Context-Aware State Management for Supporting Mobility in a Pervasive Environment

by

SIU Po Lam Pauline

A thesis submitted to

The University of Hong Kong

in fulfillment of the

thesis requirement for the degree of

Master of of Philosophy

August, 2004

Declaration

I declare that this thesis represents my own work, except where due acknowledgement is made, and that is has not been previously included in a thesis, dissertation or report submitted to this University or to any other institution for a degree, diploma, or other qualification.

SIU Po Lam Pauline

August, 2004

Acknowledgements

First of all, I would like to express my uttermost gratitude to all the people who supported me, directly or indirectly, during my research period.

I would like to thank my supervisors, Dr. Cho-Li Wang and Dr. Francis Chi-Moon Lau, for their valuable guidance and patience. They have given me stimulating ideas and constructive comments on my research. I want to thank also my project partner, Miss Laurel Kong, for her precious comments in all the discussions.

My sincere thanks should be extended to all my friends, who shared both my happy and difficult moments. They have encouraged me with enthusiasm.

Special thanks go to the Department of Computer Science, The University of Hong Kong, for providing me a nice environment during my study. I want to thank all technical staff and administrative staff, for their kindness. I must thank all the members of the Systems Research Group, for sharing with me their research work. They made me enjoy doing research.

Of course, I am profoundly indebted to my family, my father, my mother and my sister, Peggy, for their constant support and everlasting love.

Last but not the least, I would like to thank all whom I love and all who love me. Without them, this thesis would not have been possible.

Thank you.

Abstract of thesis entitled

"Context-Aware State Management for Supporting Mobility in a Pervasive Environment"

submitted by

SIU Po Lam Pauline

for the degree of Master of Philosophy

at The University of Hong Kong

in August 2004

Mobility and invisibility are the intrinsic characteristics of the pervasive computing environment. In such an environment, users are of high mobility, who roam around and access various services through resources that are available to them on the spot. These resources are heterogeneous, embedded in their surroundings, and vary from location to location. Therefore computing services should be able to adapt to different computing contexts, and it is essential for the system to support mobility and make the changing context invisible, so that users can continuously interact with the available services anytime, anywhere, and without disruption.

Researchers have worked on mobility support for many years. Mobility has been considered in many aspects, such as user mobility, device mobility, and service mobility. However, little research has so far been conducted on the issue of adaptation. There are some researchers who have worked on adaptation methods, such as data adaptation, code adaptation and functionality adaptation. But they have rarely considered the issue of mobility.

Research into mobility or adaptation alone is unlikely to achieve the goals of supporting mobility and achieving invisibility in the pervasive computing environment. Current mobility support would move an application by shipping its code with existing states. On the other hand, adaptability support is mostly concerned with the change of data or code, but very little with states. The state, i.e. the current status of the application, actually plays an important role in both mobility and adaptability, and it binds the two together.

In order to support mobility, capturing and restoring the state of an application is necessary. Instead of just transferring and restoring the state as it is, manipulation of the state could be done in between. The manipulation of the state becomes our fundamental approach to binding mobility and adaptability. To achieve better adaptability, context-awareness has to be brought into the picture. Context-aware state management is therefore needed to capture, process, transfer, and restore the state from source to destination.

We demonstrate our approach to supporting mobility and achieving invisibility through the design and implementation of a mobile functionality system. This system is integrated with our previous work on Sparkle, a component-based software architecture based on functionality adaptation. Our mobile system supports lightweight mobility through shipping functionality specification and the state of an application. Another key feature of the mobile system is the manipulation of states to achieve flexible adaptability. States are divided into three categories, namely, fixed, mutable and disposable. Based on this categorization and our proactive context reasoning rules, states can be adjusted to suit the future environment. We demonstrate the feasibility of our approach through the implementation of a universal browser application under Sparkle.

(440 words)

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Third Wave of Computing

In the beginning, computers began their existence as huge devices; there were mainly supercomputers, mainframes, and minicomputers. Computing power was centralized in these large and disconnected computing machines. This marked the first wave of computing. In this mainframe era, computers were rare and expensive, one computer could only be shared among many people.

Later, in the 1980s, the second wave came. It moved towards decentralizing information technology. Personal computers became the dominant computing paradigm. The cost of computers was getting lower and lower. Thus, in this evolution, people afforded to have their own personal computers, and there was no need to share computers with others. This is the era of personal computing.

Now, we are moving into the third wave, which will take us to pervasive computing, sometimes also known as ubiquitous computing. Unlike prior waves, this new generation is characterized by manifold pervasive computing devices. Computing devices are becoming ubiquitous, and no longer are limited to standalone computers. They could be personal digital assistants (PDAs), phones, watches, even wallets. Most people will own two or more different devices. These computers could even communicate with each other, forming a computing network, and enable people to

exchange and retrieve information anytime and anywhere. Now, computer becomes an inevitable component in our daily life.

1.2 New Needs for Mobility Support

In this era of pervasive computing, the ultimate goal is to create an environment that is fully embedded with computing devices, completely network-connected, and incessantly available. Pervasive environment is of high mobility and invisibility. For mobility, users always move from one location to another. For invisibility, devices are embedded everywhere. Users are distraction-free in using these resources to access services. Hence, the highlight is to facilitate the use of computing service continuously anywhere, anytime. Not only people can carry their devices from place to place, they can also perform their work seamlessly. As they move across environments, they still can keep on their computing tasks without any distraction. Services should be provided and adapted to the new environment.

Mobility, therefore, is the one of the vital requirements to achieve the goal. The goal of mobility allows services logically "follow" the user during execution independent of the physical location of the user. Hence, user can access service seamlessly anywhere and anytime. Researchers have delved into this field for many years, striving for device mobility, user mobility and service mobility. Device mobility means user can access the application with the same mobile device wherever he goes. When device mobility is achieved, a user can bring along his device for information access. However, the user cannot perform the same work with another device. User mobility is a kind of mobility that supports a uniform view of application regardless of where a user may be roaming. A user can access the application with different kinds of devices. Nevertheless, personal information could not be accessed in another device. Lastly, service mobility allows user to access services continuously wherever he goes, and whichever device he uses. With service mobility, a user can obtain his personalized services even if connected to different devices or in different

locations. Nonetheless, even though service mobility is achieved, the user may not be contented and satisfied just because he could access the application whenever he wants. Service mobility does not guaranteed applications could be compatible with and performed smoothly on the target device. Hence, adaptability is essential for achieving better mobility with better utilization of device. Thus, these mobilities are still not enough in the pervasive environment because they mainly emphasizing on migrating between heterogeneous devices, without adaptability.

In order to achieve mobility, migration is one of the strategies which transfer codes or resources between devices. Most of these migration strategies are based on mobile code paradigms. Mobile code paradigms, to certain extend, are nice to support mobility in mobile environment. However, devices are heterogeneous in pervasive environment. Traditional mobile code paradigms are not enough towards the direction of device heterogeneity. Therefore, we aim at designing a system based on a modified mobile code paradigm to support mobility in pervasive computing environment. The requirements of the system are:

- **Lightweight Migration.** Lightweight migration should be needed to support mobility. When migrates, necessary states are transferred between devices for application resumption. The amount of these states should be kept minimum so as to reduce overhead during migration.

- **Flexible Adaptability.** Flexible adaptability refers to the capability of the system to adapt to changes dynamically, and the capability to prepare for the adaptation. It should be flexible in the sense that adaptation could be done on either the source or the destination device. With flexible adaptability, invisibility could be achieved. More on adaptability support needed in pervasive computing is discussed in next section.

1.3 New Needs for Adaptability Support

In addition to mobility, ***adaptability*** is another vital requirement to achieve invisibility and support mobility in a pervasive computing world. It is the ability to adjust or be adjusted to suit the current circumstance and fully utilize the devices. Adaptation is a spontaneous or planned way to achieve adaptability. Mobility occurs and context changes may provoke adaptation. However, to achieve a meaningful adaptation, a system needs to analyze the current context and make suitable regulations on the application to adapt to the environment. In order words, a context-aware system with suitable adaptation is a way to achieve adaptability.

Recently, many different kinds of adaptation techniques have been proposed to deal with different contextual changes. Some of them are data adaptation, energy adaptation, and migration adaptation. Data adaptation is the changing of the application's data in some manner, for example, transforming the resolution of an image, so that user is able to continue his information access under different situations. Undoubtedly, these application data are pieces of content information, articles or graphics, accessed by the user. Data adaptation is commonly applied when the resources are limited, yet it tries its best effort to present the information. Energy adaptation is the technique to shutdown some unnecessary programs, or hibernate the device, so as to minimize the amount of energy used if necessary. Especially for mobile devices, batteries are always limited. Thus, energy conservation is important while keeping the current running program alive. For migration adaptation, it may try to move the execution to machines with more resources, for example, to achieve load-balancing. In pervasive environment, mobile devices are usually small resource-limited devices, connecting to other heterogeneous devices. Indisputably, some resource-consuming executions may not able to be performed on the current device. With migration adaptation, it could get assistant from nearby devices and migrate to these devices for application execution, while only displaying result to the end user. Hitherto, there are still lots of other researched adaptation techniques

we have not mentioned.

Although many adaptation techniques have been explored, functionality adaptation and state adaptation have seldom been mentioned. Systems usually apply adaptation methods when they realized the changes, few systems predict changes and prepare for adaptation beforehand. Hence, in our work, we will focus on these two issues for designing a system to support adaptability:

- **State Adaptation.** State adaptation involves changing the states of an application during migration. It gathers current context information and predicts the context changes. Thus, it could prepare for the adaptation on these changes. Especially on migration, it could analyze the context to decide if the adaptation should be done on the source or the target devices. State adaptation is more flexible than traditional adaptation strategies.

- **Functionality Adaptation.** Functionality adaptation involves changing the way a task is carried out in response to the changes in the mobile execution or context environment. Applications could have different implementations to carry out a specific task with different performance concerns. When application migrates and context changes, functionality adaptation allows the application to change its way according to the context environment.

1.4 State Management

In this research, we aim at providing service mobility so that user can carry out his work continuously, even roaming around. In particular, state management is constructive to achieve our goal. State management is a set of operations to manage states inside an application, which helps the application to adapt to the new environment when context changes. States are the current snapshots of the components that builds up an application, for example, the execution stack, or the content of variables, while contexts are the instantaneous environment status outside the

application but affects it, for example, the memory usage of the device, the current location, or the current weather information. States and contexts are closely related with each other as the current context affects how the application should present itself. For instance, the application should be power-save in a mobile device; or, under a slient atmosphere, the application should be mute. Through managing these states, application could be more adaptive to the current environment when context changes.

State management consists of four different processes: State Acquisition, State Manipulation, State Transmission and State Restoration. When the application migrates, states would undergo these four processes. At State Acquisition, all current information inside the application will be captured. All gathered information will be passed on to the next process – State Manipulation. As context changes from time to time, the current state may not be suitable to the application at the next moment. State Manipulation is a process to analyze the current captured state, with related context information, so as to predict the most suitable states for the application after migration. These related context information could be the context history, the current context and the predicted context. After the State Manipulation, states can be transmitted from the source device to the destination one, which is the State Transmission. Lastly, but not the least, the state will be undergo the State Restoration. The states will be restored on the destination device for the resumption of the application.

The most challenging part in this state management process is the State Manipulation. Since the context information is keep changing from time to time, we designed a system to support mobility of an application while keeping the state up with latest context information. In particular, we have introduced the categorization of states for the efficient of states processing.

1.5 Needs for Collaborating Mobility with Adaptability

As mentioned in previous sections, mobility and adaptability are two vital elements in the world of pervasive computing. Although both are important issues, researchers usually delve into either one direction. Some researchers may focus on investigating plenty of mobility mechanisms to achieve strong mobility transparently and efficiently on different levels, without taking the adaptability into account. Some may focus on exploring different adaptation methods to achieve adaptability, without considering how mobile the scheme is. Very few researchers emphasize their work on cooperating mobility and adaptability, complementing each other. The benefits of such a hybrid scheme are twofold.

First, in our approach, our adaptation methods will support our mobility. Our system will provide functionality adaptation, and different functionalities are downloaded according to the specification. Hence, when designing our mobility system, only state and specification are needed to be transferred. Codes are not necessary moved from the source to destination, and thus the amount of data transferred in our migration scheme can be reduced.

Second, in our mobility system, besides capturing and restoring the states to allow user carries out his task continuously, state processing is done before and/or after migration depending on the power of the client devices. This state processing prepares the state of an application for the future context so as to suit the destination environment. State adaptation, thus, can be applied in our mobility system to achieve better adaptability.

Hence, there is a need to incorporate mobility with adaptation to achieve lightweight migration with high adaptability.

1.6 Contributions

To our best knowledge, our mobile functionality system is the first system that only migrates states with functionality specification across network to achieve mobility. Besides, our functionality system differs from others by our state processing mechanism which provides context-awareness of states. Our work is based on the Sparkle, which is a component-based architecture. Originally, the Sparkle was introduced to support dynamic component composition and dynamic application reconfiguration. It enables application to be dynamically composed at run-time and reconfigured depending on the context changes. However, mobility support is not enough explorered in the Sparkle. We build a mobile functionality system on the Sparkle to achieve high adaptability via lightweight migration. The contribution of our work is listed as follows:

1. We have introduced a new way to incorporate mobility with adaptability. Mobility and adaptability are two indispensable issues in the pervasive computing world, they should not and could not be separated when designing a system to support applications in this environment. Therefore, we suggest building a system that allows mobility and adaptability complementing each other, in order to offer a better mobility and adaptability scheme.

2. With the incorporation of the Sparkle system, our mobile functionality system allows migration of applications from devices to devices. Through functionality adaptation, even big applications can be moved to and be continued on small thin clients, by downloading different implementations of the same functionality. Yet, with our data agreement in the container, the application can still be resumed after being migrated, even though the implementation of the application may be totally different.

3. We have also introduced the categorization of data state. For each running application, the intermittent data states can be grouped into three categories:

Fixed, *Mutable*, and *Deposable*. With categorization of data states, we can effectively process states during migration and allow state adaptation in our mobility system. Furthermore, depending on the categorization, we can achieve context-aware state management through keeping necessary data states, modifying them and dropping unnecessary one.

This thesis contributes mainly on the collaboration between mobility and adaptability, and how our mobile functionality system achieves lightweight mobility with context-aware state management and attains adaptability.

1.7 Synopsis

This dissertation focuses on how the mobile functionality system manages state with context-awareness, so as to achieve mobility support in the pervasive computing environment. It is organized as follows:

Chapter 2 gives a brief review on the literature on mobile agent systems, context-aware and adaptation systems and other background.

Chapter 3 gives our model in pervasive computing. It is then followed by a brief overview of the Sparkle architecture, including some important concepts and main entities in Sparkle.

Chapter 4 discusses about mobility and adaptability. This leads us to the needs and benefits of cooperating mobility and adaptability. It describes the approaches towards collaboration of mobility and adaptability.

Chapter 5 discusses the needs of context, the categorization and the representation of context. It is followed by our state management and categorization of state. After that, it shows our system based on context-aware rules for state management.

Chapter 6 describes our mobile functionality system which was built to support mobility in the pervasive world. It looks into the constituent entities of the mobile system and how they have been realized in implementation. After that, it shows how

to achieve context-awareness of state. It describes the stages we take in managing the state during migration.

Chapter 7 describes applications that realize our proposed architecture – the Universal Browser. It evaluates our mobile functionality system with context-aware state management.

Chapter 8 concludes the whole dissertation and describes our future work.

Chapter 2

Background Studies

In this chapter, we would have discussion on some basic concepts on mobile code, followed by some literature reviews in current mobile systems, adaptation systems and state management.

2.1 Concepts about Mobile Code

In this section, we will briefly describe the commonly used design paradigms in current computing environment identified by Fuggetta et. el. [12].

Client-Server

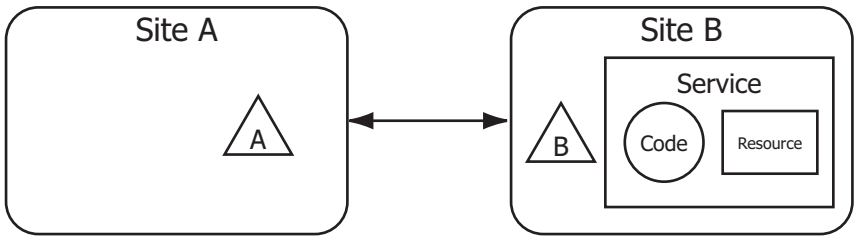

Figure 2.1: Client-Server paradigm

The client-server paradigm is most widely-used in present computing world. As shown in Fig. 2.1, both resources and code needed for providing the service are

placed at Site B. Component B, located at Site B, is responsible for providing the service. When the component A requests the execution of service, it interacts with Component B. By executing the corresponding code and accessing necessary resources, component B performs the service as requested by A. After execution, B will deliver the result of the service to A with an additional interaction.

Remote Evaluation (REV)

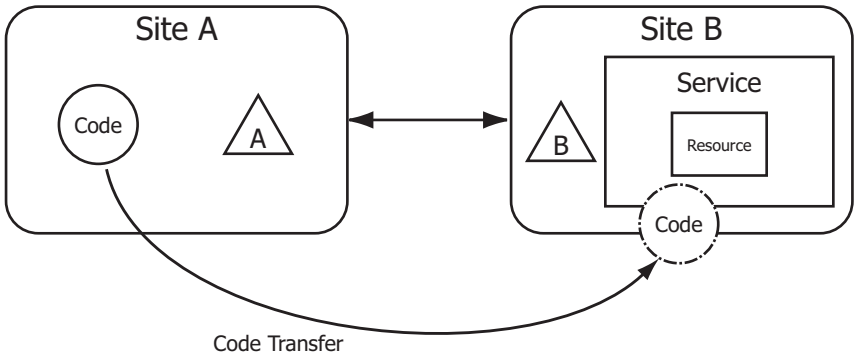

Figure 2.2: Remote Evaluation (REV) paradigm

Figure 2.2 shows the REV paradigm. In this paradigm, the code for performing the service is co-located with component A at Site A. However, component A does not have the required resources, which happen to be located at remote Site B. As a result, component A sends the code to component B, which is located at Site B, for the execution of service. Consequently, B executes the code with the necessary and available resources. With an additional interaction, component B will deliver the result back to A.

Code on Demand (COD)

As shown in Fig. 2.3, and comparing with REV paradigm (Fig. 2.2), the COD paradigm seems, in certain sense, to be the opposite of the REV paradigm. In this paradigm, resources that are needed by component A to execute the service are placed at Site A. Component A can access resources freely at Site A. However,

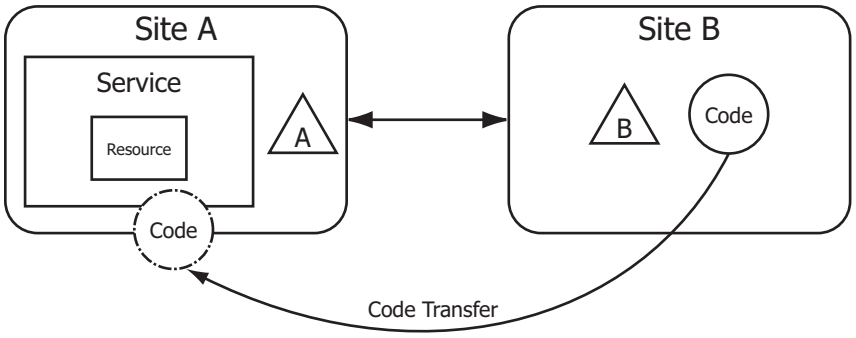

Figure 2.3: Code on Demand paradigm

component A does not have the code to perform the service. Component A, thus, requests the execution code from component B at remote Site B. Component B interacts with A, and delivers the corresponding code to A. Subsequently, component A can perform the service locally at Site A.

Mobile Agent (MA)

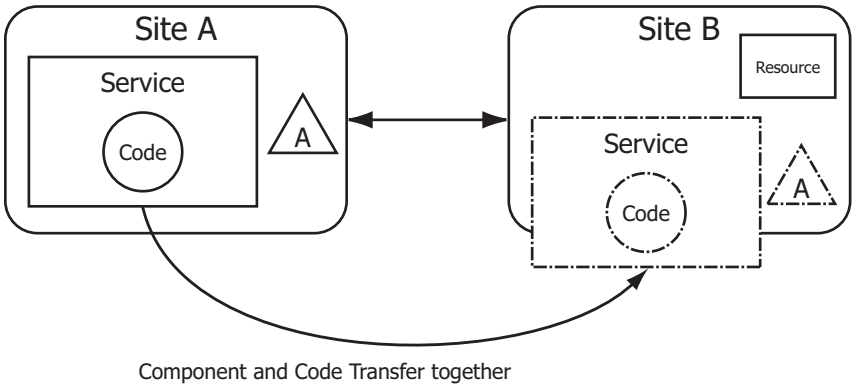

Figure 2.4: Mobile Agent paradigm

In the MA paradigm, the code required by the service is located at Site A. Component A owns the code at Site A, however, it does not have the necessary resources. These required resources are located at Site B. In this case, component A *migrates* to Site B, with its code and some intermediate execution states. After A has moved to Site B, it continues to execute the rest of the services using the

resources available there. Figure 2.4 shows Mobile Agent paradigm.

Furthermore, existing mobile agent systems offer two forms of mobility. Strong mobility allows migration of both the code and the state of an execution unit to a different computational environment. On the other hand, weak mobility allows code transfers across different environments; the code may be accompanied by some initialization data, but the execution state is not migrated.

Table 2.1 shows a summary of the mobile code paradigms concerning the transfer of data.

	Client-Server	REV	COD	MA (weak mobility)	MA (strong mobility)
Code	X	O	O	O	O
Data State	X	X	X	O	O
Execution State	X	X	X	X	O

Table 2.1: Summary of the mobile code paradigms: Circle (O) indicates the data is migrated under particular scheme and Cross (X) indicates the data is not migrated

2.2 Current systems applying mobile code

2.2.1 Java Aglets

Aglets, developed by IBM Tokyo Research Laboratory in Japan, are Java objects that can move from one host to another. They bring the program code and data state with them when they move across network. They extend Java with the support for weak mobility. The name "Aglet" means a combination of "Agent" and "Applet".

A running aglet can be stopped, dispatched to another host and resumed there. There are two ways in which aglets migrate, namely "dispatch" and "retract". For dispatch, the aglet moves to the destination with the code segment. For retract, the code segment is being fetched, so that the aglet is moved to the place which

fetches the code segment. No matter dispatch or retract, the aglet is re-executed from the beginning with some initial data state after migration. The Java Aglets API (J-AAPI) is a standard for interfacing aglets and their environment.

Our mobility system is very similar to aglet as our mobile unit also is re-executed with some data state after migration. Yet, our mobile unit does not carry code with it. For migration, it will only carry function specification and some data state. Besides, the data state is not transferred as it is. Data state is processed in our system before and after migration so as to incorporate mobility with adaptability.

2.2.2 Mole

Mole [28], developed at the University of Stuttgart, is also a Java object that supports weak mobility, which is also similar to the Aglets. Mole is an agent system that consists of a number of (abstract) places, which are the homes of various services. Agents are active entities, which may move from place to place to meet other agents and access the places' services. In the Mole model, agents may be multi-threaded entities, whose state and code is transferred to the new place when agent migration takes place. Places provide the environment for safely executing local as well as visiting agents.

Weak migration in Mole is implemented by using Remote Procedure Call (RPC) / Remote Method Invocation (RMI) and Java object serialization. RPC/RMI is used to transfer the control flow from the caller, which is the source, to the called, which is the destination, until the request is served and the result returned. Messages, while this is not the message passing between objects, are used to transfer data (objects) between agents. Messages use Java object serialization to move objects between agents.

In our system, we also make use of Java object serialization to transfer data between hosts. However, our migration method is not based on RPC/RMI, instead transferring the code to the destination. This is more suitable in the sense that

disconnection operation could be achieved. Again, Mole focuses only on mobility, in contrast to our system, which focuses on both mobility and adaptability.

2.2.3 Sumatra

Sumatra [3] is an extension of Java. It supports resource-aware mobile applications. Sumatra supports resource-awareness through three mechanisms. First, they monitor the level and quality of system resources. If there are any changes, they respond to these changes. Lastly, they adjust the libraries and other code continuously, to allow adaptation on these new updates.

For Sumatra, it supports migration of objects and threads. It achieves strong mobility through migration and remote cloning. On the other hand, it also supports weak mobility by using Java and RMI.

The main difference between Sumatra and our system is the support of strong mobility. In our system, strong mobility is not supported. Therefore, migration of threads is not possible. However, migration of threads is not quite suitable in our system as application may be reconfigured after migration. Hence, some previous threads may not be needed at the destination.

2.2.4 JADE

JADE (Java Agent DEvelopment Framework) [1] is a software framework fully implemented in Java language. It simplifies the implementation of multi-agent systems through a middleware that complies with the FIPA specifications and through a set of graphical tools that support the debugging and deployment phases. The agent platform can be distributed across heterogeneous machines and the configuration can be controlled via a remote GUI. The configuration can even be changed at run-time by moving agents from one machine to another one, as and when required.

JADE is a mobile agent system. On the contrary, our mobile system is not a mobile agent system. Our system does not provide autonomous feature to our

mobile unit, which means the mobile unit could not make any decision on where it moves. Our client system, including the mobile system, has full control on the mobile unit.

2.2.5 LMCS

In our previous work, Sparkle, the Lightweight Mobile Code System (LMCS) [8] is built for the pervasive environment. This LMCS, with the cooperation of Lightweight Mobile Agents (LMAs), supports lightweight and flexible computations in the pervasive world. It is a Java-based system with source code instrumentation scheme. Thus, the mobile agents are able to strongly migrate between devices without program restarting after migration. Besides supporting strong migration, the LMCS has implemented two on-demand schemes. The code-on-demand scheme enables the LMA to retrieve code from the source when needed, while state-on-demand scheme enables the LMA to retrieve execution state from the source when needed. These two on-demand schemes claim to help to save memory resources and network bandwidth usage.

Unfortunately, the state-on-demand scheme has some limitations. It suggests that the execution state can be dynamically downloaded from the original device to the destination device as needed. One of the main features of this state-on-demand scheme is that the state can be located in a cascade way on different devices if the LMA moves to and from many different devices. On the other hand, Sparkle is functionality adaptable, it may download new implementation code for better execution. Hence, for current state-on-demand scheme, these states may not be suitable for different code implementation.

Moreover, LMCS mainly emphasizes on the strong mobility of application. It puts less concern on the adaptability needed when migration occurs. As a result, we would like to build a new mobile system which supports mobility in the pervasive world in a lightweight and flexible way.

2.3 Component-Based Adaptation Systems

In order to achieve the goals in pervasive computing, context-aware systems are needed. Owing to the dynamic nature of pervasive computing, systems should be able to capture, understand and adapt to the changes, this is the so-called context-aware system.

Adaptation system is one kind of context-aware systems. It allows user to continue with a degraded service, instead of letting user to wait for the original services to become available.

2.3.1 Perimorph

Perimorph [17] is a system that supports compositional adaptation. The authors of [17] suggested the transfer of non-transient state between old components and their replacements, which is similar to our design. However, they only focused on the concept of collateral change. In our system, we focus on state management and adaptation. Our system supports the transformation of state in order to adapt them to new facets.

2.3.2 Puppeteer

Puppeteer [9] is a component-based adaptation system for extending component-based applications to support adaptation in mobile environments. It could support adaptation without modifying the applications. Puppeteer performs data and control adaptations by repeatedly using the policies of subsetting and versioning. A subsetting policy renders parts of the original document like text, or the first-slide of a document. A versioning policy allows the choices among multiple instantiations of a component, such as instances of an image with different resolution.

The main difference between Puppeter and our system is the focus on adaptation methods. Puppeteer is a component-based adaptation on data adaptation, while Sparkle is a component-based adaptation on functionality adaptation.

2.3.3 DACIA

The DACIA project [22], at the University of Michigan, provides a framework for building adaptive distributed applications. Similar to Sparkle, applications are made of components located on various network entities. However, in DACIA, these components are considered as a unit for implementing data streaming, processing and filtering functions. DACIA provides mechanisms for run-time reconfiguration of applications. Therefore, applications can be adapted to the environment through adjusting the connection between components. Besides, components could also be moved to another host through maintaining the connection between components.

Since in DACIA, new components are introduced along data paths, this framework seems more suited for distributed application which requires data flow from one entity to another, for example video-on-demand. This is, on the other hand, not the limitation in Sparkle.

2.4 State Management

2.4.1 ROAM

ROAM [15] is a seamless application framework that aims for distributed applications. It assists developers to build multi-platform applications that can run on heterogeneous devices. This framework also allows users to move running applications between heterogeneous devices. In ROAM, the user interface (presentation) is device-specific. During migration, it is necessary to save all states and map running states between the source presentation and the target presentation. Hence, extra time is needed to transform states between different GUI components. In our system, we categorize and process data states before migration, and less data are kept and transferred.

2.4.2 ISAM

ISAM [16], is a software architecture for adaptive and distributed mobile applications. It provides support for developing and executing adaptive distributed mobile applications. It supports adaptation through different levels at load-time and at run-time. Their system and the application collaborate to adaptation decision to allows flexibility. Their ideas are quite similar to ours as they would like to collaborate adaptation with the system. In contrast, their adaptations are mainly focuses on reallocate, reschedule and restructure the applications. Our works on adaptation are focuses functionality adaptation and state adaptation.

2.5 Summary

In this chapter, some background concepts of mobile code are described. There are four mobile code paradigms: Client-Server, Remote Evaluation, Code on Demand and Mobile Agent. These four mobile paradigms are commonly used in current systems for support mobility. Some related projects designing mobile systems, adaptation systems and state management systems are then discussed. Most of these projects are tackling one specific area, either mobility support, adaptation support, or state management. Very few systems would give a hybrid solution to support mobility with adaptation through state management.

Chapter 3

Mobility and Adaptability

In this chapter, we would focus on the two important issues in the pervasive computing environment: *Mobility* and *Adaptability*. First, we would like to discuss the two commonly used term: "state" and "context". It then follows by the discussion on mobility and adaptability in the current literature. Lastly, we would like to discuss our paradigm design, mobile functionality, in order to achieve our mobility support incorporate with adaptability.

3.1 Some background concepts

3.1.1 What is "context"?

Recently, many researchers have given plenty of different definitions on *context*. Schilit and Theimer [26], who first introduced context, refer context as a location, identities of nearby people and object, and changes to those objects. Later, Dey [10] stated that context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves. Many other researches have also provided synonyms for context sharing similar ideas.

We define "context", sharing similar idea with Dey, as information that to iden-

tity the current situation of an entity. However, we restrict context as pieces of information that are not inside the running application, but closely related to the application. For example, the location, the weather, or the device hardware status. Context does not directly related to the running application. As the context changes, the application could still serve the user with correct result without any adaptation to current context. Applications, however, could be adapted to the new context for better performance and better user's satisfaction.

3.1.2 What is "state"?

State is a term widely used to describe the current status of an entity. In pervasive computing, a state could be the current status of the environment, the person or the device. This idea shares similar ideas with "context". In our discussion on states, we would restrict states as pieces of information that are directly affect the running application.

In order to support mobility in pervasive computing, we deepen our studies to the state inside a running application. We define state as the instantaneous snapshot of an application. In such a case, states can be divided into different categories. Generally speaking, it can be classified into data states and execution states. Data states mean the current status of the data space. The data space is the variables or objects, which can be accessed by running entities. On the other hand, execution states hold some private data about the current execution. For example, program counter and frame stack.

States are absolutely important to an application as it controls the correctness of the program flows and the output. The application may misbehave if states are not correct. Thus, we still need a management scheme to manage these states for mobility support. We will discuss our management schemes in the next chapter.

3.2 Mobility

In the pervasive computing environment, ***mobility*** is an essential element to enable users continuously carry out their tasks from place to place. With mobility support, user can perform their computation seamlessly.

Nevertheless, many researchers explored mobility in different ways, thus, the word "mobility" diverts to two different meanings:

- Mobility is the "goal" we would like to achieve. For example, we would like to achieve user mobility, that means the application could "follow" the user whenever user moves.
- Mobility is the "method" we apply to achieve the "goal". For example, we could apply code mobility to achieve user mobility.

The types of mobility we would like to achieve depend on situations. For example, in mobile computing, a user brings a device from one location to another. System can achieve device mobility via connecting to different access points.

In pervasive computing, user is of high mobility, and even the system could not know exactly where the user will move in the next moment. Hence, mobility support should have a great difference from other mobilities introduced in mobile computing. Therefore, we aim at supporting service mobility which incoparated with adaptability.

Moreover, in order to support mobility, the general "method" used is mobile code. The introduction of mobile codes, undoubtedly, facilitates mobility in the pervasive world. However, we would like to introduce our design paradigm, mobile functionality, to support our service mobility for pervasive environment. Before further our discussion on mobile functionality, we first have a look on some mobility supports applying the current mobile code paradigms.

Device Mobility

Device mobility allows an application to "follow" the device. Even when the devices are connected to different access points, applications could still be accessed continuously with the connection to nearby access points dynamically. A user is able to enjoy his application access when he brings his mobile device with him. This kind of mobility is usually addressed in mobile computing with wireless networks. Certainly, this kind of mobility is not enough in pervasive computing as devices are faded out and becoming embedded to the environment. Users are no longer bringing mobile devices with them, thus, users need another way to access their applications.

User Mobility

User mobility is a kind of mobility that supports a uniform view of a user's work independent of his location. User could continue his work if he brings along his device or even if he changes his device.

Service Mobility

Service mobility allows user to carry on his services while moving or changing devices and network service providers [27]. It is the ability to allow service to "follow" the user.

3.2.1 Short Discussion on Mobility

The mentioned types of mobility are not enough in the pervasive environment as they mainly focuses on mobility only. However, in pervasive computing, mobility and invisibility are correlated. Hence, we need a correlated support to divert us towards the goal of pervasive computing.

Generally speaking, the traditional paradigms, such as client-server, are static with respect to code and location. They are not flexible in the sense that the component cannot change their location or their code after their creation. In contrast,

mobile code paradigms provide such a flexibility as they allow components to change their location dynamically. As mentioned before, the REV and MA paradigms allow the component to execute code on a remote site, while the COD paradigm enables components to retrieve code from other components at remote sites.

The mobile code paradigm provides certain extents of flexibility, which is allowing component or code to be moved. However, such a flexibility is not enough in the pervasive computing environment. Owing to the nature of constant changing of resources in the pervasive world, most of the current design paradigms are not flexible enough as they seldom consider the collaboration with adaptability.

Considering the following example, which is inspired by the chocolate cake making examples by Fuggetta et. el. [12]:

Louise wants to prepare a chocolate cake. She knows the recipe but she has at home neither the required ingredients nor an oven. Her friend Christine tells Louise that she has both at her place, yet she doesn't know how to make a chocolate cake. Louise then goes to Christine's home. However, after reaching Christine's home, Louise realizes the ingredients, which are the resources, are not enough. Then, what should Louise do?

In the above example, resources changes when Louise arrived. As in the pervasive computing environment, the context environment always changes. Although code mobility allows moving code from one site to another site, this does not ensure the code is suitable at the target site. Therefore, we introduce a new design paradigm, taking adaptability into account, going towards a hybrid approach of the mobile code paradigms, and gaining benefits from Code on Demand and Mobile Agent.

Before going into the discussion of our design paradigm, we will first briefly introduce some background on adaptability. Later, more discussion on our design paradigm will be in section 3.4.2.

3.3 Adaptability

Adaptability is another essential element to enable users continue their tasks without distraction. It means we would like to ensure the stability of an application in the face of a changing environment. On the other hand, *adaptation* is the way to achieve adaptability. It is required to overcome the intrinsic dynamic nature of pervasive computing. Adaptation can make provision in response to context modifications, through changing one dimension of parameters [4].

Presently, adaptability can be achieved by many taxonomies of adaptation. Hence, we would first have a look on some commonly used adaptation methods based on our categorization in nowadays computing world. We, thus, would like to examine the possible extensions from existing adaptation techniques, to cooperate with mobility, so as to facilitate mobility and to gain some benefits from mobility in achieving better adaptability.

3.3.1 Some common adaptation methods

Different researchers classify adaptation differently owing to the diverse approaches of adaptations; therefore, there is no standardization in grouping adaptations. Some researchers may categorize them into data and control adaptation [9]. Some researchers may classify them into application-aware and application-transparent adaptations [24]. Others may categorize them into data, network, energy, migration and functionality adaptations [4]. As we are aiming to collaborate adaptability with mobility, we categorize the adaptation methods according to the structure of an application. Hence, they are grouped into data adaptation, state adaptation, code adaptation, and functionality adaptation.

Data Adaptation

User may access different kinds of data while using applications. Data, here, means information or content that user accesses, such as emails and web pages. Data

adaptation is a method that transforms the data of an application in order to meet the client context, for example, retrieving different sets of data depending on context, rearranging the data to a more appropriate format, or changing the fidelity of images to a proper resolution. Content adaptation, which applies data adaptation techniques, addresses more on the layout of data, such as the pagination of data. Currently, a number of projects focus on data adaptation. For example, [20] [23] address the content adaptation.

State Adaptation

State adaptation is the way to change the internal state of an application in order to meet the context. This is different from data adaptation in two ways. First, state adaptation supports adaptation during migration, while data adaptation only supports adaptation after migration. Second, state adaptation addresses the internal variable state (including data state or execution state) of an application, while data adaptation focuses on the layout of data. For example, if a device does not have sufficient memory, some of the unnecessary state can be discarded. However, few researchers focus on this adaptation. More on state adaptation will be discussed in section 3.4.4.

Code Adaptation

In a pervasive world, especially we are concerning a mobile code environment, codes may be moved to any devices for execution. Code adaptation is, thus, a way to change the code of an application so as to fit the client environment. For example, adding new parameters, instrumenting the code, removing unnecessary code, or reconstructing the code for the context. It modifies the code based on the existing one.

Functionality Adaptation

Functionality adaptation involves changing the way the task is carried out in order to respond to the changes [4]. It selects different code implementations for execution depending on context. Functionality adaptation differs from code adaptation as it downloads new codes when context changes, while code adaptation transforms existing one to new one. For example, if a device does not have sufficient computation power, an application can execute another implementation of encryption using a smaller key. However, few researchers focus on this adaptation. More on functionality adaptation will be discussed in section 3.4.3.

3.3.2 Short Discussion on Adaptability

In general, adaptability is required whenever there is a change of context. Especially in a pervasive environment, user may roam around and change devices. As Katz pointed out, mobility *requires* adaptability [18]. Traditional adaptation, such as data adaptation, could certainly achieve certain kind of adaptability, as they in general provide support for adaptation after mobility. In other words, adaptability is considered as the follow-up process after mobility, they only adapt to the context after they realize the changes.

Nevertheless, in pervasive world, mobility is as important as adaptability. Hence, instead of treating adaptability as the follow-up process, it should be cooperate with mobility. In other words, adaptability should be required during migration time. With this idea, the traditional adaptation methods are not enough in the sense that they give few support during mobility. They could not prepare for the context change. For example, data adaptation can modify the application data after the changed context or mobility, yet the data would not be modified during mobility. On the other hand, for functionality adaptation and state adaptation, although few researchers have delved into these two dimensions, we strongly believe these two kinds of adaptation could support and be supported by mobility. Therefore, in the

following section, we will try to collaborate mobility and adaptability with these two kinds of adaptation.

3.4 Collaborating Mobility and Adaptability

Mobility and adaptability are of vital importance in the pervasive world. Despite this, researchers usually delve into either one direction. Some researchers may focus on investigating mobility mechanisms to achieve strong transparent mobility or efficient lightweight mobility on different levels, without taking the adaptability into account. Others may focus on exploring different adaptation techniques to achieve flexible adaptability, without considering how the mobility scheme is. Only very few researchers would emphasize their work on cooperating mobility and adaptability, by complementing each other.

In our approach, we will focus on cooperating these two issues to achieve lightweight mobility and flexible adaptability:

- Functionality adaptation allows lightweight mobility. With functionality adaptation, only the specification of functionality are needed to be transferred to the destination during migration. As a result, no code implementation is transferred. Migration could be lightweight.

- Mobile functionality enables flexible state adaptation. With mobile functionality, it is flexible to choose the point for state adaptation. The adaptation could be done on the source site or on the target site depending on the context information, for example, by comparing the processing power of the source and destination device.

- In the collaboration of mobility and adaptability, we could consider adaptation before migration with the current context. Hence, preparation of state could be done on the source site, and resources could be saved on destination site.

In this section, we first discuss how adaptability and mobility could complement each other. We will then discuss the three main techniques to achieve adaptability and mobility.

3.4.1 Approach

Researchers have put much effort in exploring mobile code which provides flexibility to the system, yet mobile code does not facilitate adaptablity. In general, mobile code concerns the moving of code, the mobile agent system would transfer state and code. However, mobile code is not enough to provide a lightweight migration.

Adaptability supporting mobility

As mentioned before, functionality adaptability allows us to change the implementation of code in response to context change. Since the Sparkle provides functionality adaptability, it enables the application to be dynamically reconfigured after migration. Owing to this characteristic, our system should also take this advantage to enhance the mobility in twofold. First, when mobility occurs, even the application is the same, different program code could be loaded in for execution. Thus, it is not necessary for us to transfer the code from the source to the destination due to the context variation. With the support of functionality adaptability, our mobility can be lightweight in the sense that only application specification is transferred for execution continuation.

Second, adaptability not only facilitates mobility, it also provides flexibility during migration. If the connection bandwidth is very limited from the source to the destination, in a traditional mobile code system, all codes as well as the execution states are needed to be transmitted through this link. On the other hand, in our approach, we only transfer the application specification, including the function specification and data agreement, through this limited bandwidth. Hence, network bandwidth could be saved. Even though if new implementations are needed to

be downloaded, we are flexible to choose the download location which maintains a better bandwidth connection.

Mobility supporting adaptability

Mobility is always needed in pervasive computing, and adaptability usually is required after mobility. However, adaptability should not just "follow" mobility. During migration, adaptability should still fully exploit the advantages of mobility. There are mainly two advantages to allow adaptability. First, it promotes state adaptability. Generally, during migration, state capturing, transmission and restoration are essential processes. In the meantime, we could process the application state for adaptation. After proper adjustment of the state, some states could be changed or some unnecessary one could be disposed. Hence, the amount of state needed to be transferred is reduced.

Second, adaptability could be more flexible. Since we address state management and adaptation on migration, this allows adaptability to occur either in the source or the destination device. For instance, if the resources are rich in source device and that are poor in destination device; state adaptation can be done on the source device, so it would not overload the destination device. This makes the state adaptation more flexible and more "adaptable" in the pervasive computing environment.

3.4.2 Mobile functionality

In our architecture, one of the key features is the ability to move application between clients. Precisely, our mobility support is a kind of weak mobility that focuses on transferring functionality specification between clients instead of transferring program codes. Figure 3.1 depicts the mechanism of mobile functionality.

As mentioned in the previous chapters, Sparkle is a component-based model providing dynamic component composition. This uniqueness of our architecture also

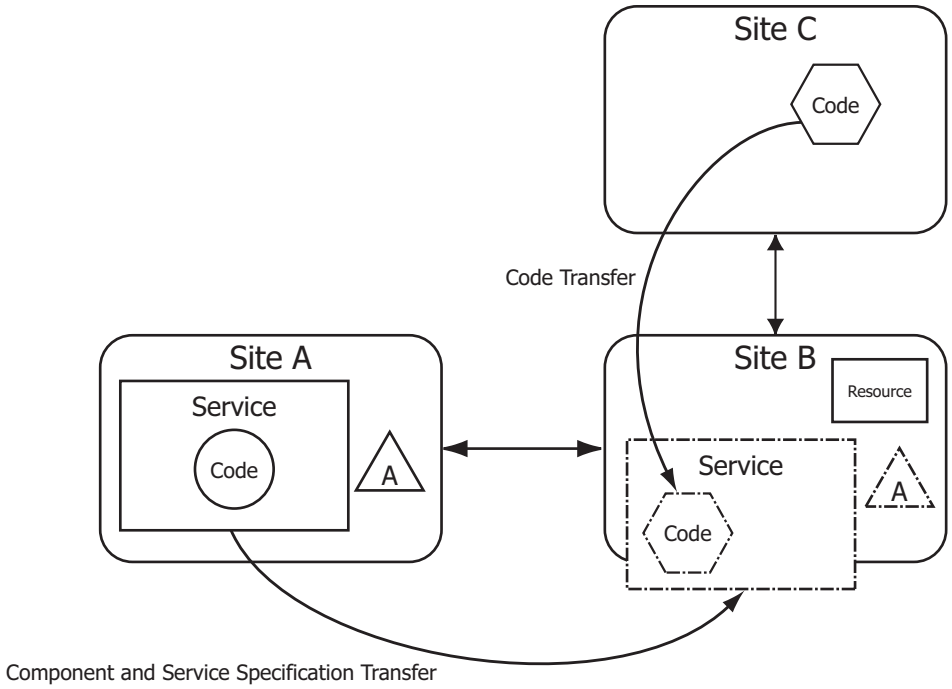

Figure 3.1: Mobile Functionality paradigm showing when the application is migrated from Site A to Site B, only component and service specification are transferred between these two sites, the code could be downloaded from another site, Site C, when needed

enables us to support mobility via transferring functionality specifications instead of transferring program codes between clients. The advantages of our scheme are twofold. First, current developed mobility systems usually support mobile application through moving states and/or program codes. This introduces a difficult problem if the resources available at the destination are very limited. Since the target device has limited resources, current code implementation may not fit into the target device, or could not be run on the target device. Hence, the application could not be able to be run on the destination. In order to solve this problem, our mobile functionality system is integrated with Sparkle system to enable dynamic application reconfiguration. Whenever location changes, application could be rebuilt via downloading suitable facets from peers or proxies. New facets of totally

different implementation could be downloaded to make up the application providing the same functionality. Our approach is different from most of the current developed mobility systems which only support moving same piece of program code from one place to another. With our architecture, application could be rebuilt with more suitable facets. Hence, better utilization of computer resources is possible. Besides, when you move your application from a resource-limited device to a device with rich resource. You may want your task to be completed in a faster way through using an algorithm which spends more spaces but runs faster. Especially in the pervasive environment, users always connect from various places, using a variety of devices having a wide range of resources. Our mobile functionality system supporting dynamic application reconfiguration could be more adaptive to the context changes.

Second, users can continue their tasks even when the application is dynamically reconfigured. Due to the characteristic of high mobility in the pervasive computing environment, mobile users always wish to be able to perform computation wherever they go continuously. The Sparkle integrated with our mobile functionality system could serve this purpose to enable user continuously do their tasks from any point.

In our architecture, we achieve mobility by moving functionality specification from one location to another. This ensures applications can be freely migrated between heterogeneous devices, as dynamic application reconfiguration is possible. In order to let users continue their tasks, we also capture the states from the source device to the target one. However, problem exists when there is a change in the implementations of facet as we need to ensure the compatibility of states between different implementation of facets.

Sparkle provides mobility at the application level. Unlike tradition mobile code systems, we aim at moving functionalities from one device to another. The key benefit of our scheme is to move the least amount of application states during migration. In our migration scheme, we would not transfer the execution state of

an application (e.g. program count, stack and registers content, thread status). The main reason is that it is not useful for us to keep all these execution states as new code might be downloaded at the destination. The execution state could not be compatible as the code is totally different. Moreover, we aim at reducing the overheads of capturing the data and restoring it at the target. Hence, we only capture the current data state in the container to support the continuation of user's task.

For state capturing and restoring, these are automated by our system. In Sparkle, the system is responsible to save the state from the container to our mobile unit. Thus, our mobile unit can be shipped to the destination for state restoration.

Furthermore, Sparkle would help the application to capture the variable value to the container from time to time, preparing for the migration. Nevertheless, in order to facilitate the application migration, application programmers could specify which variables are needed for migration and when the system should copy the variable value to the container.

The mobility functionality mechanism supports migration at a suitable facet level. The migration point should be made by the system. Thus, when migration is first initiated, our system would calculate the total resources needed by the current running facet. If the destination device has enough resources to continue with the current calling sequence, the system will start the migration at the current position. However, if the system realizes the destination device does not have enough resources for the continuation, it will wait for the completion of the current facet, and decide if the current situation is suitable for migration.

Comparison between mobile code and mobile functionality

Our mobile functionality system is based on mobile agent and code on demand paradigms. However, it differs from the goals of most mobile agent systems in several issues. First, unlike mobile agent, our mobile units are not designed to be

autonomous. In contrast, our mobile units will be migrated as a result of commands issued by our client system. This difference in design makes the mobile units more controllable. Second, generally mobile agent needs to ship state and code to the destination, or just the mobile code without state. On the other hand, we only transfer the specification of application and some necessary application states, which is considerably lightweight when compared with the transfer of whole application with all states.

Another key difference is the mode of code transfer. Usually, the current mobile code systems only transfer code between the source to the destination. In our approach, the code transfer might not be between the source to the destination, while it might be from another third party. This difference makes the download of code more flexible, as it can be done from a site with better internet connection.

3.4.3 Functionality Adaptation

Dynamic component composition provides a flexible mechanism for achieving functionality adaptation. Functionality adaptation is made possible by three factors. First, the component model, which separates applications neatly into components, and allows them to be composed at run-time and be discarded when no longer used. Second, the context managers in client devices, which maintain information about the physical resource, network connectivity and the context of devices. This information is included in a component request and is the basis on which matching occurs. Third, the proxies, which carry out matching between the request and facets. They are the main active entities for adaptation.

Functionality Adaptation, in this approach, is achieved by choosing the appropriate component among different ones which have the same functionality.

The main advantage of this approach is that it is very flexible and dynamic. Developers only need to specify which functionalities they require in an application, and provide different versions of them. The adaptation mechanism is transpar-

ent to the programmer. Which component comes in depends on the system and the matching mechanism of the proxy. In addition, since applications are linked by functionalities, rather than specific components, as new technologies or devices emerge, developers only need to write newer versions of the affected functionalities. The proxies will automatically match these components to suitable clients under the appropriate conditions. Rewriting or reinstallation of the whole program is not required. Also, since the components are thrown away at run-time after use, even the biggest programs can be used in a small device, depending on the size and the run-time behavior of each component. In short, this approach overcomes all the drawbacks in current functionality adaptation techniques.

3.4.4 State Adaptation

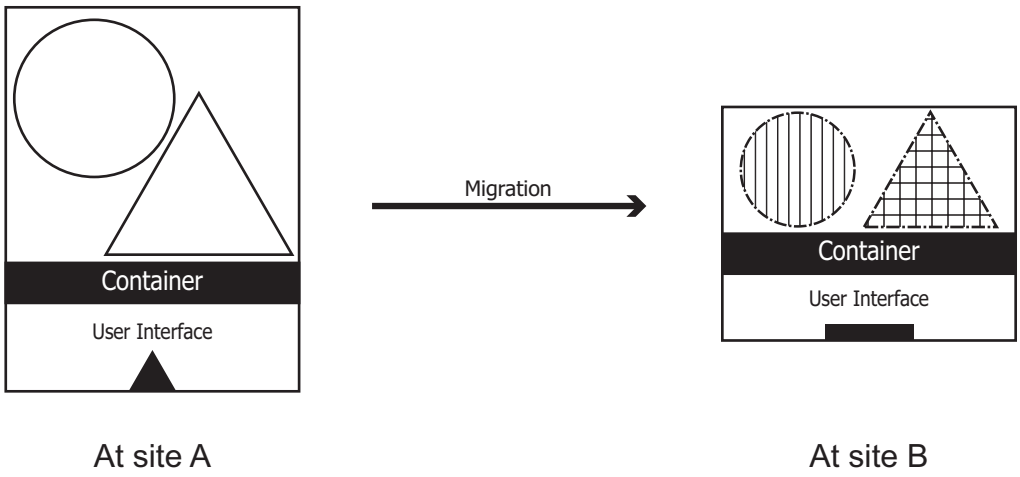

Figure 3.2: Functionality and State Adaptation. As the context changes from Site A to Site B, the codes, but not the functionalities, that built up the applications is changed (the circle and the triangle are kept the same shape but are squeezed) and the states are changed (the circle and the triangle are shadowed)

State adaptation is the manipulation of state to make the states suit the current environment, as shown in Fig 3.2. In our architecture, there are several elements in the client devices to make state adaptation possible. First, the data state man-

ager, which is responsible for capturing the state for migration. State capturing mechanism enables the retrieval of runtime internal data state, which is required for processing and migration. Second, the mobility manager, which is the core for state adaptation, will get the context from the source and the destination devices for state processing. Third, the categorization of state, which helps us to process the state systematically following some context-aware policy. More details on this context-aware state processing will be discussed in the next chapter.

The key benefit of state adaptation is the flexiblility. Depending on the context, the state adaptation can be happened at the source or the destination. The adaptation mechanism is transparent to the application. The mobility manager decides how the state will be adjusted according to the context of the source and destination. With the help of the categorization of state and context-aware policy, state could be modified to suit the destination context. In addition, the context-aware policy could be maintained separately from system implementation details. Therefore, system administrator, or application programmer, can modify these policies in a more easy way. Besides, since state is processed before migration, some unnecessary state could be disposed, the amount of data needed to be transferred and the bandwidth could be saved.

3.5 Scenario

Figure 3.3 shows a typical scenario in our system. At home, Fred is using a desktop computer. Whenever he needs a service, the system will download suitable functionality to fulfill his service. In this scenario, the system is going to download functions, namely f1, f2 and f3 which will be used sequentially. After finished using f1, the system throws it away as f1 will not be needed within a short period of time.

At this moment, Fred is using f2. In the meantime, Sparkle knows Fred will go to office soon, thus, Sparkle prepares for the migration of the service being used by Fred from the PC to the PDA. Sparkle first captures the specification of the functionality

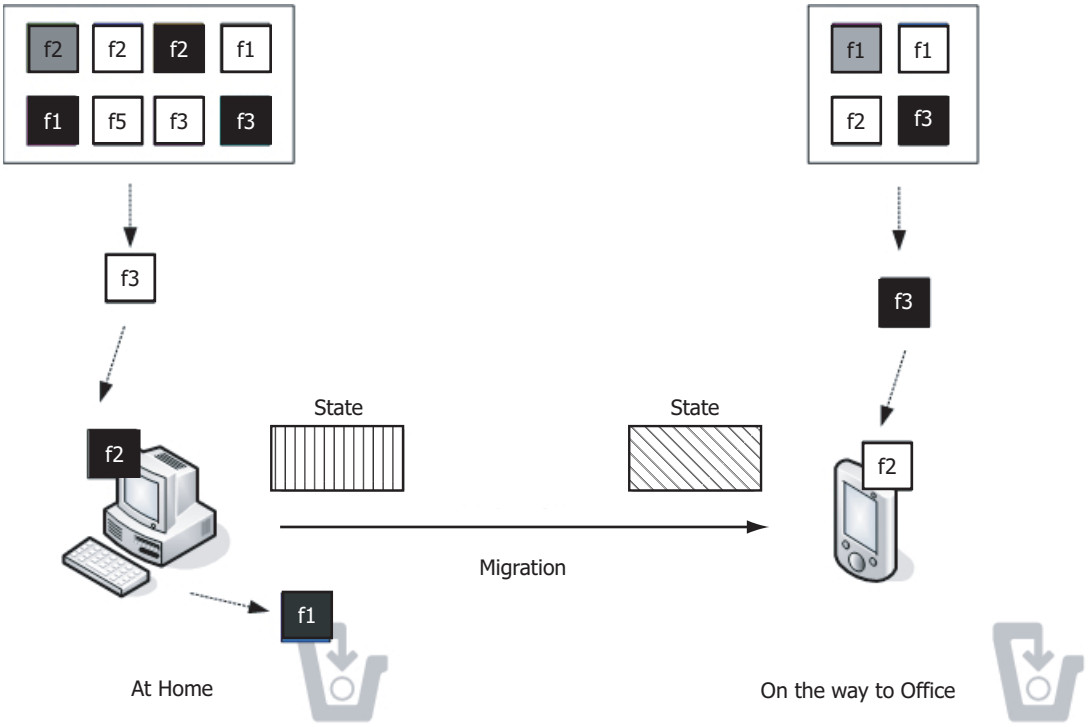

Figure 3.3: A scenario with mobile functionality and state adaptation

needed, that is f1, f2 and f3 in this scenario. Sparkle, then, captures the state of running f2. Sparkle mobility system manipulates the state of f2 depending on the context using the context-aware rules, as shown in the Figure the pattern of the state is modified.

At the destination site, the running state of f2 could be restored. Moreover, in order to continue the service, f2 should be loaded in. Since the context at destination is different from that at the source, different implementation of f2 should be downloaded for resumption. As shown in Fig. 3.3, the color of f2 has been changed as the implementation changed.

After downloading f2, the state can be restored to f2. The functionality f2 can be resumed. Fred can be continued with his work.

3.6 Summary

In this chapter, we first presented the meaning of "state". We, then, discussed the mobility and adaptability needed in pervasive computing. We pointed out that mobility and adaptability is correlated. Hence, this comes to our focus on the collaboration of mobility and adaptability.

Chapter 4

Sparkle Architecture

Before the discussion of the theory of our mobile functionality system supporting context-awareness, we introduce the Sparkle system. We briefly describe our defined model for pervasive computing environment. We, then, discussed the architecture of Sparkle, followed by the main components inside Sparkle and their interactions. These background could help better understanding the ideas of our mobile functionality system that will be described in later chapters.

4.1 Overview

4.1.1 Our Model

In our model, there are three main entities, namely person, device and environment.

Environment

An *environment* is the smart space. In an environment, it is full of resources, which can serve a person. These resources could be persons, devices, or other things. They may station in the environment. On the other hand, most of these resources are continuously coming in and out in an ad-hoc manner.

An environment usually has its own surrogate; it has the knowledge of its own environment. Furthermore, environments can communicate with each other and

exchange information through their surrogates.

Person

A *person* is an active entity that is roaming around *environment*. A *person* utilizes resources to achieve his work. He can access information anytime, anywhere. A *device* could be a channel to aid a *person* accesses information transparently, yet a *person* can still get information through other methods, such as face-to-face communications with other people.

Device

A *device*, as one of the resources, is a means to provide information to a *person*. It can be tangible or non-tangible. Especially in pervasive computing, *devices* are no longer limited to physical devices that can be seen. *Devices* are becoming invisible, being embedded in many things in our daily life.

4.1.2 Sparkle Project

In realizing our model, we have started the Sparkle project in 2000. It is still an ongoing research project in the Systems Research Group, at the University of Hong Kong. Previous work done by our team could be found in [4] [8] [19]. The architecture of Sparkle is depicted in Fig. 4.1.

In the beginning of the Sparkle project, the main goal is to solve the problems introduced by the monolithic characteristic of applications. Traditionally, applications are built as huge monolithic chunks. These applications provide lots of functionalities, yet they are too big to fit into any device, especially small devices. However, in a pervasive computing environment, small client devices become prevalent. This introduces a big problem with such a monolithic approach: The huge monolithic application in a desktop PC cannot be fitted into the small PDA. To solve this problem, you could only find another version of the application with the

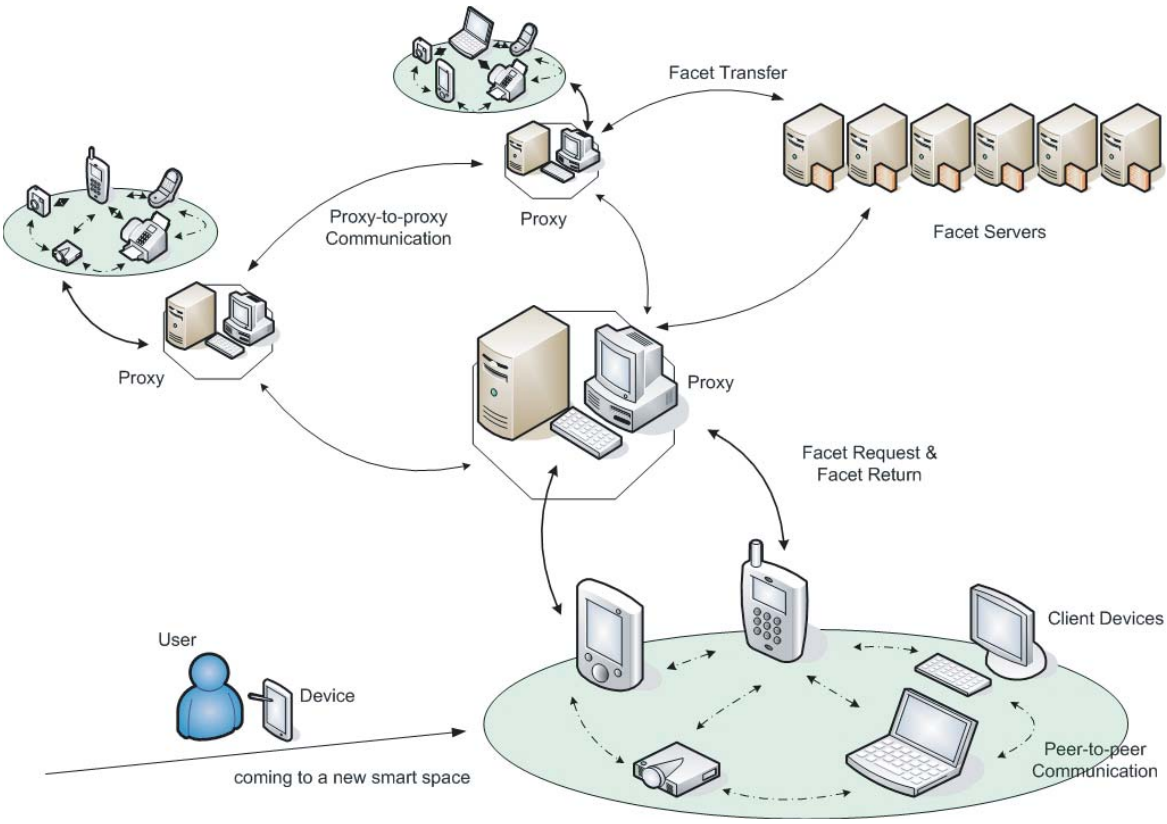

Figure 4.1: Architectural overview of Sparkle

same functionality but smaller in code size, to fit into your PDA, or, you just leave the work in your PC.

In order to solve the mentioned problem, Sparkle was introduced to support dynamic component composition and dynamic application reconfiguration. Sparkle follows a component-based software model. It enables application to be dynamically composed at run-time and reconfigured depending on the context changes. In Sparkle, applications are built from small functional units, called *facets*. These functional components could be implemented in different ways to fulfill the same functionality. When application runs, suitable functional units are downloaded from the network to the device. After the functional components have been used, Sparkle system may cache it for future use or discard it if resource is limited.

With the integration of mobile functionality system, application could also be

reconfigured dynamically. Whenever the context changes, new functional components are brought in so as to adapt to the environment. Especially when application is moved from one device to another, components of same functionality but with different implementation can be brought in to suit the target device. Although different code implementation is loaded, the application could still be resumed without causing any distraction to the user.

Figure 4.1 shows the overall architecture of the Sparkle system. It mainly consists of three entities: facets servers, client devices and intelligent proxies.

4.2 Important Concepts in Sparkle

Before mentioning the three main entities in Sparkle, we first introduce two essential concepts in Sparkle.

4.2.1 Facet

Facet plays a significant role in the Sparkle model. All applications run in Sparkle are built from these small functional components, called *facets*. Facet has two main characteristics. First, each facet only carries out single functionality. Each facet has a predefined input and output parameters, pre- and post-conditions. Second, facet is stateless. This means facet does not maintain any persistent state; any call to a facet is an independent invocation. This feature makes the facets discardable after invocation. Later, if the same functionality is needed, other facets which implements the same functionality can be brought in.

For each facet, in order to fulfill its functionality, it can also call other facets to help with this functionality. A facet may depend on other functionalities, and this is called *facet dependencies*. Different implementation of functionality may have a totally different set of facet dependencies. However, no matter how the implementation is, it is acceptable as long as it fulfills the stated functionality.

Every facet consists of two parts: shadow and code segment. Shadow describes

the properties of the facet including the facet ID, vendor number, version number, the input and output parameters, the pre- and post-conditions, its dependencies and other requirements. The shadow is represented in human- and machine-readable XML format. It can be used by proxies to locate the appropriate facet when there is a request, or it can be read by humans to learn more about the details of the facet. The second part of facet is code segment. It is the body of the execution code, which implements the functionality. In this executable code, there is only one callable method to make this service publicly accessible.

4.2.2 Container

Another remarkable concept in Sparkle is the container. The container is the core of an application composed of facets. It is a passive entity that describes the framework of the application. It stores various specifications that are needed in the application. Each container consists of three parts: the facet specification, the data agreement and the updatable user interface. For each application, it should have one and only one container.

Facet specification is a list of functionalities description provided by the application. This specification does not tell the program flow, however, it ensures the same set of functionalities provided whenever and wherever the application runs.

Data agreement is the contract ensuring data compatibility among facets. It is a list of the data that the application needs. These data are also necessary for the application to be continued when user migrates.

In the container, we have a facet specification and data agreement, however, they do not tell how the application provides these functionalities or how it manipulates the data. Hence, an updatable user interface is needed. This user interface is a machine-dependent representation that interacts with user and facets. It describes how the application looks like and how it flows.

The container also plays an important role in mobility. It keeps track of the state

information, for examples, the execution status of the facets and shared data, and some relevant information for restoring the execution. In short, we can say that the container is responsible for storing information about the execution state, in order to be able to restore it when the execution moves to another device. The details will be discussion in the later chapters.

4.3 Main entities in Sparkle

In Sparkle, the three main entities are intelligent proxies, facet servers and client devices. We shall describe them in details in this section.

4.3.1 Intelligent Proxies

Intelligent proxies act as the surrogate in our model. They have dual responsibility. First, they are responsible for receiving facet requests from Sparkle clients, matching the requirements with current facet descriptions. These requirements include the resource availability, locality and user preference, etc. After searching, proxies will return the most appropriate components to the client devices.

On the other hand, they need to keep track of some context information. This context information includes the context of the environment, some context of the devices and some context of the person. The proxies are responsible for the collection of context information from nearby sensors, such as location sensors, temperature sensors, and infrared sensors, to detect the changes of the environment, and the coming in and out of persons and devices. Moreover, proxies can communicate with devices to gather more information.

4.3.2 Facet Servers

Facet servers are the central storage area for facets, which are the functional components. After facet providers design and implement facets, they deliver these new facets to the facet server, and publish them to the proxy server to make it publicly

available on the network. On the other hand, when facet consumers want to use any functionality, they can send a request to intelligent proxies in order to bring in any suitable facets from these facet servers.

The facet servers contain the most up-to-date facets. For faster facet retrieval, a facet can be placed on more than one facet server. Since facets can be added or removed, synchronization will be made once the facet is updated.

4.3.3 Client Devices

Client devices could be any computing devices with Sparkle client system installed. With Sparkle client system, all applications running on these devices can access services provided by our system. Our client system provides all the basic necessity to the devices, so that all applications could be built up from components at run-time. Besides, with the integration with mobile functionality system, applications can logically "follow" users whenever they go between these client devices.

4.4 Summary

In this chapter, we have first presented our model in pervasive computing. The three main entities in our model is Person, Environment and Device. In order to realize the model, the Sparkle system was built following a component-based structure. In the Sparkle, the two most important concepts are the facets and the containers. Facets are small functional components that builds up an huge application, while the container is the core of an application that describes the framework of an application. Finally, we have provided the details of main entities in the Sparkle, which are the intelligent proxies, the facet servers, and the client devices. These three entities communicate with each other to provide services to end user.

Chapter 5

Context-aware State Management

In this chapter, we first look at the general definition of context, and how we categorize context in our model.

5.1 Context Management

As metioned in pervious chapter, most of the definitions conclude context is the information telling who you are, where you are, when and what you are doing. While this thesis will not emphasize on the "correctness" of the definition of context, we would focus on how we manage the context. We will emphasize on how the context information can serve user in a better way in a pervasive environment. Since user may expect to be able to retrieve any information or service no matter how the situation is, it is important that we can serve, to our best, what user want in every situation. Hence, context can be used to help in determining what kind of information or functionalities users may try to use under different circumstances. Especially in our work, when migration occurs, we would like to explore more on how the state of an application should be adjusted when context changes so as to satisfy the need of the users.

In a pervasive environment, we would like to apply the knowledge of context during migration. Knowledge of context will also be required to enable adaptation to changing environmental conditions, such as changing bandwidth and input/output devices, which can be brought about by mobility.

5.1.1 Categorization of Context

Environment	physical location, temperature, humidity, light level, noise level, UV index, wind speed
Person	identity, contact information, address book, social activity, user preference, daily schedule
Device	memory, network connectivity, bandwidth, storage, connected peripherals, local available resources

Table 5.1: Categorization of Context

Generally, most of the previous works on context-aware computing put emphasis on how the software adapts according to the context. They tried to make the applications more dynamic, adaptive and extensible via using context. Many researchers have attempted to categorize context, yet there is no standardization to classify the context.

Schilit [26] divided context into computing context, user context and physical context, while Dey [10] have defined four important context types: location, identity, time and activity. Others [14] also classified context similarly as Dey did. However, these classifications are mainly for context-aware applications, that is how applications adapt to the context. They put more concerns on the environmental context (i.e. the physical location, the temperature, etc.) while pay less attention to the computational context (i.e. memory resource, network bandwidth, etc.). By contrast, our work focuses on how the context affects the state of an application. We, thus, adapt and integrate the categorization from [26] and [6], with some

modification. These context descriptions includes the user requirements or profiles, application requirements, and device capabilities. We would divide context into three categories, as shown in Table 5.1: device, person, and environment.

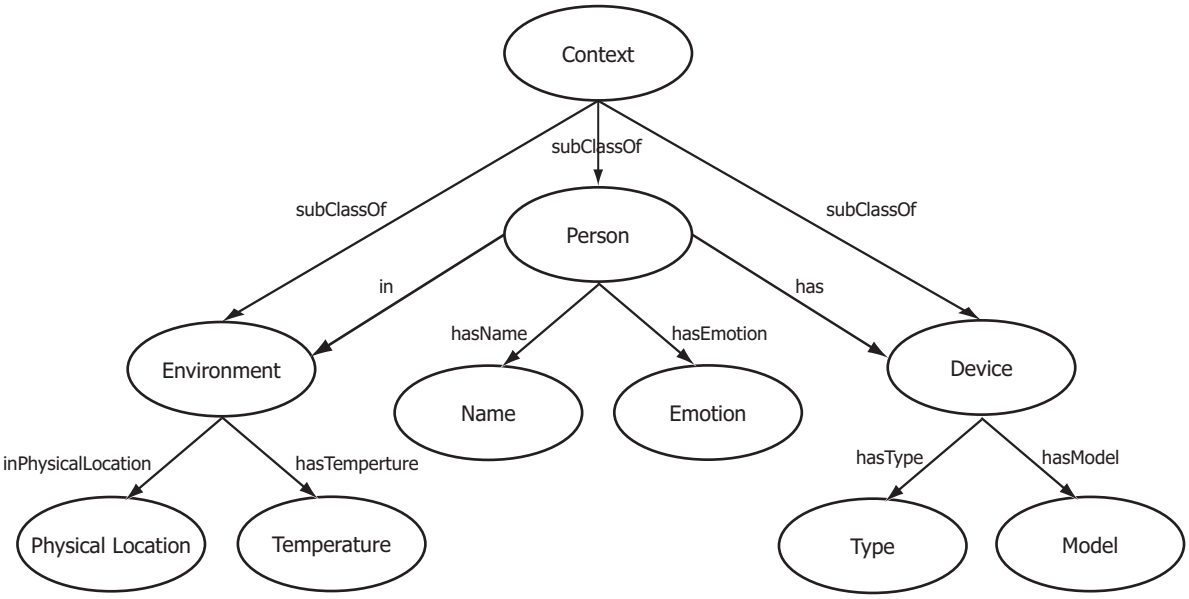

Figure 5.1: An example of categorization of context and relationship inside context

Figure 5.1 shows a graphical representation of the categorization of context, and the relationship inside the context environment. Besides acquiring the current context, we would also keep histories of context, as in [6]. It is useful and beneficial to keep histories of context, so that applications can use historical information to predict the future actions or intentions of users.

5.1.2 Context Specification

In order to achieve context-awareness, it is necessary for the system to acquire the current context periodically. Through keeping track of the context, the application could be reconfigured to suit the environment when the context changes. As the context needs to be interpreted by heterogeneous devices, as a representation language, eXtensible Markup Language (XML) would be an inevitably good choice. XML is

an extensible and structured language that allows to be exchanged between different systems. It is a common representation between systems regardless of the platforms. Thus, representing context in XML enables the knowledge to be transferred among clients and proxies.

Nevertheless, in a pervasive environment, XML alone is not enough to provide support in representing knowledge for reasoning. XML could only represent data in a common format; however, it gives no hint on how to understand the data. This introduces the need of ontology, which is an explicit specification of conceptualization [13]. It enables systems to share the common understanding of knowledge.

In our representation of context, Web Ontology Language (OWL) is used. OWL is a markup language that is built upon XML for publishing and sharing ontologies [21]. Moreover, ontology enables the reuse of domain knowledge [25], we would define our context using OWL, and based on some of the existing ontologies [2] [7].

In our design, context will be expressed differently for internal use and external use. Internally, the context will be expressed as a *profile*, which is a Java Object. The advantage is the easy management of Java Object. On the other hand, in order to communicate with a wide range of client devices as well as proxies, the context will be expressed externally in OWL, with defined schema similar to that shown in Fig. 5.2. An example of an instance using schema shown in Fig. 5.2 of under a particular context is shown in Fig. 5.3.

5.2 State Management

In our mobility support, we aim at providing service mobility so that user can carry on one's work. As a result, there is a need to manage the state so as to achieve our goal.In our system, "state management" involves four management processes:

- **State Acquisition.** During an application migration, states are required to be carried to the destination, in order to allow application to be continued

```
...
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlns:owl="http://www.w3.org/2002/07/owl#" ...>
<owl:Class rdf:ID="Context" />
<owl:Class rdf:ID="Environment" />
       <rdfs:subClassOf rdf:resource="#Context" />
</owl:Class>
<owl:Class rdf:ID="Person" />
        <rdfs:subClassOf rdf:resource="#Context" />
</owl:Class>
<owl:Class rdf:ID="Device" />
        <rdfs:subClassOf rdf:resource="#Context" />
</owl:Class>
<owl:Class rdf:ID="PhysicalLocation" />
...
<owl:ObjectProperty rdf:ID="in" />
        <rdfs:domain rdf:resource="#Person" />
        <rdfs:range rdf:resource="#Environment" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="has" />
        <rdfs:domain rdf:resource="#Person" />
        <rdfs:range rdf:resource="#Device" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasName" />
        <rdfs:domain rdf:resource="#Person" />
        <rdfs:range rdf:resource="#Name" />
</owl:ObjectProperty>
...
```

Figure 5.2: An example of the schema of context using OWL (partially)

```
...
<Person rdf:ID="Pauline.38437">
        <has rdf:resource="#Notebook.92348" />
        <in rdf:resource="#Room423.23483" />
        <hasName rdf:resource="Pauline" />
</Person>
<Device rdf:ID="Notebook.92348">
<hasMouse rdf:resource="#Optical" />
</Device>
<Environment rdf:ID="Room423.23483">
        <inPhysicalLocation rdf:resource="#Room423" />
</Environment>
...
```

Figure 5.3: An example of an instance of context (partially)

after migration. Hence, for state acquisition, it means states are captured before migration.

- **State Manipulation.** For state manipulation, it is the key of "state management". We would process and adjust the states, according to the context and following the context-aware policies. This state manipulation could be done on either the source or the destination site depending on the context environment, such as the processing power on the source and the destination device. Usually, state manipulation is preferred to be done on the source site, thus, state could be made compact to allow lightweight migration.

- **State Transmission.** As the nature of the pervasive world is highly mobile, changing of location or devices will always occur. It is necessary to transfer the state between client devices efficiently. Hence, for state transmission, it means the transferring of state from one device to another through underlying the network connection.

- **State Restoration.** After migration, states are transmitted to the destination device. Nevertheless, although states are obtained, application cannot be

resumed without proper interpretation of these states. For state restoration, it means the client system needs to restore the state so that the application can retrieve the state back for continuation.

Details of how we achieve the state management is described in Section 6.3, Migration Stages.

5.2.1 Categorization of State

One key feature in our design is the state adaptability, which allows state to be context-aware. To achieve this objective, besides acquisition of the current context, we need to manipulate state based on the context. Hence, we clssify the data state into three categories, in order to facilitate our state processing.

In the beginning, we have tried to classify data state into four categories. As data state here refers to the data shared by the running application, our categorization is based on two dimensions: (1) whether the value of that variable could be changed, and (2) whether the whole variable (i.e. the variable name and its stored value) could be thrown away.

- Type I: the variable is non-disposable, and its value is also fixed
- Type II: the variable is non-disposable, but its value is mutable
- Type III: the variable is disposable, but the value is fixed if the variable is kept
- Type IV: the variable is disposable, and its value is mutable

In our design, the data state we need to migrate is specified in the agreement inside the container. These data states listed in the agreement should be useful in migration. As a result, if the variable is of type IV, which can be modified or disposed, it is not necessary for the system to keep that state in the agreement

for migration. Consequently, we redefined the categorization as follows in order to facilitate the management of state:

- **Fixed.** This is type I. Fixed states are essential for the restoration of the application. These states are context-independent and will not change under migration. For example, bookmarks inside browser.

- **Mutable.** This is type II. States are essential for the restoration of the application but are context-dependent. They may be adjusted depending on the context of the application. For example, the screen-size of the user interfaces.

- **Disposable.** This is type III. If these states are kept for migration, they are not dependent on context and so they cannot be changed. However, if the context environment needs to minimize the amount of data transfer, these states could be disposed. For example, if there are many running frames inside the browser application, during migration with a limited bandwidth, some frames may be disposed.

Having the categorization of data states, we still need to represent the state in a suitable format. State adaptation, then, could be achieved in our mobile functionality system.

5.2.2 Representation of State

Instead of storing states in a format similar to the context information using markup languages, it is more flexible for us to store them as Java objects. There are several reasons behind it. First, context information is very helpful in reasoning the current situation, there is a strong need to represent it in a formal specification. While states are useful in the execution of application, it is sufficient for them to be interpreted in a machine-readable format. Second, context information is transferred among client devices and proxies. In order to cooperate with proxies, context should be

represented in OWL based on XML. On the other hand, states are only transferred between client devices only, there is no strong need for states to be represented as XML. Lastly, in a Java application, all variables are stored as Java object, so as the data states. If a transfer is needed, they can be transferred through Java object serialization, and object can be re-created easily. However, if the states need to be stored in XML format, conversion is always needed which is time-consuming.

Consequently, in our design, state will be expressed as a *Storage*, which is a Java object. The main benefit is the easy management of Java object.

5.3 Context-awareness

Researchers have explored context-aware system. Context-aware means the ability for a device to make use of the current context and react on it. In current literature, there are basically two types of context-aware system:

- **Reflection-based.** A reflection system keeps the data structure that materializes some aspects of the system itself [11]. Reflection means the capability of a system to reason about and act upon itself.

- **Policy-based.** A policy-based system is much simpler than a reflection-based system, in the sense that a policy-based system only specifies different policies to react on the context changes.

In our approach, the context-awareness is based on policies. In general, the policy-based approach could be applied to all systems easily, independent of the implementation language [5]. In our policy-based context-aware system, the policies are structured in the canonical "If condition(s) then action(s)" format. This rule means when the condition(s) in the "if" part is satisfied, the action(s) in the "then" part will be performed.

In our current approach, these context rules are defined inside the mobility manager, details of the entity inside the system will be discussed in the next chapter.

The mobility manager will take in the state, the context information of the source and the destination, and apply the pre-defined context rules. Later, it sends the adjusted state to other related entities.

Nevertheless, in a pervasive computing environment, the traditional policy-based context-aware system is not enough for context-aware state management. Owing to the heterogeneity of the pervasive environment, different context-aware rules to manage state are needed under different contexts. Hence, in our system, there are two sets of policies. One set consists of the *basic policies*. These policies are the basic rules for rule adaptation. These rules would not be changed under context, and help to control the context-aware rules. Another set consists of *context-aware policies*. These policies would be changed under different context, so that state could be managed differently under different contexts.

5.4 Summary

In this chapter, first, we discussed about what context is, how the categorization should be, and how the specification should be. We, then, introduced our management, categorization and representation of state. Lastly, we discussed our context-aware policy scheme in order to achieve context-aware state management.

Chapter 6

Mobile Functionality System

In this chapter, we shall first have an overview on the mobile functionality system. We shall discuss the main entities in the system in details. After that, we shall describe our state management stages when a migration is initiated.

6.1 System Overview

In order to achieve mobility, our mobile functionality system is needed to be integrated with the Sparkle client system. The mobile functionality system provides basic necessity to all devices, so that application can continue their execution from devices to devices. Thus, it is important to the whole Sparkle architecture.

Our mobile functionality system is situated in the client devices. It interacts and incorporates with the Sparkle client system to achieve context-aware mobility support.

Fig 6.1 shows the overview of the mobile functionality system.

6.2 Entities in the Mobile Functionality System

In our mobile functionality system, there are five modules which interact with each other to provide context-aware mobility support. They are the application manager,

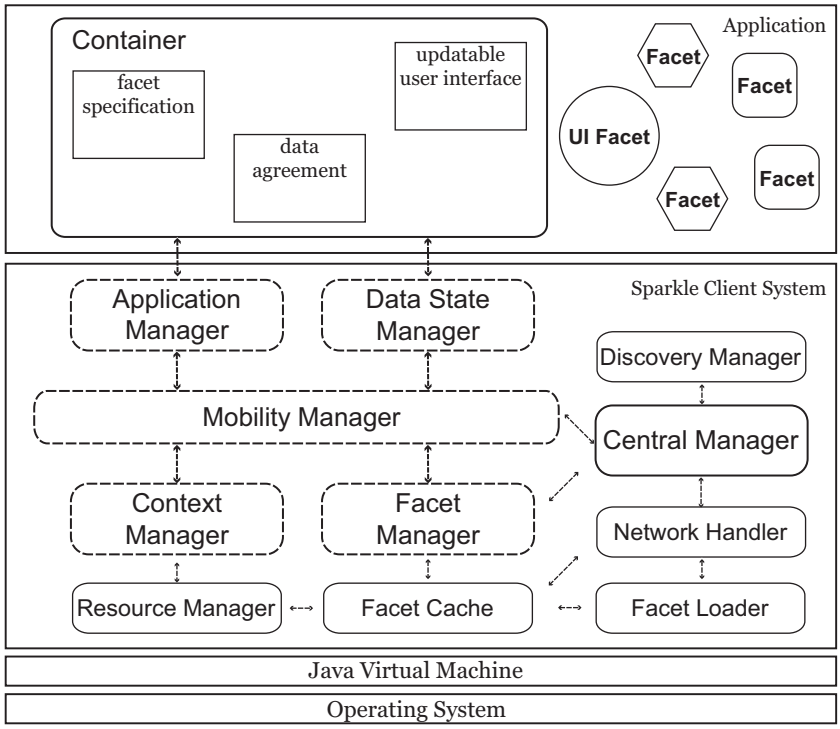

Figure 6.1: Architectural overview of the Mobile Functionality System

the data state manager, the context manager, the facet manager, and, the mobility manager. Besides, our mobile functionality system needs to cooperate with the Sparkle client system. For details of the client system, please refer to [4].

6.2.1 Application Manager

The application manager is responsible to mange one and only one application. It acts as a bridge between the Sparkle client and the application. Before the starting up of an application, the application manager helps in initializing the container for the application. During the execution, it is the runtime environment to support the execution of the container, and control the container if needed. After shutting down the application, it helps to do some housekeeping jobs of the application.

To startup and shutdown an application, we need the application manager. Actually, in Sparkle, starting up and shutting down an application means the starting and closing the container respectively. The application manager dynamically loads

the container in order to start the application.

In our current implementation, in order to load the container dynamically, we first need to check if the container is located locally. If it is, we can get the container class object using the input class name. This is a typical way to load the container by calling the forName(String className) method on the class java.lang.Class.

On the other hand, if the container is located remotely, we need to load the container class from peers through network. This needs to make use of Java's class loaders. If we can successfully load the container class into the virtual machine, a new container object can now be instantiated. After that, the container will be initialized and the environment for running the application is set up.

For any running application, there may be a need to stop it due to migration or energy saving. Our application manager also provides a method *appClose()* to do the close down and the housekeeping of the container.

6.2.2 Context Manager

Besides mobility, adaptability is another goal we would like to achieve in the pervasive environment. Context changes as time goes by. Hence, the context manager is needed to keep track of the current context situation for better adaptability. It is responsible to gather the current human context, the device context and the environment context periodically. The human context includes the identity of the user and the current activities or future plan of the user. The device context includes the current memory available, bandwidth available and input/output devices available. The environment context includes the location, temperature, light intensity and noise intensity. The context manger keeps the new context information, and backs up the old one as history. Through analyzing the history of the context, it could do some predictions for application migration, or generate controls to the central manager for facet removal, facet reloading or facet pre-fetching.

Context is important in a context-aware system. There is a need to monitor cur-

```
public class ContextManager {
        // setting the context profile
        private void getDeviceContext() {...}
        private void getEnvironmentContext() {...}
        private void getHumanContext() {...}
        public ContextProfile getCurrentContextProfile() {
                ...
                getDeviceContext();
                getEnvironmentContext();
                getHumanContext();

                return cp;
        }
        public String getCurrentContextXML() {
                ...
                return cp.generateXML()+ "\n";
        }
}
```

Figure 6.2: The ContextManager class

rent context and keep a record of it. In our implementation, the current contextual information will be stored as a *ContextProfile*. Moreover, a history of past context information will also be kept. Besides, an XML file containing context information would be generated if necessary. For example, when the remote machines ask for the current context for context-aware state management, the context in XML format will be sent out.

The context manager plays a signification role in the system. However, a comprehensive implementation of it would require an in-depth study and an integration of the multiplicity of sensing hardware, such as location sensors, temperature sensors, and so on. Hence, at current stage, our prototype does not support all dynamic tracking of the current context. We support the sensing of current device environment, for example, the screen size of the device, the memory resources, and the current machine time. The support of detecting human identity and activities are achieved only by using the calendar book of the user. For current physical environment, our system only read from an XML file, which stores the details of the physical environment. Figure 6.2 is the extracts of codes of the context manager.

```
public class FacetManager {
        // record statistics of each facet
        public void recordFacetStat(FacetProfile fProfileInput) {
                ...
                if (objCount == null) {
                        //if no key found, add new one
                        htStorage.put(fProfileInput, new Integer(intCount));
                } else {
                        intCount = Integer.parseInt(objCount.toString());
                        //else add 1 to the original one
                        htStorage.put(fProfileInput, new Integer(intCount + 1));
                }
                ...
        }

}
```

Figure 6.3: The FacetManager class

6.2.3 Facet Manager

Different facets are brought in from the network in order to adapt to current context. Although the functionalities performed are the same, different implementation of facets may be loaded in depending on the instance resource availability, network bandwidth, location, user preference. Some facets may be more suitable to the current situation than others. The facet manager helps to record the details when a facet is loaded in. It keeps information on how often a facet is loaded, under what situation a facet is loaded.

These information and statistics are essential to our system. The benefits are twofold. First, this information helps the system in facilitating the pre-fetching of facets. The system could download the facets before the application arrival during migration. Second, they also help the system to maintain the entries in the cache, so as to achieve a better hit-rate.

Every time there is a call on a facet, the facet manager will record down which facet is used. The *FacetProfile* stores some details of when this facet is called, for example, the functionality ID of the facet, the identity of the user, the device and the application is being used. From these details, the system can later make some

analysis on which facet is always called by this user, or which facets is often used by this application, and so on.

However, resources need to be spent to save the statistics. The more facets loaded in, the larger is the memory used to keep the record. Hence, we need to make a trade-off between the amount of statistics kept and the memory usage. At present, least-recently-used (LRU) algorithm is used to keep the number of statistics under a specified threshold. In order words, we only keep the statistics of facets which are more frequently used.

Moreover, as only a proof of concepts, our implemented facet manager only keeps very limited details and provides very limited functions at this stage. Figure 6.3 demonstrates the codes of the facet manager.

6.2.4 Data State Manager

When there is a migration request, states inside the container will be captured by system automatically. In Sparkle, facets are downloaded whenever current context changed. Especially, when the application migrates to a new device, new facets might be downloaded from the network to suit the current context. Data states, the internal variable values at a point during execution, are needed to be captured for application resumption. Data State Manager is responsible to capture the internal data states of the running facets.

This data state manager provides callable methods, *captureState(FacetContainer fcInput)* and *caputreContainerName(FacetContainer fcInput)* for mobility manager to access the data state inside the container. Before migration, mobility manager will call upon these methods for getting the current status of data states.

In our current implementation, we have made use the feature of Java reflection. Reflection allows the examination of internal properties of the running application. Within an executing application, through Java reflection, the data state manager can dynamically retrieve the information about the internal data state inside the

```
public class DataStateManager {
        private Storage captureState (FacetContainer fcInput) {
                //capture the data storage in container
                ...
                for (int i=0; i<myFields.length; i++) {
                        ...
                        Object objField = myFields[i].get(fcInput);
                        if (objField instanceof Storage) {
                                s = (Storage)objField;
                        }
                }
                return s;
        }
        private String captureContainerName (FacetContainer fcInput) {
                // capture the container name
                ...
                if (objField instanceof FacetContainer) {
                        return objField.getClass().getName();
                }
        }
        ...
}
```

Figure 6.4: The DataStateManager class

container. Figure 6.4 is the extracts of codes of the data state manager.

Besides capturing the data state, we need to capture the class name of the current container running. This is important as we need to start up the same container in the destination device.

6.2.5 Mobility Manager

The mobility manager is the heart of the mobile functionality system. It is the main entity that communicates with the central manager of the Sparkle client system. The mobility manager helps in coordinating with other entities to achieve successful migration of application. It monitors and determines when a user needs to migrate. It prepares for the migration, for example, it predicts the future destination, gathers context information of the predicted environment, sends facet statistics for pre-fetching facet at predicted site. Besides preparing for migration, with the help of data state manager, facet manager, and context manager, it also enables lightweight migration with context-awareness of state.

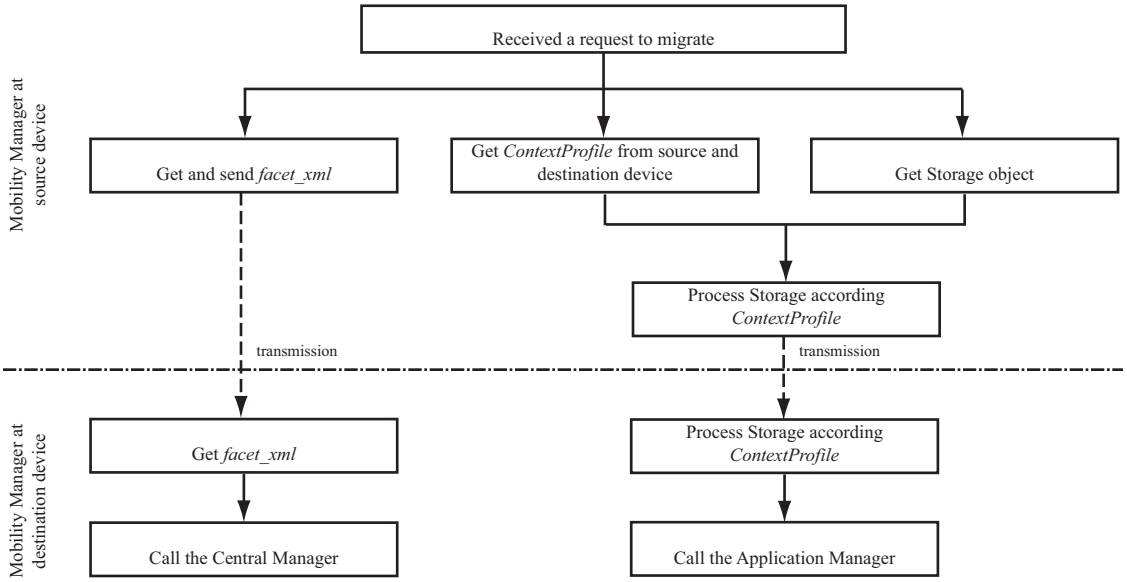

Figure 6.5: The actions carried out by Mobility Manager

Our mobility manager is the core to facilitate mobility. It controls the whole migration process. It provides a method *migrateTo(String strDestination, FacetContainer fcInput)* for initiating migration. Furthermore, the context-awareness of state is processed within this manager.

At current stage, our mobility manager will first get some context information from the destination device. It will then send some facet statistics to the destination device for facet pre-fetching. The mobility manager will then request the data manager to capture the data states. Besides, it will ask the context manager to gather the current context information. With the context data from the source and the destination device, it will first determine if the source device performs the context-aware state process, or the destination device does, depending on the resources on both devices. If the source device has richer resources, the mobility manager will then process the application states according to some pre-defined context-aware policies. After processing, it will transmit the data to the destination. At the destination, it will re-process the data state if needed. Otherwise, the mobility manager will ask

```
public class MobilityManager {
        ...
        private void processState (Storage sInput, ContextProfile cpDest) {
                ...
                // process state according the data state's category
                if (strChildType.equals(StorageChild.DISPOSABLE)) {
                        //if disposable, it can be disposed, but can't be changed
                } else if (strChildType.equals(StorageChild.MUTABLE)) {
                        //if mutable, it can be changed, but can't be disposed
                }
        }
        private void preloadFacet () { // pre-loading facets ... }

        public boolean migrateTo (String strDestination, FacetContainer fcInput) {
                // get some context information from destination
                ContextProfile cpDestination = read(p.getInputStream());
                // transfer some statistics to destination
                ..
                // capture data state
                Storage sState = captureState(fcInput);
                // process state
                processState(sState, cpDestination);
                // transfer data to destination
                ...
        }
        public void migrateFrom (ObjectInputStream oinStream) {
                // get data from source
                Storage sState = (Storage) oinStream.readObject();
                // process state
                processState(sState, cp);
                //set up the container
                new ApplicationManager(manager, strContainer, sState );
                ...
         }
}
```

Figure 6.6: The MobilityManager class

the application manager to set up the environment, initialize the container and resume the application. Figure 6.5 summarizes the actions performed by the mobility managers at the source and the destination devices. These actions will be carried out in different stages during migration. Details of the migration stages will be discussed in the coming section. Figure 6.6 is the extracts of codes of the mobility manager.

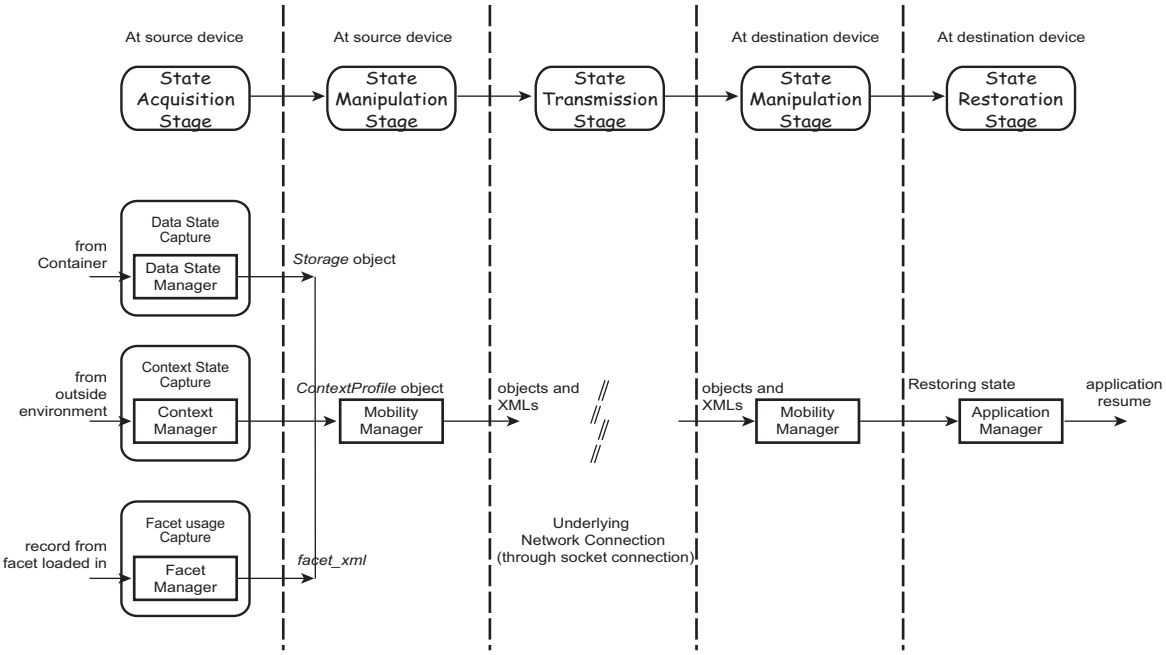

Figure 6.7: Migration Stages

6.3 Migration Stages

Our mobile functionality system not only will detect the current context to allow state adaptation, it also prepares states for the predicted future context. During migration, it will undergo five logical stages. These five stages for the state management are: State Acquisition, State Manipulation (at the source), State Transmission, State Manipulation (at the destination), and State Restoration. Depending on the context, either one of the State Manipulation could be passed over. For example, the resources at the destination are very limited, the mobility manager will ask to proceed to State Restoration Stage, while skipping the State Manipulation (at the destination). Mobility manager has the responsible to control the whole progress during migration.

In this section, we shall describe each stage, as shown in Fig. 6.7, in details.

6.3.1 State Acquisition

Whenever an application migration is triggered, no matter the command is initiated by the client system or by the user, it will enter the State Acquisition stage. It is the first stage for application migration. First of all, an appropriate checkpoint for migration will be chosen. There are three criteria in selecting the proper checkpoint: (1) All migrated facets could fit into the target environment; (2) Ensure the continuation of application; and, (3) Minimizing effort and amount of data in transmission. After taking these three criteria into account, our system would move the application at root-level facets.

Besides selecting the appropriate checkpoint for migration, some required information is required to be captured. This information is the data state of the current running application, the current context environment, and the facet usage statistic. Hence, the State Acquisition stage will be divided into three sub-stages, namely data state acquisition, context acquisition, and facet usage acquisition, to capture these information respectively. For better performance, these three sub-stages could undergo parallelly.

Pervasive computing is characterized by its full of heterogeneous devices. As our states are needed to be interpreted by these various kinds of devices, the representation of these states should be machine-independent when states are under transmission. An eXtensible Markup Language (XML) is a suitable format in our case for representing our states, because it is extensible, portable and human-legible. For example, where there is a need to transmit the current context information and state information among devices and proxies, an XML would be generated for the transmission. This is a good choice as system administrator can also read these information if necessary.

On the other hand, if states are for using within the local device only, it will be maintained as Java objects. Besides, as data states are only transmitted between system clients, and it is seldom read by the system administrator, data states will

not be represented in XML format. In addition, the data states usually are in a more complicated structure, such as multi-dimensional arrays or vectors, it could be time-consuming to convert them to and from XML format. Instead, data states will be transmitted between clients as Java objects through object serialization. The advantage of using object serialization is the ability to move objects through reading and writing entire objects to the input/output stream, without converting them to and from raw bytes data. This is much faster and easier for client to reconstruct the data state into objects after migration.

In the following, we will describe the three sub-stages in details.

Data State Capturing

Functionality adaptation is one of the key features in the Sparkle system. It allows different implementations of the same functionality be brought into the devices whenever the context changes. This feature is important in Sparkle, yet it brings new challenges to Sparkle. As new implementations may be brought in from the network when migration occurs, the states captured are required to be compatible with the new facets in order to allow the continuation of application.

The container, thus, becomes the important entity in Sparkle. Besides storing the specification of user interface and root facets, container also stores the agreement of variables for mobility support. This data agreement is essential, which acts as a contract between applications concerning the variables needed for migration.

In data state capturing, our data state manager will be responsible for the acquisition of state stored in the container. This capturing makes use of Java reflection. In this stage, all data in the container will be captured without modification. Date state manager will store the data states inside the container as a Java object of *Storage* class. After capturing, it will pass the *Storage* object to the mobility manager for future processing.

Context Capturing

Adaptation is indispensable in a pervasive environment. As mentioned in Chapter 4, adaptation could be divided into many categories. Nevertheless, adaptation is required because the context changes. Consequently, one of the ways to provide better adaptation is the ability to know the current context well. This could be achieved by sensing the current environment and detecting the current system periodically. This also marks the importance of this sub-stage.

At this sub-stage, our context manager will be responsible for the acquisition of contextual information. Usually, the contextual information will be stored as *ContextProfile* for easy maintenance and private use within the system. Hence, the *ContextProfile*, which stores the most up-to-date context, will be sent to mobility manager for reference.

Facet Usage Capturing

Besides adaptation, our system also facilitates the pre-fetching of facets during migration. In our design, the facet usage information is transferred to the destination device before the transmission of the application. Therefore, pre-fetching could be done beforehand. This states the reason behind this sub-stage.

At this stage of facet usage capturing, our facet manager will be responsible for reporting the facet usage statistics. In Sparkle, whenever a facet is brought in from the network, the facet manager keeps a record of the event. Thus, the facet manager will generate a report, the *facet_xml*, which is in XML format. This *facet_xml* will then be sent to the target device via the mobility managers.

In this stage of state acquisition, three pieces of information are generated. The data state manager generates the *Storage* object, the context manager generates the *ContextProfile* object, and the facet manager reports the usage as *facet_xml*. The mobility manager collects these information, and proceed to the second stage.

6.3.2 State Manipulation

In general, most current systems separate the concepts of mobility and adaptability. As a result, in general, for mobile systems that support mobility, states are transferred and restored *as they are*. In contrast, aiming at complementing mobility and adaptability, our system will process the state during migration. This stage, State Manipulation, makes our system having a great merit over other mobile systems.

It will come to State Manipulation Stage under two circumstances: (1) After State Acquisition Stage at the source device; and, (2) After the State Transmission stage at the destination device. Actually, the main purpose of both state manipulation stages is to support state adaptation via adjusting state, and this is the key feature in our system design.

The mobility manager plays a pivotal role in both stages. It takes the responsibility to coordinate the source and the destination devices, decide which device should process the state, and how the state should be adjusted. The mobility manager makes these decisions according to the context environment and context-aware policies.

Although the goal of both stages is the same, in the implementation, there are still minor differences. Therefore, we would like to separate the discussion of State Manipulation into two parts: at the source and at the destination.

At the source

At this stage, mobility manager will first get the *facet_xml* from facet manager and send out it to the destination device. Meanwhile, mobility manager gathers context information from the source and the destination devices. Mobility manager will get a *ContextProfile* from the source, as the profile is generated from the context manager for internal use. On the other hand, a *context_xml* is received from the destination, since the information is obtained from another source via the network connection. A *ContextProfile* object will then be generated by parsing the *context_xml* obtained

from the destination device.

According to the context information from both devices, mobility manager will make a decision on whether to continue the processing of the state at the source device, or if it is more appropriate to leave the state adjustment to the destination device. In our current implementation, this decision is defined in the context-aware policies by system administrator. For example, if the processing power is greater at the source device, the mobility manager will continue the state manipulation.

In the meantime, the mobility manager gets the *Storage* objects from data state manager. If the mobility manager confirms to process the state at the source device, it will bring the three objects together for further processing. The three objects are the *Storage* object, the *ContextProfile* object from the source and the *ContextProfile* object generated from the *context_xml* sent by the destination.

In our current implementation, some context-aware policies are predefined inside the mobility manager. As mentioned before, data states are classified into three categories. As a result, the mobility manager follows these context-aware policies, adjusts the mutable states as well as throws away some of the disposable states. A new *Storage* object will be generated. This new *Storage* object will be sent to the destination device using Java object serialization via the mobility managers. Besides, a *final_xml*, containing information and remarks, such as the state processing status and the context at the source, will also be sent to the destination together.

At the destination

In our approach, the mobility manager at the destination receives a *Storage* object and a *final_xml* in this stage. Again, it needs to make a decision on whether to continue the state processing according to the context-aware policies. For example, if the states have not been processed before, the mobility manager will continue the state manipulation.

If the mobility manager confirms to process the state, it will obtain the latest

context information from the destination device. It gathers the *ContextProfile* from the context manager at the destination, and the *Storage* object received in order to process the state according to the predefined context-aware policies.

Later, the mobility manager will pass the finalized, processed *Storage* object to the next stage.

6.3.3 State Transmission

This is a very simple stage. Network connection is set up between Sparkle clients whenever there is a migration triggered. This network connection is set up using socket connection implemented in Java. The two Sparkle clients, the source and the destination clients, will set up a peer-to-peer socket connections on TCP. They both can communicate with each other.

As taking the advantage of Java object serialization, the clients are communicate through *ObjectInputStream* and *ObjectOutputStream*. The *Storage* object and *final_xml* are converted into *Byte* object, thus, could be sent via these two streams between Sparkle clients.

6.3.4 State Restoration

Now, it comes to the final stage, the State Restoration stage. The main purpose of this stage is to restore the states and continue the execution of the application.

For the mobility manager, it will now start a new application manager to manage this new application in this device. It will also send the *Storage* object to the application manager for the initialization of application. Hence, the application manager will take up the responsibility to re-establish and initialize the container. Consequently, the application manager starts up the application for the user to continue his work.

6.4 Summary

In this chapter, we had an overview on the mobile functionality system. In our mobile functionality system, there are five main entities, namely application manager, data state manager, context manager, facet manager, and mobility manager. We have discussed their responsibilities and implementations in details. After that, we described our state management stages when a migration is initiated. There are basically five stages: State Acquisition, State Manipulation (at source), State Transmission, State Manipulation (at destination), and State Restoration. s i

Chapter 7

Application and Evaluation

In this section, we shall first describe our application implemented in Sparkle. After that, we shall present some experimental results on the application implemented in Sparkle with mobile functionality system.

7.1 Application description – Universal Browser

Universal Browser is unlike to traditional web browsers, which in majority limited to webpage browsing, it is a browser designed for pervasive environment to browse whatever you want. It is a special graphical user interface (GUI) implemented in Sparkle.

This special graphical user interface allows user to dynamically retrieve functionalities they want. As shown in fig 7.1, a user can use the Universal Browser to browse a webpage, to play a game, to edit an image. the Universal Browser can retrieve the functionality a user wants from the network for the user.

Moreover, these functionalities could be brought in from network when needed and be thrown away after use. A user does not need to worry about if the software suitable for his operating system, or if his device has enough memory to install or run the program. The Universal Browser, supported by the Sparkle system, could help a user to find the suitable facet (i.e. functionality) for his device, and discard

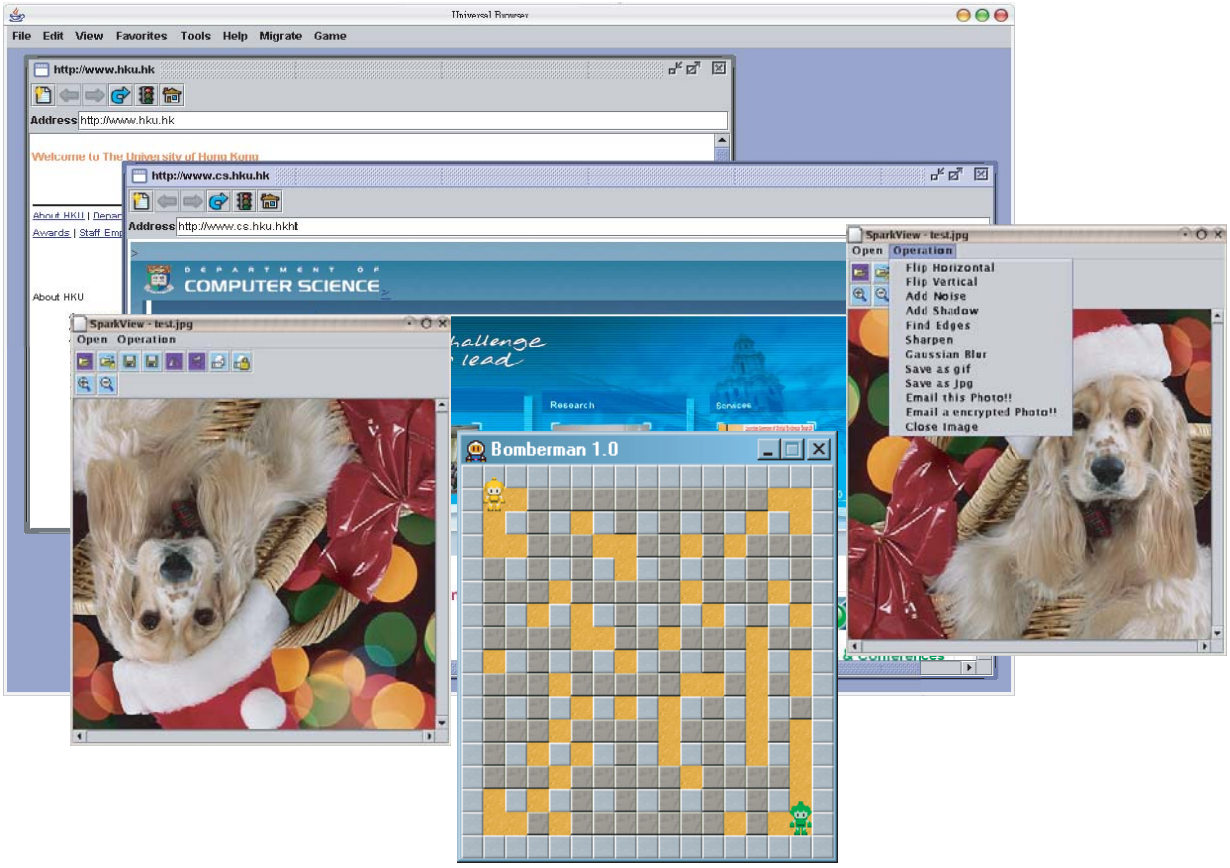

Figure 7.1: Screen Shots of the Universal Browser

it whenever that facet is no longer needed.

Furthermore, the Universal Browser is a context-aware and extensible application. The context-awareness of Universal Browser is totally different from that of state management. The former one is in application level while the latter one is in system level. The context-awareness here means the downloading of different functionalities under different contexts. Especially in the pervasive world, functionality should "follow" a user to provide him necessary functionality any where.

In the coming future, the Universal Browser would become the pre-installed application with Sparkle. The interface of the Universal Browser should be device-dependent. The functionality would be context-aware and user-centric. As a user moves around, using different Universal Browsers, he could get the same set of

functionalities and continue to carry on his work. This is the world of pervasive.

7.2 Application Composition

In our current implementation, the Universal Browser has many facets, as shown in Table 7.1, to carry out different functionalities. These facets provide the basic functions for daily life or web browsing. On the other hand, if a user would like to use other functionalities, they could be loaded later.

Functions	Facets that carry out the corresponding functions
Browse HTML files	HTMLRender
Maintain user preference	AddPreference, GetPreference
Maintain user history	AddHistory, GetHistory
Maintain user schedule	Calendar
Listen to MP3 songs	MP3Player
Play games	Bomberman, Blackjack

Table 7.1: Basic functions and related facets in the Universal Browser

7.3 Evaluation

In this section, we would like to present some experimental results using our mobile functionality system based on the Universal Browser. The original Universal Browser program has approximately 2,500 lines of code, of total size approximately 104KB. When the Universal Browser is written using facet programming, the main Universal Browser program is reduced to 1,500 lines. The size of the Universal Browser is around 60KB, where over two-third of the size is for coding the graphical user interface. When the Universal Browser needs different functionalities, it needs to bring in facets. These facets are approximately of 50 to 200 lines depending on the facet complexities. The total size of the facets is over 200KB. Bomberman and

Blackjack are two game facets built upon Universal Browser.

7.3.1 Experiment testbed

Environment Specification

The mobile functionality system is built on Java. All benchmarking programs are written and compiled with Sun Java SDK 1.3.1. It runs on a Java Virtual Machine.

We have four client devices and a proxy for testing all the experiments in this chapter. The configuration of the source client device, named as *Notebook_Source*, is a notebook equipped with Intel Pentium III Mobile CPU 1133MHz and 384MB RAM. This machine is running Windows XP operating system. The remaining two client devices are the destination devices. One of the destination devices, named as *PC_Destination*, is a standard PC with an Intel Pentium 4 2.26GHz processor and 512 MB RAM. The other one, named as *Notebook_Destination*, is a notebook with Intel Pentium II CPU 366MHz and 128MB RAM. Both of these machines are running Fedora Core 2 operating system. The last client, named as *PDA_Destination*, is a HP iPAQ H5500 Pocket PC with a XScale Intel 400MHz processor, 128MB of RAM and 48MB of ROM. This PDA is running Familiar Linux v0.8.2. The proxy had the same configuration as the destination PC.

Context rules

In our mobile functionality system, context awareness is another important issue. All benchmarking programs are performed using artificial context rule sets. The ruleset contains 100 rules and is built as follows: Each rule in the ruleset has different actions under different cases and situation. Over 60% of rules in the ruleset has at least three different cases with different actions. Figure 7.2 shows an example of simple rules with only one action concerning one context information, while Fig. 7.3 shows an example of complex rules manipulating different context information and making different decisions.

```
<baserules>
        ...
        <rule>
        <!-- This is a simple case for migrating to a more memory device>
        <!-- prioritized by number from 1 to 10, 1 for highest priority>
                <priority>3</priority>
                <case>
                        <deviceContext>
                                <deviceNearby>more memory</deviceNearby>
                        </deviceContext>
                        <action>
                                <move to:deviceNearby>as soon as possible</move>
                        </action>
                </case>
        </rule>
</baserules>
```

Figure 7.2: Example of simple context rule

7.3.2 Performance Analysis

This dissertation mainly focuses on the mobile functionality system. Hence, our experiments put emphasis on evaluating the timing performance of this mobility system, such as the timing latency and the amount of data transferred. On contrary, the evaluation of the performance on correctness of context-awareness would heavily require the sensing of the current context and the context-aware policies. Thus, it is out of the scope of this thesis.

Measuring the migration latency

We have measured the migration latency of different functionalities of our Universal Browser. Migration latency is the time elapsed from the time when the migration is initiated to the moment when the application is completely visible on the target device. This experiment is conducted using notebook at the source and the PC at the target. In addition, we have measured the size of the transferred state using our context-aware scheme. The result is depicted in Table 7.2.

Measurements show that the migration latency of the browsing tool is less than that of the other two games. This is mainly because fewer states need to be captured

```
<baserules>
        <rule>
        <!-- This is a more complex context rule to specify the action
         need to be done at 15:00 in HW520 on Fridays>
                <priority>5</priority>
                <case>
                        <environmentContext>
                                <time>15:00</time><day>Friday</day>
                                <location>HW520</location>
                        </environmentContext>
                        <deviceContext>
                                <applicationUsing>more than 5</applicationUsing>
                                <deviceNearby>PaulinePC</deviceNearby>
                        </deviceContext>
                        <action>
                                <move to=deviceNearby>immediately</move>
                                <screenSize>optimize</screenSize>
                        </action>
                </case>
                <case>
                        <environmentContext>
                                <time>15:00</time><day>Friday</day>
                                <location>HW520</location>
                                <weather>sunny</weather>
                        </environmentContext>
                        <personContext>
                                <with>Fred</with><with>Bob</with>
                                <schedule>free</schedule>
                        </personContext>
                        <action>
                                <lightIntensity>high</lightIntensity>
                                <msg>Lets have tea now!</msg>
                        </action>
                </case>
                <case>
                        <environmentContext>
                                <time>15:00</time><day>Friday</day>
                                <location>HW520</location>
                        </environmentContext>
                        <personContext>
                                <with>Susan</with><with>Bob</with>
                                <schedule>free</schedule>
                        </personContext>
                        <deviceContext>
                                <applicationUsing>more than 5</applicationUsing>
                                <deviceNearby>None</deviceNearby>
                        </deviceContext>
                        <action>
                                <speaker>off</speaker>
                                <lightIntensity>low</lightIntensity>
                        </action>
                </case> ...
        </rule>
</baserules>
```

Figure 7.3: Example of complex context rule (partial)

Applications	Without state manipulation		With state manipulation	
	Migration latency (ms)	Data transferred (bytes)	Migration latency (ms)	Data transferred (bytes)
Browsing tool	3837	1938	2913	1593
Bomberman	4183	3538	3129	2034
Blackjack	3935	2365	3293	2039

Table 7.2: Migration latency and size transferred

and processed in the browsing tool. Moreover, the size transferred in the browsing tool is much less than that of the other two games. This is due to the fact that most of the states in the browsing tool can be dropped.

The result also shows that the amount of data transferred has no great difference between the cases with and without data state manipulation before the state transmission. This is mainly due to our pre-defined context-aware rules. In our limited scale of implementation, the context-aware policies are defined to demonstrate only some states adjustment and removal, these set of rules are not very comprehensive as our main focus is not on defining these context-rules. Besides, states could be just adjusted to suit the context instead of completely disposed. Hence, the result of state manipulation is not so significant, yet it shows that there are some reductions in case some of the states are disposed.

Measuring time for migration states

In this experiment, we have measured the time needed for each migration stage. For context capturing, since we do not have real hardware for sensing the environment, we have implemented it as reading the context from external files. The reason is, in real world, gathering context information usually needs to communicate with other hardware, such as location and temperature sensors, instead of reading internal variables. Hence, we implemented this as reading files, while not just hardcoding the values inside the program. The result of this experiment is depicted in Fig. 7.4.

According to the graph, the total time of those five stages during migration is around 1100ms. Compared to the result on Table 7.2, which needs approximately

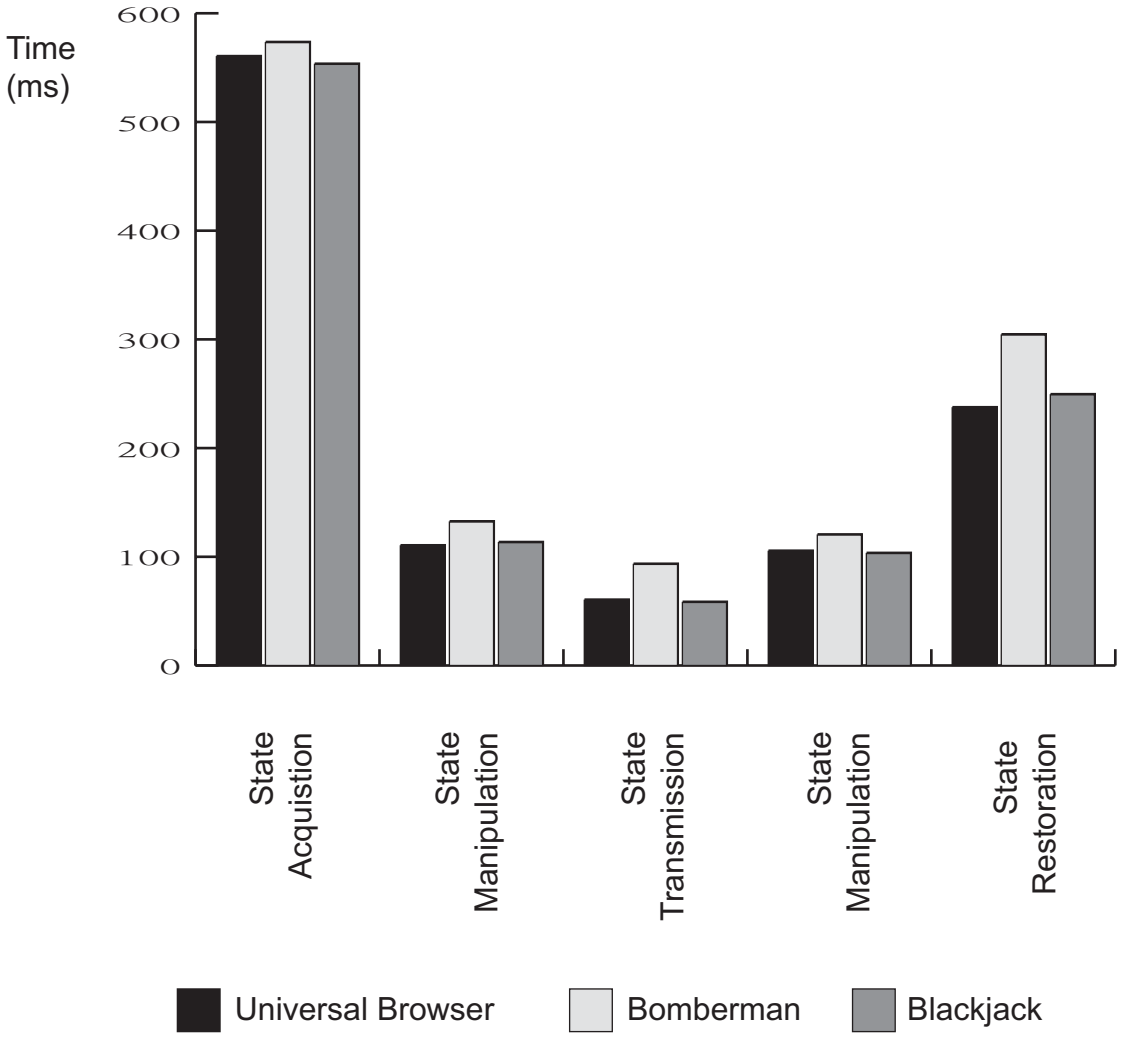

Figure 7.4: Time for each Migration Stages

4000ms for the whole migration, there is a great difference between both results. This is due to the fact that in the first measurement, the definition of migration latency is measuring the time from command initiation to application visible to user. On contrary, in this measurement, we have only taken the time of the five stages. The main reason for the difference is the time for re-drawing the graphical user interface on the destination device, which takes around 2000ms.

Furthermore, as the graph shown, the state acquisition takes the longest time. For state acquisition, it consists of three processes: data state capturing, context

capturing and facet usage capturing. The time for data state capturing and facet usage capturing only takes around 20ms, while the time for context capturing takes the longest time. The major reason is that, in our implementation, although there is no real hardware for sensing the environment context, we have made the acquisition of context through reading some external files. As a result, this takes certain a amount of time for reading the external information. In real world, this acquisition stage may take much longer time, as it needs to communicate with other sensing hardware and proxies to get the context information.

For state manipulation, it actually does not take so much time, around 100ms, compared to other stages. Hence, it is reasonable to do some state processing to achieve state adaptation. Moreover, in this experiment, we have forced them to process the states twice, before and after the migration. However, in the real world, the state processing could be done either at the source or the destination device depending on the context and system choice. Thus, time spent could be saved if the processing is done once only.

For state transmission, it takes only about 60ms. Opposing to the general belief of network connection is the bottleneck, the state transmission takes only a little amount of time compared to other states. There may be two reasons. First, the size of data transferred is only a small amount. As we only transferred the functionality specification, instead of the whole program code, and some data states, the size is about 2000 bytes. Much time should be needed to resume the application if the code is not situated in the destination. The downloading time of facets from proxies depends on the size of code, in an average of 3.8 seconds. Second, the Internet connection between two client devices is good, as they are connected within LAN inside the university. The traffic may not be too heavy during the experiment.

Lastly, for state restoration, this takes a longer time as it needs to rebuild and reinitiate the Container. If the container code is not in the destination device, it needs to retrieve it from source device. Hence, in this stage, much more time is

needed.

Measuring time for state manipulation on different machines

As the resources varied from machines to machines, under different context environment, time spent for the State Manipulation stage on these devices should be different. In this experiment, we have measured the time spent on manipulating state on different machines. The results is shown in Fig. 7.5.

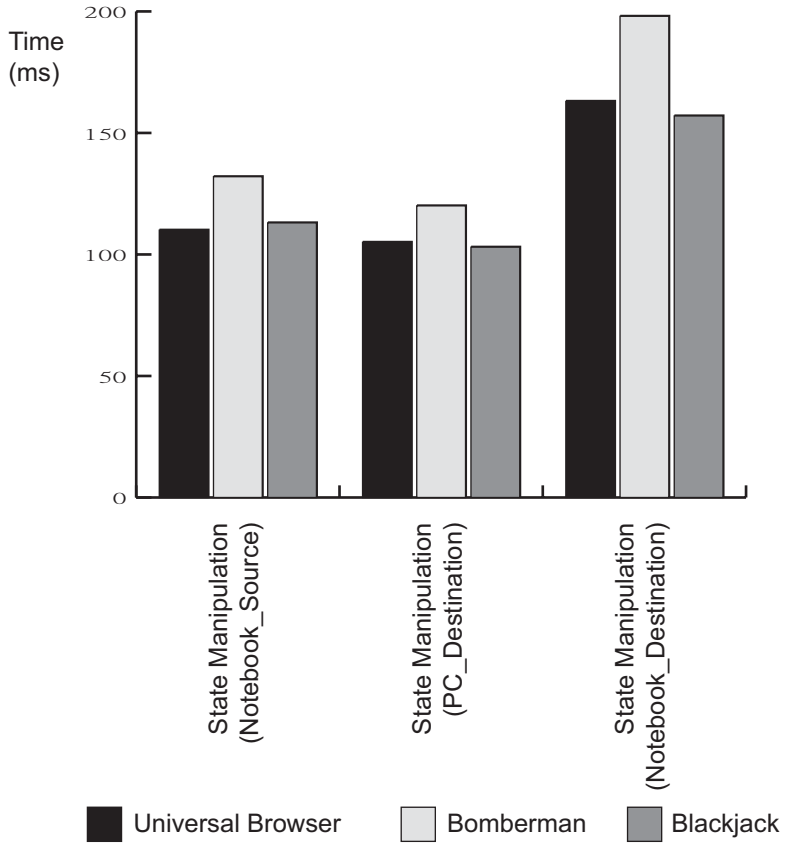

Figure 7.5: Time for the State Manipulation stage in different machines

As shown in Fig. 7.5, the time spent for processing the state is the greatest when the states are processed on PDA_Destination. This is reasonable since the PDA_Destination has the most limited resources. From the view of the processor and memory configuration, PDA_Destination and Notebook_Destination both have similar configuration. However, PDA_Destination, as a small handheld device, still

does not perform as efficent as a notebook device. On the other hand, the time spent using Notebook_Source and PC_Destination have no great difference as their resources are rich. This result is as expected and encourages us to process states in resource-rich deivces. To further stengthen this idea, we have done another experiment to measure the migration latencies under different situation on different devices. The result is depicted in table 7.3.

The measurement shows the total migration latency is the largest in Case C, when the states are processed on both Notebook_Source and PDA_Destination. This result is as expected since the states are processed twice while PDA_Destination is a resource-limited device. Under Case A, the system always spent much time to process the states twice. Another noticeable situation is Case G where the states are only processed on the PDA_Destination. The time spent under Case G is quite high while compared to Case D, Case E, and Case F, which also manipulated states on only one device. Case D is the same case shown in table 7.2, which processed the states at the source once only, and Case E and F are the cases processed the states at the destination only with a standard PC and notebook respectively. Although states are manipulated once only in Case G, the time spent is as much as Case A, which manipulates states twice on the source and destination devices. In this experiment, the result also encourages our flexible design to provide the choice on where to process the states. If we are migrating from a resource-rich device to a resource-limited one, for example, from Notebook_Source to PDA_Destination, the state manipulation should definitely be done on the source instead of the destination. Through comparing Case C and Case D, about 50

7.4 Discussion

From the experiments, we have shown that the introduction of context-aware state management would not induce a great degrade in performance, even a large number of context-aware rules is applied. The main concern is the retrieval of context

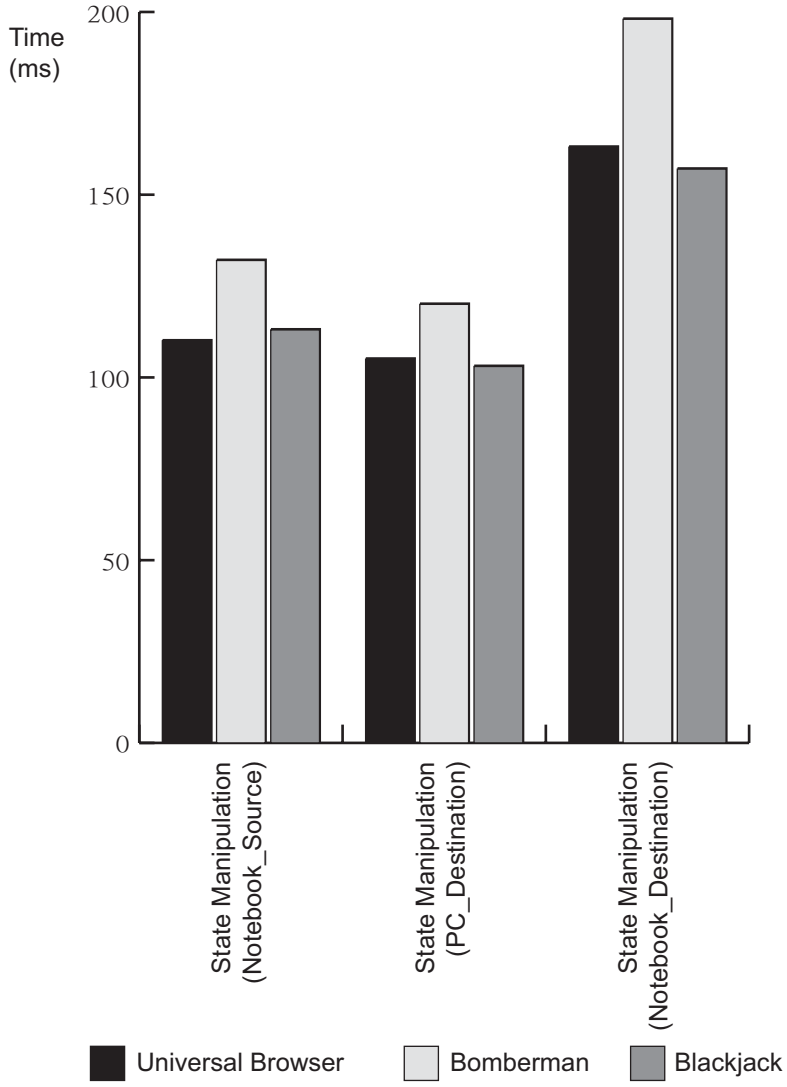

Table 7.3: The total migration latency of different applications when the states are manipulated on different devices. Case A: states are manipulated on both Notebook_Source and PC_Destination; Case B: on both Notebook_Source and Notebook_Destination; Case C: on Notebook_Source only; Case D: on PC_Destination only; Case E: on Notebook_Destination only; Case F: on Notebook_Destination only; and, Case G: on PDA_Destination only

information. Some techniques could be done to increase the performance. For example, in our system, we have already stored the context history. Instead of waiting for the most up-to-date context information, our system could try to make predictions from the context histories and process the states with these predicted one.

In our implementation, our main focus is on the mobility support, thus, the context-aware rules are hardcoded inside Sparkle. As a result, the feature of context-awareness could not be comprehensively evaluated. We believe that context-aware state management is a good way to support service mobility during migration, as it could allow state to be more suitable to the context in the destination, yet the time latency spent in state manipulation is limited.

7.5 Summary

In this chapter, we have presented our application, the Universal Browser, implemented under Sparkle. It is a context-aware and extensible browser that could download different facets under different situations. Users could have their own set of facets for information access. After that, we have done some performance analysis on the Universal Browser and our mobile functionality system. The results show that, although manipulation of state is introduced into the system to collaborate mobility and adaptability, the performance of system would not have a great increase of migration time latency. Therefore, our mobile functionality system is suitable to support mobility through context-aware state management.

Chapter 8

Conclusion

8.1 Summary

In pervasive computing, it is characterized by high mobility and invisibility. For mobility, users roam around, yet they want to continuously access their information. For invisibility, computing devices are embedded everywhere in pervasive environment. Users can use these resources without distraction. As a result, it is necessary to facilitate the use of computing service anywhere, anytime in a pervasive computing environment.

Researchers have done plenty of researches on supporting mobility through mobile code paradigms, and achieving invisibility through different kinds of adaptation methods. Normally, adaptation will be in a process after the migration is done. However, this procedure is not so suitable in a pervasive computing environment as they are not lightweight and flexible enough. Therefore, we propose to make collaboration between mobility and adaptability so as to achieve a lightweight migration and flexible adaptation.

In this thesis, we emphasize on the cooperation of mobility and adaptability. For mobility, we focus on the mobile functionality. This scheme is based on the mobile code paradigms, yet our scheme only moves function specification from the source to the destination instead of moving the code implementation as the current

mobile code paradigms. In our mobile functionality, code could be downloaded from third parties, such as proxies, when it is needed. For adaptability, we focus on two adaptations to achieve adaptability, that is the functionality adaptation and the state adaptation. Functionality adaptation is supported by the Sparkle client system. With functionality adaptation, different code implementations could be downloaded according to the context. Functionality adaptation could also facilitate our mobile functionality, which allows lightweight mobility. Secondly, with the state adaptation, states could be prepared and be adjusted in an application, so that they could be fit the destination context.

In most of the current mobility systems, adaptation is usually be done after the application is migrated to the destination device. This is not efficient and flexible enough. Hence, our mobile functionality system would like to prepare for context changes and do some adaptation before migration. State is the current snapshot inside an application, and context is the current status outside an application. Through managing the contexts and the states, we could modify the states before migration to suit the future context and prepare for adaptation.

As a result, we have developed a mobile functionality system with context-aware state management. With this system, we would like to achieve lightweight migration and flexible adaptability. The key feature of this mobile system is the state management scheme which could be divided into five stages: State Acquisition, State Manipulation (at the source), State Transmission, State Manipulation (at the destination) and State Restoration. The distinctive stage is the State Manipulation. It is challeging part to process the states according to context-aware rules, so that the states could suit the destination environment. The State Manipulation could be done either on the source site or the destination site depending on the context. This is a very flexible adaptation scheme, especially for the pervasive computing environment.

In addition, we have implemented a prototype of the mobile functionality sys-

tem. Some experiments have been carried out. The results of the experiments show that the total time of the five stages is about 1100 ms in average, while state manipulation stage only takes about 150ms on resource-rich deivce and about 300ms on resource-limited device. This is acceptable, as this does not induce great time latency. Moreover, the results also encourages us to process the states on resource-rich device to reduce time spent. This shows our designed mobile functionality system is suitable for context-aware state management in the pervasive computing environments.

8.2 Future Work

The mobile functionality system that has been implemented is only a simple prototype to demonstrate the mobility support with context-aware state management in pervasive computing environments. The prototype is only a minimal effort as a proof-of-concept. Hence, there are still rooms for improvements.

8.2.1 Migration checkpoint

In our current implementation, the facet is migrated at root level facets. This means we only migrate after the completion of root facets. However, for better migration, a better algorithm for checkpoint selection is needed. For example, the resource usage of current facets could be calculated to check if the current running lists of facets can all fit into the destination environment.

8.2.2 Context capturing

To simplify our implementation, only device context are captured. With the limitation of sensors, the environment context are not captured in real time, and the movement of people could not be capture. In the future, some sensors could be added into the system for collecting current environment context.

8.2.3 Context-awareness of state

In our current implementation, the context-aware rules are hardcoded in the mobile functionality system. In the future, policies could be maintained completely separated from the system implementation details. It could be expressed in a high-level of abstraction to simplify the modification by system administrators.

Moreover, these context-awareness rules should be also context-aware. Different sets of context rules could be chosen for State Manipulation depend on the current context. States, therefore, are manipulated differently under different context.

Finally, we conclude this dissertation. The collaboration of mobility and adaptability is a foundation to achieve lightweight migration and flexibility adaptation. Although many issues are needed to be solved for making it into a real practical system, the significance of the mobile functionality system is undeniable.

Bibliography

[1] Java agent development framework. http://jade.tilab.com.

[2] Ubisworld. http://www.u2m.org/UbisWorld/UbisWorldManager.php.

[3] Anurag Acharya, M. Ranganathan, and Joel Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222, pages 111–130. Springer-Verlag: Heidelberg, Germany, 1997.

[4] Nalini M. Belaramani. A component-based software system with functionality adaptation for mobile computing. Master's thesis, 2002.

[5] Paolo Bellavista, Antonio Corradi, and Rebecca Montanari. Context-aware middleware for resource management in the wireless internet. *IEEE Transactions on Software Engineering*, 29(12):1086–1099, 2003.

[6] Guanling Chen and David Kotz. A survey of context-aware mobile computing research. Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, November 2000.

[7] Harry Chen, Filip Perich, Timothy Finin, and Anupam Joshi. Soupa: Standard ontology for ubiquitous and pervasive applications. In *Proceedings of International Conference on Mobile and Ubiquitous Systems: Networking and Services*, pages 258–267, August 2004.

[8] Yuk Chow. A lightweight mobile code system for pervasive computing. Master's thesis, 2002.

[9] Eyal de Lara, Dan S. Wallach, and Willy Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, pages 159–170, March 2001.

[10] Anind K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5(1):4–7, February 2001.

[11] A. Finkelstein and A. Savigni. A framework for requirements engineering for context-aware services, 2001.

[12] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[13] Thomas R. Gruber. Towards principles for the design of ontologies used for knowledge sharing. In N. Guarino and R. Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*, Deventer, The Netherlands, 1993. Kluwer Academic Publishers.

[14] Lonnie harvel, Ling Liu, Gregory D. Abowd, Yu-Xi Lim, Chris Scheibe, and Chris Chatham. Context cube: Flexible and effective manipulation of sensed context data. In *Pervasive Computing: Second International Conference*, pages 51–68, April 2004.

[15] Hao hua Chu, Henry Song, Candy Wong, Shoji Kurakake, and Masaji Katagiri. Roam, a seamless application framework. 69(3):209–226, January 2004.

[16] J. L. V. Barbosa C. F. R. Geyer I. Augustin, A. C. Yamin. Isam, a software architecture for adaptive and distributed mobile applications. In *The Seventh IEEE Symposium on Computers and Communications (ISCC 2002).*

[17] E. P. Kasten and P. K. McKinley. Perimorph: Run-time composition and state management for adaptive systems.

[18] Randy H. Katz. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1:6–17, 1994.

[19] Vivian J. V. M. Kwan. A distributed proxy system for functionality adaptation in pervasive computing environments. Master's thesis, 2002.

[20] Tayeb Lemlouma and Nabil Layaida. Context-aware adaptation for mobile devices. In *Proceedings of the 2004 IEEE International Conference on Mobile Data Management*, pages 106–111, January 2004.

[21] David Linthicum. Semantic mapping, ontologies, and xml standards, April 2004.

[22] Radu Litiu and Atul Prakash. Dacia: A mobile component framework for building adaptive distributed applications. *ACM SIGOPS Operating Systems Review*, 35(2):31–42, 2001.

[23] Wai Yip Lum and Francis C. M. Lau. On balancing between transcoding overhead and spatial consumption in content adaptation. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, pages 239–250, September 2002.

[24] B. D. Noble and M. Satyanarayanan. A research status report on adaptation for mobile data access. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 24(4):10–15, December 1995.

[25] Natalya F. Noy and Deborah L. McGuinness. Ontology development 101: A guide to creating your first ontology. Technical Report KSL-01-05, Stanford Knowledge Systems Laboratory, March 2001.

[26] Bill Schilit and Marvin Theimer. Disseminating active map information to mobile hosts. *IEEE Network*, 8(5):22–32, 1994.

[27] Henning Schulzrinne and Elin Wedlund. Application-layer mobility using sip. *ACM SIGMOBILE Mobile Computing and Communications Review archive*, 4(3):47–57, July 2000.

[28] Markus Straber, Joachim Baumann, and Fritz Hohl. Mole - a java based mobile agent system. In *ECOOP '96 Workshop on Mobile Object Systems*, 1996.