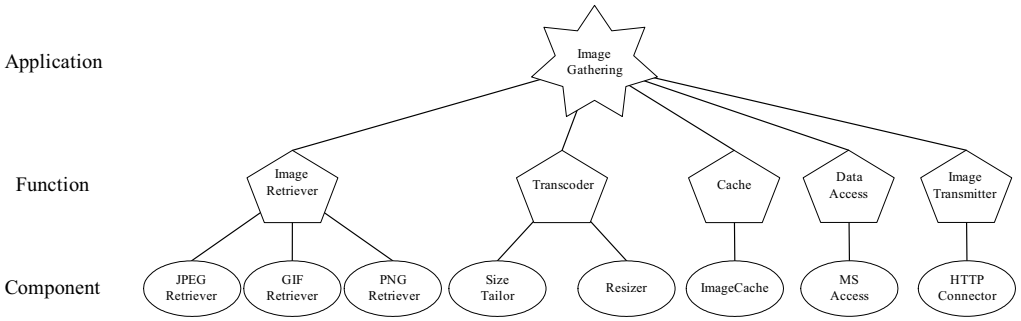

Figure 8. The structure of the application ImageGathering.

Table I. Three categories of components.

Type	Stateful/stateless	Relative/irrelative	Movable/immoveable	Example
Type 1	stateful	Relative	immoveable	Database Access
Type 2	stateful	Irrelative	moveable	Cache
Type 3	stateless	Irrelative	moveable	Transcoder

Type 1 components are the components connecting to certain resources. As demonstrated in Figure 7, candidate components of the function Data Access belong to this type. Type 2 components are those components which can be moved. The component Image Cache (in Figure 9) is declared as this type. Type 3 components are usually certain algorithms or pure computational logics, such as the XML transformation engine `javax.xml.transform.Transformer`. Table I displays the three types of components.

3.3. Application structure and component constraints

In order to enable this framework to be aware of the structures of applications, we define ASCC, an application profile description. Figure 9 illustrates the ASCC profile of the application ImageGathering.

As we can see in Figure 9, the `<application>` element includes five `<function>` elements, which can describe the five functions. In each `<function>`, the candidate components can be specified. Lines 4–28, for instance, declare that `<component id="JPEGRetriever" ...>`, `<component id="GIFRetriever" ...>`, and `<component id="PNGRetriever"...>` can implement the Image Retriever function. In addition, within a `<component>` element, the properties, `stateful`, `relative`, and `carried`, can be used to set components stateful/stateless, relative/irrelative, and carried/un-carried respectively. The `priority` property concerns the priority


```
1 <?xml version="1.0"?>
2 <ascc xmlns:ascc=http://dcsw3.cis.nctu.edu.tw/project/CAAS ...>
3  <application id="ImageGathering">
4    <function id="ImageRetriever">
5      <default idref="JPEGRetriever"/>
6      <component id="JPEGRetriever" priority="51{\%}"
7       stateful="No" relative="No" carried="No">
8        <constraints>
9         <prf:ImageCapable>Yes</prf:ImageCapable>
10        <prf:CcppAccept>image/jpeg</prf:CcppAccept>
11       </constraints>
12     </component>
13     <component id="GIFRetriever" priority="50{\%}"
14      stateful="No" relative="No" carried="No">
15       <constraints>
16         <prf:ImageCapable>Yes</prf:ImageCapable>
17         <prf:CcppAccept>image/gif</prf:CcppAccept>
18       </constraints>
19     </component>
20     <component id="PNGRetriever" priority="50{\%}"
21      stateless="No" relative="No" carried="No">
22       <constraints>
23        <prf:ImageCapable>Yes</prf:ImageCapable>
24        <prf:CcppAccept>image/png</prf:CcppAccept>
25        <prf:JavaPlatform>MIDP/1.0-compatible</prf:JavaPlatform>
26       </constraints>
27     </component>
28   </function>
29   <function id="Transcoder">
30     <default idref="SizeTailor"/>
31     <component id="SizeTailor" priority="50{\%}"
32      stateful="No" relative="No" carried="No">
33     <component id="ColorTransformer" priority="50{\%}"
34      stateful="No" relative="No" carried="No">
35   </function>
36   <function id="Cache">
37     <component id="ImageCache"
38      stateful="Yes" relative="No" carried="Yes">
39   </function>
40   <function id="DataAccess">
41     <component id="MSAccess" stateful="Yes" relative="Yes">
42   </function>
43   <function id="ImageTransmitter">
44     <component id="HTTPConnector"
46      stateful="Yes" relative="Yes"/>
47   </function>
48 </application>
49</ascc>
```

Figure 9. The ASCC profile to describe the structure of ImageGathering.

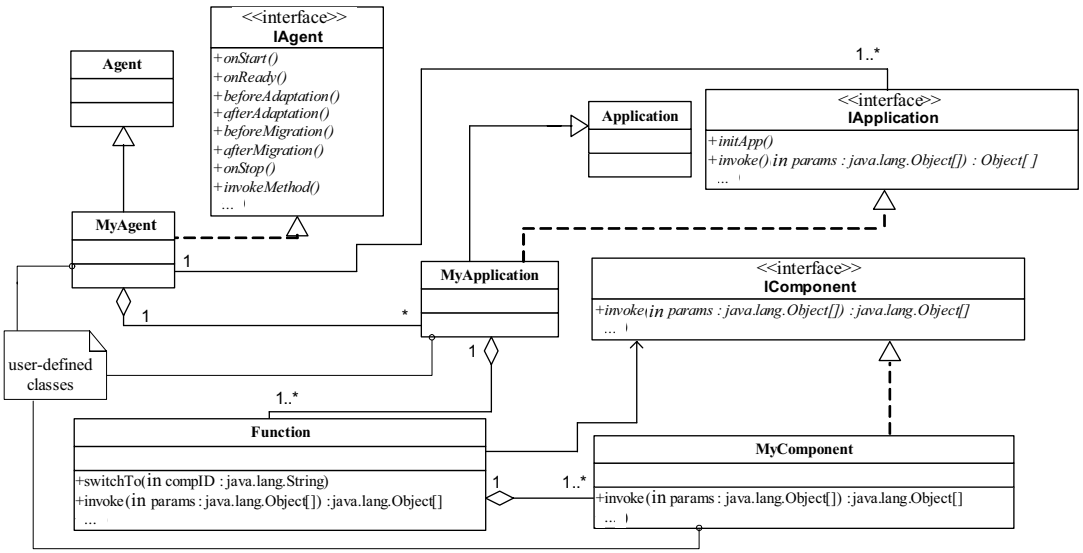

Figure 10. The class diagram of programming agents and back-end modules.

of a component, one of which is chosen in each application adaptation. Furthermore, to set a component as a default component for a function, we can use the element `<default>`. If we want to set a component implementing the function which cannot be replaced with others, we can use the property `"unchanging='Yes'"`. Figure 10 exhibits the class diagram of the implementation of the back-end module, which is made up of the classes derived from three original classes. A programmer defines a personal agent class, which is derived from the `Agent` class, and lets the agent carry the applications whose classes derived from the `Application` class. Furthermore, the programmer can define various subclasses of the class `Component` to substantiate and diversify his application. Without loss of generality, we use `MyAgent`, `MyApplication`, and `MyComponent` as the user-defined classes, which are illustrated in Figure 10.

4. AGENT MIGRATION

In the application ImageGathering, even when moving from room to room, users can continue collecting the information. In order to complete this, we need to overcome the following problems: 'How does the system perceive the situations of users' movement?' and 'How does the system instruct an agent that serves the user to migrate with the user under perceiving the situations of users' movements?'.

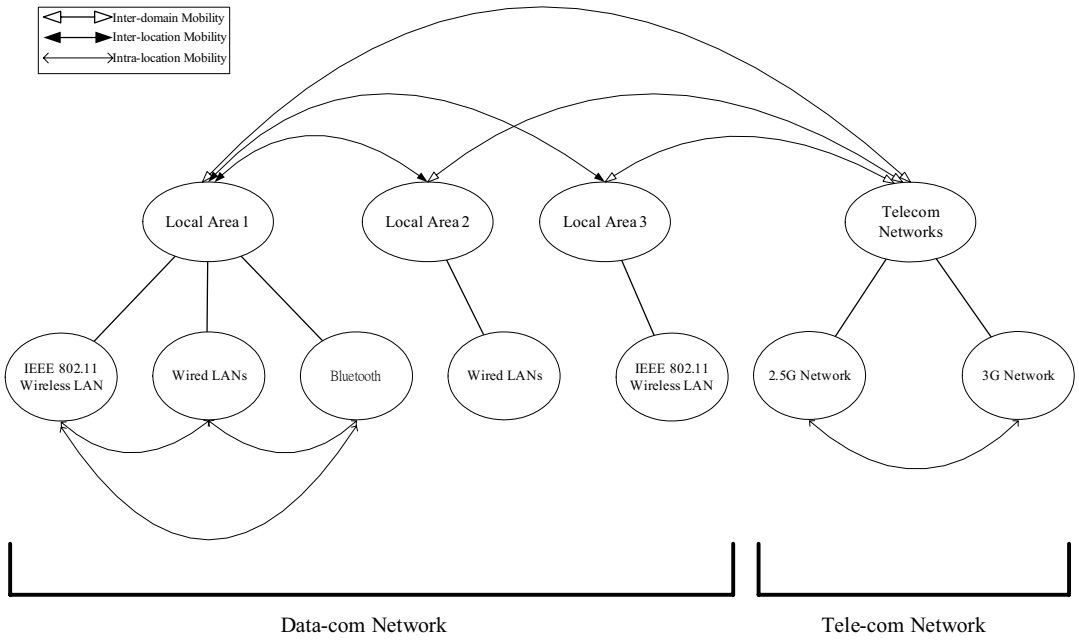

Figure 11. Cases of users' movements.

4.1. Types of mobility

We can roughly partition off networks into data-com network and tele-com network. Data network includes IEEE 802.11x [21,22], wired LANs, and bluetooth networks [22]. Tele-com network consists of 2.5/3G [23] networks (Figure 11). According to the characteristics of these networks, we define personal mobility and terminal mobility. Personal mobility means that by using any nearby computing equipment, a user does not need to carry his device wherever he moves. In other words, a user can use a device to perform his work, and also continue working via another instead. Terminal mobility indicates that a user can perform his work via his carried device.

As shown in Figure 11, the data-com network contains a large number of LANs, and the three kinds of networks may be in the same region, as Local Area 1. Furthermore, there is one possible type of network in a LAN, such as Local Area 2. In a tele-com network, numerous wireless tele-com network areas, formed by the radio coverage of base stations, are regarded as the same network in our system.

Cases of users' movements from one region to another can be grouped into three categories: *intra-location mobility*, *inter-location mobility*, and *inter-domain mobility*. *Intra-location mobility* means that the coverage of a user's movement does not exceed the range of a LAN. For instance, when a student collects information through his personal computer in his lab, *inter-location mobility* indicates that a user's movement crosses two LANs. A case of this movement might be that a user uses a certain device in Local Area 1, and then uses another after moving to Local Area 2. *Inter-domain mobility*

refers to the fact that a user's movement crosses data-com networks and tele-com networks. A student, for example, collects images by using his personal computer in his lab. Next, in place of the personal computer he uses a Java Phone when moving from his lab to a meeting room.

4.2. Agent registration

The system provides two mechanisms to perceive users' movements. We call the first mechanism *Passive-Client and Active-Server* (*PCAS*). In this mechanism, the user agent of a user's device will be notified to initiate a registration procedure when its user moves to the server's covered region. We call the second mechanism *Active-Client and Passive-Server* (*ACPS*). By using this mechanism, user agents of a user's device will actively inform the repository service if they need to connect to some CAAS server.

In PCAS, a server located in a region can detect movements of user devices entering into this region. When a user uses his device and enters this region, the server notifies the user agent on his device, thereupon the user agent will register with this server. Intrinsically, notification messages are the advertisements broadcast periodically on the wireless IEEE 802.11 network by a CAAS server. User agents of client devices continue listening to such messages. Provided that there is a user entering a new wireless LAN, then the user agent will send a requesting service message to the server sending the notification without registering to any server. Figure 12 illustrates the sequence diagram of this procedure. To inform the server of client information, we embed the CC/PP profile in the request service message. While a server receives the request message from the `ClientInterActionAgent` object, it will forward the messages to the `RequestResolver` object to resolve the CC/PP profile. The `RequestResolver` object is capable of retrieving the profile from the message and passing it to the `ProfileProcessor` object to resolve the profile. Then, the result will be passed to the `CAAS` service object, `c2`. When receiving the message, `c2` requests CAAS service `c1` for the user's personal agent. The user's personal agent, therefore, can be instructed to migrate to the server `c2` close to the user. In this mechanism, user agents on users' devices can automatically register to CAAS servers when their owners move among LANs.

The main difference between ACPS and PCAS is that in ACPS user agents on devices actively register to the repository service. Thus, ACPS can be applied to solve the condition where user agents on user devices have not connected to any server. A user agent on a user's device, for example, sends a request message to the repository service. Upon receiving a request message, the repository service redirects the connection to the nearby CAAS server closest to him. This procedure is decomposed into the steps shown in Figure 13.

It will be possible that agent migration is not needed if the covered regions of user movements are identical. Perhaps one of the conditions is that the device briefly disconnects, and then reconnects to the server. Under this condition, the user's personal agent still resides in this server. Thus, when the user agent asks the repository service about a CAAS server, the repository service will inform the user device of the original nearby CAAS server, and redirect the connection to that CAAS server. Though the user device reconnects to the CAAS server, the personal agent will not be instructed to migrate.

We explain below how the system perceives users' movements and when a CAAS server instructs a user's personal agent to migrate from another server. Further, the cases of users' movements can be considered altogether with PCAS and ACPS, as follows.

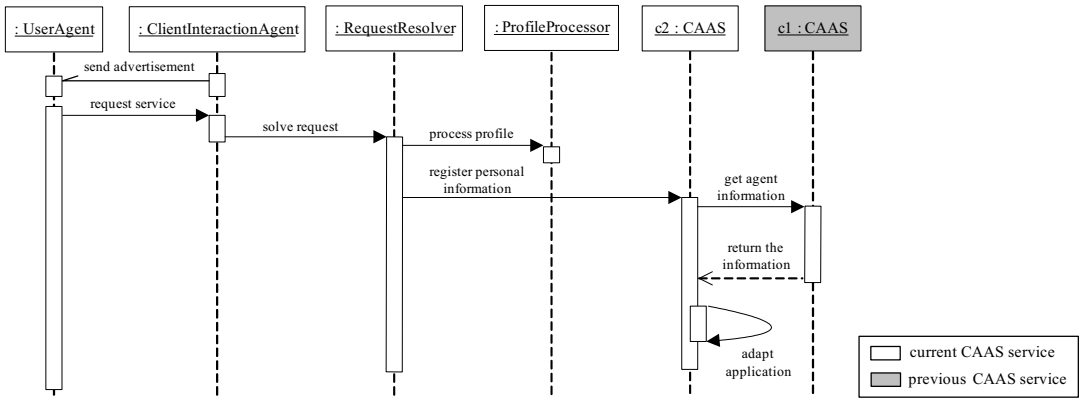

Figure 12. The sequence diagram of ACPS.

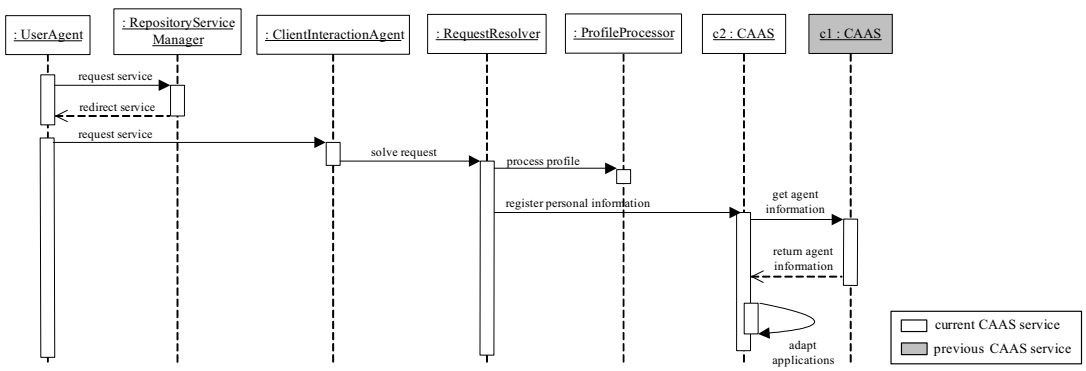

Figure 13. The sequence diagram of ACPS with agent migration.

- In inter-domain mobility, two situations are classified into this category. One condition is that the underlying network accessed is data-com network first and tele-com network subsequently. Under this condition, user agents on user devices can register automatically by using ACPS. The other condition is the opposite of the first condition. Here user agents on user devices can be notified to register through PCAS, as shown in Figure 14.
- In the cases of inter-location mobility where a user crosses two different kinds of LANs, user agents on user devices can be notified to register through PCAS.
- Under the conditions of intra-location mobility, it is unnecessary to move users' personal agents, because in this case users use their devices on the same network.

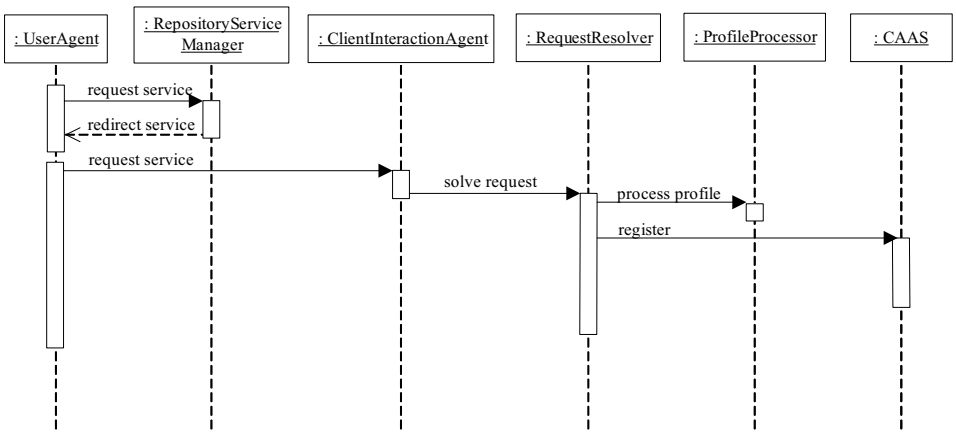

Figure 14. The sequence diagram of ACPS without agent migration.

4.3. Agent migration strategy

In agent migration, we consider the three strategies: Heavyweight Agent Migration (HAM), Flyweight Agent Migration (FAM), and Lightweight Agent Migration (LAM).

4.3.1. Heavyweight Agent Migration

In the HAM strategy, an agent will carry the components belonging to Type 2 and Type 3 while migrating, except for those specified as `"carried='No'"`** in the ASCC profile. Figure 15(a) illustrates an example. Agent migration is a procedure that serializes object instances comprising a whole agent into a byte array, and then sends the serialized binary data to the target server. Upon receiving it, the receiving server reconstructs the agent from the byte array.

Suppose that an application is carried with two functions: *fun 1* and *fun 2*. *fun 1* can be implemented by components *comp 1* and *comp 2*, and *fun 2* can be realized by *comp 3* and *comp 4*. Figure 16 exhibits the results of the HAM strategy applied to agent migration. The components *comp 1–4* are carried with agent migration. When the agent reaches CAAS server 2, the server appropriately adapts the applications carried by the agent. For function *fun 1*, the component *comp 1* is suitable for the context of the user device used previously. However, it is not suitable for that of the other used subsequently. As a result, the component *comp 2* is chosen to substitute *comp 1*.

Figure 17 shows the HAM algorithm. At the transmitter side, the immoveable components are detached and then their IDs are recorded into an absent component array in an agent state *S*.

**The term 'not carried' means nullifying the object references in the implementation code.

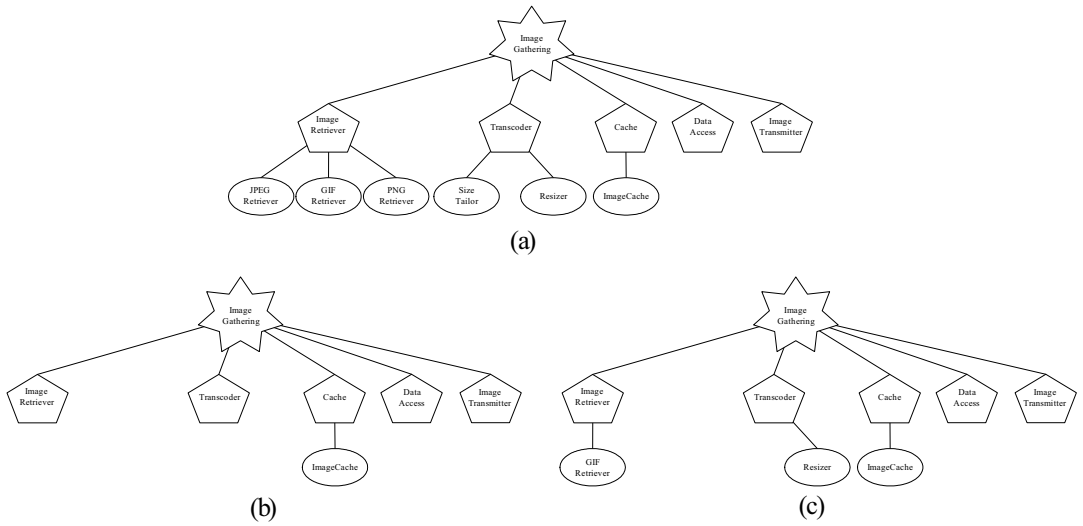

Figure 15. The structure of the application ImageGathering by using three strategies: (a) using HAM, (b) using FAM, and (c) using LAM.

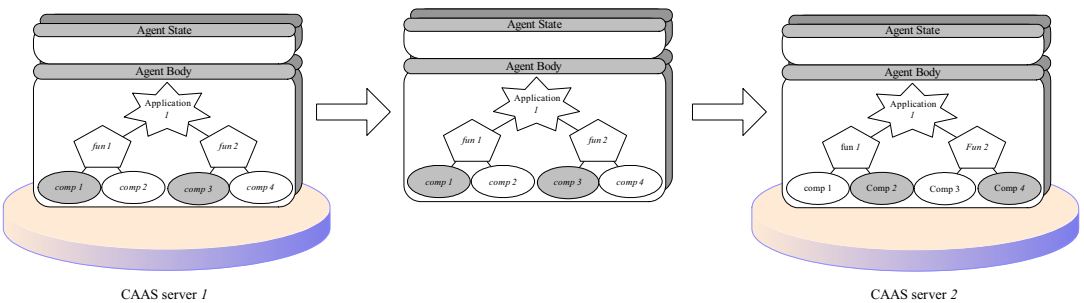

Figure 16. Agent migration using HAM.

While accepting the agent on the receiver side, the receiving server will retrieve the application IDs from the agent state and record them into array A (lines 1–2). If, for each function, a certain component implementing this function is unsuitable, another proper component will be chosen to substitute for that component by means of *DECIDE-PROPER-COMPONENT(A[i], F[j], T, Q, D)*. Eventually, line 16 switches each unsuitable attached component to a more appropriate one for each function $F[j]$ of an application $A[i]$ in the agent G.

G: an agent; S: the agent state of the agent G; A: an array recording IDs of the applications carried by the agent G; F: functions comprise an application $A[i]$; T: a previous context profile; Q: a current context profile; C: an array recording IDs of un-carried components of an application A; D: a decided components for functions$F$ of an application $A[i]$

```
HAM - TRANSMITTER(G)
1 GET - AGENT - STATE(G, S)
2 GET - APPLICATION - IDs(S, A)
3 for i ← 0 to length[A]
4 do GET - FUNCTION - IDs(S, A[i], F)
5     for j ← 0 to length[F]
6     do DETACH - IMMOVEABLE - COMPONENT(G, A[i], F[j], C)
8         for k ← 0 to length[C]
8         do ADD - TO - ABSENCE - COMPONENT(G, S, A[i], F[j], C[k])
9 return G


HAM - RECEIVER(G, T, Q)
10 GET - AGENT - STATE(G, S)
11 GET - APPLICATION - IDs(S, A)
12 for i ← 0 to length[A]
13 do GET - CHANGED - OR - ABSENCE - FUNCTION - IDs(S, A[i], F)
14     DECIDE - PROPER - COMPONENT(A[i], F, T, Q, D)
15     for j ← 0 to length[D]
16     do SWITCH - COMPONENT - TO(G, A[i], F[j], D[j])
17 return G
```

Figure 17. The algorithm of HAM.

4.3.2. *Flyweight Agent Migration*

The principle of this strategy is to minimize the data size needed to transfer an agent between servers. In other words, the components, except for those classified to Type 2 and Type 3 and indicated as `"carried='Yes'"`, are not carried with agent migration. Figure 18 illustrates this example since, except for the component Image Cache, no component is carried with agent migration. This is because Image Cache is Type 2 and declared `"carried='Yes'"`, but the other components are the cases in either Type 1 or Type 2/3 components declared `"carried='No'"` (see Figure 3). Figure 19 illustrates that the components *comp 1–4* are not carried since these components are classified to Type 2/3 but not specified `"carried='Yes'"`. Then, in CAAS server 2, for each function, proper components are decided upon. Additionally, their object instances are reconstructed if necessary.

The algorithm of FAM is shown in Figure 20, where components, except for the Type 2/3 components specified `"carried ='Yes'"`, are not carried in agent migration. At the acceptance of the agent, the server can examine the missing components (retrieved into array F). After deciding upon proper components for each function in line 14, the receiver creates object instances for those

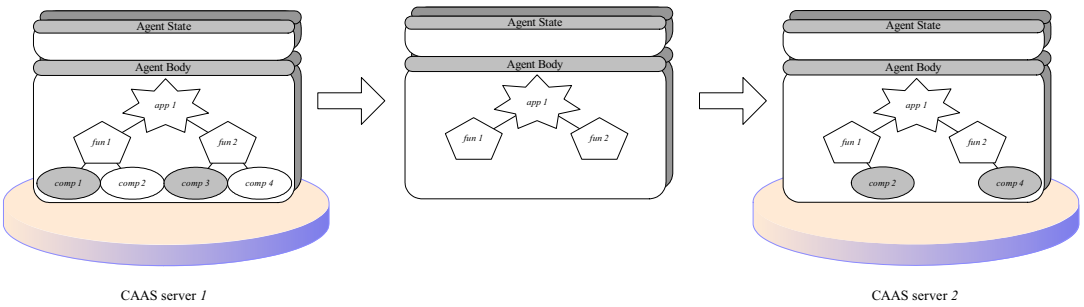

Figure 18. Agent migration using FAM.

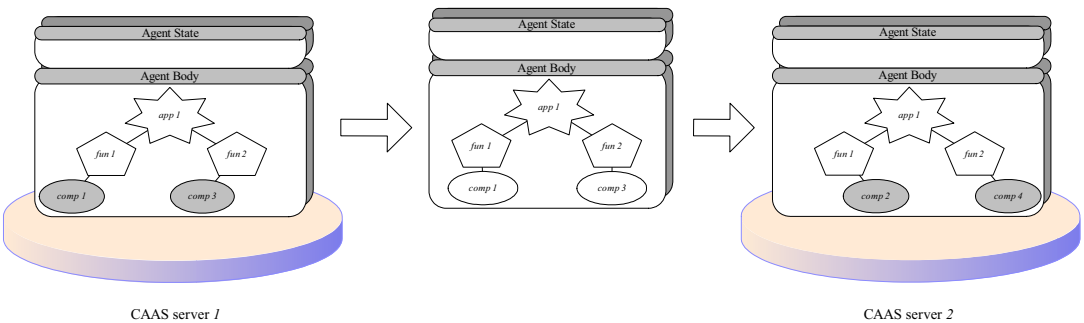

Figure 19. Agent migration using LAM.

in F, and plugs the suitable ones into their corresponding functions (in lines 15–16). However, there is likely to be a problem: the required classes (the user-defined subclasses of `Component`) are not found. If this problem occurs after an agent migrates, the needed classes can be loaded through the component manager. Details of this will be given in Section 4.4.

4.3.3. Lightweight Agent Migration

The substance of this strategy is that one component is carried for each function in an application, except for Type 1 components, when agents migrate. A possible method we propose is to carry the only components which implement its corresponding functions in agent migration. Figure 15(c) illustrates this strategy, where only the components implementing their corresponding functions are carried. Except for Type 1 components (Data Access and Image Transmitter), the components of the functions Image Retriever, Transcoder, and Cache are carried.


```
FAM - TRANSMITTER(G)
1 GET - AGENT - STATE(G, S)
2 GET - APPLICATION - IDs(S, A)
3 for i ← 0 to length[A]
4 do GET - FUNCTION - IDs(S, A[i], F)
5     for j ← 0 to length[F]
6     do DETACH - ALL - EXCEPT - CARRIED - COMPONENT(G, A[i], F[j], C)
7         for k ← 0 to length[C]
8         do ADD - TO - ABSENCE - COMPONENT(G, S, A[i], F[j], C[k])
9 return G

FAM - RECEIVER(G, T, Q)
10 GET - AGENT - STATE(G, S)
11 GET - APPLICATION - IDs(S, A)
12 for i ← 0 to length[A]
13 do GET - CHANGED - OR - ABSENCE - FUNCTION - IDs(S, A[i], F)
14      DECIDE - PROPER - COMPONENT(A[i], F, T, Q, D)
15      for j ← 0 to length[D]
16      do ATTACH - COMPONENT - TO(G, A[i], F[j], D[j])
17 return G
```

Figure 20. The algorithm of FAM.

Likewise, in Figure 19, before the agent migrates, the components *comp 2* and *comp 4* are detached from the functions *fun 1* and *fun 2*, respectively. While the agent arrives in CAAS server 2, *comp 2* and *comp 4* are chosen as the proper components for *fun 1* and *fun 2*, individually. Therefore, instances of the two components will be reconstructed, and then attached to their corresponding functions.

Figure 21 presents the algorithm. At the transmitter side, line 6 detaches all components, except for the Type 2 and Type 3 components implementing their functions. Their IDs are recorded to an agent state S (line 8). When accepting the agent, the proper components will be determined and substitute for the components that are absent or unsuitable, as shown in lines 13–16.

4.3.4. Comparison

In this section, we tested these three strategies to see how the size and number of components affect time cost (ms) of agent migration and application adaptation. In the experiments, we consider HAM, FAM and LAM under the worst case. In addition, in LAM we measure cases of LAM under the best case. The best case means that all components carried by an agent do not need to be replaced. The worst case indicates that all components carried by an agent need to be switched to the proper components. Figure 22 shows the experimental setting. We measure the round trip time during which CAAS server 1 informs CAAS server 2 to instruct an agent to migrate successfully. To analyze the results accurately, we measure each case 10 000 times to compute the average of the results.


```
LAM - TRANSMITTER(G)
1 GET - AGENT - STATE(G, S)
2 GET - APPLICATION - IDs(S, A)
3 for i ← 0 to length[A]
4 do GET - FUNCTION - IDs(S, A[i], F)
5     for j ← 0 to length[F]
6     do DETACH - ALL - EXCEPT - IMPLEMENTING - COMPONENT(G, A[i], F[j], C)
7         for k ← 0 to length[C]
8         do ADD - TO - ABSENCE - COMPONENT(G, S, A[i], F[j], C[k])
9 return G

LAM - RECEIVER(G, T, Q)
10 GET - AGENT - STATE(G, S)
11 GET - APPLICATION - IDs(S, A)
12 for i ← 0 to length[A]
13 do GET - CHANGED - OR - ABSENCE - FUNCTION - IDs(S, A[i], F)
14     DECIDE - PROPER - COMPONENT(A[i], F, T, Q, D)
15     for j ← 0 to length[D]
16     do SWITCH - COMPONENT - TO(G, A[i], F[j], D[j])
17 return G
```

Figure 21. The algorithm of LAM.

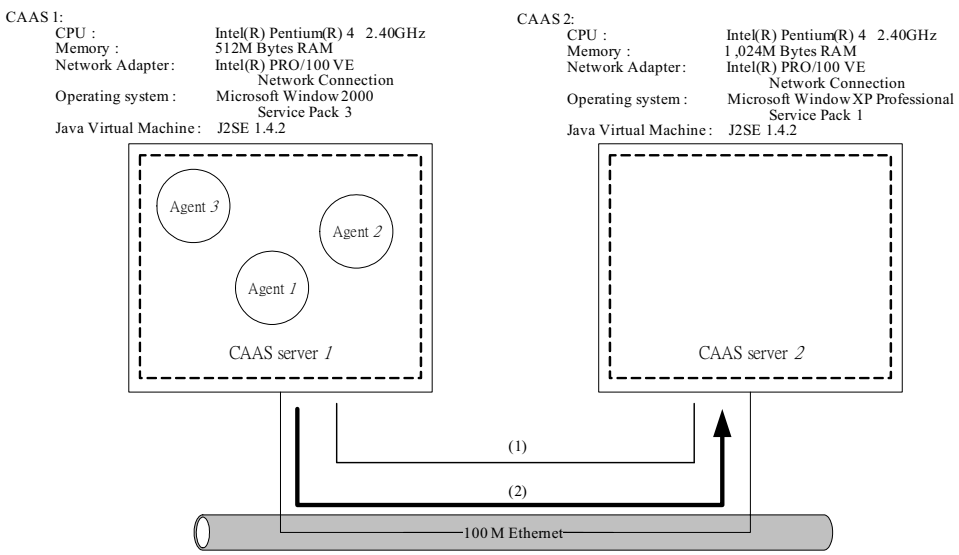

Figure 22. The experimental setting.

Table II. The results of the first measurement (one application, one function, two components).

Component size (bytes)	HAM	FAM	LAM-B	LAM-W
512	3.134	3.195	3.027	3.125
1024	3.345	3.409	3.249	3.253
2048	3.911	4.136	3.633	3.890
4096	4.250	4.333	4.192	4.284
8192	6.514	6.663	6.431	6.483
16 384	8.472	8.627	8.494	8.556
32 768	15.153	14.463	14.158	14.380
65 536	28.467	28.242	27.942	28.063
131 072	124.181	122.134	121.713	122.29

Table III. The results of the second measurement (one application, one function, one component).

Component number	HAM	FAM	LAM-B	LAM-W
50	7.016	7.021	6.816	6.800
75	9.895	10.050	9.793	9.825
100	13.866	14.656	12.473	12.029
125	14.160	14.340	14.018	14.045
150	19.667	20.514	19.430	19.444
175	20.482	20.994	20.712	20.732
200	21.844	22.141	21.931	21.841
225	24.719	24.323	24.378	24.380
250	28.770	29.221	28.443	28.422

We experiment on the strategies through two measurements. First, we let an agent carry an application containing one function, which is implemented by two components. We consider the cases of HAM, FAM, LAM-B, and LAM-W by gradually increasing the size of the two components from 512 to 128 kB. Table II and the upper part of Figure 23 demonstrate the time cost (ms) of the cases. As we can see, FAM performs worse than the other three; on the whole the cases of LAM-B and LAM-W cost less than the others, and LAM-B performs best. Second, we let an agent carry an application composed of one function, which can be implemented by 50, 75, 100, ..., 250 components separately. In Table III and the lower part of Figure 23, the results indicate that HAM and FAM perform worse than LAM-B; while LAM performs better than the others. In this situation, the time needed increases with an increasing number of components. This is because each of the algorithms performs a certain operation one by one for each component attached. For example, the HAM algorithm detaches all of the immoveable components.

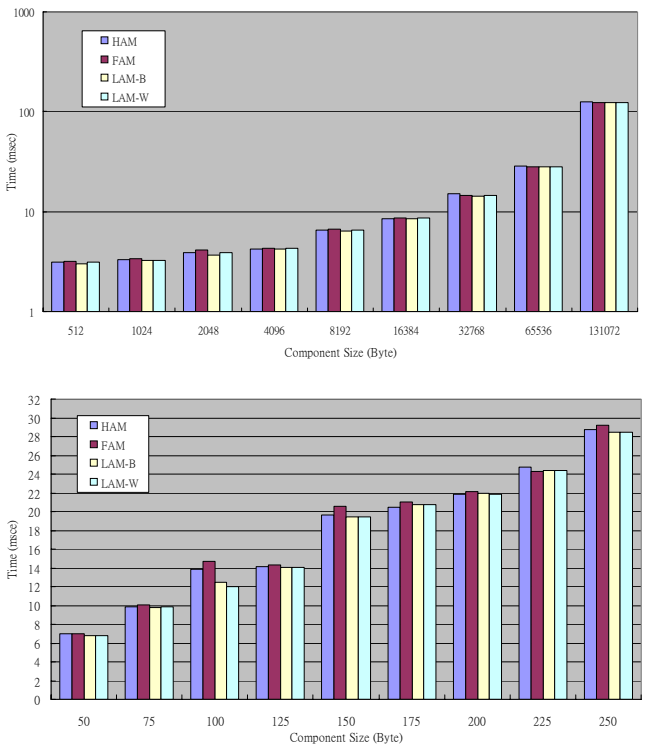

Figure 23. The experimental results for HAM, FAM, LAM-B, and LAM-W.

4.4. Class loading for composing application

We have mentioned that the proper components can be reconstructed and attached to implement their corresponding functions. Nevertheless, we encounter the problems of how to bind the necessary components to applications of agents, or how to distribute the classes and the information of the deployed applications to the CAAS server. In this section we will explain two methods used in our current system to solve these problems.

However, the problem that the required classes cannot be found may occur in class loading after an agent migrates to a new CAAS server. To solve this, we have designed two approaches: *application preloading* and *application remedying*. Application preloading is to load in advance the classes that the applications need. We can make the component managers on CAAS servers load the classes of applications while the applications are deployed into the repository service. After that, object instances of the classes required can be created and initialized without loading classes. Suppose that an agent migrates to a CAAS server, and the applications carried are adapted, we can use application preloading when these classes are not found. A CAAS server can retrieve the required classes to declare their

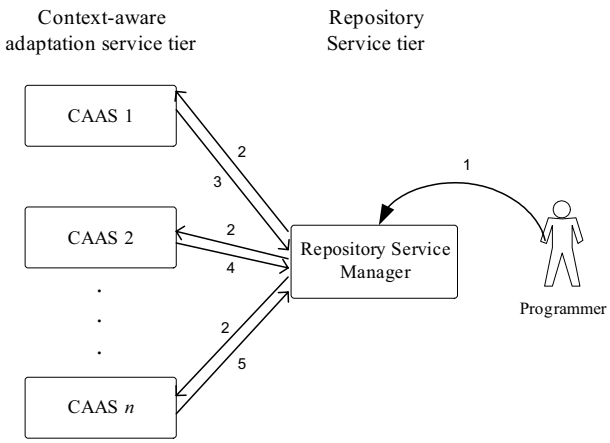

Figure 24. Application preloading.

object instances of the classes through the component manager. Figure 24 exhibits the scenario of application preloading. A programmer first finishes developing an application, and afterwards he deploys the application into the repository service. While receiving the deployment, the repository service will send the notifications to CAAS servers via IP multicasting. Once the CAAS servers get the notification, they will load the classes of the deployed applications.

However, in this scenario, the other problem is that the server has not loaded the required classes when an agent arrives in a CAAS server. This circumstance occurs right after the migration of the agent to a server which has not yet received the information of the deployed applications. This scenario might occur when the server is recovering from a crash. Therefore, we can solve this problem via *application remedying*. Provided that an agent carries back-end module 1 and migrates to CAAS server 2, an object instance of a certain class will have to be constructed. Therefore, two flows, $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 4 \rightarrow 9 \rightarrow 10$ and $1 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11$, are executed to complete this work, as shown in Figure 25. The former is where the component manager has finished loading the classes before the agent migrates. The latter is where the required classes are not found at that moment; thereupon, they will be loaded from the repository service via `URLClassLoader`, and then stored in the component repository at the CAAS server.

4.5. Component replacement issues

After application adaptation, invoking the same method will yield different results. Figure 26(a), for example, illustrates that after adaptation the component GIF Retriever replaces the component JPEG Retriever to implement the function Image Retriever. Invoking the same method of Image Retriever after adaptation, Object A can obtain a GIF image. Its implementation is shown in Figure 27. An object instance of the class `Function` can record a number of references of its candidate components.

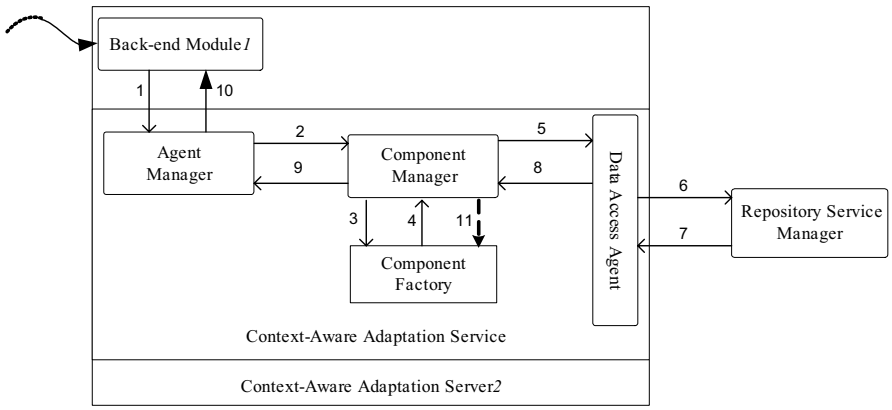

Figure 25. Loading classes through the repository service.

In addition, within a `Function` class a reference is used to keep the component that is implementing this function. If a method of a front-end module is invoked, the invocation will be sent to its back-end module. Accordingly, the method `invoke()` of the object reference (`currentComp`) kept in this back-end module will be invoked, as shown in lines 8–12. When the component implementing the function is replaced with another component, line 5 will be executed to change the kept object reference. Thus, if line 8 is invoked again, the result returned will be that of invoking the method `invoke()` of the new component. Therefore, in our system each user-defined component needs to implement the method `invoke()` (line 8) of an interface `IComponent`. As a result, we can invoke the same interface, and then our system can invoke the method `invoke()` of the component that currently implements this function.

Although adapting these components can simplify the programming, it might lead connections and data to change after application adaptation. For example, in Figure 26(b) the function Cache retains certain images, but the same result may not be retrieved after adaptation due to the replacement by Image Cache 2. On the other hand, certain components, connecting to resources and objects (TCP/IP links, HTTP request/reply, database connections), are treated as Type 1 components in our system. While a transaction is being executed or messages are delivered (in Execution state) via Type 1 components, no migration and adaptation of the agent can be performed. In some situations, new instances of the components can be established to continue execution of the same process after migration or adaptation. Figure 26(c) shows this example. In it, the component HTTP Connector 1 of the function Image Transmitter is substituted for HTTP Connector 2 after the agent migration, and images can be retrieved. However, the process of Figure 26(c) cannot guarantee the completeness of the whole transaction. Thus, exploiting Type 1 components, programmers can specify the components `"unchanging='Yes'"` in the ASCC profile and let agents anchor at a server without movements through `registerSynchronous()` in the API of the front-end module. Accordingly, agents do not follow their owners, and also components cannot be replaced. Briefly, when using Type 1 components

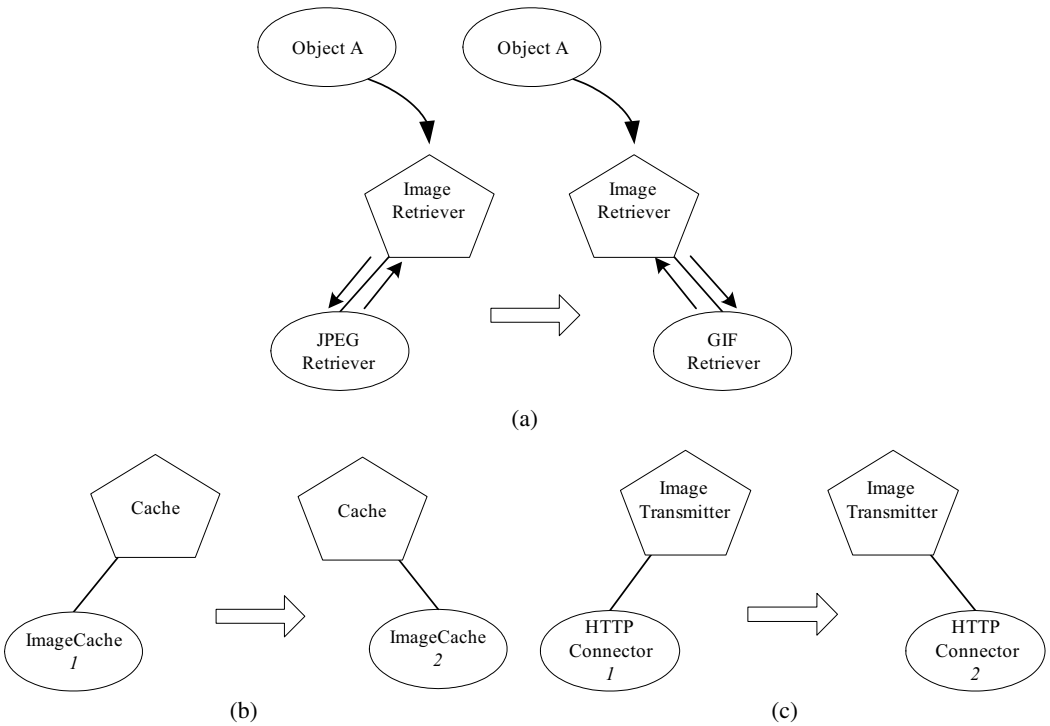

Figure 26. Cases of component replacement.

to connect to certain resources and Type 2 components to retain information, programmers need to notice the semantics of the adaptation on components.

5. CONTEXT-AWARE ADAPTATION

The use of the context profile to decide upon proper components in application adaptation is a critical issue. In this paper, we aim to develop an attribute-based algorithm that chooses components appropriately by using CC/PP and WAP UAProf profiles.

5.1. CC/PP and WAP UAProf

Figure 28 illustrates a profile of WAP UAProf in Resource Description Framework RDF format. RDF is a general-purpose language for representing information on the Web. This description covers six parts describing some characteristics of devices: Hardware Platform,


```
1<?xml version={"}1.0{"}?>
2  <rdf:RDF ...>
3   <rdf:Description rdf:ID={"}Nokia8310{"}>
4    <prf:component>
5     <rdf:Description rdf:ID={"}HardwarePlatform{"}>
6      <rdf:type rdf:resource= {"}http://www.wapforum.org/profiles
7       /UAPROF/cppschema-20010430{\#}HardwarePlatform{"}/>
8      <prf:Keyboard>PhoneKeypad</prf:Keyboard>
9      <prf:NumberOfSoftKeys>2</prf:NumberOfSoftKeys>
10     <prf:ScreenSize>84x30</prf:ScreenSize>
11     <prf:ScreenSizeChar>10x3</prf:ScreenSizeChar>
12     <prf:StandardFontProportional>
13      Yes
14     </prf:StandardFontProportional>
15     <prf:Vendor>Nokia</prf:Vendor>
16     <prf:Model>8310</prf:Model>
17     <prf:TextInputCapable>Yes</prf:TextInputCapable>
18    </rdf:Description>
19   </prf:component>
20   ...
21  </rdf:Description>
22 </rdf:RDF>
```

Figure 27. The XML serialization form of the profile in Figure F28.

Software Platform, Network Characteristics, BrowserUA, WAP Characteristics, and Push Characteristics. As in Figure 28, Hardware Platform specifies the hardware properties of devices. These properties include `prf:Keyboard`, `prf:NumberOfSoftKeys`, `prf:ScreenSize`, and `prf:ScreenSizeChar`, whose values can be `PhoneKeypad`, `2`, `84x30`, or `10x3`, respectively. Figure 27 demonstrates the XML serialization form of the context profile in RDF.

5.2. Attribute-based component decision algorithm

Take a profile Q for example. The profile Q includes a set of attributes, which can be expressed as $\{a_i \mid 1 \leqslant i \leqslant n\}$, where a_i and n denote an attribute and the total number of the attributes in the profile individually. Let $domain(a_i) = \{av_{i,k_i} \mid 1 \leqslant i \leqslant n \text{ and } 1 \leqslant k_i \leqslant v_i\}$ indicate the domain of the attribute a_i, and $value[a_i]$ be the value of the attribute, where v_i is the number of possible values of a_i. For instance, Figure 27 involves the attribute *TextInputCapable*, which has $value[TextInputCapable] = yes$ and $domain(TextInputCapable) = \{yes,no\}$.

An agent body contains a number of applications. An application comprises one or more functions $fun_1, fun_2, \ldots, fun_m$. Each of them can be implemented by at least one component, $comp^i_1, comp^i_2, \ldots, comp^i_{r_i}$, where $1 \leqslant i \leqslant n$ and r_i denotes the number of the user-defined components implementing fun_i. For example, fun_1 can be implemented by components $comp^1_1$ and $comp^1_2$ (illustrated in Figure 29). Each component $comp^x_y$ has a *constraint set* $CS_{x,y}$, which contains

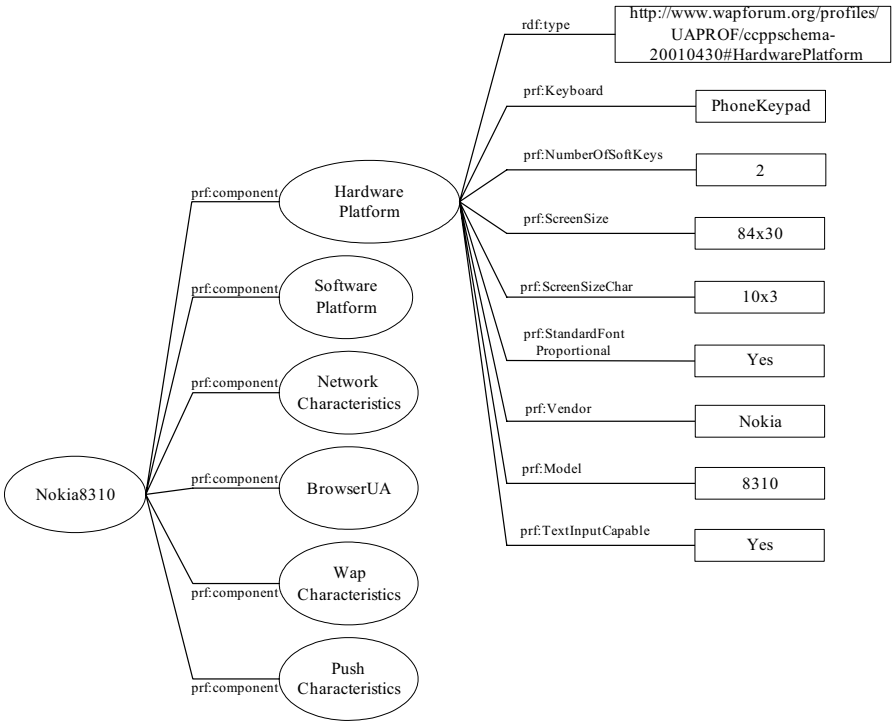

Figure 28. A segmentation of Nokia 8310's WAP UAProf profile.

zero or more tuples (a_i, av_{i,k_i}), where $1 \leqslant i \leqslant n$ and $1 \leqslant k_i \leqslant |domain(a_i)|$, annotated under each component shown in Figure 29. We can accomplish the testing of a component to see if it can be chosen to implement its corresponding function by using this constraint set. For a component $comp_y^x$, if for the given profile Q, $comp_y^x$ can be chosen, it must be true that each attribute value av_{i,k_i} of (a_i, av_{i,k_i}) in its $CS_{x,y}$ is equal to the value of the same attribute a_i in Q. If so, we say that the component is satisfied. For example, assume that a certain profile and two components $comp_1^1$ and $comp_2^1$, and the function fun_1 are given. The component $comp_1^1$ has the constraint set $\{(ColorCapable, yes)\}$ and $comp_2^1$ has $\{(ColorCapable, no)\}$. Because the value of the same attribute *ColorCapable* in this profile is *yes*, $comp_1^1$ is satisfied. $comp_1^1$ can be chosen to implement fun_1 accordingly. A constraint set, in implementation, can be established by a `<constraints>` element in the ASCC description. As in Figure 3, lines 9–10 describe two elements, `<prf:ImageCapable>` and `<prf:CcppAccept>`. Therefore, the component $comp_1^1$ is declared suitable for processing JPEG files. As a result, the constraint set $CS_{1,1} = \{(ImageCapable, yes), (CcppAccept, image/jpeg)\}$ will be generated.

A component decision tree can be seen as a tree hierarchy. It comprises a number of attribute nodes, each of which has several branches linked to other attribute nodes as its child nodes. Let an_{i,d_i} indicate

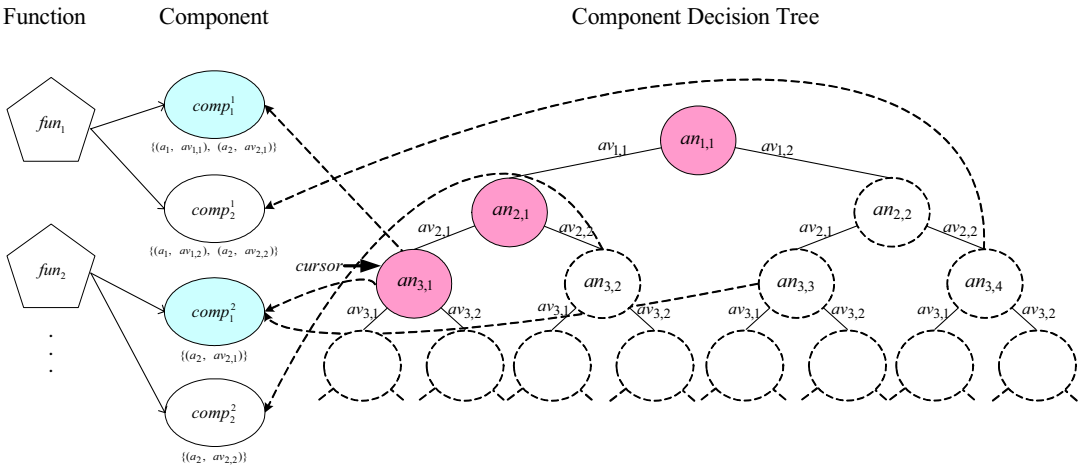

Figure 29. A component decision tree and its linked components.

an attribute node, which is semantically equivalent to the attribute a_i with the same name in the given profile Q. Let av_{i,k_i} denote a branch of an attribute node an_{i,d_i}, where $1 \leqslant k_i \leqslant |domain(a_i)|$, d_i is between 1 and the component number at the same level in a tree, and $1 \leqslant i \leqslant n$.

Each attribute node an_{i,d_i} has a *linked component set* LC_{i,d_i} that includes the components associated via dotted lines in the component decision tree, illustrated in Figure 29. As in the figure, the linked component sets of the attribute nodes $an_{3,1}$ and $an_{3,2}$ are $LC_{3,1} = \{comp_{1,1}, comp_{2,1}\}$ and $LC_{3,2} = \{comp_{2,2}\}$, respectively.

To operate a component decision tree, there are a pointer *cursor* and two operations, $NEXT(an_{i,d_i}, an_{i+1,d_{i+1}})$ capable of moving *cursor* from an attribute node an_{i,d_i} to its child node $an_{i+1,d_{i+1}}$, and $VISIT(an_{i+1,d_{i+1}})$ representing *cursor* visiting an attribute node $an_{i+1,d_{i+1}}$. Taking Figure 29, for example, the pointer *cursor* will point to the attribute $an_{3,1}$ when the operation $NEXT(an_{2,1}, an_{3,1})$ is applied; thereupon $an_{3,1}$ is visited, denoted by $VISIT(an_{3,1})$. Furthermore, let t denote a traverse from the root to a certain leaf node. A traverse t, a sequence of $VISIT()$ and $NEXT()$, can be expressed as $SEQ(t) = <VISIT(an_{1,1}), NEXT(an_{1,1}, an_{2,d_2}), VISIT(an_{2,d_2}), NEXT(an_{2,d_2}, an_{3,d_3}), \ldots, NEXT(an_{i,d_i}, an_{i+1,d_{i+1}}), \ldots, VISIT(an_{n,d_n})>$. In Figure 29, for instance, a traverse t starts from the attribute node $an_{1,1}$ to the attribute node $an_{3,1}$. Thus, $SEQ(t)$ is equal to $<VISIT(an_{1,1}), NEXT(an_{1,1}, an_{2,1}), VISIT(an_{2,1}), NEXT(an_{2,1}, an_{3,1}), VISIT(an_{3,1})>$. Accordingly, while a traverse t is built, the linked component set LC_{i,d_i} of each attribute node an_{i,d_i} visited can be united to establish a proper component set

$$P(t) = \bigcup_{\substack{\text{for each } i \text{ and } d_i, \\ \text{where } VISIT(an_{i,d_i}) \text{ in } SEQ(T)}} LC(an_{i,d_i}).$$

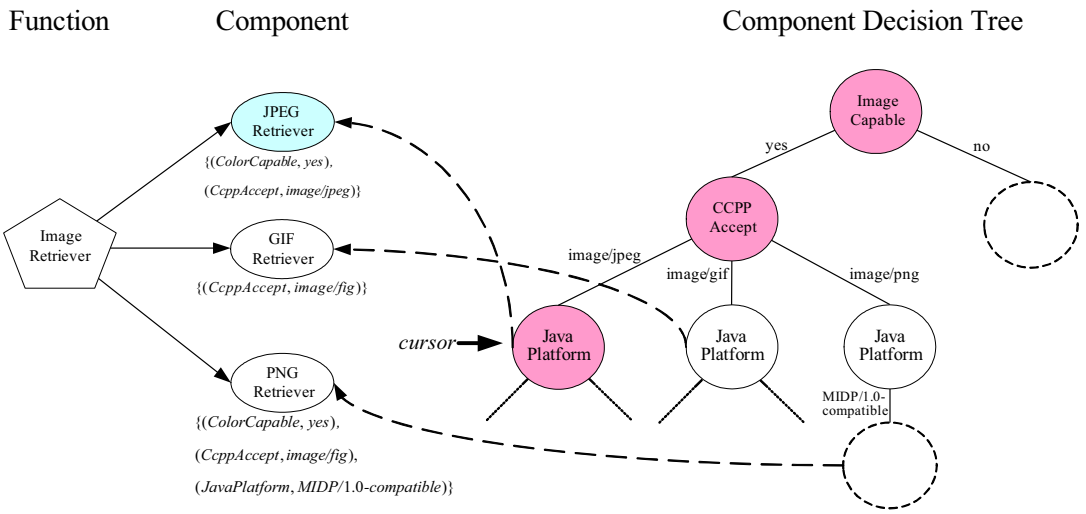

Figure 30. An instance of a component decision tree on the right-hand side, and the associated components of the application ImageGathering on the left-hand side.

For example, suppose that there is a profile {*ImageCapable*, *CCPPAccept*, *JavaPlatform*, ...}, and their domains can be expressed as *domain*(*ImageCapable*)={*yes*, *no*}, *domain*(*CCPPAccept*)={*yes*, *no*}, etc. Figure 30 demonstrates the structure of the functions and components of an application. The function Image Retriever can be implemented by three components: JPEGRetriever, GIFRetriever, and PNGRetriever. The constraint set of the first component is $CS_{1,1}$ = {(*ColorCapable*, *yes*), (*CcppAccept*, *image/jpeg*)}, and that of the second component is $CS_{1,2}$ = {(*ColorCapable*, *yes*), (*CcppAccept*, *image/gif*)}. Moreover, in the component decision tree, each attribute node an_{i,d_i} has a number of branches and a linked component set LC_{i,d_i}. As in Figure 30, the attribute node *ImageCapable* has two branches, *yes* and *no*. The attribute node *CCPPAccept* has three branches encompassing *image/jpeg*, *image/gif*, and *image/png*. In addition, the attribute node $an_{3,1}$ is a *JavaPlatform* whose linked component set is $LC_{3,1}$ = {*JPEGRetriever*}, and that of $an_{3,2}$ is $LC_{3,2}$ = {ImageRetriever}.

Let us assume that a traverse t is made by moving *cursor* from the root $an_{1,1}$ (*ImageCapable*) to the leaf node $an_{3,1}$ (*JavaPlatform*). As a result, the sequence $SEQ(t)$ = <*VISIT(ImageCapable)*, *NEXT(ImageCapable, CCPPAccept)*, *VISIT(CCPPAccept)*, *NEXT(CCPPAccept,JavaPlatform)*, *VISIT(JavaPlatform)*> and the proper component set of t, $P(t)$ = {*JPEGRetriever*} are established.

The problem to be solved through the attribute-based component decision algorithm is how to decide upon a proper component to implement each function f if given an application p and each function f of the application p; or how to adapt an application p. This is because $SEQ(t)$ and $P(t)$ will be generated after traversing from the root to a leaf node. In $SEQ(t)$, $NEXT(an_{i,d_i}, an_{i+1,d_{i+1}})$ implies $VISIT(an_{i,d_i})$ and $value[an_{i,d_i}] = av_{i,d_i}$. Therefore, if a component $comp_y^x$ exists in $P(t)$, then $\forall i, k_i$ $value[an_{i,d_i}] = av_{i,d_i} = value[a_i]$, where $an_{i,d_i} = a_i$ and a_i in $CS_{x,y}$. In other words, for a

traverse t the proper component set $P(t)$ contains the components which are satisfied. Specifically speaking, given an application, if a suitable component exists for each function, this component can be chosen from $P(t)$. Moreover, if there are two or more suitable components at the same time, the last-examined component will be chosen as the default. This algorithm solves the problem and eliminates the need for traversing a tree from the root to a leaf node. Once sufficient components exist in the proper component set $P(t)$, traversing a component decision tree can terminate at some internal attribute node which is not a leaf node.

In implementation, instead of realizing this algorithm by using the data structure tree, we realize this algorithm by means of a linking list. The reason for this is that using the tree as the data structure consumes more memory space to choose proper components. For each attribute node an_{i,d_i} at the same level of a component decision tree, the information recorded for the nodes seems different, except for the linked component set LC_{i,d_i}. However, the information is essentially identical. Take the previous profile Q and the tree in Figure 29 as an example. At level 3, $an_{3,1}$, $an_{3,2}$, $an_{3,3}$, and $an_{3,4}$ are semantically equivalent to the attribute a_3 in the profile Q. Therefore, to implement the concept tree, we use a linking list. In this way, for each level in a tree, attribute nodes an_{i,d_i}, for all d_i, where $1 \leqslant i \leqslant n$ and $1 \leqslant d_i \leqslant v_i$, are regarded as one node in a linking list. Figure 31 represents a linking list that starts from the root attribute node connecting to its child attribute node in the tree as the next node, which also links to its child node as the next node, and so on. This hierarchy of the linking list equals that of the component decision tree. In this list, an attribute node an_i has two links: one connects to a child attribute node an_{i+1}; the other binds its linked component set (a hash table in practice). In Figure 31, for example, the linking list, kept by a table index, starts from the attribute node an_1 to the attribute an_4, each of which binds a linked component set. For instance, the attribute node an_3 retains a link component set containing two components $comp_1^1$ and $comp_2^1$.

Figure 32 illustrates the implementation of the component decision tree (Figure 30). Symmetrically, by traversing from the root node Image Capable to the node Java Platform, the proper component JPEG Retriever for the function Image Retriever can be chosen.

In our system, we apply an attribute-based component decision algorithm to the application adaptation. Applications carried by the agent are adapted when the agent migrates to a new CAAS server. Implementing the component decision tree by a linking list simplifies the maintenance of attribute nodes. The space complexity is the sum of linked component hash tables $|\sum_{\text{all } d_i} LC_{i,d_i}|$ for all i, where $1 < i \leqslant M$. It is less than $n_m * M$, where n_m is the number of attributes, and M is the size of the maximum linked component hash table. M is a constant. Therefore, the space complexity is $O(n_m)$.

In terms of time complexity, the time complexity is constant for the attribute-based algorithm as the processing time does not depend on the number of components. By contrast, we can consider a simple algorithm that decides upon proper components by examining each component. Thus, we inspect the constraint set $CS_{x,y}$ for each component $comp_y^x$. This costs $O(n_c * n_m)$ worst-case time, where n_c denotes the total number of components of an application, and $n_m = \max(|CS_{x,y}|)$ indicates the total number of attributes. The cost of the attribute-based algorithm is merely affected by the length of the linking list (the height of the component decision tree). In addition, the link can be built from the attributes in $CS_{x,y}$ for all components $comp_y^x$ in an application instead of generating it from all attributes in a given profile. Therefore, its time complexity costs $n_m = \max(|CS_{x,y}|)$. This means that the time complexity is dominated by the size of the maximum constraint set. As it can be seen, using the attribute-based algorithm to support decisions about component selection facilitates

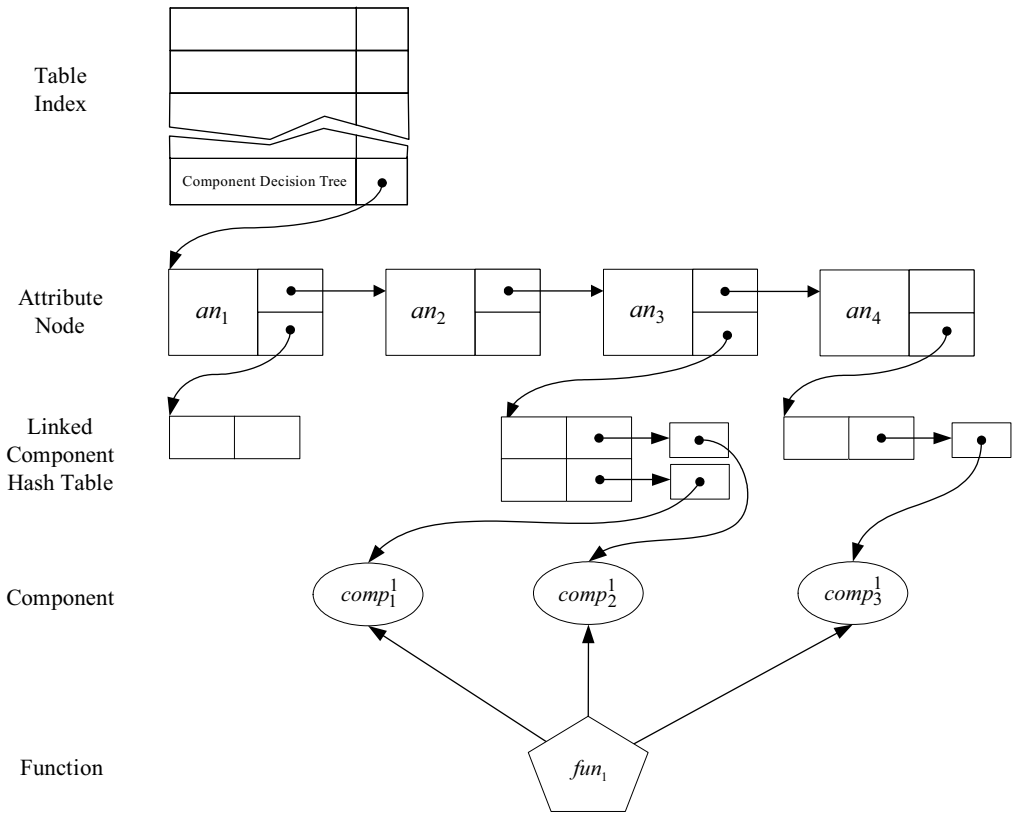

Figure 31. The implementation (linking list) of a decision tree.

programming of adaptive applications. It can support a large-scale system with a large number of diverse implementations of particular functions.

5.3. Hash tables in the application context

We explained the algorithm in the previous section. The algorithm uses auxiliary hash tables providing the functionalities that include choosing the component, creating the component, and constructing application for the application adaptation. Therefore, the time cost of searching for the required components is decreased to $O(1)$ constant time.

Figure 33 illustrates the structure of these hash tables in an `ApplicationContext` class. This class contains the application structure and component constraints generated from an ASCC profile and keeps them by using hash tables. In the top-left of this figure, an index hash table is

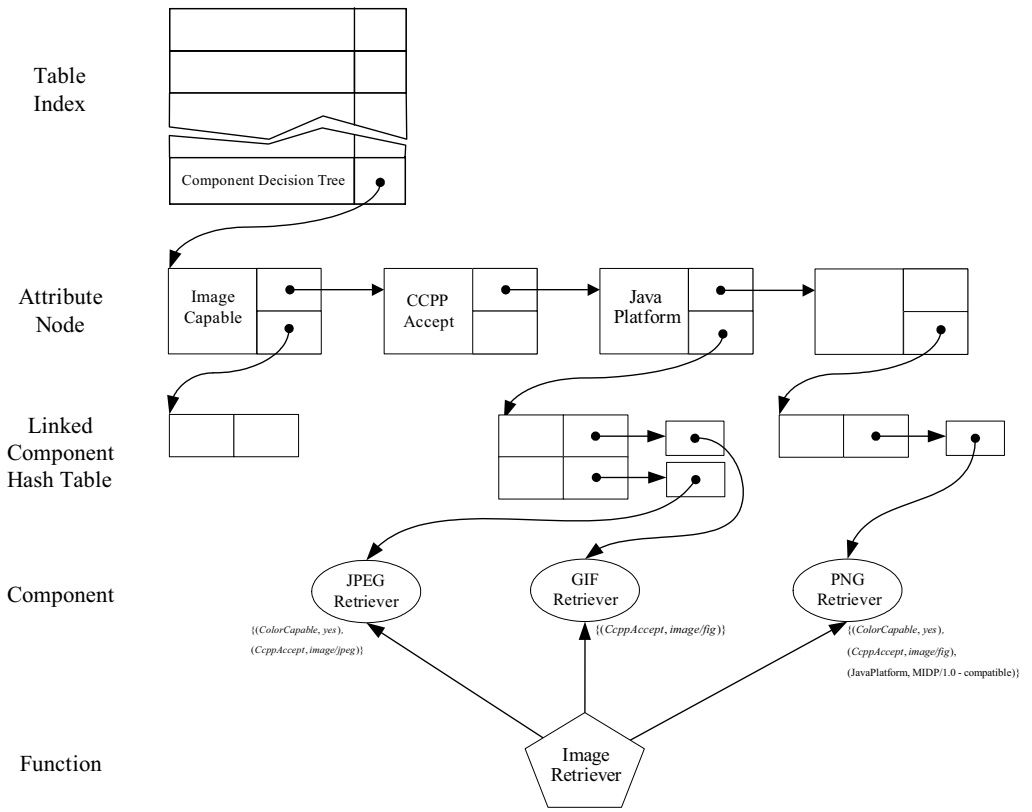

Figure 32. The linking list of the decision tree illustrated in Figure 30.

associated with the hash tables: a changed components hash table, a component hash table, a function hash table, and so forth. Furthermore, it has a component context table, which includes stateful/stateless component hash tables, relative/irrelative component hash tables, etc.

A *function hash table* records the functions that make up an application, and a *component hash table* retains the components capable of implementing these functions. Through a *changed component hash table* and a *component context table*, information on components can also be retrieved. The former is used to retrieve the information related to the components needed to switch in application adaptation. The latter can be used to get components with particular semantics depending on their characteristics, including stateful/stateless, relative/irrelative, and moveable/immoveable. As in Figure 33, a stateless component hash table keeps a number of pairs, each of which keeps two references: one is the reference of a certain component and the other is its corresponding function. Component 1, for instance, is declared a stateless component and *fun 1* its corresponding function.

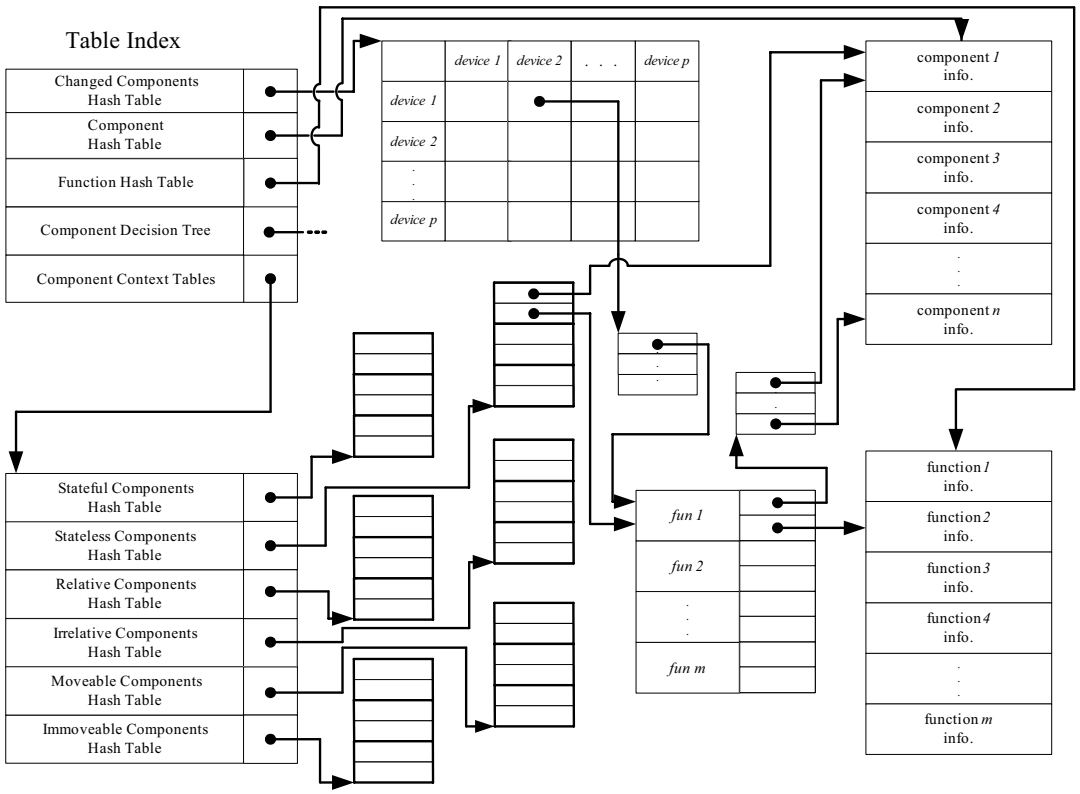

Figure 33. Hash tables in an application context.

6. REMOTE DYNAMIC INVOCATION

We have designed APIs for the development of the front-end module (Figure 34) and the back-end module (Figure 10). For programming on user devices, a programmer can use the `IUserAgent` interface. Through the API, programmers can set the synchronous/asynchronous mode of their personal agents through `registerSynchrony()`. Furthermore, we can obtain the `Method` object associated with a method of an application in the back-end module by means of the `getMethod()` method. In this way, if a programmer wants to let his front-end module invoke the method of the back-end module, he can use `getMethod()` to get the method he wants by specifying the method name as the parameter of `getMethod()`. Actually, while the `invoke()` method of the `Method` object is invoked, the invocation is essentially serialized into a byte message and transmitted to the remote back-end module. Figure 35 demonstrates the execution sequence of this procedure.

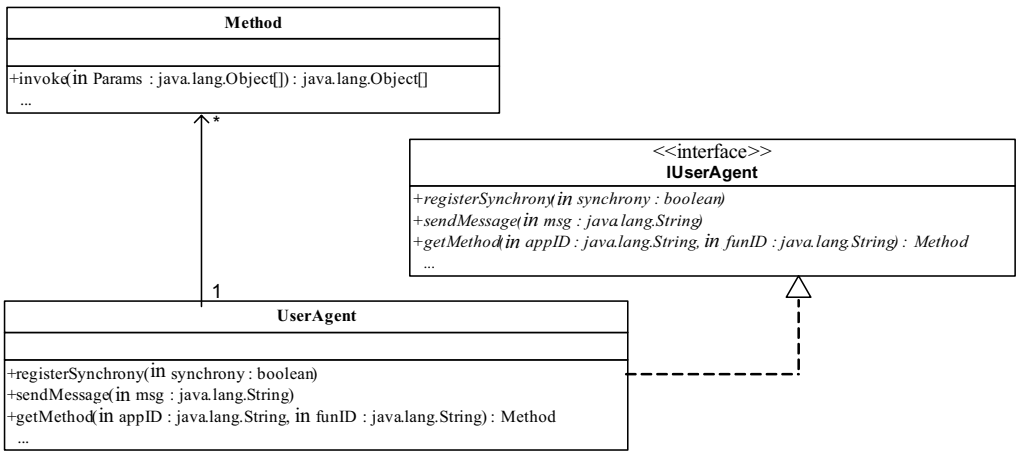

Figure 34. Application interfaces (APIs) on client devices.

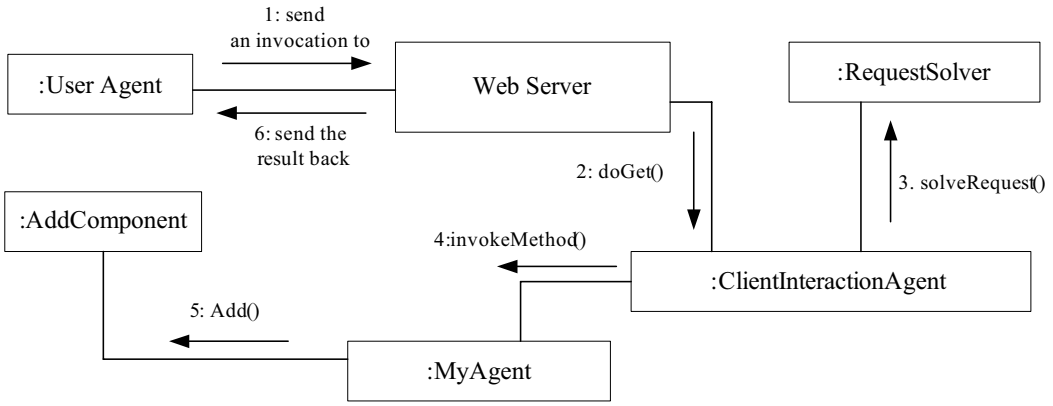

Figure 35. Sequence flow of remote dynamic invocation.

To design applications in back-end modules, programmers can declare three kinds of user-defined classes (`MyAgent`, `MyApplication`, and `MyComponent`, as mentioned in Section 3.3). The first class can be used to construct a personal agent. An application carried by the agent can be created through the `MyApplication` class. Further, the components of the application can be derived from the `Component` class. Each of these classes has the substantial computational logic of the application. Programmers can declare the subclasses of the `Component` class to actualize the functions of applications. For instance, programmers can define `AddComponent` and `SubComponent` components to sum and subtract two numbers, respectively.

Figure 35 exhibits a process flow of *remote dynamic invocation*. The front-end module and the back-end module, in effect, run on a client device and on the server side, respectively. The classes related to `Agent`, `Function`, `Component` can be referred to in Figure 10 (Section 3.3). Suppose that there is a simple calculator application developed based on our system, users can add or subtract two numbers. A user can input two numbers and then press a compute button to calculate the two numbers by using the two components `AddComponent` or `SubComponent`. In this application, we let the back-end module compute the two numbers using `AddComponent` when users utilize Nokia 6600, or instead using `SubComponent` when users utilize Nokia 3300. This can be described in the ASCC profile of the application. Depending on the description, CAAS servers can choose one of the two components in application adaptation. We consider the following scenario. A user uses a J2ME-enabled phone and inputs two numbers. After he presses the compute button, `ClientDevice` of the front-end module of the application can send the serialized byte string of the invocation to the back-end module. While receiving the message, the system component `ClientInteractionAgent` will invoke `invokeMethod()` of the `MyAgent` class, and the method `invoke()` of the `Function` object appointed thereupon. Subsequently, the component implementing this `Function` object will be invoked. At that moment, if the user is utilizing Nokia 6600, the method `invoke()` of `AddComponent` will be invoked; if instead he is utilizing Nokia 3300, that of `SubComponent` will be invoked.

7. RELATED WORK

In our design, remote dynamic invocation acts to complete invocations between the front-end and back-end modules. RMI, a method invocation on remote objects, is a widely used interaction paradigm. However, not all devices support RMI. Java reflection [24] (Section 4.4) lets programmers invoke the appointed method of the object determined dynamically at run-time. The mature RMI and Java reflection techniques enable programmers to develop applications efficiently, but have not been completely supported in mobile execution environments. For example, in the J2ME run-time environment, Sun Microsystems has not defined the RMI mechanism in the J2ME specification. Though Sun Microsystems defined the RMI interfaces on the CDC environment (an optional package of CDC), it did not provide the RMI interfaces on the CLDC environment. As can be seen in Figure 2, the devices being used in the CDC environment are PDA, Palm, Pocket PC, and Smart Phone, while the devices with lower computational power only provide the CLDC environment.

Most mobile agent systems [25,26] provide abundant functions, including agent migration, communication of agents with other agents and with the underlying system, as well as support for security, transactions and controlling agents. For instance, MOLE [27] offers an agent migration

infrastructure with all of these functions, such as a protocol for fault-tolerant execution of mobile agents, accounting and billing, and control algorithms for finding agents, terminating agents, and orphan detection. Though complete functions support the mobile agent, adapting application according to the characteristics of the small and handheld devices has not been provided yet.

Some previous research has focused on the intrinsic structure of mobile agents and mobility behaviors of mobile agents, such as MobileSpaces [28]. MobileSpaces proposes agent hierarchy and inter-agent migration. The former is so that an agent can have several child agents, each of which also has agents as its child agents, and so on. The latter means that an agent is capable of migrating into another computer or to within an agent. Also, this framework makes agents adaptable. It regards a mobile agent as a component, and can combine a collection of agents into a single agent. Several agents are organized hierarchically into one agent. Additionally, this compound mobile agent can be adapted to the target environments. Although the hierarchical structure and adaptable concept for the mobile agents are provided in this framework, it does not structure the application or consider the context-aware adaptation for various mobile devices.

m-P@gent [27,29] provides environment-aware mobile agents capable of running on resource-limited devices and appliances. In addition, it supports the run-time environment with mobile applications on the mobile devices, and contains four subsystems—@Desk for the PC platform, @Palm for the Palm device platform, @Pocket for the PocketPC platform, and @TINI for the TINI device platform. Moreover, it divides a mobile agent into two parts: a core and add-on functional modules. Then, it can adapt add-on modules of the agent to a run-time environment via a specific profile for each run-time environment, such as the profile for J2SE and another profile for J2ME. Yet, this framework lacks the ability to distribute the computational loading of applications on the small and handheld devices. In other words, the capabilities of the applications on this mobile agent system are restricted by the limitations of the devices. Furthermore, to adapt each component of the mobile agent, it is necessary to describe the type and class of a component for each run-time environment. In our system, only a description of the component constraints in an ASCC profile is needed for the same purpose.

On the other hand, some researchers [1,2,6] have explored the follow-me applications. Harter *et al*. [2] describe a sensor-driven, or sentient, platform for context-aware computing that enables applications to follow users while they move around a building. Takashio *et al*. [6] also propose a mobile agent framework f-Desktop for the migration mechanisms of follow-me applications in an ubiquitous computing environment and evaluate its basic performance. Even though the basic functions of migration and adaptation of applications are provided, this framework does not concern the real context profiles of mobile devices for adaptation, and does not help run applications on these mobile and embedded devices.

In context sensing and modeling, Schmidt *et al*. have explored context acquisition from sensors [30], and aim to model the context information [31,32]. Gray and Salber [31] present a way of analyzing sensed context information formulated to help in the generation, documentation and assessment of the designs of context-aware applications. Furthermore, to use CC/PP as the context information, Indulska *et al*. [33] address a context model and a context management system able to offer pervasive systems, and discuss the pros and cons of the CC/PP framework.

For developing context-aware applications, Dey *et al*. [34] describe a distributed software infrastructure to support context-aware applications in the Aware Home, a prototype smart environment. Their infrastructure is similar to the Situated Computing Service [35]. Both of them discuss polling and notification mechanisms to impart application information of context changes.

Kermarrec *et al*. [36] focus on a contextual object, a conceptual object model, for developing applications toward adaptation on the continuous changes of the mobile environment. A contextual object has a context-sensitivity list (similar to component constraints in our framework) for describing the dependencies of an object and the kind of context that it senses. In addition, it has a reference to some real object (e.g. HTML page, Java Class, etc.) to represent the value of this object in the current context. A conceptual framework for context-aware applications in current mobile and Internet environments has also been proposed [37]. The framework contains three parts. The first is the context management part capable of sensing and aggregating data, and managing the set of context groups. The second is the service management part that selects the appropriate services with context information from the context management part, and returns the services to the adaptive user interface part. The third is the adaptive user interface part, which provides users with the adaptive and Web-based user interface with selected services. All of the frameworks can facilitate the development of context-aware applications and a fundamental adaptation infrastructure for the applications on a ubiquitous computing environment. Nevertheless, the weakness of their frameworks lies in the decision of the appropriate component or service for application adaptation according to context information.

8. CONCLUSION AND FUTURE WORK

In summary, we have explained our focus on transmitting agents efficiently and adapting applications to cope with the variability of user devices. By means of the front-end module and the back-end module, the restrictions in developing applications on small and mobile devices can be decreased. Furthermore, agents can synchronously migrate with their owners or be asynchronously anchored to their resident server. To transmit the agent efficiently, we experiment on agent migration strategies, and use the LAM as the default strategy for the agent migration. Additionally, by structuring applications in ASCC profiles, and leveraging CC/PP and WAP UAProf frameworks, the attribute-based component decision algorithm can choose the components suitable for the context of the user's devices.

Currently, there are some issues that need addressing, including the replacement of the stateful and relative components, the conflict of the component property declaration, the consistency between the ASCC profile and the back-end module, and the lack of proper component declaration. Therefore, in the future we will attempt to design a software development kit (SDK) to aid programming and consistency checking. To further enhance this framework, some services related to the integration of this framework will be discussed in the future. Transaction and security handling, as well as load balancing and faulty recovery can be achieved by including services of distributed computing platforms, such as J2EE [38]. The J2EE environment offers a distributed application model, a unified security model, flexible transaction control, etc. In transaction, several invocations between the front-end module and the back-end module of an application are regarded as an atomic unit. This transaction can be handled through some particular operations, such as commit or abort, and the two phases commit protocol. Security consists of authentication and authorization, which can be used to protect servers against malicious applications, and *vice versa*. In addition to the methodologies, we will attempt to integrate our framework with some mobile agent systems. IBM Aglet [25] and MOLE [26], for instance, have full-fledged mechanisms of security, transaction, etc. Furthermore, we intend to exploit the context sensing and modeling technologies to increase the use of contextual information toward adaptation in ubiquitous computing environment.

ACKNOWLEDGEMENT

This work was partially supported by the National Science Council under grant No. NSC93-2752-E-009-006-PAE, Advanced Technologies and Applications for Next Generation Information Networks (II)—Sub-project 5: Network Security.

REFERENCES

1. Schilit BN, Adams N, Want R. Context-aware computing applications. *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, 1994; 85–90.
2. Harter A, Hopper A, Steggles P, Ward A, Webster P. The anatomy of a context-aware application. *Mobile Computing and Networking* 1999; 59–68.
3. Chen G, Kotz D. A survey of context-aware mobile computing research. *Technical Report TR2000-381*, Department of Computer Science, Dartmouth College, Hanover, NH, November 2000.
4. Cheverst K, Davies N, Mitchell K, Friday A, Efstratiou. Developing a context-aware electronic tourist guide: Some issues and experiences. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2000; 17–24.
5. Asthana A, Cravatts M, Krzyanowski P. An indoor wireless system for personalized shopping assistance. *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, 1994; 69–74.
6. Takashio K, Soeda G, Tokuda H. A mobile agent framework for follow-me applications in ubiquitous computing environment. *Proceedings of the 21st International Conference on Distributed Computing Systems Workshops (ICDCSW '01)*, Mesa, AZ, 2001.
7. Klyne G, Reynolds F, Woodrow C, Ohto H, Hjelm J, Butler MH, Tran L. Composite Capability/Preference Profiles (CC/PP): Structure and vocabularies. W3C Working Draft, March 2003. http://www.w3.org/TR/CCPP-struct-vocab/.
8. Reynolds F, Hjelm J, Dawkins S, Singhal S. Composite Capability/Preference Profiles(CC/PP): A user side framework for content negotiation. W3C Note, 1999. http://www.w3.org/TR/NOTE-CCPP/.
9. Hjelm H, Suryanarayana L. CC/PP implementors guide: Harmonization with existing vocabularies and content transformation heuristics. W3C Note, December 2001. http://www.w3.org/TR/CCPP-COORDINATION/.
10. Ohto H, Hjelm J. CC/PP exchange protocol based on HTTP extension framework. W3C Note, June 1999. http://www.w3.org/TR/NOTE-CCPPexchange.
11. Butler MH. Implementing content negotiation using CC/PP and WAP UAProf. *External Technical Report HPL-2001-190*, 2001. Available at http://www.hpl.hp.com/techreports/2001/HPL-2001-190.html.
12. AP Forum. User agent profiling specification, October 2001. http://www.wapforum.org/tech/terms.asp?doc=WAP-248-UAProf-20011020-a.pdf.
13. Sun Microsystems. Java 2Platform micro edition technology for creating mobile device. *Sun Microsystems, Inc*, 2000.
14. JSR 118 Expert Group. JSR-000118 Mobile Information Device Profile 2.0 (Final Release), May 2002. http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html.
15. Brickley D, Guha RV, McBride B. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Working Draft, January 2003. http://www.w3.org/TR/rdf-schema/.
16. Lassila O, Swick RR. Resource Description Framework (RDF) model and syntax specification. W3C Recommendation, February 1999. http://www.w3.org/TR/REC-rdf-syntax.
17. Maruyama H *et al.* XML and Java. *Developing Web Applications* (2nd edn). Addison-Wesley: Reading, MA, 2002.
18. Wollrath A, Waldo J. Trail: RMI. http://java.sun.com/docs/books/tutorial/rmi/index.html.
19. Butler MH. DELI: A DElivery context LIbrary for CC/PP and UAProf. *External Technical Report HPL-2001-260*, February 2002. Available at: http://www.hpl.hp.com/personal/marbut/DeliUserGuideWEB.htm.
20. McBride B, Seaborne A, Carroll J. Jena Tutorial for Release 1.4.0, April 2002. http://www.hpl.hp.com/semweb/.
21. Geier J. Wireless LANs. *SAMS*, 2002.
22. Held G. *Data Over Wireless Networks: Bluetooth, WAP, and Wireless LANs*. McGraw-Hill: New York, 2001.
23. Wang J. *Broadband Wireless Communications: 3G, 4G, and Wireless LAN*. Kluwer Academic: Dordrecht, 2001.
24. Green D. Trail: The reflection API. http://java.sun.com/docs/books/tutorial/reflect/index.html.
25. Lange D, Oshima M. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley: Reading, MA, 1998.
26. Baumann J, Hohl F, Rothermel K, Strasser M, Theilmann W. MOLE: A mobile agent system. *Software—Practice and Experience* 2002; **32**:575–603.
27. Takashio K, Mori M, Funayama M, Tokuda H. *Constructing Environment-Aware Mobile Applications Adaptive to Small, Networked Appliances in Ubiquitous Computing Environment* (*Lecture Notes in Computer Science*, vol. 2574). Springer: Berlin, 2002; 230–246.

28. Satoh I. MobileSpaces: A framework for building adaptive distributed applications using a hierachical mobile agent system. *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, Taipei, Taiwan, 2000.
29. Takashio K, Mori M, Tokuda H. m-P@gent: A framework of environment-aware mobile applications for small, networked appliances. *Proceedings of the 2002 IEEE 4th International Workshop on Networked Appliances*, Gaithersburg, MD, 2001.
30. Schmidt A, Asante Aidoo K, Takaluoma A, Tuomela U, Van Laerhoven K, Van de Velde W. Advanced interaction in context. *Proceedings of the First International Symposium on Handheld and Ubiquitous Computing*, Karlsruhe, Germany, September 1999; 89–101.
31. Gray PD, Salber D. Modelling and using sensed context information in the design of interactive applications. *Proceedings of the 8th IFIP Working Conference on Engineering for Human-Computer Interaction (EHCI'01)*, Toronto, 2001.
32. Henricksen K, Indulska J, Rakotonirainy A. Modeling context information in pervasive computing systems. *Proceedings of the 1st International Conference on Pervasive Computing, Pervasive 2002* (*Lecture Notes in Computer Science*, vol. 2414). Springer: Berlin, 2002; 169–180.
33. Indulska J, Robinson R, Rakotonirainy A, Henricksen K. Experiences in using CC/PP in context-aware systems. *Proceedings of Mobile Data Management, 4th International Conference, MDM 2003* (*Lecture Notes in Computer Science*, vol. 2574). Springer: Berlin, 2003.
34. Dey AK, Salber D, Abowd GD. A context-based infrastructure for smart environments. *Proceedings of the 1st International Workshop on Managing Interactions in Smart Environments (MANSE '99)*, 1999; 114–128.
35. Hull R, Neaves P, Bedford-Roberts J. Towards situated computing. *Proceedings of the International Symposium on Wearable Computers*, 1997; 146–153.
36. Kermarrec A-M, Couderc P, Banatre M. Introducing contextual objects in an adaptive framework for wide-area global computing. *Proceedings of the 8th ACM SIGOPS European Workshop*, September 1998; 229–236.
37. Jang S-I, Kim J-H, Ramakrishna RS. Framework for building mobile context-aware applications. *Proceedings of the 1st International Conference on The Human Society and the Internet—Internet Related Socio-Economic Issues Citation*, 4–6 July 2001.
38. Armstrong E *et al.* The J2EE. 1.4 Tutorial. *Sun Microsystems, Inc.*, 16 November 2003.

