# SEAMLESS MOBILITY IN UBIQUITOUS COMPUTING ENVIRONMENTS

A Thesis
Presented to
The Academic Faculty

by

**Xiang Song**

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August 2008

# SEAMLESS MOBILITY IN UBIQUITOUS COMPUTING ENVIRONMENTS

Approved by:

Dr. Umakishore Ramachandran,
Advisor
College of Computing
*Georgia Institute of Technology*

Dr. Ling Liu
College of Computing
*Georgia Institute of Technology*

Dr. Mustaque Ahamed
College of Computing
*Georgia Institute of Technology*

Dr. Sang-bum Suh
Virtualization Department
*Samsung Electronics, Korea*

Dr. Keith Edwards
College of Computing
*Georgia Institute of Technology*

Date Approved: 20 June 2008

*To Yiyi and my parents,*

*for all their love and support*

# ACKNOWLEDGEMENTS

I would like to thank all the people who help me and support me in this dissertation. Without them, this thesis work would not have been possible.

My greatest gratitude goes to my advisor, Dr. Umakishore Ramachandran for his consistent support and encouragement of my work. He is always inspiring and patient in the discussion with a smiling face that helps me think positively about all the obstacles on the way to this dissertation. It is my pleasure to work with him for all these years. Thank you, Kishore!

I would also like to thank all my other committee members, Dr. Mustaque Ahamed, Dr. Keith Edwards, Dr. Ling Liu and Dr. Sang-bum Suh for their interests and support on my thesis work. Their insightful comments help me significantly improve the quality and presentation of this dissertation. I also want to thank Samsung-Tech Advanced Technology center (STAR) for the support and funding for my dissertation.

I appreciate the support from Rajendra Kumar, my mentor in Hewlett Packard Research Lab when I was an intern. My dissertation idea comes from the project I did there and Raj gave a lot of insightful thought in that. Without my internship experience and Raj's kindly support, it is not possible for me to develop the dissertation idea initially. I also like to convey my appreciation to Phillip Hutto, the research scientist in our group who works with me on the first few projects in my Ph.D. career. I learned a lot of Phil on how to do research in academia as a Ph.D. student. Thanks Phil!

My lab mates made my work enjoyable and all my other friends made my life in Georgia Tech memorable. I appreciate the help from Sam Young and Jatin Kumar

for their help in some implementations of the system. I would also like to thank Dave Lillethun, David Hilley, Hasnain A. Mandviwala, Nova Ahmed, Junsuk Shin,Namgeun Jeong, Dushmanta Mohapatra, Rajnish Kumar, Pei Yin, Jianxin Wu, Howard Zhou, Ziru Zhu, Han Xu, Zhongtang Cai and Yi Zhang, for all their support and assistance. With your friendship, I could survive any hardship in my life.

My last but deepest thanks belongs to Yiyi Huang and my parents. My parents always stand behind me and support me in any circumstance. Their supports are always powerful and encouraging, which makes all my work worthwhile. I would give a special appreciation to Yiyi Huang, for all her exceptional kindness, caring and loves. It is you who always takes me out of the negative feelings and supports me to move forward.

# TABLE OF CONTENTS


DEDICATION . . . . . iii

ACKNOWLEDGEMENTS . . . . . iv

LIST OF TABLES . . . . . x

LIST OF FIGURES . . . . . xi

SUMMARY . . . . . xiii

I INTRODUCTION . . . . . 1

II PROBLEM STATEMENT . . . . . 5

2.1 Ubiquitous Service System . . . . . 5

2.2 Dimensions . . . . . 8

2.2.1 What to move . . . . . 8

2.2.2 Where to move . . . . . 10

2.2.3 When to move . . . . . 11

2.2.4 How to move . . . . . 11

2.3 Requirement for Seamless Mobility . . . . . 12

2.4 Approaches . . . . . 14

2.4.1 Service Level Virtualization . . . . . 15

2.4.2 Device Level Virtualization . . . . . 17

2.4.3 Comparisons of two approaches . . . . . 22

2.5 Service categories . . . . . 23

2.6 Goal and Contributions . . . . . 25

III RELATED WORK . . . . . 27

3.1 State Migration . . . . . 27

3.2 Capability Adaptation . . . . . 31

3.3 Summary . . . . . 33

IV SERVICE LEVEL VIRTUALIZATION . . . . . 35
4.1 Design choices . . . . . 36
4.2 MobiGO: a middleware level virtualization system . . . . . 37
4.2.1 Architecture . . . . . 37
4.2.2 Discovery and description of the services . . . . . 39
4.2.3 Choices of the user . . . . . 40
4.2.4 State Management . . . . . 42
4.2.5 Trust . . . . . 45
4.3 Supported Applications . . . . . 46
4.3.1 Characteristics of video service . . . . . 46
4.3.2 An application scenario . . . . . 47
4.4 MobiGo Implementation . . . . . 48
4.4.1 Soft State and Hard State Management . . . . . 48
4.4.2 Application Wrapper and Service Repository . . . . . 49
4.4.3 Location Selector . . . . . 49
4.4.4 Authentication and Discovery . . . . . 50
4.4.5 Communication Message Structure . . . . . 51
4.4.6 Sample User Interface . . . . . 53
4.4.7 UPnP Media Server and Render . . . . . 56
4.5 Performance of MobiGo . . . . . 56
4.5.1 Latency for I/O, soft and hard state migration . . . . . 57
4.5.2 Latency improvement from location selector for hard state migration . . . . . 58
4.5.3 Simulating different network conditions . . . . . 58
4.6 Summary . . . . . 61
V DEVICE LEVEL VIRTUALIZATION . . . . . 63
5.1 Device level virtualization requirements . . . . . 64
5.1.1 Device state migration . . . . . 64
5.1.2 Capability adaptation . . . . . 65

5.2 Design principles . . . . . 65
5.3 Base System . . . . . 67
5.3.1 Xen and virtual device . . . . . 67
5.3.2 Framebuffer device and virtual framebuffer device . . . . . 70
5.4 Design choices . . . . . 71
5.5 Chameleon Architecture . . . . . 75
5.5.1 Discovery . . . . . 77
5.5.2 Connection establishment . . . . . 78
5.5.3 Virtual device to physical device mapping . . . . . 78
5.5.4 Capability adaptation . . . . . 79
5.5.5 Data flow . . . . . 79
5.5.6 Device state migration . . . . . 81
5.6 Implementation . . . . . 81
5.6.1 Data structure for virtual device, physical resource and their mapping . . . . . 81
5.6.2 Capability adaptor interface . . . . . 84
5.6.3 Registry and match maker . . . . . 86
5.6.4 Framebuffer device - the backend . . . . . 87
5.6.5 Capability adaptation algorithm . . . . . 90
5.6.6 Frontend and its communication to the backend . . . . . 91
5.7 Performance . . . . . 92
5.7.1 Performance of capability adaptation . . . . . 93
5.7.2 Performance of the state migration . . . . . 97
5.7.3 VNC framebuffer vs. physical framebuffer . . . . . 99
5.8 Summary . . . . . 100
VI DISCUSSION . . . . . 102
6.1 Other category of services . . . . . 102
6.2 Discovery and authentication . . . . . 103
6.3 Guest domain migration for device level virtualization . . . . . 104

6.4 Security vs. mobility . . . . . 105
6.5 Interaction between two levels . . . . . 106
VII CONCLUSIONS AND FUTURE WORK . . . . . 107
VITA . . . . . 117

# LIST OF TABLES


1 Requirements for Seamless Mobility . . . . . 13

2 Virtualization levels . . . . . 27

3 Summary of related works with respect to state migration . . . . . 31

4 Fetching vs. carrying strategy in different scenarios . . . . . 42

5 Structure of Ubiquitous Virtual State . . . . . 48

6 MobiGo Command Parameters . . . . . 52

7 Comparison of different designs . . . . . 75

8 NVIDIA framebuffer format . . . . . 90

# LIST OF FIGURES


1 A Ubiquitous Service System . . . . . 6
2 An example of hard, soft and I/O states . . . . . 9
3 Virtualization at different layers . . . . . 15
4 Scenario 1: multiple devices connected to a single environment . . . 18
5 Scenario 2: migration of entire operating system . . . . . 19
6 Architecture for MobiGo . . . . . 38
7 I/O Migration . . . . . 43
8 Soft State Migration . . . . . 44
9 Hard State Migration . . . . . 45
10 Hard State Migration . . . . . 51
11 UI: Initial Screen . . . . . 53
12 UI: Services are discovered . . . . . 54
13 UI: Service launched . . . . . 55
14 UI: Switching I/O or service . . . . . 55
15 Switching latency for I/O, soft and hard state migration . . . . . 57
16 Latency improvement from location selector . . . . . 59
17 Migration latency affected by network bandwidth (for different sizes) 60
18 Xen Architecture . . . . . 68
19 Communications in Xen Split Device Driver Model . . . . . 69
20 Design 1:changes in FE and BE, no change in VMM . . . . . 72
21 Design 2:No BE and FE changes, CA in VMM . . . . . 73
22 Design 3: No changes in FE and VMM, CA in host domain . . . . . 74
23 Chameleon System Architecture . . . . . 76
24 Data flow in Chameleon . . . . . 80
25 Performance of Chameleon and Original Xen . . . . . 94
26 Additional memory copy for Chameleon . . . . . 96
27 Time for copying different sizes of framebuffers . . . . . 97

28 Device state migration costs . . . . . 98
29 VNC framebuffer vs. Physical framebuffer . . . . . 100

# SUMMARY

Nominally, one can expect any user of modern technology to at least carry a handheld device of the class of an iPAQ (perhaps in the form of a cellphone). The availability of technology in the environment (home, office, public spaces) also continues to grow at an amazing pace. With advances in technology, it is feasible to remain connected and enjoy services that we care about, be it entertainment, sports, or plain work, anytime anywhere. We need a system that supports seamless migration of services from handhelds to the environment (or vice versa) and between environments. Virtualization technology is able to support such a migration by providing a common virtualized interface at both source and destination.

In this dissertation, we focus on two levels of virtualization to address issues for seamless mobility. We first identify three different kinds of spaces and three axes to support mobility in these spaces. Then we present two systems that address these dimensions from different perspectives. For service level virtualization, we have built a system called MobiGo that can capture the application states and restore the service execution with saved states at the destination platform. It provides the architectural elements for efficiently managing different states in the different spaces. Evaluation suggests that the overhead of the system is relatively small and meets user's expectation. Service level virtualization has certain limitations, specifically when the application state and device state are not visible to the middleware. On the other hand, for device level virtualization, Chameleon is a Xen-like system level virtualization system to support device level migration and automatic capability adaptation at the operating system level. Chameleon is able to capture and restore device states

and automatically accommodate the heterogeneity of devices to provide the migration of services. Device level virtualization can address some issues that cannot be addressed in service level virtualization. It also has less requirements than service level virtualization in order to be applied to existing systems. Through performance measurements, we demonstrate that Chameleon introduces minimal overhead while providing capability adaptation and device state migration for seamless mobility in ubiquitous computing environments.

# CHAPTER I

# INTRODUCTION

With the rapid advance in technology, it is becoming increasingly feasible for people to take advantage of the devices and services in the environment to remain "connected" and continuously enjoy the activity they are engaged in, be it sports, entertainment, or work. Examples include stock tickers and sports highlights streamed to cellphones, and electronic tour guides in museums and art galleries that provide exhaustive information about artifacts [14]. We believe that in the near future there will be many more devices and services available for use in office buildings, squares, parks and other publicly accessible places.

The focus of our work is to provide seamless mobility in such a ubiquitous computing environment. For example, a user who is listening to music in her iPod should be able to continue listening to it in the sound system available in the environment that she walked into if she so chooses. Similarly, the sports highlights streaming to one's cellphone should be displayed on a high quality display that became available in the environment. There are three properties that a user would expect for such migration of his/her activity to the environment that is being done with the intent of increasing the user experience. First and foremost, the experience should be seamless, i.e., there should be no discontinuity in the user experience. Second, the user should have control over the mobility (if so desired). Third, the user should be able to trust the environment.

These properties require the entire migration of user's states and activities at all levels across heterogeneous platforms in a controllable and secure way. Such migration assumes heterogenous source and destination platforms can understand each other

and have common facilities available. Virtualization technology can easily support interoperability by providing a common interface on top of heterogeneous platforms. Such a common interface can be at different levels. Service level virtualization hides the details of the service instantiation (launching a new application, fetching files, redirecting I/O, etc.) from the user, and provides users with *functionalities* (such as a video player) that become available in the environment utilizing the concrete platform specific services (such as *MediaPlayer* on Windows, and *mplayer* on Linux). Device level virtualization provides a common virtual machine under the operating system and can host a variety of devices by providing automatic device capability adaptation for each device category (input device, display device etc.). Again, the goal of these virtualization approaches is to provide the continuity of services when a mobile user moves from one environment to the other.

There is much work in supporting the aforementioned mobility in ubiquitous setting including dynamic discovery of devices and services by protocol families such as Bluetooth [16], operating system support for virtualization such as Xen [17], and whole system virtualization such as Microsoft's desktop on a memory stick [6]. Classic work in process migration in distributed systems [18] is also related to the problem being addressed in this dissertation. Device level migration and capability adaptation has also been discussed in several projects such as device-capability-on-demand (DCOD) framework [35] and DCC (Dynamic Composable Computing) [40]. However, we are not aware of any work that provides comprehensive frameworks to address issues of seamless mobility at different levels to meet users' needs.

In this dissertation, we focus on two level of virtualization approaches to address issues for seamless mobility: service level virtualization and device level virtualization. We first identify three different kinds of spaces, self-owned, familiar, and totally-new, and three axes to support mobility, namely, hard state, soft state, and I/O state in

these spaces. We consider these spaces and states as critical dimensions for the seamless mobility problem in ubiquitous computing environment. After presenting these dimensions, we present two systems that address these dimensions at different levels. For service level virtualization, we have built a system called MobiGo that can restore the service execution with captured application states at the destination platform. It provides the architectural elements for efficiently managing different types of states in the different types of spaces. Through evaluation, we demonstrate that the overhead of the system meets user's expectation. However, service level virtualization has its limitations, especially when the application state and device state are not visible to the middleware. Therefore, we have another level of virtualization, the device level virtualization to accommodate these limitations. For device level virtualization, we have built Chameleon on top of Xen to support device level migration and automatic capability adaptation at the operating system level. Chameleon is able to capture and restore device states and automatically accommodate the heterogeneity of devices to provide the migration of services. As a device level virtualization system, Chameleon can address some issues that cannot be addressed in MobiGo such as device state migration and capability adaptation. It also has less requirements than MobiGo (such as application hooks required in MobiGo to control applications) in order to be applied to existing systems. Through performance measurements, we demonstrate that Chameleon introduces minimal overhead while providing capability adaptation and device state migration for seamless mobility in ubiquitous computing environments.

In the rest of this dissertation, we first describe the problem of seamless mobility and present different dimensions of the problem. Then we survey some related work in this space. After that, we present the two systems we have built to address the issues for seamless mobility at different levels: MobiGo at service level and Chameleon at device level. We then describe each system in detail, including the system architecture, implementation, performance evaluations. Finally, we draw some conclusions

and present possible future work in the last chapter.

# CHAPTER II

# PROBLEM STATEMENT

In this chapter, we will describe our target problem in detail. We first define the ubiquitous service system in which seamless mobility becomes an important issue. Then we present several dimensions of the seamless mobility problem that need to be addressed. Finally, we describe the requirements for a successful seamless mobility system that can fully address all issues in a ubiquitous service system.

## *2.1 Ubiquitous Service System*

With the advances in computing technology, we can imagine in the near future, people will rely on their mobile platforms for various daily activities, such as watching movies, sending emails or editing presentations. They will also take advantage of the surrounding resources (including hardware resources and software services) to help enhance the experience. For example, Alice can switch from her iPod to her in-car stereo system to continuously listen to her favorite music when she sits in the car. A tourist can use the airport-provided language translation service to talk to people for help. We call such a system a *ubiquitous service system*. As Figure 1 shows, a ubiquitous service system consists of two components: an environment and a mobile platform. Users carry the mobile platform, such as a PDA or a cellphone, and enter/leave environments on the go. Whereas the mobile platform may already have some resources installed, the environment can always provide more resources that are impossible to carry (such as a big screen), or not available on the devices (such as language translation). The mobile platform could be an iPAQ, a cellphone, a laptop computer or whatever device a user is comfortable carrying. A typical environment is an office building, an airport lounge or a hotel room where additional resources and

services are available for use.

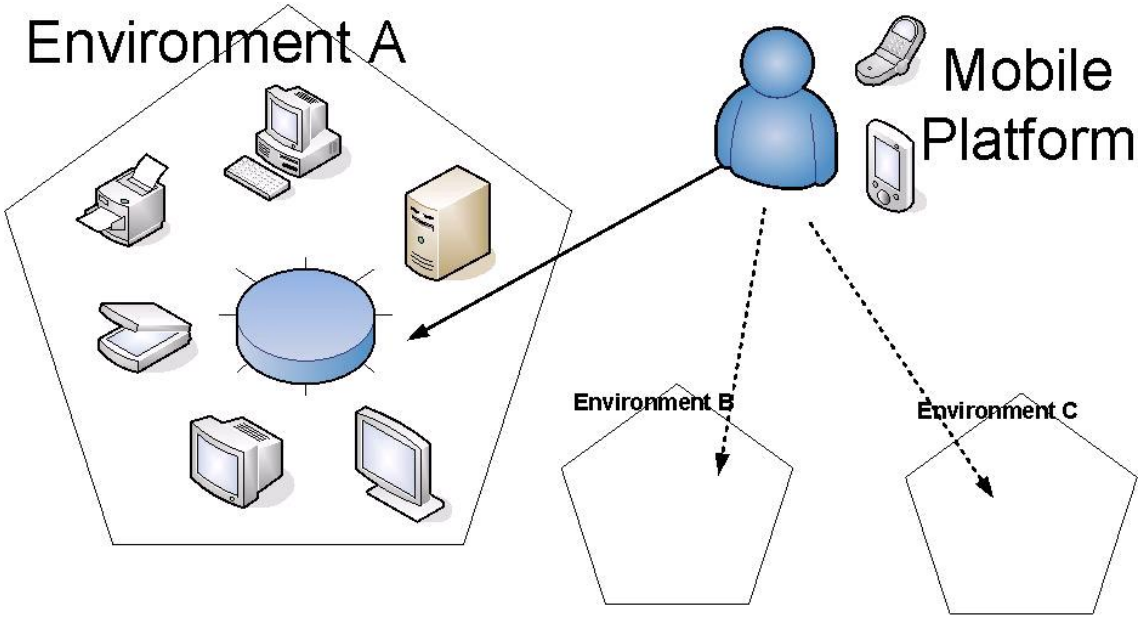


**Figure 1:** A Ubiquitous Service System

One critical problem in such a ubiquitous service system is how to provide the continuity of services to users when they move around. To migrate a service from one platform to another without discontinuity, the system has to find a common way understandable by both the source and destination. Virtualization technology provides such a common interface for the migration by establishing a homogeneous virtualized platform on top of heterogeneous devices/services.

A service in such a system typically consists of three parts: (1) software applications, (2) input and output devices, and (3) user specific files stored on disks. Each of these parts can be either on the mobile platform or in the environment as necessary in order to enhance a user's experience in the best possible way. They also should be migrated on the "go" as users move around. For example, when Bob is watching a movie at home, all three parts may be located in his home environment. However, when he suddenly gets a call from the company and is asked to fly to another city to see a customer, he can "move" his movie from his home environment to his mobile

platform and continuously watch the movie on his way to the airport. When he sits in the airport lounge waiting for his flight, Bob can move the I/O to a nearby high resolution display (the movie player and the movie file still remain on his mobile platform) and enjoy the movie on the big screen (I/O is migrated). If the airport wireless network happens to be too slow to stream the movie from the mobile platform to the environment, the system (or Bob himself) may choose to use a wired network that is available in the airport environment to stream the movie from a website where he bought the movie. In this way, the remote website becomes part of the environment currently servicing Bob's request.

Software applications in ubiquitous services systems are heterogeneous in two ways: (1) different applications may support different protocol families for communication, and (2) different applications may provide same functionality to users (e.g. QuickTime player and Realplayer both provide movie playing service). A ubiquitous service system should be able to switch from one application in an environment to another similar application (i.e., with the same functionality but perhaps speaking a different protocol) in another environment. In such a way, the system provides a *virtualized* service to user across heterogeneous platforms. Such a virtualized service provides a functionality to user by utilizing a local available service in user's current environment. Such *service level* virtualization is crucial to enhance the user's experience.

Similarly, input and output devices in a ubiquitous computing environment are also heterogeneous. Different I/O devices may have same functionalities but different capabilities. For example, the display on an iPAQ has the same functionality as a large video wall, except for the difference in resolution and quality. Devices in each category (such as display devices) should be interchangeable regardless of their capability. A ubiquitous service migration system should be able to switch from one device to the other and dynamically adapt to the capabilities of individual devices.

In summary, seamless mobility in ubiquitous computing environment means (1) mobile users can take advantage of environmental resources to enhance their experiences, and (2) mobile users can migrate their activities from one environment to the other without any interruption of the services. Virtualization technology can achieve these two goals by providing homogeneous interface on top of heterogeneous platforms. We do need different levels of virtualization to address heterogeneity problems from different perspectives.

## *2.2 Dimensions*

In order to provide seamless mobility of user's activities, the user *states* have to be migrated across the above-mentioned heterogeneous platforms. By user states, we mean the user specific information that is necessary for resuming the activities user did previously. State migration is the key problem for seamless mobility and consists of four dimensions: what to move, where to move, when to move and how to move. We will explore these different dimensions in this section.

### 2.2.1 What to move

Seamless mobility boils down to migrating the dynamic state of a service from one platform to another. We recognize three categories of state associated with a service: *hard state*, *soft state*, and *I/O state*.

Hard state refers to data or preferences stored in persistent storage (such as files or databases). It can be part of the mobile platform that a user is carrying with him (e.g., music stored in one's iPod, or files in an Intel Personal Server [20]). However, such a strategy may not work very well for more demanding services such as video wherein a single movie may be several hundreds of megabytes. Therefore a comprehensive solution for migrating hard state in a ubiquitous setting should allow for the content being served from a repository on the Internet (such as a Web server or a file server).

Soft state refers to the volatile state of the service currently being accessed by

the user. An example of soft state is the current scene of the movie being viewed, or the pause point of a movie, when a user stops a movie at one location and wants to resume it at another location.

I/O state, as the name suggests, refers to the current state of the input and output devices the service uses. Clearly, from the user's perspective, I/O state migration is the most impactful since it directly affects the user experience. The overall user experience depends on how well the three categories of state are managed by the infrastructure that provides seamless mobility.

Figure 2 shows an example of these states in the context of a movie service. The hard state in this case refers to the actual movie file (typically hundreds of megabytes in size) stored on persistent storage such as disks or tapes. Soft state refers to the movie player's internal states while the movie is being played. Soft state includes current position of the movie, user's preference in brightness or contrast, and maybe a play list that the user defined. These states can be recovered if lost. However, migrating these states can improve user experience since user doesn't need to recover them manually at the destination. I/O state in this case refers to the raw video stream that the movie player outputs to the actual display device for rendering. Some of I/O states may reach the display device but have not been played due to the timing issues. These states also need to be migrated as a user moves.

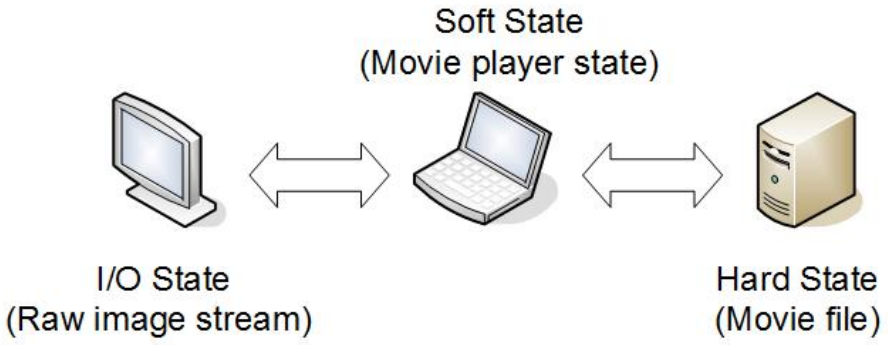


**Figure 2:** An example of hard, soft and I/O states

### 2.2.2 Where to move

The motivation for seamless mobility arises from the fundamental premise that the ambient environment may have more resources to enhance the user experience than the mobile platform (such as a cellphone or handheld) carried by the user. Since the environment plays a key role in seamless mobility, we need to understand the characteristics of the environment in order to fully understand how to achieve seamless mobility. We classify the ambient environment into three groups: self-owned space, totally-new space and familiar space.

In a self-owned space, a user has full control over the environment. The hardware and software configuration and settings can be changed at will. An example ubiquitous service in such a space is video watching: the video will seamlessly move from the TV screen in the living room to the one in the bedroom when the user moves from one to the other. The service itself can be fully customized and the user can specify the service policy such as whether the movement of the video should be implicit with her movement or explicit.

In a totally-new space, a user has no control over the environment or the specific services provided. In such a scenario, dynamically discovering new services and security of the environment becomes important. Building on the video service example, the environment (say a rest area along an Interstate highway) may offer a VCR service on a large display. The user will be able to discover this service and avail it for watching his favorite movie from the point where he left off prior to starting on the trip.

The familiar space is in between the above two categories. An example would be a preferred traveller's lounge in an airport or a hotel where the user has stayed previously. The user may be familiar with most of the services available in this environment but does not have control over the environment as she does in her own home. On the other hand, the environment may allow customization of the services

that it offers knowing the preferences of the user who has just arrived.

Depending on the space into which migration is desired, the requirements on the infrastructure for seamless mobility is likely different.

### 2.2.3 When to move

The third dimension to seamless mobility has to do with when to migrate the service from one platform to another. The choice is either explicit under user control or implicit on recognizing some user cues. The choice is quite intimately tied to the "where" question addressed in the previous subsection. For example, in a self-owned space, a user may desire the migration to be implicit (such as based on location information of the user). On the other hand, irrespective of the space she is in, the user may want explicit control on when to migrate. For example, a salesman may want to show a short video that is on his office server at a customer site. In this case, he may want to explicitly control the state of the video service available in the environment according to his presentation style.

### 2.2.4 How to move

In order to launch a service in a specific platform, all the three states (hard, soft and I/O) have to be present in the local environment and have to be connected to perform the service. It comes to the question of how those states can be transferred to the target environment. One straightforward way may be using the mobile device to store all the state information. On the other hand, users may only carry minimum information, such as a globally unique user ID, and expect the local environment to retrieve the state from the previous environment based on this unique ID. Since the goal of seamless mobility is to provide continuity of service for users in the best possible manner, a good ubiquitous service system should dynamically decide which is the best way, in term of user experience, to fetch the states necessary to launch the service.

## *2.3 Requirement for Seamless Mobility*

Based on the above discussion on the dimensions of seamless mobility, we can summarize some general requirements for a ubiquitous service system:

- We need rapid discovery of available services in the environment.
- We need an efficient management of the different states associated with a service. It is important that the state management results in maximizing the user experience.
- We should give the choice of explicit or implicit control of the mobility to the user.
- We should build an intuitive model of trust that allows the user and the environment to respect the privacy of the user and ensure the integrity of the environment in the presence of malicious or unintended attacks on the infrastructure.

Note that these requirements may or may not be critical in some spaces. Table 1 shows the importance of the various requirements in different spaces. For example, in a self-owned space, the discovery is not quite important since everything is under user's control and the mobile platform knows exactly what are available in the environment. However, in a totally-new space, discovery is crucial because a new entrant can almost do nothing before he/she discovers what services are available. In familiar space, discovery is important but not as critical as in totally-new space. Since user always knows some services in familiar space, he/she can always utilize services that he/she is aware of previously without discovery. However, when new services are introduced or old services are removed, user may need discovery for these updates.

Similarly, in self-owned space, user always has a local copy of hard state, which makes the hard state management less important than other spaces. However, soft

**Table 1:** Requirements for Seamless Mobility

| | Discovery | State Management | | | State Migration | | Trust |
|---|---|---|---|---|---|---|---|
| | | Hard | Soft | I/O | Explicit | Implicit | |
| Self-owned | ❍ | ❍ | ● | ● | ❍ | ● | ❍ |
| Familiar | ◗ | ● | ● | ● | ● | ◗ | ◗ |
| Totally-new | ● | ● | ● | ● | ● | ❍ | ● |

Legend: ❍: less important ◗: mid important ●: more important

state and I/O state management are still important in self-owned space since when user is moving inside the self-owned space, soft state and I/O state still need to be migrated. All the three state managements are necessary for the other two spaces, since user may need to migrate all the three types of states in these spaces.

In self-owned space, since user knows everything in the space, he/she may always want the migration to be implicit since it gives him/her the most convenience as he/she does not need to issue commands for the migration and the system can automatically move the service with his/her movement. For example, consider a user sitting in her living room watching a movie. When she goes to kitchen to grab a drink, she wants her movie to be "automatically" migrated to her kitchen's monitor without her explicit command. However, in the other two spaces, an explicit command may be necessary since the user is unfamiliar with the environment. User may not want, for example, his unfinished slides to be "automatically" migrated to the big display equipped in a hallway when he walks from his office to his boss's office. Such migration needs to be explicitly done upon user's request. Normally, implicit migration is not desired in a totally-new space for privacy reasons since the user may be uncertain about such a space.

As a user always trusts the self-owned space most, the need for trust management in self-owned space is minimal. However, in familiar and totally-new space, such a need is very critical since user wants to ensure his/her data is not compromised

and the environment does the right thing. A well-established trust management subsystem is crucial to a successful ubiquitous service system since security is one of the main concerns of mobile users while trying to utilize environmental resources.

## *2.4 Approaches*

In this dissertation work, we plan to use virtualization technology at both service/middleware level and device/hardware level to address the problem for seamless mobility. In short, we call them service level virtualization (SLV) and device level virtualization (DLV), respectively. The ultimate goal of both approaches is to provide the seamless migration of a user's activities across heterogeneous software and devices in ubiquitous environments.

Figure 3 shows the locations of the two virtualization layers in the traditional software stack. In a typical ubiquitous computing environment, a variety of applications, operating system and devices are available. However, some applications and/or devices provide similar functionalities to users. For example, Windows Media Player and Real Player both provide movie playing services; Small displays and video walls can both render video streams. In order to enable users to move across these similar application services or devices, virtualization may be placed at two levels: a middleware/service level virtualization can be placed between applications and operating systems to allow different applications to be able to provide a virtualized service to end users; a device/hardware virtualization can be placed between operating systems and physical devices to provide virtualized devices on top of heterogeneous physical devices to operating systems. In this section, we describe these two approaches in detail and present the contribution of this thesis work.

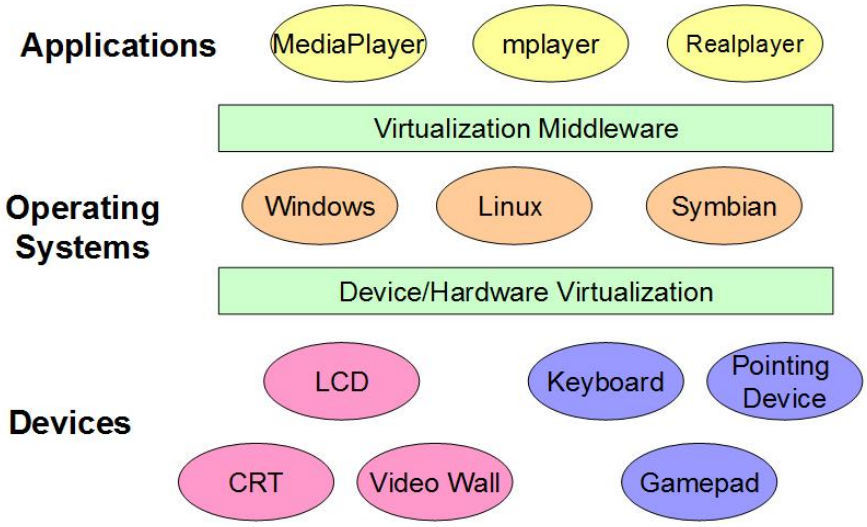


**Figure 3:** Virtualization at different layers

### 2.4.1 Service Level Virtualization

#### *2.4.1.1 What is service level virtualization*

A service level virtualization system migrates user activities at middleware/application level. Such a system assumes both the source and destination systems have their own system stack, including applications, operating systems and devices. However, these components in the system stack may be totally different at the two ends of the migration. Service level virtualization system runs at both the source and the destination to enable these heterogenous components to communicate with each other correspondingly and ensures the proper delivery of the states. Using service level virtualization system, users have a unique view of a category of service (e.g., movie playing service) without worrying about the details of which application the system uses to provide such a service. For example, at the source, the system may use a Windows Mediaplayer to play a movie for the user, while at the destination the system may use an mplayer on Linux if it is the only movie player available there.

The key problem of service level virtualization is to define a common state structure that is understandable by a category of service, and then properly migrate such

a state structure across heterogeneous platforms to ensure the continuity of services. There are also several other issues, such as discovery and security, that need to be addressed in order to build a complete system for the migration. We will also present how we address these issues later.

#### *2.4.1.2 Why service level virtualization*

Service level virtualization is good because it is built on top of existing system stack and doesn't need to change legacy systems to enable the migration. What is needed is the installation of the service virtualization system on both the source and the destination and proper configuration of these systems. Therefore, service level virtualization system can be deployed very easily and used in a variety of environments.

However, as we will see later in a later chapter, service level virtualization system requires special interface that applications have to provide in order to control these applications. Through the interface, service level virtualization system can issue commands to the application to do certain operations programmatically (without user's involvement). Since applications are used off-the-shelf and not built on top of the service level virtualization system, such an interface is crucial for the control of the applications.

Fortunately, most applications in common use today provide such an interface. Some applications have Software Development Kit (SDK) or Application Program Interface (API) that can be used to reach into the internals of the application. Windows Mediaplayer is an example of such an application. Other applications provide control knobs for the internal state via external input (via a keyboard for example). By using a simulated keyboard, we can easily use such an interface to make these applications "controllable".

In summary, service level virtualization system is easy to be deployed and used on

top of existing system stack without any change to the application or the OS available on the platforms. It requires special application interface but many applications have such an interface available. In the next section, we will introduce our service level virtualization system, called MobiGO, that targets the fundamental problems for seamless mobility at middleware level.

### 2.4.2 Device Level Virtualization

#### *2.4.2.1 What is device level virtualization*

Device level virtualization can provide virtualized devices to operating systems to support seamless mobility at a lower level than middleware level virtualization. This level of virtualization presents a uniform view of a category of devices to operating systems to work on. Internally, device level virtualization system maps virtualized devices to physical devices to do actual device functionality. Device level virtualization may also switch the mappings from one physical device to another physical device to enable the device state migration. Operating systems are not aware of such a mapping or switching.

Device level virtualization can be used in two types of seamless mobility scenarios. First, when multiple input/output devices are available on the same machine and the switch from one device to the other is needed, device level virtualization can achieve this by changing the mappings internally without making the application aware of such a lower level change. Such a device mapping and switching can be applied to any type of devices such as display devices, audio devices and/or input devices. Take a display device switching as an example: in a home environment where different monitors/displays are equipped in different rooms but connected to a single computer, when a user is moving from his bedroom to kitchen, his movie is also moved from the bedroom display to the monitor in the kitchen to provide continuous movie service to him. Figure 4 shows an example of this scenario, in which two display devices, a LCD monitor and a video wall, are available in the same environment.

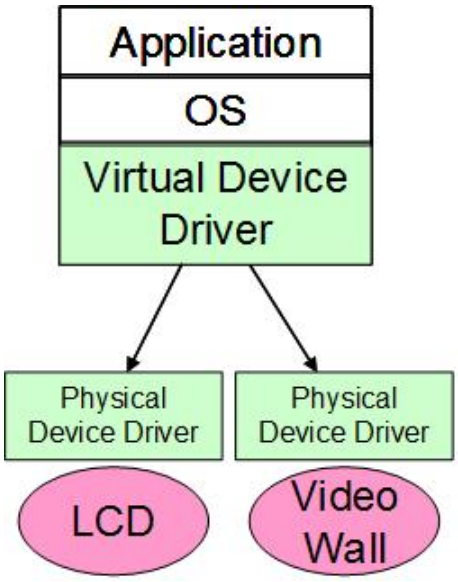


**Figure 4:** Scenario 1: multiple devices connected to a single environment

A second scenario of device level virtualization is the migration of the entire operating system image (including the operating system and the applications) from one environment to another. In such a scenario, local devices available in the new environment may be different from those in the old environment. In order to resume the operating system image in the new environment without changing the OS image itself, devices in both environments should be virtualized and should present a uniform view to the operating system. However, when the OS image is moved, the internal mappings of virtualized devices to physical devices are changed. The OS is not aware of such a change and is still using the virtualized devices as in the old environment. But the virtualized devices have already been mapped to the new devices so the input/output are redirected to the local devices in the new environment. Figure 5 shows this scenario. Note that this scenario requires the operating system to be small in order to be migrated in a small amount of time. Typical operating systems on handheld devices meet this requirement. Large complex operating systems that usually run on desktop machines may not be suitable for such a migration.

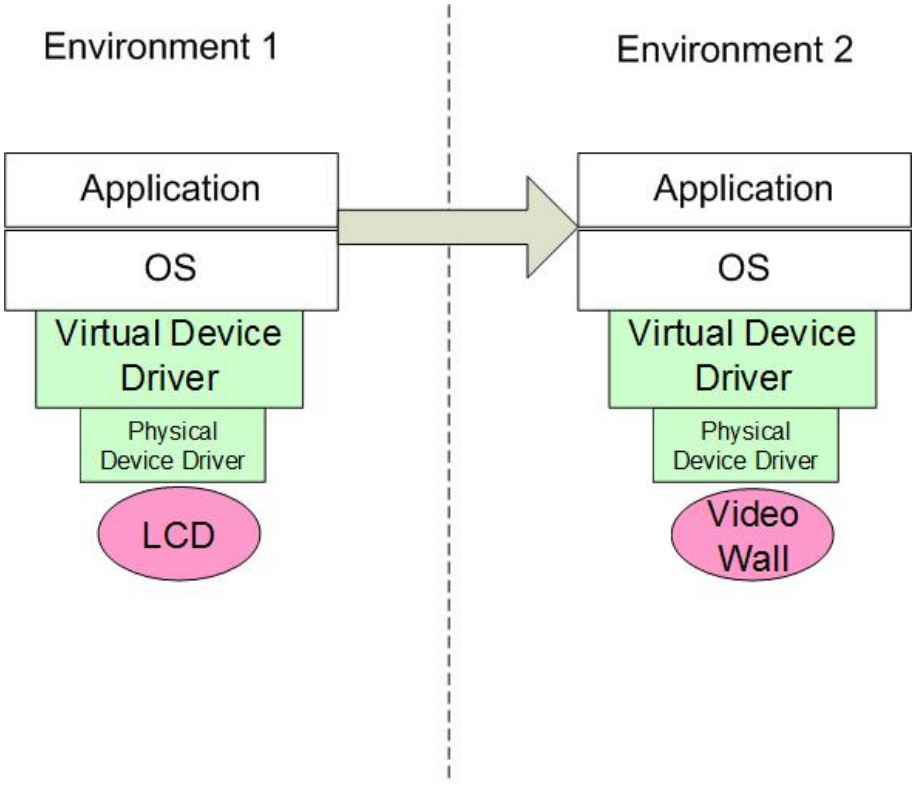


**Figure 5:** Scenario 2: migration of entire operating system

#### *2.4.2.2 Why device level virtualization*

Similar to service level virtualization, device level virtualization provides continuous service to mobile users as we stated at the beginning of this dissertation. However, device level virtualization can solve some fundamental problems at device level, which cannot be addressed at any upper layer (middleware layer or application layer). In this section, we present the reasons why we need device level virtualization even though middleware level virtualization has similar functionalities. We will see that device level virtualization can be complementary to middleware level virtualization to fully solve the seamless mobility problem.

**Removal of the need for applications hooks.** The first advantage of device level virtualization over middleware level virtualization is that the former approach does not require any application level hooks to provide seamless mobility. As we will

see later in this dissertation, application wrappers in middleware level virtualization require applications to provide some type of interface in order to allow the application wrappers to be able to "control" them. For example, application wrappers should be able to issue *play* or *stop* command to the movie players in order to control the playback without a user's involvement. Such interface (or application hooks) has to be provided by applications so that they can be integrated into MobiGo, the middleware level virtualization system. Otherwise, if application wrappers have no such interface to communicate to applications internally, MobiGo will not be able to interact with the applications and migrate their internal states.

In contrast to middleware level virtualization, device level virtualization does not require such an interface (application hooks) to be available. Device level virtualization considers applications and the operating system as a whole "upper layer image". This image uses virtualized devices as a black box provided by the device level virtualization system. Internally, device level virtualization re-maps virtualized devices to physical devices to enable the migration. Therefore, in device level virtualization, application requirements are removed and seamless migration becomes possible transparent to the "upper layer image", namely the applications and the operating system.

**Device manipulation.** The second reason why we need device level virtualization is that lower level state can be migrated along with application state. Lower level state refer to device specific state. Such state includes buffers in the device driver of an I/O device, detailed configuration of devices and device specifications (e.g., resolution of the screen). Such state is device specific and not always available to upper layer applications. Therefore, middleware level migration system may not have access to migrate this state along with application state, causing this state to be lost in the migration process.

Device level virtualization, on the other hand, is able to manipulate devices when mapping virtualized devices to physical devices. Hence it can dump low level device state and configurations, and then migrate this device specific state in one of the two ways corresponding to the two scenarios we mentioned earlier. For example, it can migrate the state from one video device to the other on the same machine to "switch" the display, or it can migrate device state with the operating system image to fully migrate the entire system. In either way, device level virtualization can help the state migration at a lower level, the device level, to fully address the migration issues for seamless mobility.

**Dealing with heterogeneity of devices.** The third reason for using device level virtualization is to deal with the heterogeneity of devices. In ubiquitous computing environments, devices can vary considerably from one environment to the other. When a user is moving from one environment to the other, the devices he/she uses are likely to be different in size, resolution (for output devices) or key mappings (for input devices). However, we do not want such heterogeneity to be an inconvenience to the user. Therefore, we need to consider the differences in the devices and bridge the gap between them to provide the same service to a user despite such heterogeneity.

Service level virtualization relies on the applications themselves to deal with device heterogeneity. This level of virtualization requires applications to be aware of physical devices and adapt to the device changes during migration. However, such an approach requires application programmers to have physical devices in mind when developing applications, which adds burden to the programmers.

Device level virtualization can deal with heterogeneity of devices at a lower level to remove the burden from application programmers. When switching the mapping between virtualized devices to physical devices, device level virtualization automatically receives the information of both source and destination devices, and therefore

can do some adaptation/conversion when re-building the mapping. Applications need not be aware of such adaptations and do not need to build any special support for the adaptations. Hence device level virtualization can deal with device heterogeneity more efficiently and transparently.

### 2.4.3 Comparisons of two approaches

Service level virtualization and device level virtualization address the problem of seamless mobility from different prospectives. Service level virtualization provides virtualized service to end users and enables minimal state migration for each service category across heterogeneous platforms. It is a user-oriented approach at a level above the operating system. On the other hand, device level virtualization assumes the entire OS and application image to be available at the destination in order to do capability adaptation for the target platform. It is a system level approach at a level below the operating system.

These two approaches both have pros and cons. Service level virtualization does not require the entire application and OS image available at the destination platform and only needs minimal state for a virtualized service to be migrated. Therefore, the state in service level virtualization is light-weight and easy to maintain. Another advantage of service level virtualization is that it does not require a virtual machine manager to be available at all environments. It can be built on top of existing operating system stack as shown later in this dissertation. However, service level virtualization cannot provide device state migration as it cannot reach into the device state from middleware level. Also, service level virtualization is normally not aware of the device capabilities and therefore cannot adapt different capabilities when a user moves into an environment with a different set of devices with different capabilities.

Device level virtualization, on the other hand, can solve the aforementioned problems very easily. It can operate the device directly and migrate device specific states

and do capability adaptation based on the source and destination device capabilities. However, a drawback of device level virtualization is that it assumes the entire operating system and application image to be available at destination. In some cases, it might be difficult to migrate such an image due to its size (a typical Linux OS and application might be a few gigabytes). Compared to minimal state migration in service level, the entire OS and application image in device level virtualization is heavy-weight and hard to maintain. Another drawback of device level virtualization is that if the application state consists of some server state that is out of the control of the OS image, device level virtualization cannot fully migrate the user activities. These issues can be relatively easy to address at service level.

In summary, service level virtualization and device level virtualization address different issues for seamless mobility. They both have pros and cons individually but the combination of these two approaches can help us address more issues and meet all the requirements we identified in the previous sections. We know that our solution may not be able to solve all the problems in this space. We will discuss the limitations and future extension of this work in Chapter 6.

## *2.5 Service categories*

A number of service that people use on a daily basis demand seamless mobility. Examples of such services include email, web browsing, video playing and music listening. Different services have different characteristics. The mobility of some type of services can be easily supported by service level virtualization and some others can be supported by device level virtualization. In this section, we will discuss categories of services in this space and the features of each category. We will show in this section what type of service can be supported by our virtualization.

One category of service is stand-alone applications running on desktop machines.

Such applications include text editors and single machine games. This type of applications have all their state saved on the local machine and always do not require a network connection to operate. When a user moves from one environment to the other, all the local state needs to be migrated from the source environment to the destination environment. The size of the local state is application dependent.

A second category of service requires network connection but the state of the service is local to the platform that hosts the service. Examples of such services include online movie playing , music listening and static web browsing (i.e., there is no interaction with any server that maintains state.). For example, in an online movie playing service, the movie file streamed from a streaming server does not change from time to time. Therefore, no matter where a user connects to the server, he can see the same movie content. Similarly, a music file can be streamed from anywhere to anywhere else on the Internet assuing the same playback. For such services, any state received from the network is always static but may be huge in size (such as a movie file). When a user moves from one environment to the other, she can make a connection to the state provider without the need for any information about her previous connection.

A third category of service requires dynamic state stored both in the network and on the local machine hosting the service. Examples of such services include online gaming and E-commerce. These services have both client state and server state that need to be migrated when a user moves from one place to the other. The connection to the server needs to be maintained at the destination of migration in order to fully resume the service. In other words, such services require strong connection to the server in order to retrieve server state with client state in order to be migrated from one place to the other.

Service level virtualization can be applied to the first and the second category of services but is especially good for the second category. Both of these two categories of

services have only local dynamic state that is under control of user's mobile platform, which makes it easy to retrieve in order to resume the service. The second category has less local state than the first category, which can help reduce the amount of common state for a certain service type (such as movie playing service) and therefore make it easy to define the common state structure in service level virtualization. The third category of service cannot use service level virtualization since there is network state (or connection to the network state) that needs to be maintained during migration. Such state (or the connection) is out of the control of user's mobile platform and therefore very difficult, if not impossible, to migrate when a user physically move.

Device level virtualization can also be applied to the first and second category of service but is especially good for the first category. Similar to service level virtualization, device level virtualization can access and migrate local state with user's movement. However, different from service level virtualization, device level virtualization packs and migrates the entire OS and application image with device state instead of a common state structure. Such a mechanism makes the migration of the first category of service very easy because these services are mostly stand-alone applications and easy to pack and resume. The second category of services may also be able to use device level virtualization but a connection to a remote networked state provider may be necessary although such a connection does not require the information of the previous connection. Similar to service level virtualization, device level virtualization cannot be applied to the third category of services for the same reason: the dynamic network state is out of the control of user's mobile platform and cannot be migrated when user moves.

## *2.6 Goal and Contributions*

In this dissertation, we plan to explore the virtualization technology at both service/middleware level and device/hardware level to see how a service in a ubiquitous

computing environment can be migrated continuously. The contribution of this dissertation is also split into two parts:

For service level virtualization,

- We identify different dimensions for seamless mobility (i.e. what, where, when and how).
- We design MobiGo middleware to address the seamless mobility problem along the above dimensions.
- We implement a prototype system for efficient management of different types of states for different types of environments for a movie playing service.

For device level virtualization,

- We design a set of device independent abstractions for each device category.
- We develop the mechanism for packaging and migrating device states.
- We develop the device capability adaptation for each device category in order to switch across similar devices.

In the next three chapters, after surveying the related work, we explore the design space for each of these approaches and show the design decisions we made for building two systems using the above approaches: MobiGo system using middleware level virtualization and Chameleon using device level virtualization. We also present the performance evaluation results of these systems to demonstrate our systems introduce minimal overhead while addressing critical issues for seamless mobility.

# CHAPTER III

# RELATED WORK

As we presented in the previous chapter, the main focus of this work is to solve seamless mobility problems at different levels. There are a number of related projects that target these problems from different perspectives. In this chapter, we present the related work to the best of our knowledge covering two different perspectives: state migration and capability adaptation. As will become evident in the following discussion, most of the related projects have their own goals, which might be different from ours. Therefore, these studies, while relevant to serve as building blocks for ideas, may not be sufficient to meet the requirements for seamless mobility.

## *3.1 State Migration*

Seamless mobility implies that the source and destination platforms should have something in common. Virtualization technology can easily provide interoperability among heterogeneous platforms. There are different levels of virtualization as shown in Table 2. In this section we review prior research projects that target the mobility with respect to the levels of virtualization and the dimensions described in the previous section. To the best of our knowledge, no other system focuses on service level virtualization as well as device level virtualization to support seamless mobility.

**Table 2:** Virtualization levels

| Levels of virtualization | Example systems and related work |
|---|---|
| Service Level | MobiGo |
| Middleware Level | CORBA |
| Device Level | VNC, Sun Ray, uMiddle |
| Operating System/VM Level | Xen, IBM SoulPad, Microsoft Desktop on Keychain |
| Hardware Level | Register renaming in processors |

While not representing a truly seamless mobility solution, systems such as Microsoft Remote desktop [27] allow a user to access their favorite applications and data ubiquitously (so long as a connection to the "home" machine can be established by the environment). Virtual Network Computing (VNC) [4] and Sun Ray [15] represent a thin-client approach to allow a user's display to be dynamically moved to suit the user's convenience and preference.

Microsoft's "Your Desktop on Your Keychain" project [6] brings to bear system virtualization technology to remove the necessity of connecting to a "home" machine for accessing the user's application and data. The system state is saved with all the open applications and their data "as is" (equivalent to closing the lid of the laptop) to a USB drive (typically 1GB to 2 GB); the user can take the USB drive and plug it into another machine and essentially recreate the original desktop on the new host. Both the original and target machines run a Windows virtualization layer to support this user mobility. The system assumes that a backup network file server is available if in case the USB drive fills up completely, as well as for accessing data and applications not saved on the drive. IBM's SoulPad [13] takes a similar approach to migrate the entire VM state as well as file system to the USB drive and boots from this USB drive on the host machine. These approaches fall into the category of operating system level virtualization and involve migration of soft state and hard states to achieve the mobility.

HP's Cooltown project [14] centers around giving ubiquitous access to information for users. The basic philosophy is to give a "Web presence" to every artifact and thus every artifact becomes self-describing. This is particularly effective for self-guided tours of art galleries and the like. Clearly, the goals of the Cooltown project are quite different from our goal of seamless mobility and it does not use any virtualization technology. Nevertheless, it provides a rich user experience for the environment it targets for.

The Gaia project [19] from UIUC intelligently customizes services to meet different user's requirements. An example is the use of voice recognition techniques to identify a user and pull out the working desktop for that particular user. It also uses vision technologies to track the user in order to move the display from screen to screen as the user moves. Their "smart space" solution, however, is designed to run in a fixed environment (what we termed as self-owned space in the chapter 2) with a fixed set of devices. It is not designed for seamless mobility in different spaces identified in the previous section.

There are several other research projects trying to migrate the state to mobile users for remote access. Networked File System (NFS) [2] is an effort to provide users with access to files on a remote server by mounting the remote file system on the desktop of the user. In a sense, the USB drive solution (both Microsoft's and IBM's alternatives to remote desktops) has a similar but more sophisticated functionality. Ubidata [5] is a project targeting ubiquitous access to data files, i.e., the hard state. The system can provide a consistent view of locally stored files and remotely accessible files on the user's mobile device. Ubidata gives an illusion to a user as if she has a big file system on her mobile platform and can potentially access any data she has wherever the files are located. CoFi [23] is another system that enables authoring multimedia content and collaborative work on mobile devices, which is another example of a mechanism for hard state migration through low bandwidth wireless network. Xmove [11] is an effort at trying to move the I/O state for mobile users. It enables users to map the virtual X protocol server to different physical X servers so that the I/O state can be moved from one I/O device to another (in a single environment). Once again these solutions are appropriate for the goals they set out to accomplish but are at best complementary to our goal of seamless mobility.

Cyber foraging [21][22], like traditional process migration [18], is a technique for exploiting the resources available in the environment to increase the productivity of

a mobile user. Xiaohui Gu and her colleagues designed and implemented a system that can offload java objects to the environment to reduce the burden of computing and storage on resource-constrained devices [39]. The work by Goyal and Carter allows mobile devices to install their own applications in the environment by providing a virtual operating system to each mobile client [8]. These approaches are both operating system/virtual machine level virtualization and are good examples of how a mobile device can take advantage of the computational facilities in the environment to hide its resource limitations. The problem they address is orthogonal to what we are targeting and can complement our solution in achieving continuity of services in multiple environments.

uMiddle [12] is a middleware system that supports interoperability across multiple protocol families at device level. It makes devices speaking different protocols (like UPnP [28] and Bluetooth [16] for example) talk to each other seamlessly through the middleware infrastructure. uMiddle is a device level virtualization technology and lays down the foundations for potential migrations among devices (and device specific services).

As should be evident from the above discussion, the related work surveyed in this section have different goals from ours. Consequently, they do not address all the requirements we identified in chapter 2. Nevertheless, they represent technologies that are very relevant to the overall theme of supporting mobility in a ubiquitous computing setting.

It is interesting to look at these projects with respect to the different spaces we mentioned in chapter 2 and with respect to the state management needed for service migration. This information is summarized in Table 3 for the relevant projects. The slots where we do not have any identified projects suggest that there is an opportunity for new work.

The granularity of migration is another interesting aspect to look at. A complete

**Table 3:** Summary of related works with respect to state migration

| | | Hard | Soft | I/O |
|---|---|---|---|---|
| Own | E | USB Drive<br>SoulPad | Key-chain<br>SoulPad | xmove |
| | I | NFS, Ubidata<br>Gaia | Cyber Foraging<br>Gaia | Gaia |
| Familiar | E | USB Drive<br>SoulPad | SoulPad | VNC, Sun Ray<br>Remote Desktop |
| | I | Ubidata | Cyber Foraging | – |
| Totally-new | E | USB Drive<br>SoulPad | SoulPad | VNC<br>Remote Desktop |
| | I | Ubidata | – | – |

Legend: E - Explicit control; I - Implicit control

migration of virtual machine (VM) states, like Microsoft's keychain project [6], is a coarse-grained approach where users need to carry all VM states in the USB drive and resume the entire virtual machine on the target environment. At the other extreme, a mobile disk drive carries only hard states and users rely on the software services in the target environment to open the files and manually resume the soft state after the software is running (e.g., drag the time bar to the appropriate position to resume a movie). The former approach forces users to carry too much if they just want to move some services but not the whole desktop/virtual machine. The latter migrates only hard state and needs users' inputs to resume soft state. Since a user may want to *dynamically* decide on the need for a specific service in a ubiquitous service system we need to find the right granularity to support seamless mobility for the user: we should migrate all states that are required to provide continuity of a particular service but no more (i.e., not entire VM). Furthermore, it also caters for the dynamic needs of a user on the go.

## *3.2 Capability Adaptation*

Capability adaptation for mobile devices is a relatively new field of research. But there has been a variety of efforts at solving this issue in a number of ways. The

mechanism for adaptation can be provided at various layers of software stack e.g. at the application level, middleware level or at the operating system/hypervisor level.

In [30], Edmonds et al. discuss various models for the structuring of an adaptive system. An application-transparent model performs all the adaptation at operating system level. An application-specific adaptation model places all the responsibilities with the application. The first approach has the advantage of centralized resource control but treats the application as a black box. The second approach might lead to contention for resources among applications due to lack of centralized coordination. An integrated approach is also possible where the resource monitoring (and allocation) is done by operating system and adaptation is performed by the application.The authors have developed a framework for development of application adhering to this integrated approach. The work by Becker et al. [32] mentions the requirements for adaptation at various levels. The devices available to an application and also the characteristics of a device may vary over time and location. An application needs to adapt to these changing scenarios. Device capabilities as well as characteristics of local and remote services need to be uniformly accessible to the application. Typically middleware is responsible for abstracting the communication with remote services while the operating system is responsible for abstracting the access to device capabilities.

Proper representation of the hardware and software context in which an application is operating is very essential for adaptation to take place. W3C's *Composite Capability/Preference Profiles (CC/PP)* is predominantly used for this purpose. Buchholz et al. [31] describe an alternate language named *Comprehensive Structured Context Profiles* for context representation. The primary objective of this work is to provide highly structured representation of context information for easing the task of dynamic composition(/decomposition) of context profiles. In the device-capability-on-demand (DCOD) framework [35], Fu et al. have described a model for enabling an application to make use of various devices as and when they become available.

The framework virtualizes the access to devices and has a matchmaking engine which matches the application requirements with the capabilities of devices and selects a device (or a group of devices) for the purpose.

Naughton et al. in their work [34] discuss a possible mechanism for providing dynamic modification of the Xen hypervisor. Their approach is based on Linux's loadable module mechanism that allows dynamic insertion and removal of kernel modules. In [38] Chen et al. have a slightly different objective: to make live updates to Linux kernels running in virtual machines.

Dynamic Composable Computing [40] targets adaptation from a different perspective. Users of DCC can choose the environmental resources to connect to their mobile devices and take advantages of these resources to enhance their experience. Users do adaptation manually by choosing the appropriate resources to match the application requirements. Different from DCC, the system we propose for device level virtualization called Chameleon (Chapter 5) uses an automatic mechanism to choose capability adaptation algorithms for users based on the application requirements and the environment resource information. While DCC focuses on the usage of surrounding resources for mobile users, Chameleon targets the adaptation mechanisms for service migration across heterogeneous platforms. The two projects are orthogonal and can be complementary to each other.

## *3.3 Summary*

As we can see from the above discussion, there are a number of projects that try to solve the migration or adaptation from a particular perspective. However, we need a comprehensive framework to meet all the requirements of seamless mobility we stated in the previous chapter. Since each of these project is specific to its individual goals, it may not be easy to combine them trivially to solve the issues involved with creating such a framework. In the following two chapters, we present our approach

to address these issues. We also describe in detail some of these previous projects in context with our approach and why the solutions therein would not fit into our comprehensive framework. We understand that our system may not be applicable to all the potential application scenarios in this space. We will discuss the limitations of our system in Chapter 6 later in this dissertation.

# CHAPTER IV

# SERVICE LEVEL VIRTUALIZATION

As we mentioned in Chapter 2, service level virtualization targets the heterogeneity at middleware level. The principles behind service level virtualization is established in our system MobiGO, which can capture the application states at the middleware level and restore them at the destination to resume the execution of user's activity. MobiGo is able to accommodate the heterogeneity of software by defining a common state structure for each service category (for example, a movie playing service) and supplying the common states to various applications in different ubiquitous computing environments. In addition, MobiGo also considers different strategies for the migration of different types of states in different spaces. The state that needs to be migrated at service level is always small in size and easy to carry since the system can decide what states are necessary at the destination to restore the execution. However, it requires hooks into the application to capture and restore the states. If the application does not provide such an interface for other programs to reach into its running state, it becomes difficult, if not impossible, to virtualize the service at the middleware level. In addition, there may be some device specific states that are not visible to the application (for example, the movie streams that has been buffered in the video card memory but has not been played). Service level virtualization may not be able to capture those states for a complete migration of user's activity.

In this chapter, we explore service level virtualization in detail. First, we describe our design choices for service level virtualization. Then we present the system architecture of MobiGo. After describing some implementation details, we show through performance evaluation that MobiGo has acceptable performance while providing

continuous mobile service to end users.

## *4.1 Design choices*

We considered several designs choices for middleware level virtualization. One of the design decisions we made is about the amount of computation done on the mobile platform and in the environment. We decide to make the mobile platform as light as possible while putting most supports into the environment since the environment is generally more powerful. The other choice can be the other way around: the mobile platform holds a lot of functionalities while the environment is thin. We think that the resource imbalance of the two sides is the key factor to consider in this design choice. The limitation of the mobile platform constraint their capability to support many features, and therefore it should be made as thin as possible. However, as the mobile platform keeps developing, we may consider to put more modules in there.

The second design choice is how to construct the service in our system. We may be able to write the service customized to our system from scratch. Another way to construct services is to take advantages of existing application and write a very simple wrapper to make them controllable. We choose the latter because of two reasons: (1) We may not be able to construct every service from scratch. It is not easy and quite time consuming. (2) The service construction is not the focus of system. We should use the existing applications as much as possible.

The third design choice is whether or not we should use existing technologies (such UPnP or Web Service) as the base system for the middleware level virtualization. We decide not to use these technologies because of the following reasons: (1) Existing solutions always provide some functionalities but don't meet all requirements of our system. Modification of existing solutions is possible. However, the complexity of existing solutions makes it difficult, if not impossible, to integrate their missing functionalities seamlessly in them. (2) Existing solutions may be heavy-weight since they

have their own goals and need to consider issues other than the focus of our requirements. We would like to arrive at a minimal system design for seamless mobility so that we can clearly understand the design and implementation issues for realizing such a system. (3) We can always construct a wrapper service to bridge our system to existing solutions and make them interoperate.

Based on these design choice, we have built a system, call MobiGO, that addresses the critical issues of seamless mobility at middleware level. We will show our system architecture and its details in the next section.

## *4.2 MobiGO: a middleware level virtualization system*

MobiGo is the middleware system providing mechanisms for migrating I/O state, soft state and hard state to achieve seamless mobility. We first describe the key components in the system and then discuss how application states (I/O, soft and hard) can be migrated seamlessly. We should emphasize that we are not trying to build a "smart space" that expects people to come and use the services. Instead, we are dealing with how people can *move* a service from one place to another seamlessly.

### 4.2.1 Architecture

As Figure 6 shows, the key components in MobiGo are Service Control, Application Wrapper, Authentication and Service Repository, all of which are critical to seamlessly move a service from one place to the other.

Service Control

Service Control is the core of the entire system. Its main functions are (1) collecting necessary information (including soft state, hard state and user's preference such as selected I/O) to launch a service (2) pausing/resuming services according to user's need and (3) collecting necessary information from the service when user wants to move to another environment and storing it in a proper way for later retrieval. Service Control contains Soft State Management and Hard State Management to help with

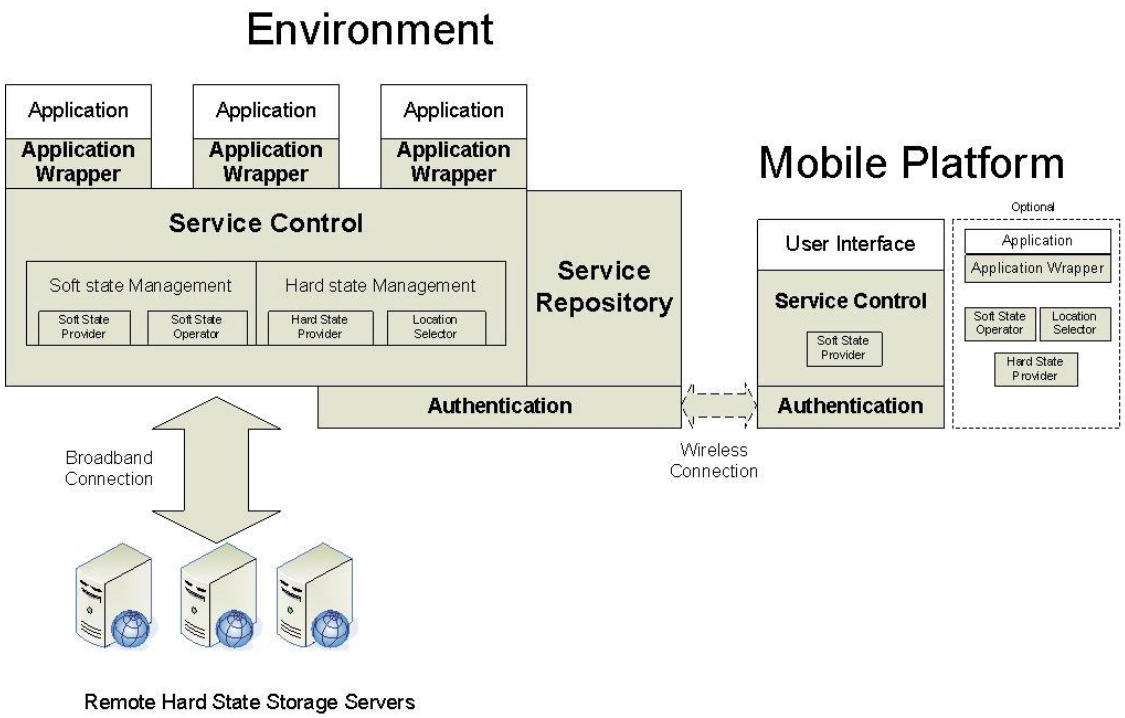


**Figure 6:** Architecture for MobiGo

state storage and retrieval.

Soft State Management decides how the soft state can be transferred to the Service Control to launch services. It has a Soft State Provider module, which stores all the necessary soft states for resuming a previously paused service, and Soft State Operator module, which (1) retrieves the soft state from the service and feeds it into the Soft State Provider for later use when a service is paused, and (2) fetches the state from the Soft State Provider and passes it to Service Control to continue the service when a service is being resumed.

Hard State Management deals with hard state in a different way. Our current hard state management only considers read-only data (mainly for video service). Thus, hard state does not need to be stored when user pauses the service. However, since the hard state is typically large (especially for video files), it is necessary to select where to retrieve the state if it is available in multiple places because bandwidth and

latency of different links will significantly affect user experience. Therefore, we have Hard State Provider and Location Selector in Hard State Management module. Upon a start/resume request from the user, Service Control will notify the Location Selector to find out the best place to fetch the state and then contact Hard State Provider to retrieve the hard state (either copying the entire file if it is small or preparing to stream if it is a large video file for example).

Application Wrapper

Application Wrapper is a small, per-application module that makes legacy applications "controllable" by MobiGo. It can either use available API of a particular application or simulate the behavior of keyboard and mouse to send commands to applications just as if a real person is manipulating the application.

Authentication

The authentication module has built-in mutual authentication mechanisms that (1) verifies users' identity and checks their privilege to use the services in the environment and (2) helps users to ensure the environment is not compromised.

Service Repository

Service repository stores a list of available services in the current environment for mobile users to dynamically discover and use. Environment administrators manage the repository by adding or removing services. Service Repository module listens to a designated multi-cast address and is the first place for the mobile platform to contact when a new user enters the environment. After providing service list to the mobile platform, it passes the control to Service Control module to start/resume a service upon a request from the mobile platform.

### 4.2.2 Discovery and description of the services

We have our own discovery protocol between the mobile platform and environment, allowing the mobile platform to obtain service list from the service repository

in the current local environment. Environment administrators add/remove services from service repository to configure the environment. Each service description contains three parts: service name, service type and I/O devices associated with the service. Services with the same type are interoperable. Since I/O device is the most impactful from the point of view of enhancing the user experience, the UI on the mobile platform clusters the available services in the environment with respect to the I/O devices. User first chooses the desired I/O device and then selects a service on that device. She can also optionally select from a list of previous saved states to resume a paused service. In case the paused service is not available in the current environment, she can select from a list of alternative services that are interoperable with the paused service.

We use a new discovery protocol instead of existing protocol families like Bluetooth or UPnP because we target *service* level interoperability, i.e., user should be able to move between two different services with the same type but perhaps speaking different protocols. By using a new discovery protocol, user can potentially use any *virtualized* service (e.g., a movie playing service) regardless of the protocols. MobiGo moves the burden of dealing with interoperability, which may be computationally intensive for the mobile platform, to the ambient powerful environments.

### 4.2.3 Choices of the user

In order to run a service in an environment, all three states (hard, soft and I/O) have to be present in the environment for the application. Whereas I/O is always local to the environment, users can choose to either explicitly or implicitly control the storage of the hard and soft states for later retrieval. In an explicit control, a user clearly expresses his intention to leave an environment and asks the environment to save the states to his mobile platform for future use. For an implicit control, he may simply leave the environment without prior notification. In the new environment

where he wants to resume the service, the system needs to locate his states regardless of whether he explicitly or implicitly leaves his previous environment.

We consider two strategies to retrieve those states: *carrying* and *fetching*. For carrying strategy, the mobile platform carries all necessary states to run a service. For fetching strategy, the mobile platform provides minimum information that identifies the user (such as a unique user ID) to the environment, and the environment finds the best way to fetch all the necessary states from somewhere else (most likely from the user's previous environment).

Fetching is desirable for best user experience as the user is not required to notify the environment before leaving. But the environment in this strategy has to be *stateful*, i.e., it has to know the user and be able to locate her previous state. When she enters a new environment and presents a unique user ID, the new environment looks for her previous environment and fetches the states from there. However, it may be difficult, if not impossible, to use fetching strategy in some cases, especially in an isolated environment with no connection to the outside world. In such a scenario, carrying strategy is the best choice, which requires the mobile platform to be *stateful* but the environment can be *stateless*. All states are stored on user's mobile platform and can be retrieved through local wireless connection by the environment. But carrying strategy requires the user to explicitly inform the environment when she leaves. It also takes some time to download the states to the mobile platform before she can actually depart.

It is obvious that the environment and the mobile platform cannot be both *stateless*, since user's state has to be retrieved from somewhere in order to launch the service. However, when both the environment and the mobile platform are *stateful*, we have the choice of selecting either fetching or carrying strategy according to user's preference. MobiGo system is able to allow user to specify which strategy to use in such a scenario. Table 4 summarizes the above discussion.

**Table 4:** Fetching vs. carrying strategy in different scenarios

| | **Stateful environment** | **Stateless environment** |
|---|---|---|
| **Stateful mobile platform** | either | carrying |
| **Stateless mobile platform** | fetching | none |

Fetching strategy may be a little slower than carrying, especially when user's previous environment is far away from the current one (e.g., the connection between the two environments is through a high latency and low bandwidth network). But it releases the burden of maintaining the state from the mobile platform to the environment so that the resources on devices (such as storage and computing power) can be saved for other use. It is useful when the resources on the mobile platform are highly constrained and users don't always know when they will enter and leave the environment ahead of time. Carrying strategy requires more resources on a mobile device (to store the state information) and requires explicit notification of departure. However, it can be used more generally in most scenarios, regardless of whether the environment is standalone or connected.

### 4.2.4 State Management

I/O state migration is the simplest among all three state migration. I/O migration means moving a service from one I/O device to the other in the same environment. For example, when a user is watching a movie at home, he will probably want to move it from the living room to the bedroom when he retires for the night. Since hard state and soft state remains unchanged in I/O migration, application wrapper plays a key role in the entire process. For example, it has to find a way to tell the application to move from one display to the other without discontinuity. When a mobile platform issues a *Switch* request (either explicitly issued by the user or implicitly determined by the system), Service Control will extract the target I/O device from the request message and send a command *IOSwitch* to the application

wrapper. Different application wrappers may have different techniques to do the actual switching (we show details on how we do the switching in the implementation section below). There is no state storing/retrieving in the entire process. The entire process is illustrated in Figure 7.

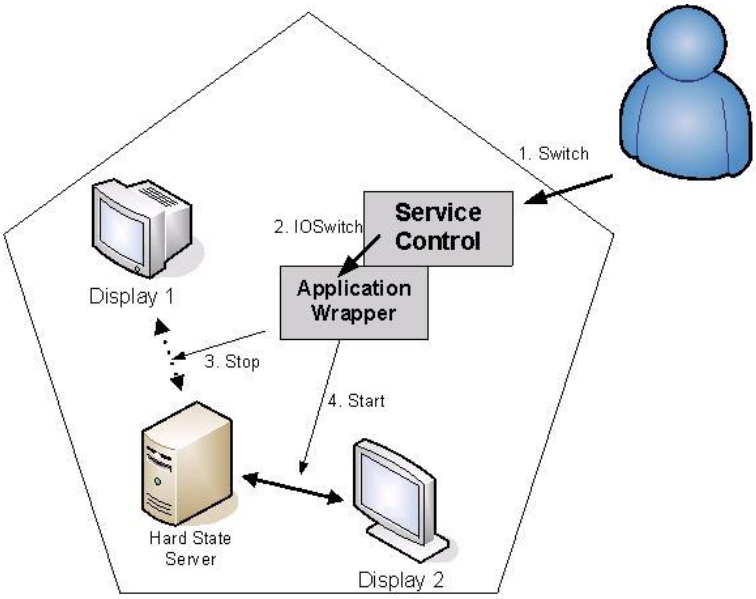


**Figure 7:** I/O Migration

Soft state migration, as aforementioned, has carrying strategy and fetching strategy. Figure 8 shows the migration using carrying strategy: when a user is leaving an environment, the mobile platform sends out a *PAUSE* request (similarly, either explicitly issued by user or implicitly determined by the system), Service Control first pauses the service, retrieves the soft state (e.g., pause point of a movie) and transfers it to the mobile platform. Then, when the user leaves environment A and enters environment B, the mobile device discovers available services in environment B, contacts Service Control in environment B and feeds it with the soft state that was stored from environment A and issues a *RESUME* request. Service Control in environment B will then interpret the soft state and instruct application wrapper to resume the execution of the service according to the state provided by the device.

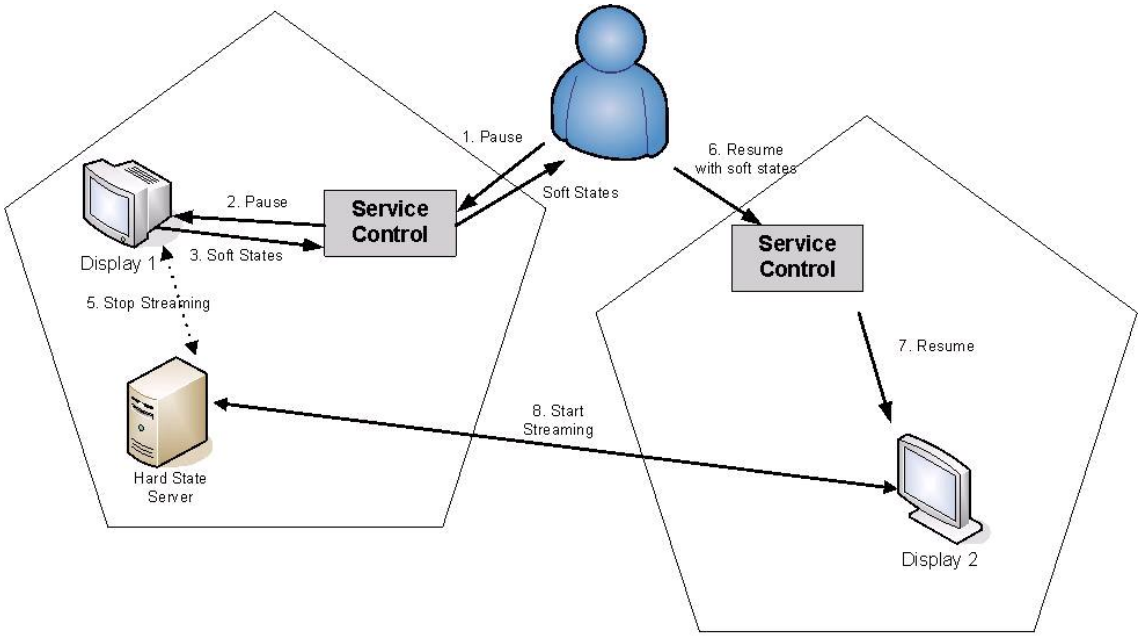


**Figure 8:** Soft State Migration

Hard state migration is more complex than soft state migration. Similar to soft state migration, we can employ carrying strategy and fetching strategy in hard state migration. However, for highly demanding service like a video service, the typical wireless connection between the mobile platform and the environment is usually not rich enough to retrieve/stream the contents in a timely manner. Thus, if a video file is duplicated in multiple sources, it is necessary to find a best source (i.e., low latency and high bandwidth connection between local environment and file source), to get the movie, even if the movie is also carried in the mobile device itself. Therefore, the choice of file source in one environment may be different in another environment depending on the connectivity. Upon entering a new environment, the mobile platform informs the environment of several possible locations for retrieving/streaming files. Then the location selector evaluates different connections (including wireless connection between device and environment if carrying strategy applies) and then decide on a best choice to serve the hard state. It then tells the file source to transfer the file or prepare to stream the file to the local environment. Figure 9 shows a scenario where

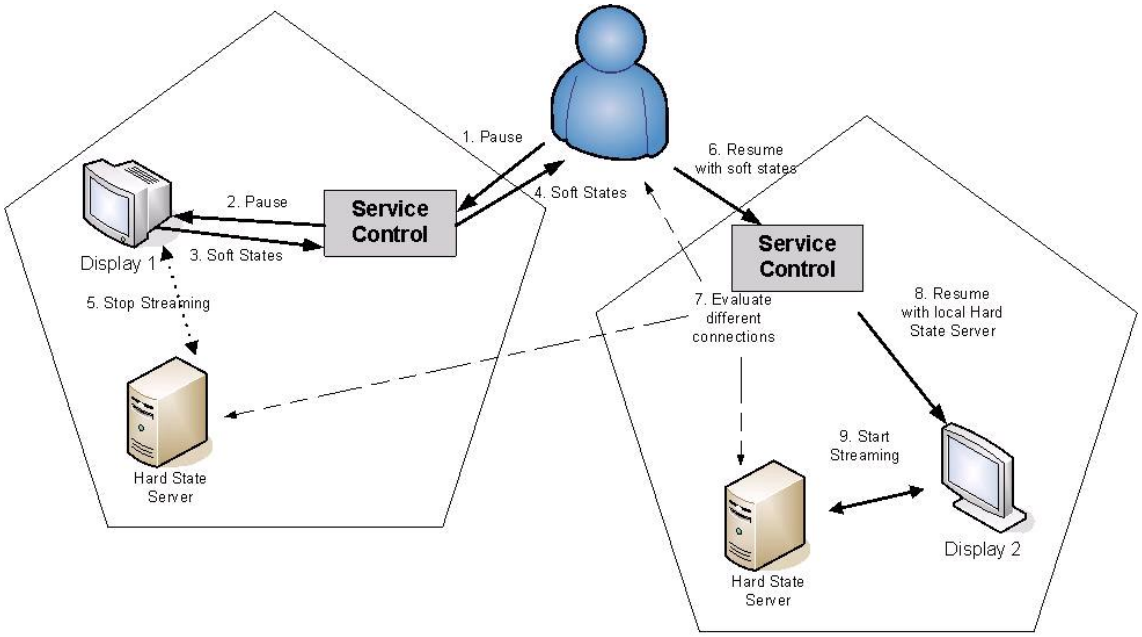


**Figure 9:** Hard State Migration

a local file source is available in the new environment.

### 4.2.5 Trust

The authentication modules on both the environment and the mobile platform ensure the security and trust of the entire system. The trust should be mutual: Only users that are authenticated can use the environment services and users send out private information only after checking the validity of the environment's credentials. To separate private data from different users, the system stores each user's state in an isolated space and retrieves the state for a remote environment (if necessary) only if the user's credential is verified. To avoid conflict among users in using local non-sharable resources (such as display devices), we only allow one user to utilize one such a resource at any given time. Resources in use are not discoverable by new entrants of the environment (Service control periodically contacts the service repository to temporarily block/unblock the advertisement of new resources). Users will get an error message if they want to launch a service on a resource that is in their resource

list but currently used by another user.

We leverage other people's work [29] into MobiGo to manage user's identity and trust of the environment. We are also aware of the complexity of a real ubiquitous computing environment and understand that more secure models may be necessary in some scenarios (such as DoS attacks). Further improvement of the security and trust model constitute the future work of this dissertation.

## *4.3 Supported Applications*

Previous section draws the big picture of MobiGo system architecture. A number of applications that people use on a daily basis can be supported by MobiGo, such as email, browsing, video playing and music listening. As we discussed in Chapter 2, the first and the second category of services are good candidates for MobiGo since it is a service level virtualization system. To make the discussion concrete, we narrow down our focus on a specific application that belongs to the second category: ubiquitous video service. In this section, we explain why we choose this application and present the features and challenges of supporting seamless mobility for this service.

### 4.3.1 Characteristics of video service

There are a number of characteristics of video service that makes it a compelling and challenging candidate for seamless mobility:

- **It demands continuity.** For example, a user may want the movie she is watching to be paused and restarted exactly where she left off (whether that is an instantaneous transition from one display to another or a delayed transition from home to a waiting lounge in an airport).

- **It demands high quality.** If the environment has a number of choices in terms of display devices and stream sources for providing this service, making the best choice is imperative to enhance user experience.

- **It demands efficient management of state.** A full-length high quality movie may be several hundreds of megabytes. Both the hard and soft states associated with a video service have to be managed efficiently for enhancing the user experience. For example, downloading the movie (hard state) from the mobile platform to the environment may just not be a feasible solution from the point of view of latency observed by the user. On the other hand, the movie may be available for download from multiple locations and the environment may be able to make an intelligent dynamic decision.

### 4.3.2 An application scenario

Here is an imaginary application scenario for this video service. Bob (a frequent traveller on business trips) has a collection of movies that he has access to from a server somewhere on the web. One day, he is at home in San Francisco, watching one of his favorite movies, when he suddenly gets a phone call from his company, asking him to travel to New York to meet a customer. He uses his PDA to "pause" the movie and drives to the airport. Upon arrival at the airport, he finds out that his flight is delayed by 2 hours due to "weather" in Chicago. In the Crown lounge while waiting for his flight, he "resumes" the movie where he left off on a big screen (available in the lounge). Upon boarding call, he leaves the Crown room still watching the remaining minutes of the movie on his PDA at the gate area.

To enable the above scenario, the environment first suggests through its discovery service the available video service and displays for migrating the I/O state; upon explicit selection by the user, the environment implicitly migrates the soft state (pause point) from the PDA, and the hard state (video file) which may be streamed either from the web server or from the user's home machine.

## *4.4 MobiGo Implementation*

In this section, we describe some implementation details of MobiGo system. Please note that some of the implementation details presented in this section are not original contribution of our work but are important for a complete prototype system (e.g., security). In this sense, we have adopted simple but workable solutions from the literature to fill in the gaps to realize a complete system. More complex algorithms and solutions may be easily integrated into our system architecture to replace the current ones as we have carefully constructed the system architecture to be modular.

### 4.4.1 Soft State and Hard State Management

We use a pivotal data structure, called Ubiquitous Virtual State (shown in Table 5), associated with each video that contains the meta information necessary to facilitate migration of soft and hard states. The four fields are pivotal in supporting seamless mobility of the service to/from the mobile platform from/to the environment. There may be additional bookkeeping information in this data structure beyond these four fields (such as a textual summary of the contents, owner, and creator). The LOCATION information is used for efficient management of the hard state and the STATE information is used similarly for the efficient management of the soft state.

**Table 5:** Structure of Ubiquitous Virtual State

| Field Name | Description | Example |
|---|---|---|
| FILENAME | The name of the file | Terminator-II.mpg |
| LOCATION | An array that includes all possible sources of the file | user@video.foo.org:/share/Terminator-II.mpg |
| STATE | Internal soft state of the file (such as how much of the file has been played already) | 10000 ms from start of file |
| MODIFIED | Last modified timestamp | Oct 31. 2006, 11:09s |

UVS is created when user downloads or gets access to the video file and will be updated when user is moving around. For example, when a user buys a movie from a website, he may download a UVS along with the video file. If the website has streaming service that allows users to stream the movie in addition to simply downloading the video file, it will specify the streaming server addresses in the UVS. Thus, upon entering a new environment, user has the choice to stream the movie from his/her mobile platform where the movie file is stored or from the remote streaming server, depending on which can provide best possible service (we will discuss how the decision is made later in this section). When the user leaves an environment, the STATE field is updated and transferred to the new environment later for resuming the service.

### 4.4.2 Application Wrapper and Service Repository

In the current implementation, we have two application wrappers: (1) a native mplayer wrapper that simulates the input commands from standard input (stdin) to control mplayer to play/pause a movie and get the pause point and (2) a UPnP windows media player wrapper receiving UPnP commands. Service Control uses different functional calls for different wrappers (for example, upnpPause() for UPnP wrapper and nativePause() for native wrapper). Service Repository maintains a list of services, which also includes the types of service wrappers. It helps Service Control to choose which functions to call when it issues a command. Environment administrators maintain Service Repository and can add/remove services according to local environment policies.

### 4.4.3 Location Selector

When the hard state of a service, especially large video files, are duplicated in multiple locations, it is necessary to select one place to achieve the best quality of service to users. In our current implementation, we use an intuitive algorithm: select

the place that has the lowest latency connection to the local environment with sufficient bandwidth for the specific service. Our algorithm first determines the required bandwidth of streaming a video from a server based on its frame rate and resolution. For example, a $352 \times 240$, 30fps MPEG-1 video requires 1.5Mbit/s bandwidth. Then, the algorithm chooses the lowest latency connection from all the candidates that can cater to this bandwidth requirement. We will show our evaluation of how this algorithm performs in enhancing user experience.

We also notice that the measurement of latency and bandwidth of connections may be time-consuming, especially for wireless link between device and environment. Therefore, the environment profiles the wireless link connectivity of the mobile device in the background as soon as the user enters the environment and is authenticated by the environment, and even before she chooses any service provided by the environment. Since profiling the wireless link can be quite slow, performing this pre-pinging helps in reducing the latency experienced by the user. The pre-ping statistic serves as a baseline for evaluating other sources of hard state specified in the UVS.

### 4.4.4 Authentication and Discovery

As we mentioned earlier, our infrastructure provides *service level* discovery. We have a simple home-grown protocol for authenticating users and discovering/transferring available service list to the mobile device. A message from the mobile platform, containing the user's identity (a hash to username and password), the name of the environment (N) and a timestamp (real time T) is sent periodically to a designated multi-cast address. This message informs the environment that the mobile platform has a service list at time T from the environment N. After checking the user's identity, the environment matches N to itself; upon a match, the environment will either send out an updated service list (if the list has changed since time T) or an empty message to signify no change. If N does not match this environment, then obviously the mobile


**Figure 10:** Hard State Migration

platform is new to the environment, and hence the environment will send its identity along with the current service list. At the time of explicit departure from an environment (i.e., user explicitly requests to leave an environment), the mobile platform sends a LEAVE message and collects the necessary state to resume the service at a later time. An explicit LEAVE has to be followed by an explicit DISCOVER request to be issued by the user, allowing the system to know the user is willing to discover surrounding services again.

### 4.4.5 Communication Message Structure

The structure of the messages for communication between the mobile platform and the environment is shown in Figure 10:

The first byte of the message indicates the requested command. The command can be any of the following:

- Discover (D): The mobile device looks for services in the current environment. A list of available services/resources is expected to be returned.
- Launch (L): The mobile device wants to launch a service that is available in the current environment.
- Pause (P): The mobile device requests to pause the service. The internal states are expected to be returned.
- Redirect/Switch (R): The mobile device requests to switch to another display at application/middleware level or to another similar service to continue the

**Table 6:** MobiGo Command Parameters

| Command | Parameters |
|---|---|
| Discover | NULL |
| Launch | Service ID |
| Pause | Service ID and Pause mode (with or without returned state) |
| Redirect | Destination display ID |
| Quit | Quit mode (with or without saving the states in local env) |

activity. If it is a display switching, it requires the application to be able to handle it internally.

- Quit/Disconnect (Q): The mobile device gracefully disconnects from the environment. The current state of the service may be saved or discarded according to the parameters of the command.

The second byte of the message indicates the length of the message in byte. Currently, we only have 1 byte for the length of the message, and therefore only support messages less than 256 bytes long, because we expect the communication between mobile device and environment is very light and the messages are normally very short. However, this is a design parameter that can be easily changed to support larger message formats, if necessary.

The rest of the message contains the parameters of the command. It is command specific as shown in the Table 6. As we listed above, we have only five commands in our system: Discover, Launch, Pause, Redirect and Quit.

The returned results have the similar structures as the command, except that they don't have the command byte. The results start with the length of the message followed by the contents of the returned values. The structure is pre-defined so every mobile platform and environment can understand.

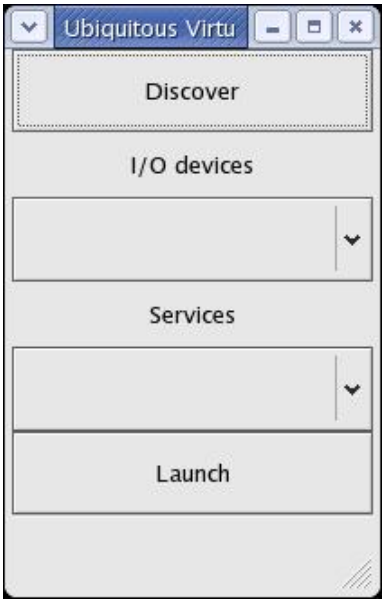


**Figure 11:** UI: Initial Screen

### 4.4.6 Sample User Interface

We have developed a user interface application on HP iPAQ using GTK libraries [54] for mobile users to control their activities. Since the screen of iPAQ is limited in size, the interface application needs to be as simple as possible. Figure 11 shows our initial interface when the application is started on iPAQ.

There are two buttons and two drop-down boxes in the interface application. Users can use two buttons to issue commands to the environment and use two drop down boxes to select desired services and display resources for the services. Initially, users can only click the top button "Discover" to look for available services in the surrounding environment. The other button and the two drop-down boxes have no use at the beginning.

If the mobile device discovers a new environment, the available services and I/O resources are downloaded into the iPAQ and populated into the two drop-down boxes. As shown in Figure 12, the top button is changed to "Quit" that allows a user to gracefully exit the connected environment at any time. By selecting corresponding

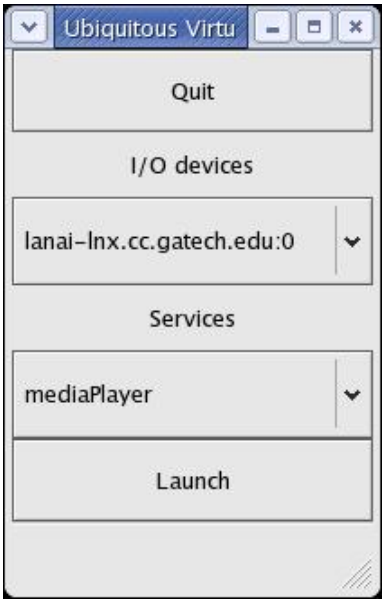


**Figure 12:** UI: Services are discovered

I/O device and service, users can click the bottom button "Launch" to start the service.

After the service is launched, the bottom button is changed to "Pause" that allows a user to pause the service and dump the state at any time they want (as shown in Figure 13). The state may be saved to iPAQ as a .uvs file that can be resumed later using the "Launch" button in another environment.

If users choose another service or I/O device while the current service is still running, the bottom button is changed to "Redirect", which allows users to switch to the selected service or display resource if they want. The Redirect button only appears when the running service/display resource is different from the selected service/resource (as shown in Figure 14). By clicking the Redirect button, the system can (1) migrate the I/O to the target device at application level or (2) migrate the internal application state from one service to the other.

The purpose of this interface is to show an example application that can be developed on top of our platform to assist users to control the migration. This simple

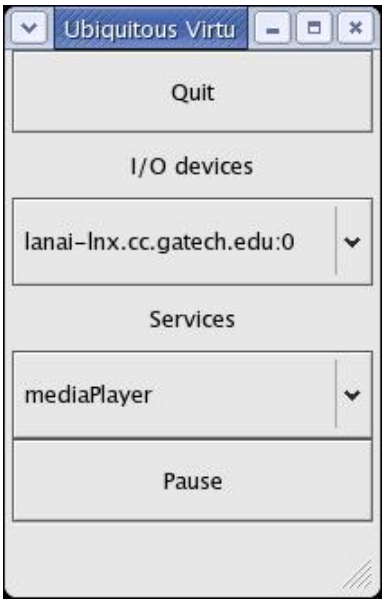


**Figure 13:** UI: Service launched

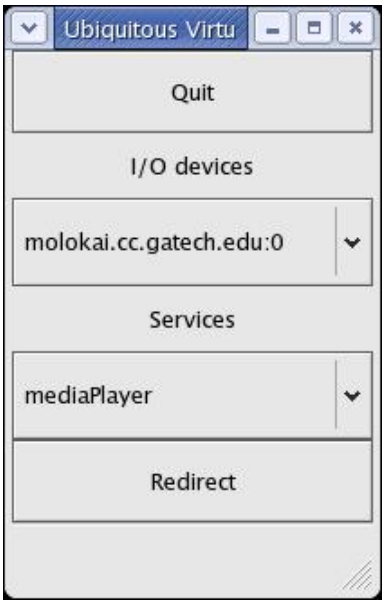


**Figure 14:** UI: Switching I/O or service

design is enough to demonstrate the functionalities of our system.

### 4.4.7 UPnP Media Server and Render

uShare [55] is the Media Server we used for serving media contents to UPnP renders. It hosts multimedia files for UPnP enabled device to consume the audio and video media. Following UPnP protocol stack, uShare is built on top of http and uses libupnp [56] to stream the files to clients.

We have used two types of media renders to render media streams on different platforms: Intel UPnP media render built on top of Windows Media Player (WMP) [59] and GMediaRender [60] built on top of GStreamer [61] in Linux. We do not change these existing tools but we do send UPnP messages to control their behaviors in order to meet our needs.

## *4.5 Performance of MobiGo*

The "goodness" of the MobiGo architecture is in the user experience. However, this is a qualitative measure of the performance of the system. To make it quantitative, we conduct a series of experiments to convince the reader that the system results in enhancing user experience.

All experiments discussed in this section use the following set up: (1) a mobile platform: an Arm-Linux iPAQ equipped with 802.11b wireless connectivity (2) two environments: <A> a 4-way SMP (4 × 450MHz UltraSPARC-II processors, 4 GB memory) running Solaris operating system, and <B> a 2-way SMP (2×3.2GHz Xeon, 4 GB memory) running Red Hat Enterprise Linux 4.0. These two machines have Gigabit wired connectivity to each other and both of them have full-fledged MobiGo environment software installed (including service repository, service control and hard state provider). We use a video service for all tests below. In these experiments, we assume that the video file is streamed to the player. Most players require 512KBytes - 1024KBytes buffering of the video before starting.

### 4.5.1 Latency for I/O, soft and hard state migration

The first experiment is to measure the switching cost of I/O, soft and hard state migration when users move a service. I/O state migration cost is the latency observed by Service Control when it informs application wrapper to switch from one display to another in a single environment (as shown in Figure 7). The cost of soft state migration, which implies a I/O migration, includes pausing the video, dumping the soft state to the mobile device (from environment <A>), feeding the soft state to the other environment (environment <B>) and resuming the video (as illustrated in Figure 8). Hard state migration cost, while including all the soft state migration cost, also involves the time to evaluate links from local environment to the mobile platform and a media server (located in environment <B>) (as illustrated in Figure 9). We use carrying strategy for the migration of soft state (1K bytes in size) and fetching strategy for the migration of hard state. The buffer size (i.e., size of the hard state) is 512KBytes in this experiment. Figure 15 shows our results for these three state migration costs.

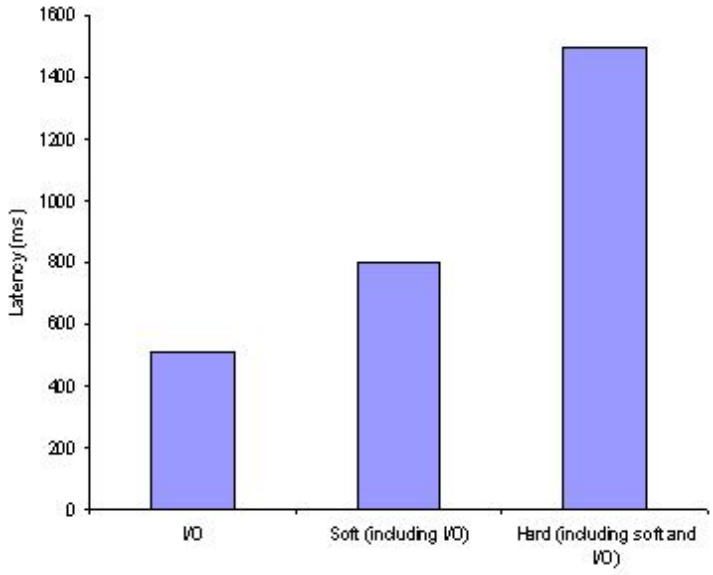


**Figure 15:** Switching latency for I/O, soft and hard state migration

We can see from Figure 15 that the one-time switching cost of all three state migrations are from 0.5s - 1.5s, which are acceptable from user experience perspective [1]. Therefore, we conclude that while providing continuity of services to users, the system overhead is in reasonable range.

### 4.5.2 Latency improvement from location selector for hard state migration

In this experiment, we measure how the location selector can help reduce the end-to-end latency. We assume that the hard state is available on the mobile platform as well as at a remote server (e.g., video server in environment <B>) that has a high-bandwidth low-latency connection to the current environment (environment <A>). Figure 16 shows the relative cost of downloading the data using the location selector (the light bar) and not using the location selector (the dark bar). For the former case, there is a hidden cost wherein the location selector first ascertains the relative cost of downloading from the mobile platform versus the remote server. As can be seen from the figure, for small file sizes (up to 100 Kbytes), there is no advantage to downloading from the remote server. This result suggests that it is best to carry the soft state (which is typically on the order of a few kilobytes) on the mobile platform and not use the location selector for transferring the soft state to the environment. On the other hand, there is upwards of 60% performance improvement for large file sizes (1 Mbyte or more). Since the buffering needed by streaming video players is in the range of 512 KBytes - 1024 KBytes, it is best to use the location selector for determining the best source for downloading/streaming the hard state.

### 4.5.3 Simulating different network conditions

The above two experiments measured the performance of our current MobiGo implementation in a laboratory setting. To further understand how MobiGo performs

[1] We assume a 2 second latency between initiating an action and observing its effect is tolerable from a user experience standpoint [62].

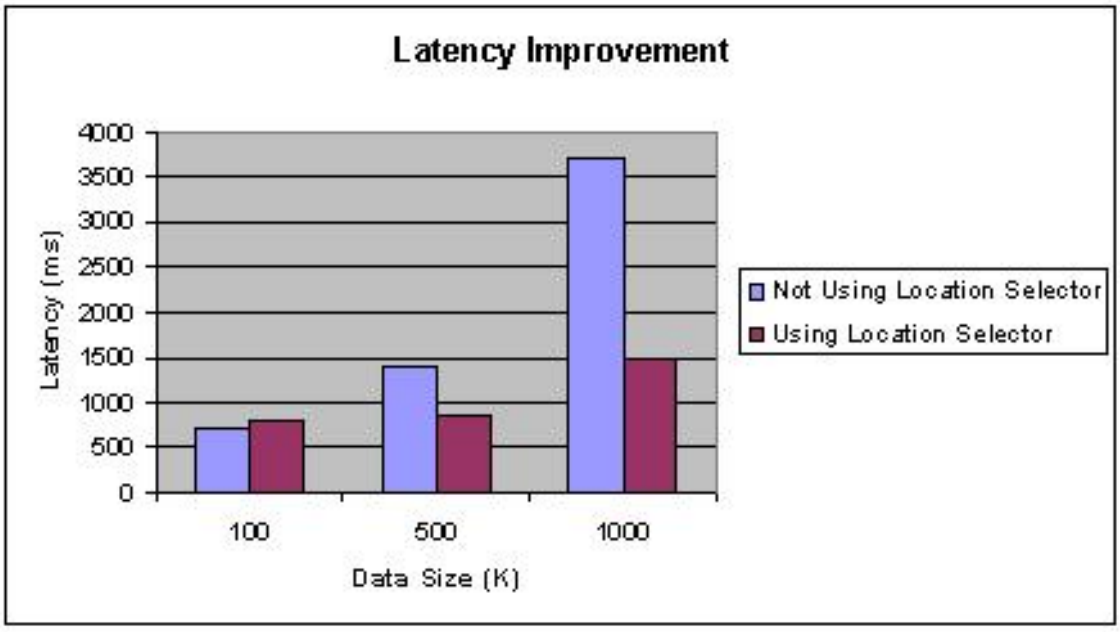


**Figure 16:** Latency improvement from location selector

under different conditions, we may need to answer questions such as "what if I deployed MobiGo in my home where the highest bandwidth is only 1.5Mbit/sec?" or "what if I change the strategy from carrying to fetching and how different is the performance?". Therefore, we first analyzed different components of the end-to-end latency and then conducted controlled experiments to simulate different network bandwidth conditions to show their influence on the end-to-end latency.

The end-to-end latency is the elapsed time between pressing the SWITCH button and the resumption of the migrated service. We divide this latency into three parts: (1) the overhead of MobiGo system to process messages and do bookkeeping, (2) the latency for the movie player to play/pause/seek the movie and (3) the network latency to transfer command messages and/or states.

The first two types of latency are relatively constant where the third one varies in different network conditions. In our current implementation, MobiGo's bookkeeping and message processing takes about 100-150ms for each migration and the movie player takes 60-80ms to complete single commands (e.g., play, pause or get_paused_position).

Every migration contains 4-6 commands sent to the player, depending on the player's capability [2]. These two types of overhead are fixed and remain unchanged from one environment to another.

In order to evaluate the end-to-end latency under different network conditions, we have modified the MobiGo implementation to add network delays commensurate with the network bandwidths to be simulated. This gives us a methodology for answering the "what if" questions we posed earlier in this subsection. Figure 17 shows the results of these experiments. X-axis represents the bandwidth between the local environment and the state provider and Y-axis is the end-to-end latency for seamless mobility [3].

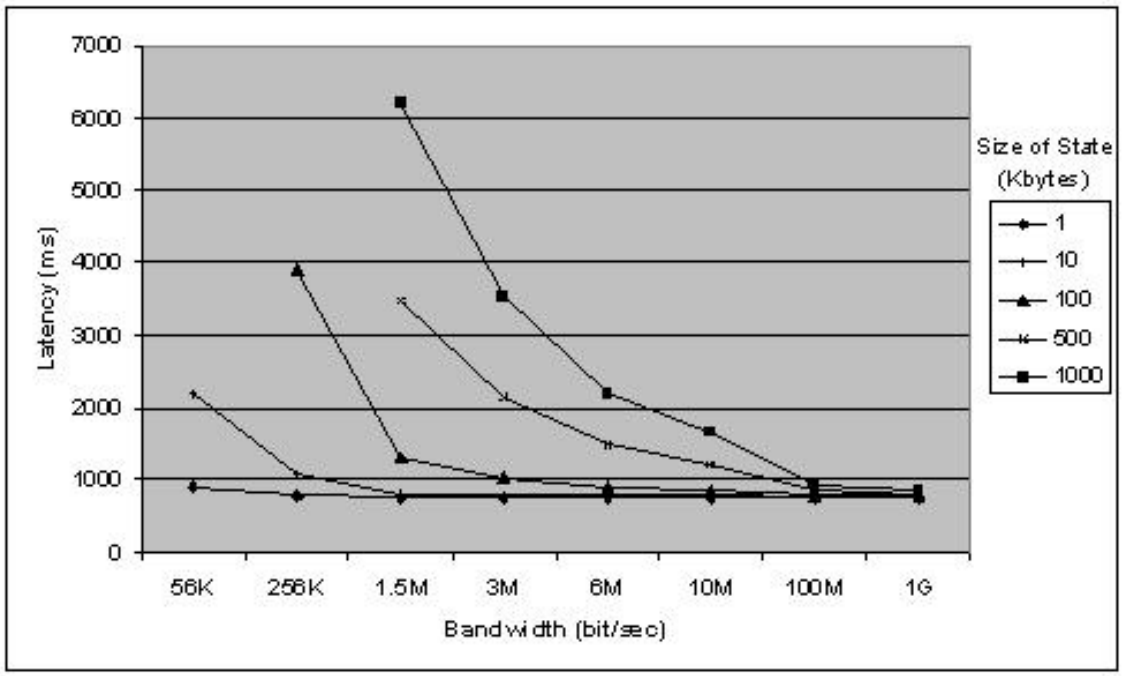


**Figure 17:** Migration latency affected by network bandwidth (for different sizes)

Since the fixed latencies are incurred for all downloads (small or large), the influence of the network bandwidth is not significant for small downloads (e.g., soft state migration). On the other hand, we can see that the larger the size of the state to

[2]Some players need a play and a pause before the seek command that sets the current position of the movie.

[3]All the latencies more than 7 seconds are removed from the figure for better view.

be migrated, the more the influence of the bandwidth on end-to-end latency. For a 1.5Mbit/sec (less than 200KByte/sec) connection, like a typical DSL Internet connection for home, it may take up to 6-7 sec to buffer 1MByte of a movie file from a remote site. This may be acceptable in some cases but it will be better if the system can choose a closer state provider, perhaps a server in the home with a 100Mbit/sec Ethernet connection, for better performance. Further, movie players tend to buffer more than 1 MByte for high quality movies (image size $640 \times 480$). Therefore, the importance of the location selector becomes more apparent under different network conditions.

The above analysis investigates the possible factors that affect the latency prior to starting the playing of the movie. An equally important consideration is the user experience during the actual movie playing. This depends on the bit rate of the movie and the available network bandwidth. The bit rate of a typical $352 \times 240$ 30fps MPEG-1 movie is approximately 1.5Mbit/s (200KByte/s). Thus, if the network bandwidth is less than 1.5Mbit/sec, the movie playback will not be smooth (i.e., the user will experience significant start/stop by the player and/or jitter depending on the player specifics). The location selector can play an important role in selecting a download/streaming site to ensure that the *average* network bandwidth is greater than the bitrate needed for the movie playback. Further, the location selector can also help instruct the player on the expected amount of buffering to be done to ensure that there is no jitter and/or start/stop during playback and thus enhance the user experience.

## *4.6 Summary*

In this chapter, we have presented our middleware level virtualization work in detail: the motivation, design choice, system architecture, implementation and performance evaluation. We have demonstrated that our system can make similar service

interoperable and enable users to migrate across these services running on heterogeneous platforms. However, as we discussed in Chapter 2, service level virtualization has its limitations. There are situations in seamless mobility that are not handled at all or not handled well by service level virtualization. Therefore, in the next chapter, we present another approach, device level virtualization, to further address issues in seamless mobility.

# CHAPTER V

# DEVICE LEVEL VIRTUALIZATION

Device level virtualization (DLV) targets the heterogeneity at device/hardware level. Devices in different environments, while providing similar services, are always different in terms of type and quality. This heterogeneity of device capabilities makes the service migration difficult since the source and destination platforms are not always compatible. Device virtualization technology is one of the practical approaches to address this problem. Device virtualization layer is placed between hardware and operating systems. Such a layer shields the target device from unintended or malicious behavior of an application from compromising the resources. DLV is able to capture the running state of the OS, applications, as well as devices, and migrate the entire state image from one environment to the other regardless of device capability. In contrast to service level virtualization, DLV considers the applications and OS as a package and therefore does not need to care about the application states that is difficult to capture when application hooks are not available. Virtualization at device level also makes it easier to capture the state of devices, which is difficult, if not impossible, to do at middleware/service level. One drawback of the device level virtualization approach is that the migration may be heavy since the entire application plus OS, and not just the state of a single application, needs to be moved from one device to the other. However, for typical handheld devices, the footprint of the applications and OS is relatively small compared to desktop PCs. This makes the device level virtualization practical, especially for these small handheld devices.

In this chapter, we describe device level virtualization in detail. We first describe the requirements of device level virtualization and why we choose Xen as our base

system to implement device level virtualization. After presenting some possible design choices, we describe Chameleon system, a device level virtualization system that we have implemented on top of Xen hypervisor. Finally, we present performance results to show that the overhead of our modification to Xen is minimal compared to the original Xen virtualization system.

## *5.1 Device level virtualization requirements*

As we described in chapter 2, device level virtualization can be applied in two types of scenarios: single-machine-multiple-I/O migration and entire OS migration. However, the key problem in these two scenarios is the same: we want to switch the mapping internally between virtualized devices and physical devices. Therefore, we identify two requirements for device level virtualization to address this need: device state migration and capability adaptation.

### 5.1.1 Device state migration

Device state migration is the first requirement for device level virtualization. When switching from one device (source) to the other (destination), the internal states of the source device has to be migrated to the destination device seamlessly to ensure the continuity of input/output. For example, when a movie is being played, there might be several frames stored in the display devices themselves, waiting to be rendered on the display. These frames, representing a type of device state, are in the physical devices and out of the control of operating system/applications. We do not consider such a state for our work. In contrast to this type of state, there are also some frames stored in the device drivers that have not been transferred to the physical device. This type of state is out of the control of the application but still part of the operating system. In other words, the applications "think" that these frames have already been rendered but they are actually not. When we switch the mapping from one physical device to the other physical device, these frames have to

be migrated to the destination device. Similarly, on the input side, there might be a few ASCII codes for the key strokes saved in the input buffer of a keyboard device driver that have not been delivered to applications. When we switch from one input device to another, these ASCII codes need to be migrated as part of the device state.

### 5.1.2 Capability adaptation

Another requirement of device level virtualization is capability adaptation. As can be imagined, different devices may have different capabilities. For example, display devices may have different sizes, resolutions or color settings (i.e., the same pixel may appear differently on different display devices). Input devices may have different key layout or key code mappings. When switching from one device to the other, we need to consider such differences in device capabilities and conduct proper adaptation to address the differences and help enhance the user experience. We call such an adaptation mechanism *capability adaptation*.

There are a variety of capability adaptation algorithms for different types of adaptation. However, the development of such algorithms are out of the scope of this dissertation. The focus of this work is to construct a comprehensive architecture to enable the dynamic selection and utilization of capability adaptation algorithms at run-time. The goal is to enable the integration of any adaptation algorithm into our system easily and efficiently.

## *5.2 Design principles*

There are a few important issues that we have to consider in order to design a successful system to address the migration needs. In this section, we discuss these issues and design principles underlying these issues.

Before discussing the design principles, we want to describe the migration system briefly. As we stated in chapter 2, users may desire movement of their activities with their physical movement. Therefore, in our migration system, there is always a mobile

platform that is moved with the user and a stationary host system (environment) that can provide resources locally to the user. The mobile platform can also be called a "guest" system, which means it takes the resources offered by the local host and serves the user directly. The guest may be migrated to a different host system if the user moves.

The first design principle is that we want to keep the mobile platform unchanged and make the environment adapt to the mobile platform. Since in a typical ubiquitous environment there are a variety of resources and systems, adaptation has to be done in order to enable the migration across heterogenous platforms. Therefore, there is always a question of where the adaptation should happen. We think that instead of making the mobile platform adapt to the target environment, it is better for the environment to be able to adapt to the mobile platform, because the hosting environment always knows the local resources better and therefore understands how to best adapt a generic guest system to its local resources. Another (not desired) way is to make the guest system adapt to environmental resources. This approach may not work well because the guest has to know every environment and all types of resources in order to be able to adapt to any new environment. Again, in a typical ubiquitous computing environment, there are hundreds of different types of resources. It makes the guest difficult (if not impossible) to learn all of them beforehand.

The second design principle is that we want to build a system that can support the dynamic installation and uninstallation of different capability adaptors for adaptation. The adaptation algorithms, however, are not the focus of this dissertation. While new adaptation algorithm may be developed from time to time, the aim of our system is to make the integration of such algorithms easy and efficient. Our focus is to add the management of capability adaptation (dynamic installation and selection) and device state migration to existing ubiquitous computing systems.

The third design principle is that we want to make the selection of adaptation

algorithms automatic and dynamic at run-time. Our system architecture allows the coexistence of a variety of adaptation algorithms and choosing one algorithm over the other is always a challenge. We would like to facilitate the selection of an algorithm based on requirements presented to the system at runtime.

The fourth design principle is that we want to make the migration as seamless as possible. Therefore, we may need to migrate the device state (such as the frame buffer for display device) in addition to capability adaptation. By dumping and resuming the device states, we can ensure the seamless migration of the guest at device level.

## *5.3 Base System*

In order to meet all the design principles described in the previous section, we choose Xen as our base operating system to work on. In this section, we briefly describe Xen and the reasons for our choice.

### 5.3.1 Xen and virtual device

Xen is an open-source virtualization system that can support multiple guest operating systems running on top of a hypervisor (also called a virtual machine manager), sharing the same hardware resources. Xen always needs a host operating system (usually Xeno-Linux) running on top of the hypervisor to interact with the guest operating system. By virtualizing the hardware resources, the hypervisor gives a guest operating system the illusion of having total control of all resources while internally managing resources across different guests. In addition, the hypervisor creates different domains for these operating systems in separate address spaces, called virtual machines (VM), and Xen provides facilities to suspend and resume these VMs.

Split device driver model is a key feature in Xen that facilitates capability adaption without significant change to the system. Figure 18 shows the basic architecture of Xen. On top of the virtual machine monitor, there is a host operating system that hosts backend device drivers (BE) similar to physical device drivers in a traditional

operating systems. These backend drivers are device specific and they take control over physical devices directly. There might be one or more guest operating system(s) with generic frontend device drivers (FE) that seek to access those devices controlled by the backend. Frontend drivers connect to backend drivers when the guest OS is loaded into the system. After a complex handshaking process and connection establishment, FE can forward the application requests of accessing a device (such as reading or writing) to BE. BE then processes those requests to fulfill FE's needs, just as a traditional device drivers would do.

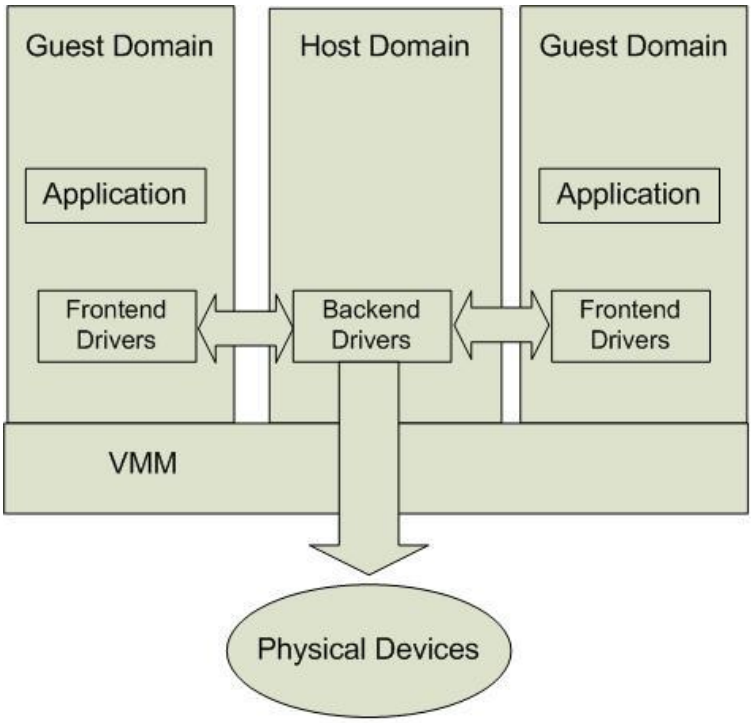


**Figure 18:** Xen Architecture

Backend drivers and frontend drivers communicate through Xenbus [57], which is a virtual bus located in the virtual machine manager (VMM). The frontend always initiates the connection by writing a particular entry in Xenstore [42], a centralized storage space that is also located in VMM and can be accessed by any domain. The backend can see the change and trigger the connection establishment process (we will

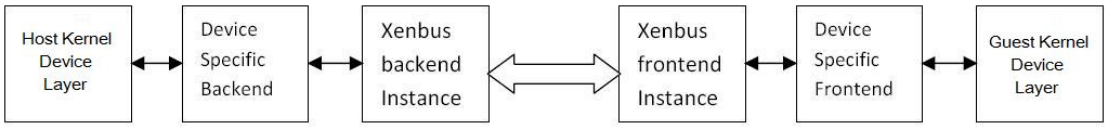


**Figure 19:** Communications in Xen Split Device Driver Model

present the details later in this section). Upon connection, Xenbus creates frontend and backend instances inside it to represent the two communication entities. The two Xenbus instances communicate to the actual frontend and backend respectively inside their domain (frontend in guest domain and backend in host domain). Figure 19 shows this model. The left two boxes in Figure 19 are running in host domain (backend part) whereas the right two boxes are running in guest domain (frontend part). The middle two boxes are Xenbus components that live in the VMM of the Xen architecture and connect frontend and backend together.

In such a split device driver model, the frontend driver can be seen as a *virtual* device driver that is generic and takes no device specific information. The virtual device driver becomes concrete when it connects to a backend that has information about the physical device (e.g., resolution and color settings for a display device). Applications running in the guest operating system only see virtual device drivers and do not need to worry about device specific settings and configurations. This gives application developers the freedom to focus on their application logic instead of dealing with physical device related issues.

Therefore, we can see some key features in Xen that can help us develop an interface virtualization system and capability adaptation:

- Xen has generic frontend device drivers, which gives us a very good starting point to develop virtualized interface for output devices (see next section for details).

- In the split device driver model, we can break the connection between frontend

and backend drivers and then insert the capability adaptor to enable the capability adaptation, which only requires a small modification of the connection part of original Xen's driver code.

- Xen is an open source operating system that makes it possible to change the operating system internals for the adaptation mechanism.

### 5.3.2 Framebuffer device and virtual framebuffer device

As a concrete example of how Xen virtualizes devices, let us consider framebuffer, an important device for several reasons. First of all, to support seamless mobility in a ubiquitous setting, the display device plays a crucial role in enhancing user experience. Second, the display device has significant amount of complexity (aspect ratio, color, intensity, etc.) that makes virtualization and capability adaptation of this device challenging. Third, the framebuffer is an excellent vehicle from the point of view of demonstrating the power of the capability adaptation framework of Chameleon.

Framebuffer device drivers in Linux provide common interface for applications to draw pixels on physical devices without knowing the details of each device. Typically, framebuffer of these devices, which contains the color values of every pixel on the screen, is mapped into main memory and can be read and written through regular memory operations. Internally, framebuffer device drivers use low level system calls (such as `inb` and `outb` system calls [58], which access device directly) or assembly language to operate the physical devices according to their detailed specifications.

In Xen virtualization system, the virtual framebuffer device drivers are provided to guest operating systems in order to virtualize these drivers on top of physical framebuffer devices. In Xen, virtual framebuffer device driver is also split into two parts: frontend driver and backend driver. Frontend drivers are located in guest domains and serve as traditional Linux framebuffer device drivers for any guest operating system. Backend drivers are located in the host domain, which can connect

to frontend drivers when guest domains start. Backend device drivers map the guest domain's framebuffer requests to physical device driver framebuffer to execute the actual operations (such as reading or writing).

Virtual framebuffer device drivers provide a common interface for us to operate output devices without worrying about their details. Such an interface helps us to focus on the upper layer logic (such as state migration and capability adaptation) rather than device control commands. We choose to use virtual framebuffer device drivers as the base device drivers in Chameleon since that simplifies the design for dealing with device specific operations.

## *5.4 Design choices*

There are several design choices to build a system that can support capability adaptation and device state migration. In this section, we discuss these design choices and in next section we present the final design of our system, Chameleon, to meet the requirements.

Since we need to add capability adaptation and device state migration on top of existing Xen's architecture, we do need to change Xen's device driver code to some extent. However, we wish to minimize the changes to the existing code. At the same time, we also wish to minimize the overhead we introduce into the system. Therefore, we consider three possible architectures for Chameleon.

Figure 20 shows our first design choice. In this design, we need to modify the frontend and backend drivers and add capability adaptation module and the virtual device interface module to each of them respectively. In this way, the two additional module communicate through regular Xen's mechanism and the requests from frontend are redirected through these two additional module instead of directly reaching the backend. Both the host and guest domains in this design have to be modified to support capability adaptation. However, regular Xen's split device driver mechanisms

remain unchanged.

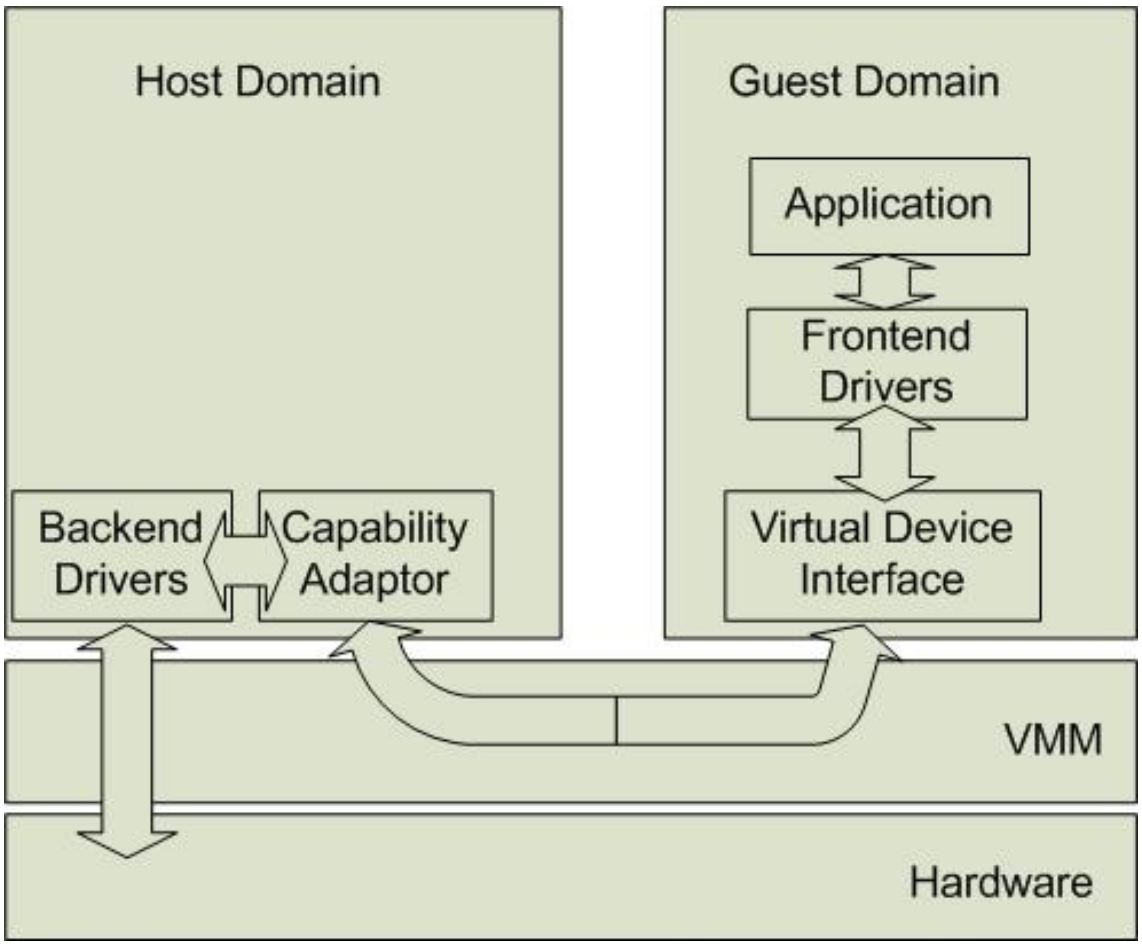


**Figure 20:** Design 1:changes in FE and BE, no change in VMM

Figure 21 shows the second possible design of the system. In this design, we insert capability adaptors into VMM and keep both the frontend and backend unchanged. However, we need to change the mechanisms in Xen to make the FE/BE connection. The changes will apply to any host environment system but only inside VMM. We do not need to change any host domain or guest domain in this design.

Figure 22 shows the thrid possible design for the system. In this design, we do not change the guest domain as well as the VMM. All the modifications happen in the host domain and the backend drivers to insert the capability adaptor in the existing Xen's split device driver model. This requires some modifications to the host domain

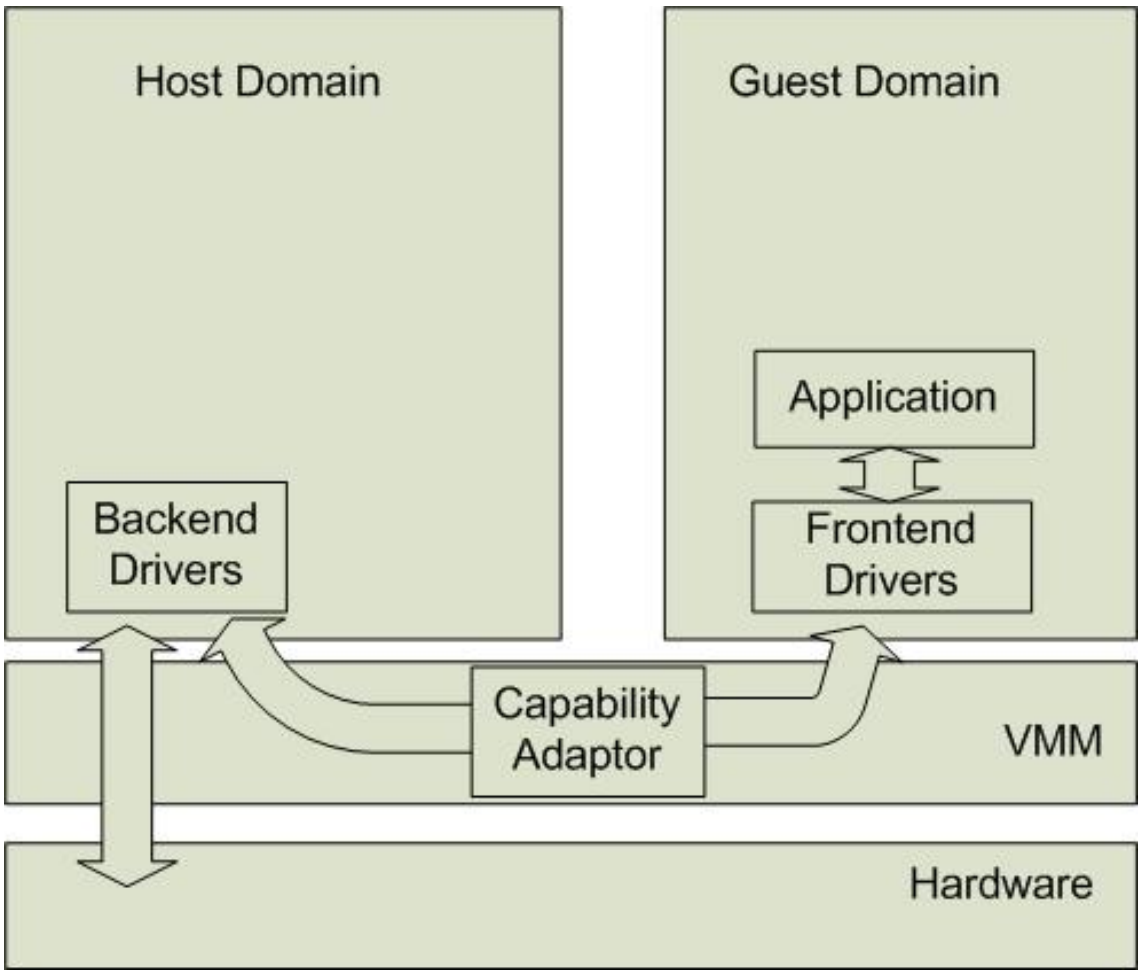


**Figure 21:** Design 2:No BE and FE changes, CA in VMM

but not anywhere else.

Table 7 summarizes the above descriptions of the three designs in term of the amount of modification they need and where the modification needs to be done. A comment on the qualitative tagging of changes as "Small" and "Big" in this table. Based on our experience with Xen, intercepting the connection between the frontend and backend is relatively easy since it only requires splitting the connection code from the driver code. Therefore the amount of code that needs to be modified is minimal. However, changing the existing Xen's split device driver mechanism requires big effort in changing the core code of Xen and needs greatest effort in debugging and testing since it touches the heart of Xen. Therefore, we tag the effort involved in the

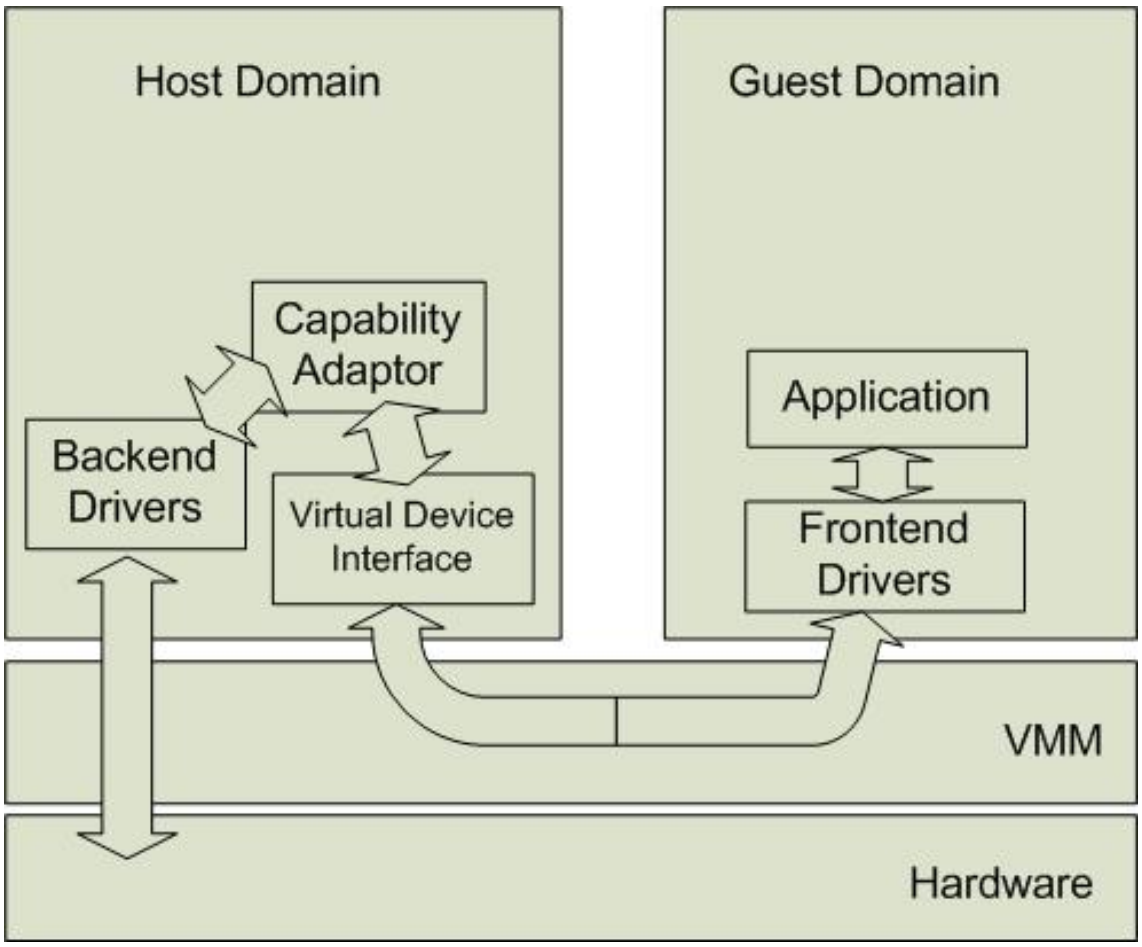


**Figure 22:** Design 3: No changes in FE and VMM, CA in host domain

envisioned changes to support the design choices as "Small" and "Big" in the table accordingly.

According to our design principles, design 1 is not desirable since it requires the change to the guest domain. We want to keep the guest domain unchanged so that the host environment can always know the guest domain and how to adapt to it. Compared to design 3, design 2 requires significant changes to Xen's VMM and core code, which enlarges the amount of modified code and requires clear understanding of every core module in VMM. Design 3, on the other hand, only requires small amount of code changes and further no changes to the guest domains, which is more desirable than the other two designs. Therefore, if a new version of Xen is released, it can be

**Table 7:** Comparison of different designs

| | Design 1 | Design 2 | Design 3 |
|---|---|---|---|
| Guest domain modification | Small | None | None |
| Host domain modification | Small | None | Small |
| VMM modification | None | Big | None |

easily integrated into our system with minimal effort if we use design 3.

## *5.5 Chameleon Architecture*

According to the discussion in the previous section, we design our system architecture as shown in Figure 23. This architecture follows the third design we discussed previously with additional modules that are required to support our design principles.

As shown in the figure, we have a pool of backend device drivers that control physical devices. Different from original Xen backend drivers, these drivers in our system do not connect to frontend drivers directly. Instead, the connection manager connects them to appropriate capability adaptors (discussed later) and then to an instance of the virtual device interface, which frontend drivers can discover and connect when the guest OS comes online. After the connection is made, the data flows without the involvement of the connection manager, as the arrows indicate in Figure 23.

We also have a registry that keeps the information about all the capability adaptors that are registered in the system. The match maker takes the information and decides which algorithm to use at run-time based on the information provided by the connection manager. For example, the connection manager can tell the match maker that the virtual device interface expects an 800*600 screen but the only backed driver available is for a 400*300 display. The match maker may take the "shrinking" algorithm but not "expanding" algorithm in this case. The connection manager can take the decision made by the match maker and connect the appropriate adaptor with other system modules.

Capability adaptation is done by breaking the frontend/backend connection and

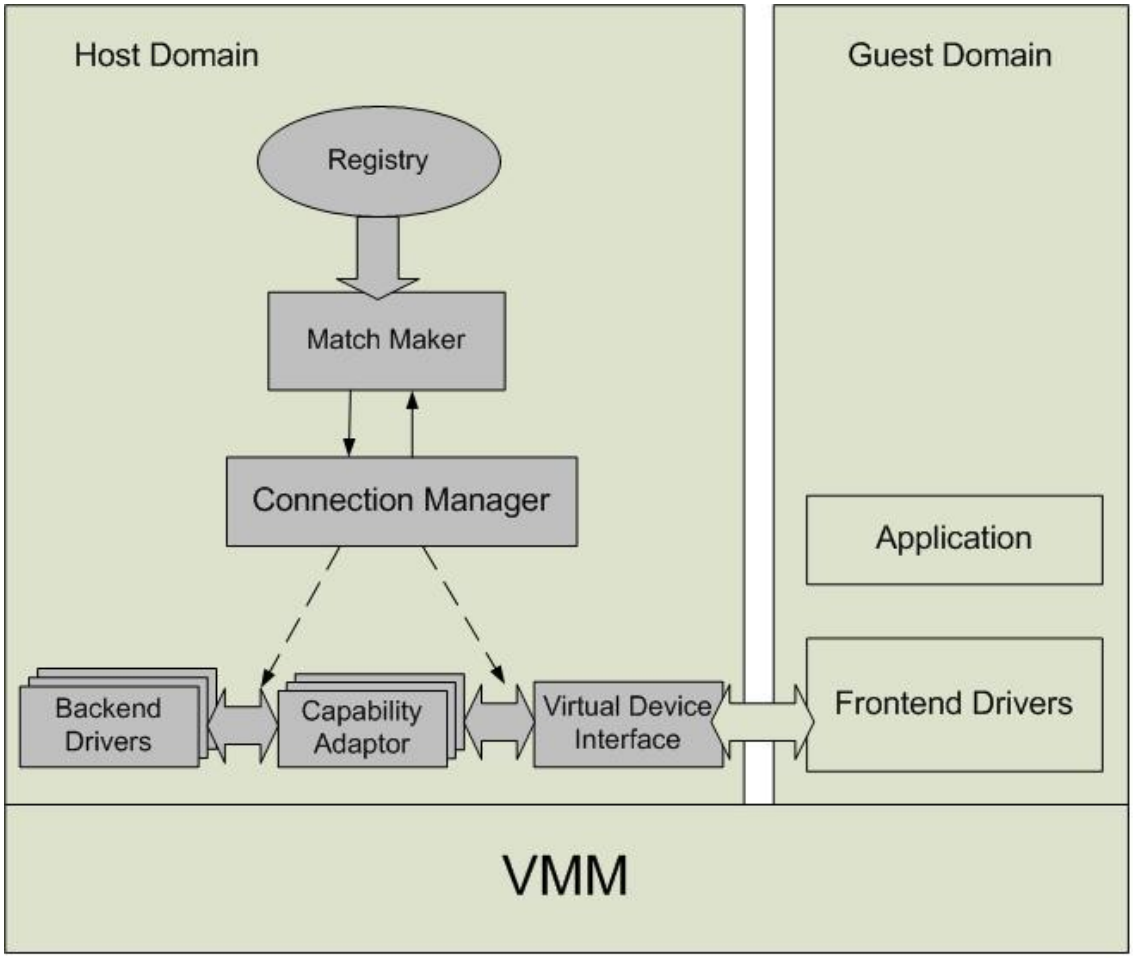


**Figure 23:** Chameleon System Architecture

adding capability adaptors (CA) between them. In this case, FE issues requests to CA instead of directly to backend drivers. CA can do adaptations on the requests based on its information about the frontend and backend and then forward the requests to backend to process. For example, if the guest OS was originally connected to an 800*600 display but is now migrated to a 400*300 display, the capability adaptor, by knowing that information, can shrink the frame buffer accordingly when frontend issues the updates on the 800*600 frame buffer (frontend still thinks it has an 800*600 screen in this case). Similarly, another capability adaptor can expand the screen if the destination framebuffer is larger than the original.

More specifically, the guest domain in our system is running on mobile platform

and is moved with users. The host domain runs the environment system that provides resources to the guest domain for better user experience. When user leaves an environment, the guest domain is paused and the state is dumped into the mobile device. At the destination environment, the guest domain is resumed and user's activities are continued. Note that the guest domain that is migrated is always minimized to reduce the cost for migration. For a movie service, the actual movie file is not included in the guest domain. The movie is normally streamed from another source on the Internet.

Our system design meets all the design principles that we presented in section 5.2. All the modules in our system are in the host domain and we keep the guest domain unchanged and generic for the migration. The registry in our system allows the dynamic registration of capability adaptation algorithm, which can be used later for selection. Removing an entry in the registry prevents the future selection of the corresponding algorithm and therefore is an "uninstallation" of the adaptor. The match maker in our system takes charge of the algorithm selection for particular adaptation needs. The connection manager can dump the device state by calling the virtual device interface and packing the state with the domain image for later resumption of the domain on the destination platform.

### 5.5.1 Discovery

Device discovery in Xen means the actions taken by the frontend drivers to find the corresponding backend driver when the guest domain comes online. In original Xen system, Frontend drivers do it by writing to the appropriate device type's entry in Xenstore [42], a registry database in VMM that is accessible from any domain. Backend driver will be notified if particular entry in Xenstore changes and then start the connection initialization. After that, FE and BE communicate the basic connection information through Xenstore to set up event channels and shared pages to make

the connection.

In our system, we separate the discovery from the connection establishment. The connection manager will set up notification of changes at specific entries in Xenstore and wait for frontend drivers to discover. After a frontend driver initializes the connection by changing entries in Xenstore, the connection manager initiates an instance of virtual device interface and hands over to the virtual device interface for the connection establishment with the frontend. The separation of discovery from connection establishment moves the connection manager out of the common path for data transfer while still setting up a central point of contact for discovery.

### 5.5.2 Connection establishment

The virtual device interface takes charge of the connection with the frontend driver as described above. It sets up the event channels and shared pages with the frontend for later data communication based on Xen's mechanism. The virtual device interface also takes suggestions from the connection manager as to which capability adaptors it should connect to and which backend driver will handle the requests. In our current implementation, we took the connection code from original Xen's backend for the connection to frontend drivers since we wish to simulate the exact behavior of Xen's backend to make our system transparent to the frontend driver. We use dynamic linked library for the capability adaptors in order to be able to load them at run-time. The CA library names are recorded in the registry and provided to the virtual device interface upon request. These CA libraries are required to implement several functions in order to make the connection happen successfully.

### 5.5.3 Virtual device to physical device mapping

Virtual device interface is mapped to physical device backend dynamically at run-time. At startup of a guest domain, the connection manager chooses the best resources for the guest domain and maps the virtual device interface to the best

possible physical resource backend. If the guest domain is migrated from another environment, the virtual device interface learns the source device specification from the frontend drivers and passes it to the connection manager. The connection manager then selects the physical resource backend that best matches that specification (e.g., screen size, resolution) for the guest domain. If a match is found, then the mapping is made directly from the virtual device interface to the physical resource. However, in most cases, the exact match may not be found. Therefore, we need capability adaptation to solve the heterogeneity problem.

### 5.5.4 Capability adaptation

In case the virtual device is different from the physical device, capability adaptation has to be done by capability adaptors in our system. Capability adaptors are classified by categories based on their functionalities. For example, video adaptors are able to adjust frames based on their original size and target size. Keyboard adaptor can change the key codes accordingly to make the source and destination input device look like the same to users. Each adaptor has to implement a certain set of functions that are required for its category. This set of unique functions implemented by all adaptors in a category enables the dynamic change of adaptors in this category at run-time. The implementation details of these functions are presented in the implementation section later.

Note that the adaptation algorithms themselves are not the focus of our system and therefore are not the main contribution of this work. Our goal is to enable the dynamic installation and selection of capability adaptation algorithm at run-time to ensure the seamless migration of user's activities.

### 5.5.5 Data flow

Different types of devices have different data flow in Chameleon. Figure 24 shows a typical data flow. Applications in guest domain issue requests for operations on a

device. The requests are sent to the frontend virtual device in the guest domain, which connects to the virtual device interface in the host domain. Typically, a capability adaptor is needed and the access requests are forwarded to the capability adaptor to do the required adaptation before they reach the physical backend device for the actual operations.

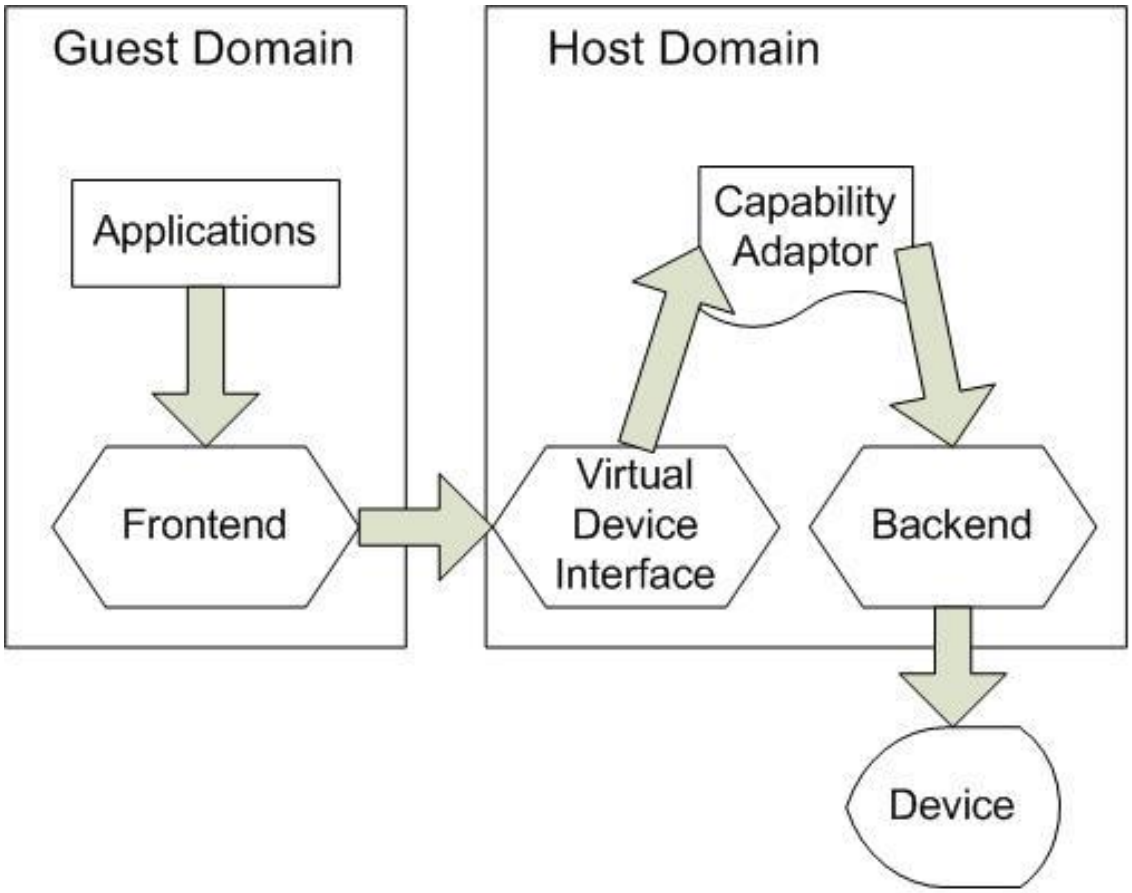


**Figure 24:** Data flow in Chameleon

Note that the connection manager is not in this common path. Therefore, we do not have additional management overhead in the entire process as every module is critical to process the data. Compared to data flow in the original Xen split device driver model, the only additional module that the requests need to go through is the capability adaptor. However, this module is not even required if the virtual device interface matches the physical device backend exactly.

### 5.5.6 Device state migration

There are a variety of internal states of backend drivers that may be necessary to be migrated in order to ensure seamless operation. Framebuffer in a display device drivers is one such example of such states. These states are internal to physical device drivers and out of the control of guest domains. Therefore, only migrating the guest domain may cause these states to be lost. We need to capture and resume such states with the migration of guest domain in order to provide the real unstopped service to mobile users.

Device state is captured by the connection manager when the migration is triggered. The connection manager collects all the internal states of the physical backend driver and packs them up with the migration of the guest domain. At the destination, the connection manager extracts the states and dumps them into the selected physical backend accordingly. These states may also subject to be adapted if the source and destination devices are heterogeneous.

## *5.6 Implementation*

In this section, we present implementation details of Chameleon to achieve our goal of device state migration and capability adaptation. To make the discussion concrete, we present details of two categories of devices (display and keyboard). However, it should be noted that our architecture is general and applies to any device that adheres to the split device driver model of Xen. The performance evaluation results that we present later are based on our current implementation of the system.

### 5.6.1 Data structure for virtual device, physical resource and their mapping

We use different data structures to represent virtual device, physical backend device (resource) and the mapping between them. Given below are these data structures.

(1) virtual device structure:

```
struct virtdevice_
{
    int devType;
    char devName[64];
    int devID;
    int domID; //DomU id
    struct xenfb* xenfb; //virtual framebuffer
    void* extraInfo;
    struct virtdevice_* nextdev;
};
```

The virtual device structure records the information of the virtual device interface. The virtual device is discoverable by the frontend drivers from its device type (devType) and device name (devName). The `xenfb` field in the structure is used for virtual framebuffer to identify the framebuffer it uses. The virtual device structure is a linked list and the `nextdev` pointer points to the next node in the linked list.

(2) resource structure (physical devices):

```
struct resource__
{
    const char* resName;
    int resID;
    int resType;
    int resStatus;
    void* extraInfo;
    adapter_t* padapter;
    void (*useResource)(void*);
    void (*reuseResource)(void*);
    struct resource__* next;
};
```

The physical driver (resource) structure is an internal structure of our system. We use it to maintain the physical backend device driver's information and whether it has been used or not (`resStatus`). The two function pointers in the structure points to two required functions of the physical backend device driver that can be used to utilize the device and resume the device state. The `extraInfo` field stores device specific information for each device category. The resource structure is also a linked list and the `next` pointer points to the next node in the list.

(3) structure for virtual device to physical device mapping:

```
struct mapping__
{
    int mapID;
    int backendID;
    int frontID;
    state_t MapState;
    pthread_t tid;
    pthread_mutex_t mapMutex;
    virtdevice_t* pVirtdev;
    resource_t* pResource;
    adapter_t* pAdapter;
    struct mapping__* pnext;
};
```

The mapping structure stores the mapping information between physical backend resources, capability adaptor and virtual devices. Since our system creates a separated thread to handle the requests for a particular mapping in order to avoid the interference between devices, this structure also records the thread information as well as a mutex for the thread. The mapping structure is a linked list and the pnext pointer points to the next node in the list.

### 5.6.2 Capability adaptor interface

In our system, we consider two types of capability adaptation: frame buffer adaptation for display device and key stroke adaptation for keyboard device. Both of these adaptors have to implement certain functions that can link to the entire system dynamically at run-time. Given below is the code for these functions and the frame buffer update structure definition:

```
updateDisplay(void* src_buf, void* dst_buf,
        struct fbu *src, struct fbu *dst);
updateKB(char *src_code, char *dst_code,
        int src, int dst);
struct fbu //Frame buffer update info
{
    // Update Area info
    int x; // start point in x axis
    int y; // start point in y axis
    int w; // width of the update area
    int h; // height of the update area

// Setting info
    int size_w; // width of the full screen
    int size_h; // height of the full screen
    int r; // color setting for red
    int g; // color setting for green
    int b; // color setting for blue
};
```

These two update functions are self-descriptive: they take the source frame buffer or key stroke and convert it into destination based on the source and destination information (`src` and `dst`). The `updateDisplay` function also sets `x, y, w` and `h` field in the `dst` as an output since these values will be used later to update the actual device. (For example, if the source device is 800*600 and it updates the area ((100,200),(240,300)), then the destination device with a screen size of 400*300 should update the area of ((50,100),(120,150)) accordingly.)

The capability adaptor is represented by the following structure in our system:

```
struct adapter__
{
    int adapterID;
    char adapterName[32];
    int state;
    void* (*adaptfunc)(void* src, int x, int y, int width, int height);
    struct adapter__ *next;
};
```

This structure maintains the basic information about the capability adaptors such as their IDs, names, state (used or unused) and adaptation function pointers. This structure is a linked list and the `next` pointer points to the next element in the list.

### 5.6.3 Registry and match maker

We store the capability adaptor information into our registry for the match maker to select at run-time. Each entry in the registry contains two parts: capability adaptor library name and a flag field that shows the types of adaptation that can be done for the corresponding algorithm. The flag has 5 binary bits that represent 5 types of adaptations:

- Extend the screen horizontally
- Shrink the screen horizontally
- Extend the screen vertically
- Shrink the screen vertically
- Do color setting change

For example, an adaptation algorithm that is very good at extending the screen both horizontally and vertically but cannot do the color setting change will have a flag of 10100 in the registry. Another algorithm that is particularly good in shrinking the screen vertically and can do color setting change will have a flag of 00011.

At run-time, the match maker takes the source and destination device information and selects the appropriate algorithm based on the flags in the registry. It passes the library name to the connection manager for loading and connecting to other modules.

We also provide a small tool for adding/removing entries to/from our registry for the capability adaptors. Basically, a developer of the adaptation algorithm compiles the algorithm code to get a dynamically linkable library. Then they can use our tools to specify the types of adaptation presented by the algorithm together with the library name to add it into our system. Our tool also allows users to delete an entry from the registry in case the adaptor library is removed or no long needed.

### 5.6.4 Framebuffer device - the backend

As we discussed before, we choose to use virtual framebuffer support of Xen in Chameleon. However, virtual framebuffer device drivers are always mapped to real framebuffer device in order to draw pixels on the screen. In the current implementation of Chameleon, we use two types of real framebuffer: a simulated framebuffer device using a VNC server, and a physical framebuffer device using NVIDIA framebuffer device drivers. Both of these framebuffers use a modified `xenfb` structure (used in legacy Xen system to represent framebuffer device) to store critical information about the device, which can be retrieved by capability adaptor for proper adaptation. The modified `xenfb` structure is defined as follows:

```
struct xenfb
{
    void *pixels;
    int row_stride;
    int depth;
    int width;
    int height;
    int abs_pointer_wanted;
    void *user_data;
    void (*update)(struct xenfb *xenfb,int x,int y,int width,int height);
    void* pResource;
};
```

#### *5.6.4.1 VNC framebuffer*

The simulated VNC-enabled framebuffer device driver starts a VNC server and uses it as the target device for display. This framebuffer device can be initialized and updated as regular physical framebuffer device. Different from physical framebuffer device, this simulated device draws pixels in the VNC server framebuffer instead of drawing to the physical framebuffer on the physical device. Any VNC client can connect to the VNC server to see the screen that is supposed to be displayed on the physical device. The advantages of using the simulated driver are the following:

- Original Xen's virtual frame buffer driver is a similar VNC-enabled simulated driver. We can take advantage of that code to implement the VNC server and the updates.
- By using a simulated driver, all the codes are running at user level, which makes the development and debugging much easier.

- The simulated driver behaves exactly like the physical driver. The only difference is that instead of sending commands to physical device, it sends commands to the VNC server.

The VNC framebuffer uses the VNC library, libvnc [41], to create a VNC server screen and does updates on the screen pixel by pixel. VNC protocol allows the update of certain part of the screen instead of the full screen as long as the correct coordinates are clearly specified. We create a VNC framebuffer with the size of 800*600 and true color setting (32 bits per pixel). Each pixel value is stored in a continuous 4-byte space and can be updated individually in Red, Green, Blue and Transparency accordingly.

#### *5.6.4.2 Physical framebuffer*

In addition to simulated framebuffer, we also integrate NVIDIA framebuffer, a physical device driver, into Chameleon to test the framebuffer in real scenarios. This physical framebuffer, similar to the simulated framebuffer, can be initialized and updated to reflect any changes on the screen. Internally, different from simulated framebuffer, this NVIDIA framebuffer uses lower level system calls and assembly instructions to operate the video card on the machine. The implementation gives us hand-on experience on how the physical device drivers behave in real scenarios. We will show our performance evaluation results in the next section and compare physical drivers with simulated drivers.

This NVIDIA framebuffer has a screen size of 640*480 and 4-bit color setting. Each pixel can have 16 different colors in this physical framebuffer. In addition, different from VNC framebuffer, NVIDIA framebuffer is not organized pixel by pixel. It uses 4 bytes to store the color value of continuous 8 pixels in a row. Each byte represents blue, green, red and transparency respectively for all 8 pixels. Table 8 shows an example of how the first four bytes in the framebuffer store the color of the first 8 pixels on the screen. As shown in the table, the first pixel has the color value

**Table 8:** NVIDIA framebuffer format

| Pixel | (0,0) | (0,1) | (0,2) | (0,3) | (0,4) | (0,5) | (0,6) | (0,7) |
|---|---|---|---|---|---|---|---|---|
| Byte 0 (Blue) | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| Byte 1 (Green) | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| Byte 2 (Red) | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| Byte 3 (Transparency) | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |

of (0,1,0,1) in the column (first bit of these four bytes). Similarly pixel (0,5) has the color value of (0,0,1,1).

This is an example to show different framebuffer may have different resolutions, color settings or even byte format. Therefore, we do need capability adaptation for the framebuffer in order to bridge the gap between different framebuffers and allow the seamless migration of the screen from one framebuffer to the other.

### 5.6.5 Capability adaptation algorithm

We have implemented two types of capability adaptation algorithms in our system. One type of algorithm is the basic scaling algorithm to target the screen size difference between the source and destination devices. This scaling algorithm scans the screen line by line and computes the average value of each pixel according to the source and destination device configurations. For example, if the source screen is 800*600 and the destination screen is 400*300, this algorithm uses the average color value of 4 pixels on the source screen to compute the color value of one pixel on the destination screen (i.e., D(x,y) = avg(S(2x,2y), S(2x+1,2y), S(2x,2y+1), S(2x+1,2y+1)) where D(x,y) is the color value of the pixel (x,y) on the destination screen and S(x,y) is the color value of the pixel (x,y) on the source screen.). Weights will be assigned to the avg function for the pixels if the scaling is complex (i.e., not exactly 4 to 1 mapping). Again, our system does not focus on specific adaptation algorithms themselves but their dynamic linkage at real-time when the guest domain is migrated.

### 5.6.6 Frontend and its communication to the backend

In this subsection, we describe the existing mechanism for device driver communications in Xen and Linux. This description is essential to understand how we have integrated capability adaptation into the existing Xen code base.

In our system, frontend drivers follows the requirements of a regular device driver that runs in legacy Linux system. Two critical functions need to be implemented in the frontend drivers in order to make guest domain operating system aware of its presence:

```
void module_init(void *);
void module_exit(void *);
```

The frontend device driver can be installed into the kernel by using the tool "modprobe" and the guest operating system will call the above function module_init to start the initialization process. The modprobe tool can also be used to uninstall the driver, at which point the above function module_exit will be executed to clean up the data structures of the driver and release resources.

Frontend starts to communicate with the backend through Xenstore [42] at the beginning to do initialization. Xenstore is a centralized storage space that can be accessed by any domain. Similar to Windows registry [43], Xenstore normally contains critical configuration information rather than large chunks of data that needs to be transferred across domains. The following functions are important to access Xenstore from frontend/backend drivers:

```
void *xs_read(struct xs_handle *h, xs_transaction_t t,
        const char *path, unsigned int *len);
bool xs_write(struct xs_handle *h, xs_transaction_t t,
        const char *path, const void *data, unsigned int len);
```

The initialization includes the allocation of resources, the set up shared pages and event channels, and the change of internal states. The frontend driver also needs to register itself to Xen hypervisor in order to be recognized. The following hypercalls are used for these purposes:

```
int xenbus_register_frontend(struct xenbus_driver* driver);
int xc_evtchn_open();
evtchn_port_t xc_evtchn_bind_interdomain(int xce_handle,
        int domid, evtchn_port_t remote_port);
void *xc_map_foreign_batch(int xc_handle, uint32_t dom,
        int prot, xen_pfn_t *arr, int num );
```

After initialization through Xenstore, the frontend and backend use the event channel and shared pages for communication instead of Xenstore. When the frontend needs to send some data to the backend, it first writes the data into shared pages and then sends an event to the backend through the event channel. Backend can recognize the event immediately and may process the data, write the results back to shared pages and then send an event back to frontend through the event channel. Frontend gets the event and retrieves the results from shared pages.

## *5.7 Performance*

In order to justify how our system affects the operating system by providing device migration and capability adaptation, we conduct some experiments to measure the overhead of the system. We use two identical AMD Athlon(TM) 64X2 dual-core machines with 1G memory and Ethernet connection to each other. Both machines have Fedora Core 5 Linux with Xen 3.1 installed as the software platform.

### 5.7.1 Performance of capability adaptation

The first experiment is to evaluate the overall performance of our capability adaptation system and how much additional overhead we introduce compared to original Xen's split device driver model, since we build our system based on Xen. This measurement can indicate if our system is good enough in terms of performance when adding capabilities adaptation to Xen.

We measure the following two types of costs in this experiment:

- **Guest domain initialization cost**: it measures the total latency to setup a new guest domain for capability adaptation. This cost includes the latency for backend device driver discovery, backend-frontend connection establishment and the first frame buffer initialization.

- **Incremental capability adaptation cost**: it is the latency for processing guest domain requests to access virtualized devices after initialization.

We compare both of these costs to original Xen's costs to demonstrate that our system does not introduce significant overhead when providing additional functionalities (i.e., capability adaptation) to Xen.

In our current implementation, we use framebuffer devices to demonstrate these two costs because the update to a certain part of the screen may include a large amount of memory copying, which is time-consuming compared to updates in other types of devices (such as inputting ASCII codes corresponding to key strokes in keyboard devices). Our experiment is intended to show the "worst case cost" for the two figures of merit (guest system initialization and incremental adaptation) among different categories of devices.

Figure 25 shows our results for framebuffer devices with different capability adaptors. Please note that the incremental capability adaptation cost is shown as "cursor

blink" in the figure because the cursor that blinks on the screen necessitates the frequent update of a small portion of the screen. Such costs match our description of incremental adaptation cost so we use the latency of "cursor blink" as an example to show the incremental cost of our system with regard to capability adaptation..

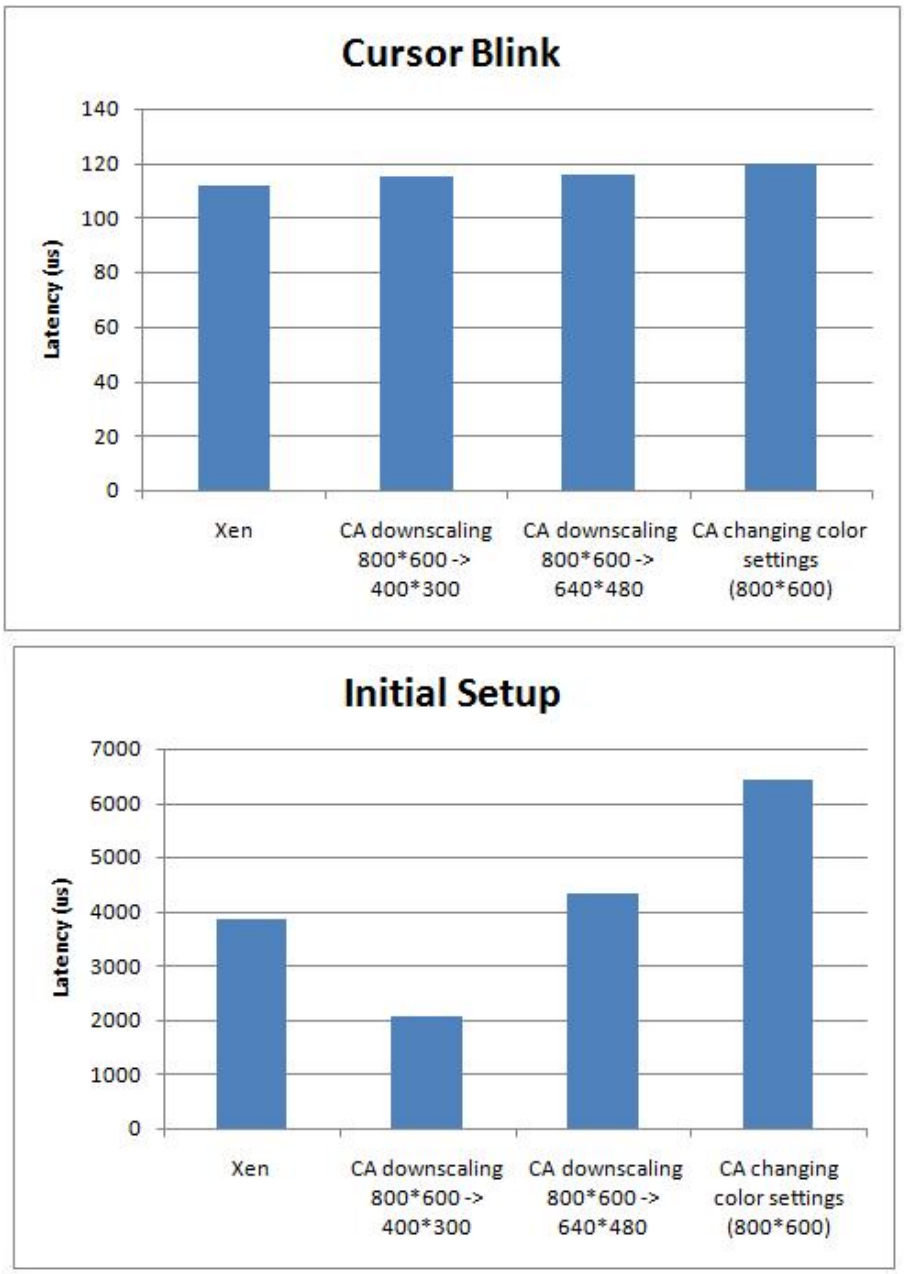


**Figure 25:** Performance of Chameleon and Original Xen

In the figure, we can see that for a cursor blink, our system with different capability

adaptors has similar performance as original Xen's virtual frame buffer device. We only introduce 2% to 7% overhead compared to Xen. This additional overhead does include the overhead of capability adaptation algorithm and therefore does depend on the performance of these algorithms. However, we use very simple algorithm in these experiments so it indeed shows the performance of the "overall" performance with simple algorithm. However, we do not always have control over the performance of the adaptation algorithm. If the algorithm is more complex, the latency might be bigger. Since the capability adaptation algorithms are not focus of our system, we want to exclude the influence of particular adaptation algorithm as much as possible. This experiment is analogous to measuring the cost of a "null message" in a message communication library. In other words, the experiment is designed to measure the cost incurred by the code path corresponding to the dataflow for device updates shown in Figure 24. From Figure 25, we can see that our system performs well compared to Xen while providing capability adaptation in addition to Xen.

In contrast to cursor blink, the guest domain initialization of Chameleon introduces noticeable costs compared to original Xen for the two adaptation algorithms. However, for one adaptation algorithm (800*600->400*300), our system even outperforms Xen. The reason for such a difference is mainly due to the memory copy cost: the guest domain initialization requires a copy of entire screen to the physical framebuffer once. With capability adaptation, we need to first "adapt" the screen and then copy it into the physical framebuffer. This additional copy of entire screen causes the difference in performance. In the 800*600->400*300 downscaling adaptation algorithm, we need to adapt and copy only 400*300 screen instead of the entire 800*600 screen in Xen. Therefore, the performance of our system with this algorithm even outperforms Xen. Figure 26 shows the details procedure of this memory copy and Figure 27 shows the time for copying different sizes of the memory.

Again, this experiment shows the performance of our system is good compared to

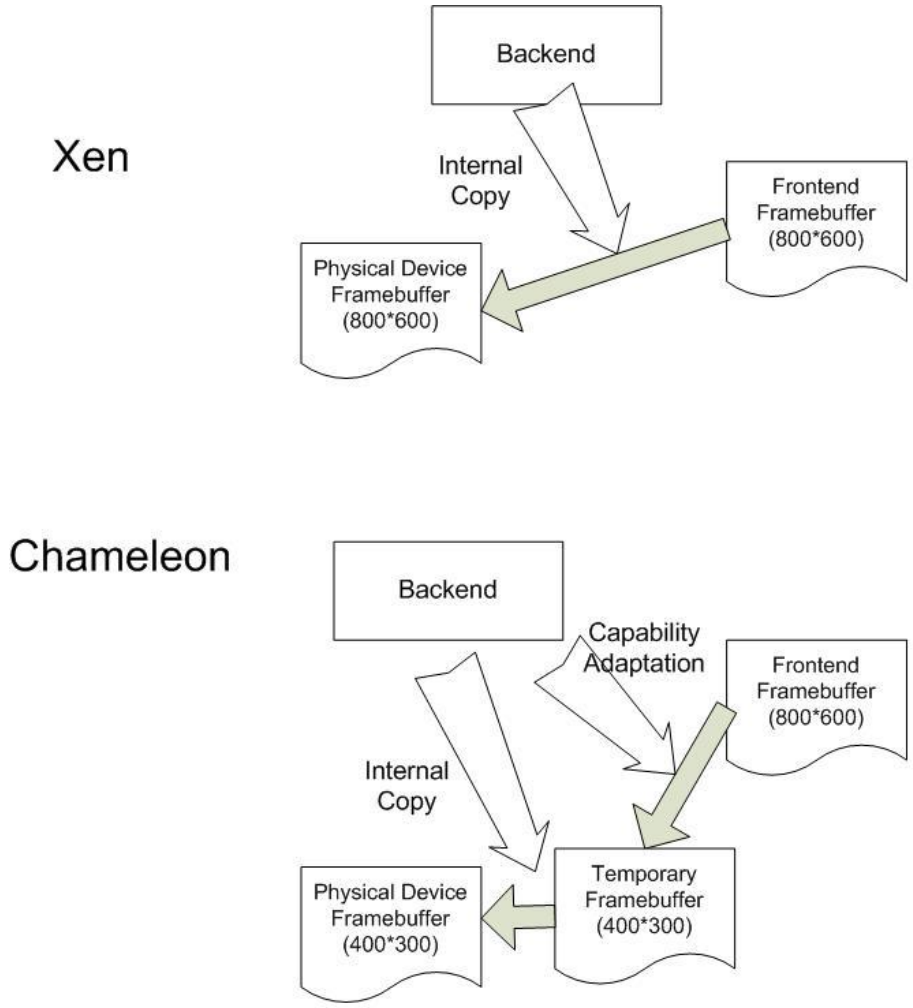


**Figure 26:** Additional memory copy for Chameleon

Xen. Similarly, we want to measure the performance of our system but not specific adaptation algorithm. By using these simple adaptation algorithms, we are able to demonstrate our system performs well with basic adaptation. We do know that these performance results are algorithm dependant. However, since the design and implementation of particular algorithms are out of our control, we only use simple algorithms to show the performance of our system compared to Xen.

To generalize our result, we can expect similar or better results for devices other than framebuffer. For example, input devices do not always have such a big memory cost shown earlier. Therefore, different capability adaptation algorithm will not

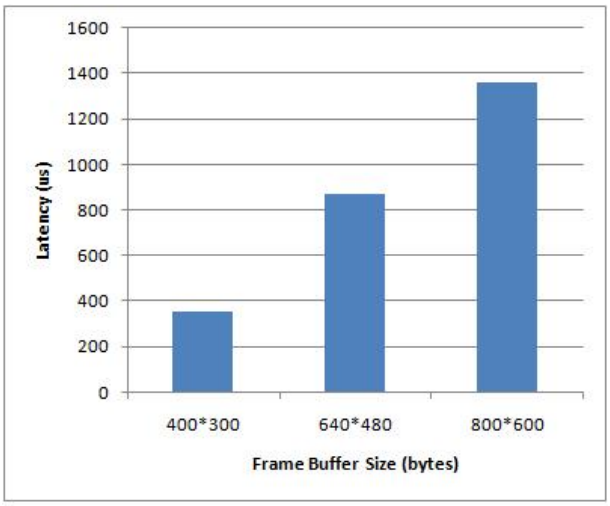


**Figure 27:** Time for copying different sizes of framebuffers

make a difference on the performance numbers. The incremental adaptation cost for input devices can be also similar to the results presented for framebuffer device since the update request of input device, similar to the update of a cursor on a screen, is typically small and match the results shown for framebuffer devices. Audio devices may also have similar results for incremental adaptation cost but may have big difference for guest domain initialization, since the adaptation algorithm for audio may be complex and costly.

### 5.7.2 Performance of the state migration

The second experiment is to measure the cost for migrating the device state from one physical device to another physical device. Since the device state migration is one key feature of Chameleon, we want to measure the cost for such a migration. Such a migration is a user initiated operation so we will judge our result in term of user experience. We expect a small amount of latency for such a migration from a user's perspective.

In this experiment, we start two framebuffer backend driver and make one of them connect to the frontend driver and display the screen. Then, we issue a command

to the running backend driver to make it migrate to the other framebuffer backend driver at run-time. The results we present in Figure 28 shows the costs for such a migration.

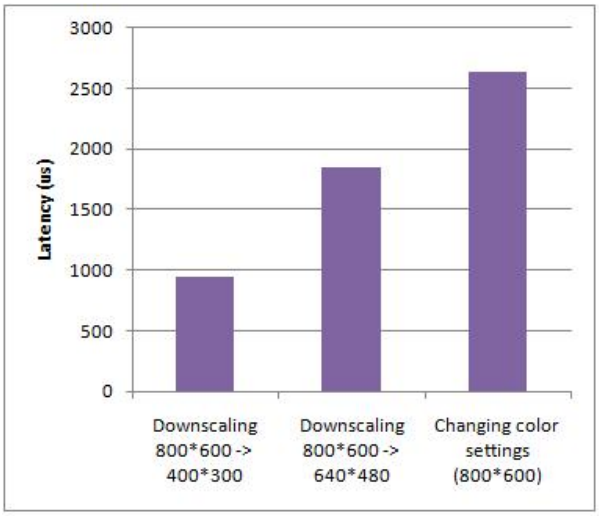


**Figure 28:** Device state migration costs

In the figure, we can see that the cost for such a migration is 1ms - 2.5ms in total. With reference to Figure 27, we can see that most of the cost is due to the memory copy of the frame buffer from one driver to the other. As we stated, this experiment is to measure the migration cost and it is a user-centric measurement. An additional 1-2.5ms for the migration from one device to the other is not noticeable from a user's point of view. Therefore, such an overhead is acceptable in terms of user experience.

To generalize our evaluation, we can expect similar or better results for other types of devices. State on other devices may be lighter than display devices. For example, a keyboard device driver may only store a few ASCII codes in its buffer as device state. Compared to a full screen of pixels, such ASCII codes are very light and therefore do not take significant time to be migrated to another input device driver. Similarly, audio device may have state such as a few millisecond of sound that will be played. The amount of such state may vary depending on the format of the sound. However, we expect the amount of state stored on most types of device drivers to be less or

similar to framebuffer device drivers. Therefore, it is fair to say that the performance results reflect a worst case cost for system initialization and adaptation incurred by our modified Xen system that supports device level capability adaptation.

### 5.7.3 VNC framebuffer vs. physical framebuffer

In addition to the two experiments of Chameleon shown above, we present an additional evaluation with regard to framebuffer device in Xen. In Xen, the framebuffer device is implemented using VNC [4] enabled backend. Such a simulated VNC backend allows any VNC client to connect to see the actual screen. However, we would like to see what is the difference if we use a physical framebuffer device rather than the simulated VNC framebuffer provided by Xen. We can also understand if the results we presented previously are particular to a certain implementation of framebuffer or can be applied to any type of framebuffer devices.

In this experiment, we use a physical framebuffer device with the screen size of 640*480. We compare the overhead of VNC framebuffer with the same size and conducted evaluation on both guest domain initialization cost and incremental capability adaptation cost. Figure 29 shows our results.

In the figure, we can see that both costs of the physical framebuffer are less than the VNC framebuffer. We think that difference is due to the relatively slow in-memory processing of VNC framebuffer since it needs to "simulate" the behavior of physical framebuffer. The physical framebuffer, in contrast, does not need such a simulation and therefore performs faster than VNC framebuffer. However, with this additional overhead, the simulated framebuffer can be easily managed for the development of a system on top of it without worrying about details of framebuffer devices.

We also note that although there are differences between VNC framebuffer and physical framebuffer, the overheads of both framebuffers are relatively small compared to original Xen's framebuffer without capability adaptation (it even performs better

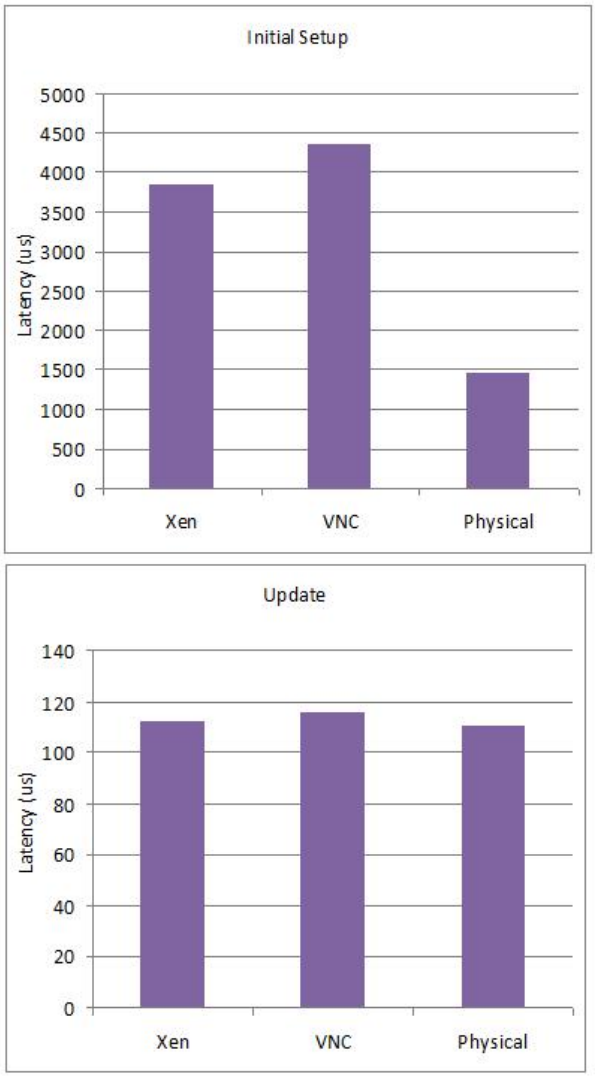


**Figure 29:** VNC framebuffer vs. Physical framebuffer

than original Xen for physical framebuffer since Xen originally uses VNC framebuffer). Therefore, we conclude that the performance results we presented previously are not tied to a particular implementation of framebuffer.

## *5.8 Summary*

In this chapter, we have described our device level virtualization in detail. First, we have presented our design choice and then the base system on top of which our

system has been built. After that, we have shown our system architecture and implementation details, followed by performance evaluation results, which demonstrate that our system has minimal overhead compared to the base system Xen.

Device level virtualization can solve some of the problems in seamless mobility that cannot be addressed by service level virtualization, such as device state migration and capability adaptation. However, it has it own limitations as we present in Chapter 2: it requires the entire guest domain (operating system plus applications) to be available at destination platform. In some cases, this might be difficult. We consider possible interactions between the two levels that may help to reduce the size of the entire guest domain image. We will discuss these future extensions in the next chapter.

# CHAPTER VI

# DISCUSSION

In the earlier chapters, we have presented our approach to the problem of seamless mobility at two different levels. However, there are many open problems in this space that need to be further addressed. In this chapter, we discuss these problems and possible solution approaches that may form the basis for future extensions of this work.

## *6.1 Other category of services*

As we mentioned in the Chapter 2, we identified three categories of services and the approaches we use in this dissertation solve problems for the first two categories but not the third one. The main reason for this is because the dynamic network state (server side state) is not accessible to user's mobile platform (client). For example, if a user wishes to buy a computer from Dell, there are typically several pages of forms for a particular order. If he cannot finish his purchase before he needs to move to a new place, he has the server state (which is his submitted partial order) and the client state (which is the current incomplete order page) for such an E-commerce service. Seamless migration of such a service may not be possible since this user may have a different IP address in the new environment and Dell has to re-authenticate the user (which makes the migration not seamless). The client has no control over the server state and therefore cannot migrate such a type of state. Even if the client has the connection information, the seamless migration of the connection may not be possible since a re-connection and re-authentication requires an interaction between the client and the server.

A possible extension of our system is to take the server side state into consideration

in addition to client side state. Although the server is out of the mobile user's control, we can still develop another supporting system on server side to enable servers to support the resumption of their own state upon a re-connection of the client. By combining both the client side and server side state, we can further extend our system to support more services in addition to the first two categories.

## *6.2 Discovery and authentication*

We present our work to address issues for seamless mobility in previous chapters. However, we do know there are some other issues that are critical to a successful seamless mobility system. These issues, such as security and discovery, are not central to this dissertation. Therefore, we have used simple solutions to address them in the overall system architecture. We discuss these issues in this section and present the limitations of the current solutions used in the systems we have built, and suggest possible extension to further address these issues.

The current discovery model allows a mobile platform to discover the environmental services using a designated multicast address. This multi-cast address has to be available in all environments that users want to move into in order to enable the discovery. A possible extension of this model might be location-based discovery, where the mobile platform can identify its location by a location service (GPS [63], RFID based location sensing [64] etc.). The mobile platform may contact a central server with its current location information to locate the discovery server IP address in the current environment.

In addition, the current authentication model used in MobiGo assumes the environment knows the user (represented by a user ID provided by the mobile device) ahead of time, even for the totally-new space. Clearly, this may not be a viable assumption in a totally new space. There are several ways to remove this restriction using more sophisticated solutions that are already available. For example, we could

use a decentralized authentication model [65, 66]. Similarly, we could public key infrastructure [67, 68] to enable the two-way authentication without revealing sensitive information from the mobile device to the environment. Further, there is opportunity for new work in this space to address this problem taking into consideration the specificity of the ubiquitous computing environment.

## *6.3 Guest domain migration for device level virtualization*

In device level migration, we consider the device state migration and capability adaptation. However, we always assume the guest domain is available at the destination before we can do device state migration and capability adaptation. The problem of how to migrate the guest domain to the destination environment is not discussed in previous chapter but that might be a problem in some cases.

We use Xen as our base system for device level virtualization. The virtual machine migration is a built-in functionality of Xen and therefore we can simply use it for the guest domain migration if we need to migrate the guest virtual machine (guest domain) from one place to the other. Our system takes charge of the device state migration and capability adaptation after the guest domain starts on the virtual machine manager (VMM) and tries to find a backend driver to resume its activity. We rely on existing Xen mechanisms to pack the entire application and operating system (the entire virtual machine) and resume on a different VMM. Our system mimics the behavior of the original Xen's host domain and can give the guest domain the illusion of connecting to a normal host domain. Therefore, we do not need to change the mechanism of VM migration in Xen.

Another question is how we can migrate the entire VM to the destination. As we know, a typical VM running a Linux operating system with basic desktop manager and applications is usually several gigabyte in size. It might be difficult to copy the entire VM image to mobile devices for the migration. However, a typical embedded

operating system running on handheld device or a stripped down Linux may be only tens to hundreds of megabyte, which may make it possible to copy of the entire image. Another way to think about this problem is: the applications that mobile users need to migrate may not be significantly big. Therefore, it might be possible to "customize" a light-weight operating system + applications image for a particular user for migration. The size of the VM image is a critical issue for its migration and we do consider such customizable VM as possible future extension of this dissertation.

## *6.4 Security vs. mobility*

In the device level virtualization work, our main idea is to keep the guest unchanged and make changes to the host only. The purpose of this design choice is to provide a generic guest to the host environment and the environment can adapt its capability easily to the generic guest. This design choice maximizes the mobility but may increase security risks. Since the host domain is more privileged than the guest domains, any modification to the code base of the host domain increases the vulnerability of the whole system. The design principle that ensures that the modification to the base system is kept small is precisely intended to address this concern. Nevertheless, it is still the case that the code base that has to be "trusted" is increased by our design decision to keep the guest unchanged and modify only the host domain.

By choosing to make changes in the host domain, we do not mean that security is less important than mobility. Since mobility but not security is the main contribution of our work, we choose to maximize the mobility that we can provide to mobile users. We use simple security models to ensure the integrity of the system and rely on the work of other researchers for enhanced security. A composite security and mobility model that simultaneously optimizes both is a potential direction for future research.

## *6.5 Interaction between two levels*

In this dissertation, we present two approaches to address different aspects of seamless mobility problems at different levels. As a further extension of this dissertation, we do consider the possible interactions between these two approaches. Such an interaction can be the communication of necessary information to better enhance user experience. For example, by knowing the device capabilities provided by device level virtualization, service level virtualization can inform the application with such information to make the application do the application level adaptation accordingly. In this way, we combine device level capability adaptation and application level adaptation together to make the migration more smooth. Similar optimization is possible in the reverse direction as well. By knowing application specific information, device level virtualization can decide if the entire virtual machine needs to be migrated or only part of it is necessary. Such information flow between the levels would reduce the size of migrated state if device level virtualization can know, at migration time, which application(s) user is really interested in. A customized migration can be provided if application specific information is available to device level virtualization.

Above are only two examples of possible interactions between the two levels of virtualization. Other types of communications between them may also be helpful to enhance user experience. As the two approaches presented in this dissertation address different problems in seamless mobility, we believe their communication and interaction can help to better combine these approaches rather than placing them separately at two levels. In other words, the sum may be greater than the parts. A very fruitful direction for future research is developing a composite system that combines virtualization at various levels.

# CHAPTER VII

# CONCLUSIONS AND FUTURE WORK

In this dissertation, we have investigated seamless mobility problem from different perspectives. In order to provide uninterrupted service to mobile users, we need to address issues for state migration across different environments. By identifying four dimensions of state migration, we have clarified our goal and requirements to provide continuous service to mobile users.

State migration requires commonality of the source and the destination platforms for state saving, retrieving and resuming. However, common platforms are not always available in ubiquitous computing setting due to the heterogeneity of such environments. Virtualization technology can provide such a common platform on top of heterogeneous environments by virtualizing system resources in an abstract form. The virtualized resources can be used interchangeably when the mapping between virtualized resources and physical resources is changed.

We proposed two approaches to address issues with seamless mobility using virtualization technology. Service level virtualization addresses the issue at middleware/application level. It migrates application internal states across applications for the migration of a service. A common state structure is defined for each category of service and the virtualized services running in every environment understands such a common structure. The state can be dumped from any virtualized service and resumed on any other virtualized service. We have built a system, called MobiGo, to demonstrate how service level virtualization may be accomplished. An example service, movie playing is built on top of MobiGo to demonstrate the service level virtualization: the service can be migrated across heterogeneous platforms using different

applications (e.g., from Mediaplayer on Windows platform to mPlayer on Linux platform). Through performance evaluation, we have shown that MobiGo has acceptable performance while providing seamless service for mobile users.

Another approach, device level virtualization, addresses issues for seamless mobility at a lower level: device/hardware level. Device level virtualization removes some application requirements for middleware level virtualization and can support the dynamic migration across different devices by re-mapping virtualized devices to physical devices. Capability adaptation is an important feature for device level virtualization, which provides seamless migration across heterogeneous I/O devices without the involvement of the applications. We have built a system, called Chameleon, to demonstrate how device level virtualization may be accomplished. Chameleon allows the registration of capability adaptors in the system and the selection of an adaptor at runtime given the source and destination device settings. Although our focus is on the dynamic linkage of capability adaptors at runtime but not capability adaptation algorithms, we have implemented some basic algorithms to demonstrate our architecture in real scenarios. Through performance evaluation, we demonstrate that Chameleon, while providing device level state migration and capability adaptation, has minimal overhead compared to the original Xen split device driver system, which we use as a base system for Chameleon.

This dissertation may be extended in several different directions as discussed in Chapter 6. First, we can consider the third category of services we presented in Chapter 2: services with dynamic network state. For example, interactive gaming services may need to address the consistency problem for shared contents. A connection management module may be necessary for an E-commerce service that maintains the connection [1] between the user and the state provider in the network. These services demand the consideration of many new issues in state management and migration.

[1] We mean software level connection, not physical network connection.

Secondly, we can investigate the possible interaction between service level virtualization and device level virtualization to better enhance user experience. For example, a message from device level virtualization may inform the service level virtualization the types of devices available in the environment. Knowing this information, service level virtualization may select the best service that makes the best usage of the environmental resources. It may also inform the application to do appropriate adaptation (application level adaptation) when the physical device information is known. Similarly, service level virtualization may help device level virtualization to customize the operating system and application image that needs to be migrated by knowing the applications of relevance to the user. A reduced OS and application image may be sufficient for migration if service level information is available to device level virtualization.

Finally, we can also investigate other issues that are critical in the ubiquitous computing environment to ensure the proper state migration across heterogeneous platforms. Security and privacy are two examples of these issues. For example, how can we enable a user to authenticate the environment when he/she enters a totally-new environment? How can we make sure user's private data is not compromised or released to other users when multiple users may share the same resources? There may be potential tension between ease of use, efficient mobility support, and security/privacy. For example, immediate removal of any residual state of the user may aid security and privacy concerns, but it could come at the cost of poor user experience if he/she decides to re-enter the same environment in a short period of time. How can we ensure security while still provide maximal level of user satisfaction? Such issues pose interesting research challenges as follow-on work to this dissertation.

# REFERENCES


[1] M. Satyanarayanan and et al., "Pervasive Computing: Vision and Challenges", IEEE PCM August 2001, pp. 10 – 17

[2] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh and Bob Lyon, "Design and Implementation of the Sun Network Filesystem", Proc. Summer 1985 USENIX Conf.,

[3] Shankar R. Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, Terry Winograd. "ICrafter: A service framework for ubiquitous computing environments", UBICOMP 2001

[4] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood and Andy Hopper, "Virtual Network Computing", IEEE Internet Computing, volume 2, pages 33-38, 1998

[5] Jinsuo Zhang, Abdelsalam (Sumi) Helal and Joachim Hammer, "UbiData: Ubiquitous Mobile File Service", Proceedings of the ACM Symposium on Applied Computing (SAC), Melbourne, Florida, March 2003

[6] Your Desktop on Your Keychain, `http://research.microsoft.com/research/sv/keychain/`

[7] Swaroop Kalasapur, Mohan Kumar and Behrooz Shirazi, "Seamless service composition (SeSCo) in pervasive environments", Proceedings of the first ACM international workshop on Multimedia service composition, 2005

[8] Sachin Goyal and John Carter. "A lightweight secure cyber foraging infrastructure for resource-constrained devices". In Sixth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA), 2004

[9] Xiaohui Gu, Alan Messer, Ira Greenberg, Dejan Milojicic and Klara Nahrstedt, "Adaptive Offloading for Pervasive Computing", IEEE Pervasive Computing Magazine, vol. 3, num. 3, July, 2004

[10] Alan Messer, Ira Greenberg, Philippe Bernadat, Dejan Milojicic, Deqing Chen, T.J. Giuli and Xiaohui Gu. "Towards a Distributed Platform for Resource-Constrained Devices", In Proceedings of the IEEE 22nd International Conferene on Distributed Computing Systems (ICDCS'2002)

[11] Ethan Solomita, James Kempf and Dan Duchamp, "XMOVE: A Pseudoserver for X Window Movement", The X Resource, volume 11, page 143–170, 1994

[12] Jin Nakazawa, Hideyuki Tokuda, W. Keith Edwards and Umakishore Ramachandran, "A Bridging Framework for Universal Interoperability in Pervasive Systems", ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems

[13] Ramon Caceres, Casey Carter, Chandra Narayanaswami, M. T. Raghunath, "Reincarnating PCs with Portable SoulPads", Proc of ACM/USENIX MobiSys 2005

[14] HP Cooltown, `http://www.champignon.net/TimKindberg/cooltown.php`

[15] Sun Ray (TM) Deployment on Shared Networks, `http://www.sun.com/blueprints/0204/817-5490.pdf`

[16] Bluetooth Technology, `http://www.bluetooth.com`

[17] Boris Dragovic, Keir Fraser, Steve Hand, Tim Harris, Alex Ho, Ian Pratt, Andrew Warfield, Paul Barham, and Rolf Neugebauer, "Xen and the Art of Virtualization", Proceedings of the ACM Symposium on Operating Systems Principles 2003

[18] Fred Douglis and John K. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation", Software - Practice and Experience, 1991

[19] Gaia project in UIUC, http://gaia.cs.uiuc.edu/

[20] Intel Personal Server,
http://www.intel.com/research/exploratory/personal_server.htm

[21] Rajesh Balan, Jason Flinn, M. Satyanarayanan, Shafeeq Sinnamohideen and Hen-I Yang. "The Case for Cyber Foraging" In the 10th ACM SIGOPS European Workshop, September 2002

[22] Devices Rajesh Krishna, Balan Darren Gergle, Mahadev Satyanarayanan and Jim Herbsleb, "Simplifying Cyber Foraging for Mobile", CMU tech report, CMU-CS-05-157

[23] Eyal de Lara, Rajnish Kumar, Dan S. Wallach and Willy Zwaenepoel, "Collaboration and Multimedia Authoring on Mobile Devices", Proc. of International Conference on Mobile Systems, Applications, and Services (MobiSys), 2003

[24] Mplayer, http://www.mplayerhq.hu

[25] Rich Wolski and Neil T. Spring and Jim Hayes, "The network weather service: a distributed resource performance forecasting service for metacomputing", Future Generation Computer Systems, volume 15, pages 757-768, 1999

[26] Helix Streaming Server, https://helixcommunity.org/

[27] Microsoft Remote Desktop,
http://www.microsoft.com/windowsxp/using/mobility/default.mspx

[28] Universal Plug and Play Device Architecture,
http://www.upnp.org/download/UPnPDA10_20000613.htm

[29] TYC Woo, SS Lam, "Authentication for Distributed Systems", in IEEE Computer (January 1992) pp 39–52

[30] Edmonds, T.; Hodges, S.; Hopper, A., "Pervasive adaptation for mobile computing", Proceedings. of 15th International Conference on Information Networking, 2001 Page(s):111 - 118

[31] Buchholz, S. Hamann, T. Hubsch, G., "Comprehensive structured context profiles (CSCP): design and experiences", Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops, 2004. March 14-17 2004

[32] Becker, C. Schiele, G., "Middleware and application adaptation requirements and their support in pervasive computing", Proceedings. 23rd International Conference on Distributed Computing Systems Workshops, 2003., May 19-22, 2003

[33] Microsoft, Plug and Play Device Driver Migration in Windows Vista, http://www.microsoft.com/whdc/driver/install/ Pnpmigration.mspx

[34] Thomas Naughton Geoffroy Vallee Stephen L. Scott, "Dynamic Adaptation using Xen", First Workshop on System-level Virtualization for High Performance Computing (HPCVirt 2007)

[35] R. Y. Fu, H. Su, J. C. Fletcher, W. Li, X. X. Liu, S. W. Zhao, and C. Y. Chi, "A framework for device capability on demand and virtual device user experience", http://www.research.ibm.com/journal/rd/485/fu.html

[36] Kientzle, Tim, "Scaling Bitmaps with Bresenham", C/C++ User's Journal, October 1995

[37] Smith, Alvy R., "A Pixel Is Not A Little Square!", Microsoft Technical Memo, July 17, 1995

[38] Chen, Haibo;Chen,Rong;Zhang,Fengzhe;Zang,Binyu andYew,Pen-Chung, "Live Updating Operating Systems Using Virtualization", VEE 2006: Proceedings of the 2nd international conference on Virtual execution environments, Pages 35-44, NY

[39] Xiaohui Gu, Alan Messer, Ira Greenberg, Dejan Milojicic and Klara Nahrstedt, "Adaptive Offloading for Pervasive Computing", IEEE Pervasive Computing Magazine, vol. 3, num. 3, July, 2004

[40] Roy Want, Trevor Pering, Shivani Sud, Barbara Rosario, "Dynamic Composable Computing", ACM HotMobile 2008: The Ninth Workshop on Mobile Computing Systems and Applications

[41] LibVNCServer/LibVNCClient, http://libvncserver.sourceforge.net/

[42] Xenstore, http://wiki.xensource.com/xenwiki/XenStore

[43] Windows Registry, http://en.wikipedia.org/wiki/Windows_Registry

[44] Xen wiki on split device drivers, http://wiki.xensource.com/xenwiki/XenSplitDrivers

[45] David Chisnall, "The Definitive Guide to the Xen Hypervisor", Prentice Hall PTR; 1 edition (November 19, 2007)

[46] Eric Roman, Lawrence Berkeley National Laboratory. Berkeley, CA., "A Survey of Checkpoint/Restart Implementations", Technical report, Berkeley Lab, 2002.

[47] Dejan S. Milojicic, Fred douglis, Yves Paindaveine, Richard Wheeler, Songnian Zhou, "Process Migration", ACM Computing Surveys, Vol. 32, No. 3, September 2000, pp. 241C299.

[48] Martin Modahl, Bikash Agarwalla, T. Scott Saponas, Gregory Abowd and Umakishore Ramachandran, "UbiqStack: a taxonomy for a ubiquitous computing software stack", Personal and Ubiquitous Computing , Volume 10(1):21-27, Feb 2006.

[49] Umakishore Ramachandran, Martin Modahl, Ilya Bagrak, Matthew Wolenetz, David Lillethun, Bin Liu, James Kim, Phillip Hutto and Ramesh Jain, "Mediabroker: a pervasive computing infrastructure for adaptive transformation and sharing of stream data", Pervasive and Mobile Computing , Volume 1, Issue 2 , July 2005, Pages 257-276.

[50] Sang-bum Suh, "Secure Architecture and Implementation of Xen on ARM for Mobile Devices", Presented at Xen Summit Spring 2007, IBM TJ Watson.

[51] Xiang Song and Umakishore Ramachandran, "MobiGo: A Middleware for Seamless Mobility", The 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, August 21-24, 2007, Daegu, Korea.

[52] David Lillethun, David Hilley, Seth Horrigan and Umakishore Ramachandran, "MB++: An Integrated Architecture for Pervasive Computing and High-Performance Computing", The 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, August 21-24, 2007, Daegu, Korea.

[53] James E. Smith, Ravi Nair, "Virtual Machine", ELSEVIER, 2005.

[54] GTK+ Project, http://www.gtk.org/

[55] uShare, a UPnP (TM) A/V Media Server, http://sourceforge.net/projects/ushare

[56] Linux SDK for UPnP Devices (libupnp) - An Open Source UPnP Development Kit, http://upnp.sourceforge.net/

[57] Xenbus, http://wiki.xensource.com/xenwiki/XenBus

[58] Riku Saikkonen, Linux I/O port programming mini-HOWTO, http://www.faqs.org/docs/Linux-mini/IO-Port-Programming.html

[59] Intel(R) Tools for UPnP Technologies, `http://www.intel.com/cd/ids/developer/asmo-na/eng/downloads/upnp/overview/index.htm#anchor\_3`

[60] GMediaRender, `http://gmrender.nongnu.org/`

[61] GStreamer: open source multimedia framework, `http://www.gstreamer.net/`

[62] Fiona Fui-Hoon Nah, "A study on tolerable waiting time: how long are web users willing to wait?", Behaviour and Information Technology, Volume 23, Number 3, May-June 2004 , pp. 153-163

[63] Tomasz Imieliński and Julio C. Navas, "GPS-based geographic addressing, routing, and resource discovery", Commun. ACM, vol. 42, 1999

[64] Aware Home, http://awarehome.imtc.gatech.edu/

[65] Michael Kaminsky, George Savvides, David Mazi eres, and M. Frans Kaashoek. "Decentralized user authentication in a global file system", In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03), pages 60–73, Bolton Landing, New York, October 2003

[66] Andrew D. Birrell, Butler W. Lampson, Roger M. Needham, and Michael D., "A Global Authentication Service without Global Trust", In Proc. IEEE Symposium on Security and Privacy, Oakland, CA, April 1986,

[67] Peter Gutmann, "Plug-and-play PKI: a PKI your mother can use", SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium, 2003

[68] Amir Herzberg and Yosi Mass and Joris Michaeli and Yiftach Ravid and Dalit Naor, "Access Control Meets Public Key Infrastructure, Or: Assigning Roles to Strangers", SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy, 2000

# VITA

Xiang Song was born in Beijing, China. He got his bachelor of science degree in computer science at Peking University, Beijing, China in 2002. After that, he joined the Ph.D. program in College of Computing at Georgia Institute of Technology, Atlanta, GA, USA. He was working with Prof. Umakishore Ramachandran as a research assistant in the area of ubiquitous computing. His research interests include mobile computing, ubiquitous/pervasive computing, middleware system and virtualization. In 2004 and 2005, he has been an intern in Hewlett Packard research lab at Palo Alto, CA for 9 months. In HP, he was working with Dr. Rajendra Kumar in GridLite project. He also went to Samsung Electronics Research Lab in Seoul, Korea in summer 2007, working on a device virtualization project on handheld devices.