University of Oslo
Department of Informatics

Policybased networking in a mobile, multiconnected environment

Master's Thesis

Kjetil Myhre

4th May 2003

Abstract

In classic networking, sessions are ordered communication between two fixed end-points. In this thesis we introduce the concept of session migration where one of the end-points of the session can change without losing the integrity of the session. We examine if session migration can be used to increase the user perceived quality of service in heterogeneous, mobile networks.

We introduce the concept of session migration into the research field of service centric networking. In service centric networks, the services are located using a discovery and lookup system which allows the application layer to view the network as a collection of services rather than as a collection of hosts.

All these services will have attributes describing them, and different applications will have different opinions of which attributes a service should have to be well suited for the application. We propose to use a separate policy language to allow applications to easily define what it considers a well suited service.

In session migration enabled service centric networking systems, a service user can start a session using one service provider and continue it on another. In this manner, the service user can select the best suited service provider when the session is initiated. If a better suited service provider becomes available, the service user can migrate the session to the new service provider.

We propose an architecture called Embouchure which includes middleware that provides support for policy controlled session migration.

We use Embouchure to show that policy controlled service migration is one possible way to harness the heterogeneous mobile networks in the future.

Contents

Chapter 1

Preface

This thesis is submitted to the Department of Computer Science at the University of Oslo in partial fulfillment of the Cand. Scient degree. The Cand. Scient degree is a Norwegian degree approximately the same as the international Master degree.

The work has been supervised by Tore Urnes at Telenor Research and Development and Stein Gjessing at Simula research laboratory and University of Oslo.

1.1 Choice of English as the Thesis language

English was chosen as the language for this thesis for two main reasons:

- English is the accepted language in this field of research. By writing the thesis in English it will be available to a much larger group of people.
- All the technical terms in Computer Science have English names. To translate these names into Norwegian could confuse the reader and make understanding harder. A possibility that was considered was writing the thesis in Norwegian but use English technical terms. We discarded this possibility, because the thesis then would be neither Norwegian nor English but something of a hybrid.

1.2 Choice of Embouchure as the name of the architecture.

Embouchure is a French word and translates to "lips" or "mouth". It is also a musical term describing the muscles in and around the lips, and how these muscles are used when playing a wind instrument. The first thing an aspiring musician has to learn is how to use these muscles, and the embouchure of a wind player is the foundation on which most of the performance is based. In the same manner, Embouchure the software architecture strives to be a foundation which software developers can build mobility aware applications on.

1.3 Abbreviations

In mobile computing there are numerous abbreviations. When abbreviations are encountered in this thesis, the full name will be used the first time with the abbreviation in parenthesis. After the first time, only the abbreviation will be used, e.g. Global System for Mobile Communication (GSM).

1.4 Acknowledgments

I would like to use this opportunity to thank all those who have helped me produce this thesis. It would have been close to impossible to complete this work without you.

In particular I would like to thank my supervisors for their unending optimism and skill at pointing me in the right direction when it all seemed hopeless. Tore and Stein have both in their own way offered feedback and motivation which have been to the point and very important for the completion of this thesis.

I would also like to single out my fellow students who have created a friendly environment for me to work in. In particular the people at the Multimedia Communication Laboratory (MMCL) have played an important part in motivating me to work on the thesis.

Kjetil Myhre
Oslo, Norway
May 2nd, 2003

Chapter 2

Introduction

The wireless networks of the future will be diverse and dynamic. Networks as wide-ranging as low-orbit satellite systems on one end and spontaneous short-range network links between two nearby hand-held devices on the other will seamlessly integrate to create a wireless distributed environment. The wireless network links available to a device will change over time, as the device moves in and out of the coverage of base stations and other wireless devices.

We believe that in order to take advantage of this multitude of networks, a mobile device needs more than one wireless network interface. This way it can use the expensive satellite system when no other connection is available, and seamlessly switch to a less expensive system when one becomes available.

Applications which use networked resources vary widely in their demand for quality of service and the willingness to pay for it. Wireless networks also show a great degree of diversity in quality and pricing. We believe that it is impossible for a device to make optimal choices about which network resource to use without allowing the applications running on the device to influence this decision.

Every participant in this distributed environment may provide *services* for others to use. One example could be file sharing like that implemented today by GnuTella[20]. Other examples are access to CPU or hardware resources like printers or displays. As the network links available to a device changes, so do the *services* that are available to it.

In such a dynamic environment, many classic network applications fail. There is no guarantee that the service provider you connect to will stay online until the application is done using it. If the device the application is running on or the service provider is moving around, it is even *likely* that the host will become unavailable sometime during a long lasting session. Classic mobility software like Mobile IP[26] can solve this problem through redirecting the connection through another network, but what if another service provider which is cheaper or better in some other way becomes available? It may then be preferable to migrate the session to the new service provider.

Embouchure In this thesis, we introduce Embouchure. Embouchure is a middleware software module which allows applications to set up *sessions* with service *types*. The application informs Embouchure it wants a session with, for example, a file sharing service and Embouchure locates the best suited file sharing service available.

What is the best suited service will tend to vary widely from application to application. Some applications may want to pay a high price to get premium service, while other applications may want to use the cheapest service available. An application using printer services could be interested in how far away a printer is, an application using file sharing services could be interested in the available bandwidth for downloading files.

Using the application's concept of what constitutes a suited service, Embouchure selects the best service provider and initiates a session with it.

If another service provider becomes available which is better suited to provide the service, Embouchure could start using the new service instead. The process of moving from one service provider to another is referred to as *session migration*.

Session migration is not a trivial issue. When moving a session from one service provider to another, the target service provider has to learn the state of the session in order to able to take over. One also has to take steps to ensure that no data is lost during migration.

Like selecting which service provider is the best in the first place, selecting one to migrate to is complex. In this case, Embouchure have to consider the cost of the migration itself when it decides if migration should be initiated and to which service provider the session should be moved.

A classic problem for such middleware modules is the trade-off between complexity of use and functionality. If the middleware allows the application developer to tune every aspect of its inner workings, the application developer will need detailed knowledge on how the middleware works. On the other side, if the middleware hides too much complexity from the application developer, it is likely that, in some cases, the middleware makes a wrong choice of which service provider to use.

In Embouchure, we propose to use *policies* to allow application developers to tune how the middleware selects which service provider to use. By using a separate policy language that allows application developers to guide

Embouchure in selecting which service providers, we separate the policies used to select service providers from the design and implementation of how it is done. Our aim is to hide the complexity of service selection and migration from the application developer, and instead allow him to define policies governing how the choices should be made. The application developer could provide policies for Embouchure if the application has special needs, or just use a default policy if the application does not have special needs.

The rest of this chapter will go into a bit more depth about areas fundamental to this thesis: heterogeneous networks, service centric networking and the need for applications to adapt to the ever changing network environment. Following this, we present the research goals we hope to reach in this thesis along with the methodology we have used to reach those goals. Finally, we present an overview of the rest of the thesis.

2.1 Heterogeneous Networks

More and more companies install some sort of wireless networks to cover their workspace. Telecom companies and ISPs are constantly installing new hardware for 2nd Generation Mobile Systems (2G) and 2nd Second Generation Mobile Systems — Improved version (2.5G) systems like Global System for Mobile Communication (GSM)[21] and General Packet Radio Service (GPRS)[8]. Some people believe that 3rd Generation Mobile Communication (3G) systems like Universal Mobile Telecommunications System (UMTS)[1] will be deployed soon.

In addition to the wireless networks offered by the telecom companies, unlicensed wireless networks like Wireless Local Area Networks (WLAN) and BlueTooth are getting more and more popular. Anyone can buy a WLAN card and install his own wireless network.

The result is a heterogeneous bundle of different network technologies. All these network technologies have different characteristics. Some are available in a large area, but slow and expensive, others are only available in small cells but fast and inexpensive, others still fill the gaps between these extremities.

Consider the scenario depicted in figure 2.1 . At the office, Hege has access to a WLAN base-station as well as the docking station for her laptop. When she is seated at her desk, she wants to use the wired network access provided by her docking-station. At an internal sales department meeting in the office meeting area, she wants her laptop to stay connected, now using the WLAN network at the office. Later the same day, she brings her laptop along with her on a meeting with a customer. This meeting takes place at a coffee bar, and there is no WLAN or other fast wireless link available.

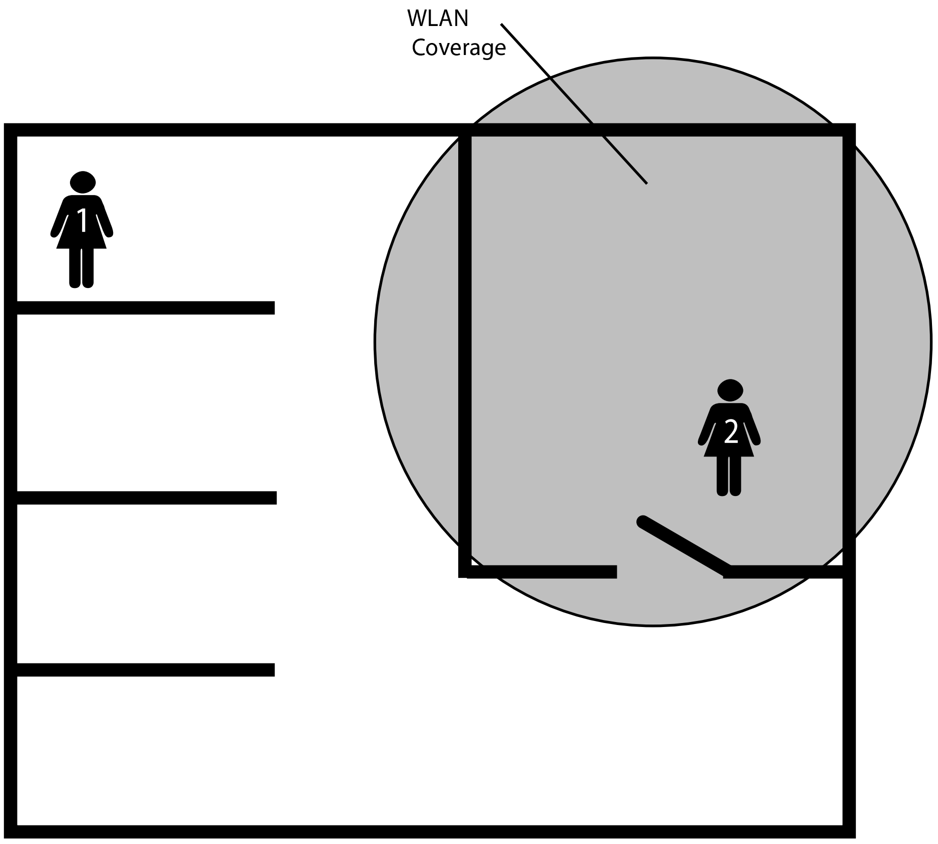

Figure 2.1: To be connected constantly at her workplace, Hege relies on different types of networks. At her office desk(1) she uses a wired link, in the close-by meeting room(2) she uses WLAN

Hege then opts to use the GSM network, inserting the GSM card phone into her laptop.

This scenario shows a typical use of wireless networks today. It requires the computer user to actively choose which network technology to use at a given point in time. This very act of choosing which network to use requires knowledge about the different networks available, and may prohibit some people from taking advantage of the wireless resources. This becomes even more true in the following scenario, depicting a possible use of wireless resources in the future.

Consider the scenario in figure 2.2 . Sitting in the coffee bar with her customer, Hege discovers that she has no access to a fast wireless network base-station. Petter is seated at the cafeteria across the street working with his laptop, and he is just barely within the range of a WLAN base-station with access to the Internet. Petter has decided to share his Internet connec-

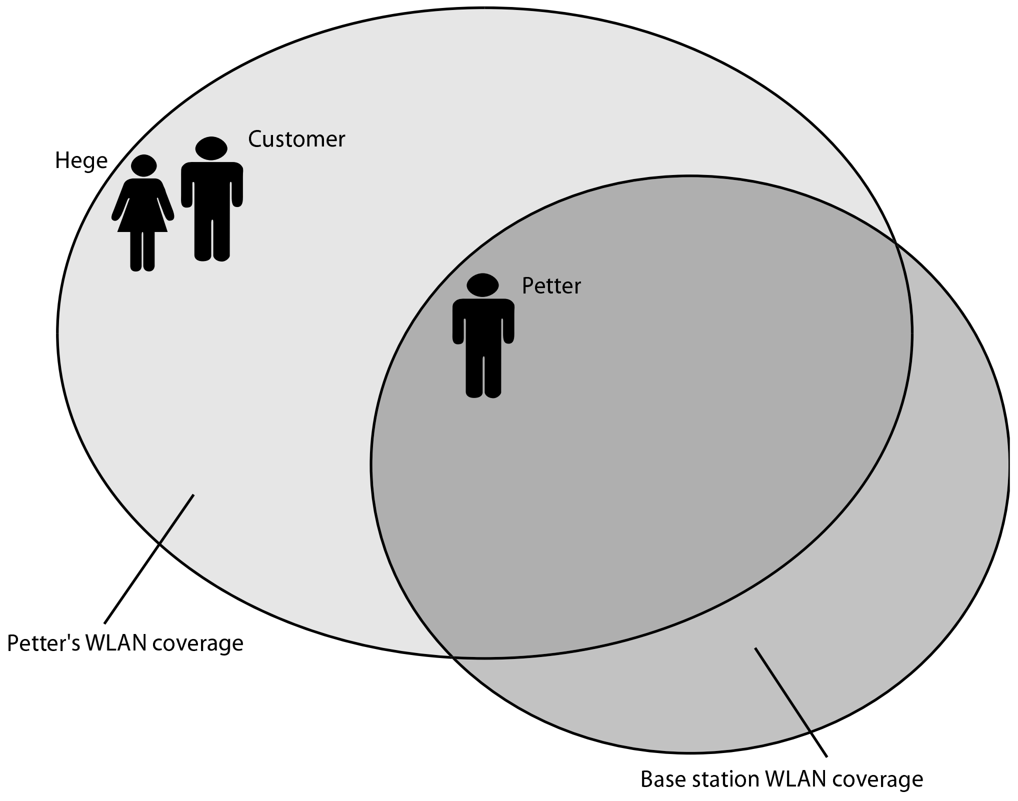

Figure 2.2: Hege uses Petter's connection to a base station WLAN instead of the expensive UMTS connection

tion for a small fee. Hege and Petter are within the range of each other, and therefore able to share resources on each other's computers. Hege compares the cost and quality of connecting to the Internet through the UMTS connection with the cost and quality of connecting via Petter's laptop, and decides that using Petter's link is the better solution.

This kind of network usage is likely too complex for the average user to understand. Also, no matter how technologically advanced the user is, comparing and selecting which network or which service to use takes time.

2.2 Service Centric Networking

One of the most important developments in distributed systems over the past several years is the arrival of service centric networking. This concept moves the focus of distributed systems away from host names and IP addresses and toward service types. Rather than connecting to the printer server with the network address printer.somewhere.com, the user connects to the printer that is currently most desirable, be it because it is close or of good quality. Rather than focusing on the network address of the printer, the user focuses on the *concept* of a printer itself. Prevalent among the research efforts in this area are JINI[30], Home Audio Video interoperability (HAVi)[24] and Universal Plug and Play (UPnP)[50]. These software infrastructures provide a platform where service providers can deploy and describe their services, while service users can search among the deployed resources and select the one that best suits their demand.

Consider the scenario depicted in figure 2.3 . Hege is a sales represent-

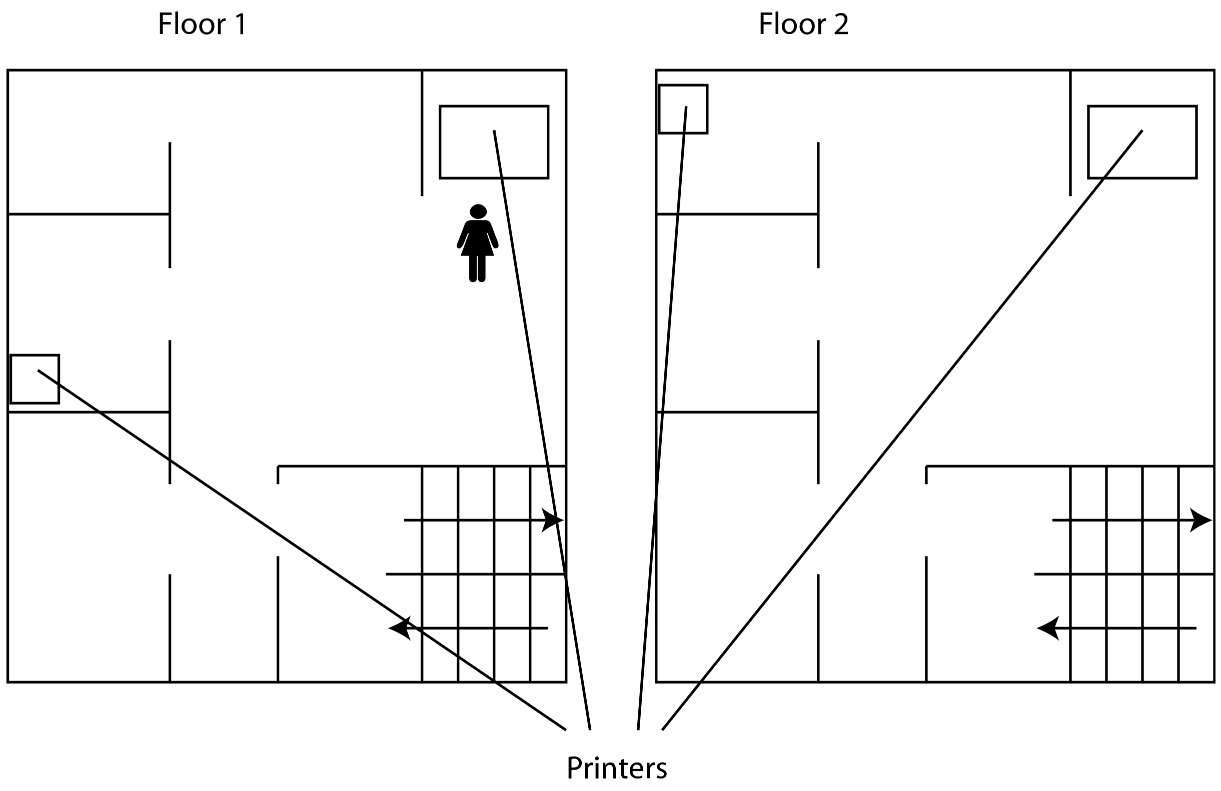

Figure 2.3: The floor plan of the office building of a customer where Hege is visiting. Hege needs to print a document and her portable computer is able to discover nearby printers that are made available as printer services.

ative for her company visiting a customer's office. During the meeting, the customer asks Hege for a printout of the sales contract. Hege simply opens the contract on her laptop and hits the print button. The laptop samples the local network, discovers the four printers, and selects the one closest to Hege 's location. It then displays a floor plan showing Hege where to pick up the sales contract.

Here, Hege did not care about the network address of the printer nor whether it was running Linux or Windows. The printer was providing a service of the *Printer* type, and had registered itself with the local service registry. Hege 's laptop simply asked the registry for a listing of printers, and then selected the most appropriate one. Hege had previously defined the best fitting printer to always be the printer closest to her current location.

The drawback of this scenario is that Hege and the local network administrators have to agree on how the *Printer* service type is defined. However, once the *Printer* service type has been agreed upon, anyone can write a printer spooler which everyone can use, no matter what programming language or operating system the spooler is based on. All that is required is that the printer spooler adheres by the defined interface.

2.2.1 Session migration

Sometimes more than one service can provide the service needed by the service user. The service user has to select which of the services it wishes to use. After the service user has opened a session with the chosen service provider, the environment might change. Other service provider may become available, or the ones that are already available might change. As a result of these changes, another service provider may have become better suited to use than the currently active one. Maybe the mobile device moved away from the WLAN base station the active service provider is reached through causing the link to become unstable, or maybe a cheaper service provider has become available.

At this point, the service user might want to migrate the session to another service provider.

We believe that service centric networking will be important in the future. When a service user wishes to use a service type, he will have several service providers to choose from. These service providers may be located on stationary hosts or they may be located on mobile devices.

In the section on heterogeneous networks we pointed out that using heterogeneous networks to the full extent of their performance requires a lot of choices on which network link to use at a given point in time. When we in addition to this have to choose among a volatile set of service providers, the task of always getting maximum performance from the networked resources grows even more complex.

To alleviate this complexity and utilize the potential of the service centric, heterogeneous network environment, the mobile device has to be able to adapt to the changing environment.

2.3 Adaptation

Another central point to be made about the network resources of the future is the dynamic nature of the environment. What happens in the scenario of figure 2.2 if Petter finishes his coffee and leaves while Hege is downloading data through his laptop? Or what if Rune comes into range and offers a much higher quality network link that Hege can use instead?

Traditional network thinking would suggest that the solution is to simply treat the mobile device as a router with an outgoing line for each of its network interfaces. However, we believe that different applications and users will have different requirements and cost thresholds. For example: When Hege is meeting with a customer through a video-chat, she is more inclined to pay a high price for good quality networking than when she is downloading her personal email. There is no point paying for an expensive network link if another less expensive link is available that can provide sufficient quality of service to the application.

An application using networked resources from a mobile host with multiple network interfaces can often benefit from changing which network interface it uses as it moves around. Applications capable of detecting opportunities for and performing such changes are called mobile aware. Mobile-aware applications strive to adapt to the ever changing network and services.

This kind of adaptation is difficult to implement. The application has to monitor the available networked resources and make decisions on whether to change the one currently being used.

In order to simplify the task of adaptation, we propose using policies in this thesis. Embouchure defines a separate policy language that the application developers can use to define what that particular application deems a good service. Using this information, Embouchure automatically selects the most appropriate service provider. Additionally, Embouchure migrates the session to another service provider if one appears that is better suited than the one currently being used.

This allows the application to easily define what it considers a well suited service. Rather than having to focus on the complex issues of monitoring and migration, the application can focus on the concepts which describe the service. These concepts could be the bandwidth available for a video streaming service or the amount of money charged for the use of the service.

The application can define what is a good service without knowledge of how this information is used to connect to a good service. The policy for choosing services is separated from the complexity of the actual choosing and migration.

2.4 Research Goals and Methodology

In this section we present the research goals we strive to reach in this thesis and the methodology we use to attain those goals.

2.4.1 Research goal

We base our research on three assumptions:

1. The network of the future will be a heterogeneous mixture of service providers, some located close to the service user, some far away. Some service providers will be reachable only through a local network while others will be globally addressable. The service providers that are available to a mobile device will change over time as the network quality or the service providers themselves change.

2. Applications which wish to fully utilize these service providers have to adapt to their volatile nature. The application has to monitor the available service providers and be able to migrate session from one service provider to another if the environment dictates it.

3. How the application adapts to the changing environment is application dependent. It is not possible to design an adaptation scheme which will adapt perfectly for all applications without input from the application.

In this thesis we wish to examine if policy controlled service migration can aid the development of distributed software for heterogeneous, mobile networking.

Central to this goal is the complexity of use of such a system. In order to aid the development of distributed software, policy controlled service migration has to made available without requiring the application developer to handle additional complexity.

We want to examine if it is possible to introduce the added functionality of policy controlled service migration without increasing the complexity to use networked resources.

We also wish to examine if policy controlled service migration can be introduced without causing a large overhead in resource and time usage.

2.4.2 Methodology

In order to reach our research goals we start out by examining research done by others. We want to find out if someone else has proposed to use policies and service migration. We also want to cover research in related areas. The research covered here will later be the benchmarks to which we compare our solution when we evaluate it. The background research will also cover several technologies which we will use as foundation when designing Embouchure .

Following our coverage of related works and background research, we provide the partial design and implementation of Embouchure . As previously mentioned in this chapter, Embouchure is an architecture which allows service migration based on policies. We want to examine if Embouchure can aid the development of distributed software.

In order to evaluate if Embouchure is successful in aiding the development of distributed software, we provide a case study as a partial proof of concept. We will not try to cover all the aspects of the system, but show that Embouchure can be used in practice. In the case study we implement a simple service provider and shows how a session can be migrated between different instances of this service provider. Some parts of the system will be simulated.

Following the case study we evaluate Embouchure and compare it to other systems which offer some of the same functionality. We evaluate how complex Embouchure is to use both for the service provider developers as well as for the mobile application developer. We also consider some of the error situations that might occur, and evaluate how these are handled.

Finally, we discuss how successful we have been in finding an answer to our research goals.

2.5 Overview of the following chapters

The rest of the thesis is built up as follows:

In chapter three, we cover the related works and background material required for the rest of the thesis. We consider how far existing technology go in the areas of interest for the thesis, and we discuss the technologies that are to lie at the foundation of Embouchure .

In chapter four we discuss the choices we made when developing the design of Embouchure .

In chapter five we present an overall design of Embouchure , with particular focus on session migration and policy control of the middleware.

In chapter six we implement the parts of the system we have designed in detail.

In chapter seven we present a case study which uses Embouchure as a proof of concept.

In chapter eight we run tests and evaluate Embouchure . We also compare Embouchure to other similar systems.

In chapter nine we summarize the thesis. We present the contributions we have made, the deficiencies of Embouchure and some suggestions for future work.

Chapter 3

Background Research

In this chapter we present related research work and background information. We start out by examining the philosophical ideas that drive developments in mobile systems and continue by covering the necessary technologies.

The area of next generation networks is currently undergoing research in research-groups all over the world, and new projects and ideas are published all the time. The volume of research is overwhelming; we limit ourselves to works most relevant to this thesis.

3.1 The story so far

In this section we will give a brief account the research done in mobile systems up the present.

3.1.1 The world of telecom

The telecom industry has long held a monopoly on providing mobile connectivity to the roaming user. Classically the application provided was the voice connection while on the move. Today's second generation mobile standards(2G) (e.g. GSM[21]) are designed to support this voice connection. All second generation systems are circuit switched. In the same manner as fixed phone lines, a connection is established all the way through the net to provide a channel for the voice data, regardless of whether the users are actually talking or not. Data transfers over these standards are limited to the data that can be sent on voice channels. Generally this is around 10kbs of bandwidth. In addition, data connections suffer from large connection time latency while the cellular system generates and allocates a path through the phone system.

As demand for better bandwidth and latency figures on the mobile net emerged, the standards evolved. People started talking about the third generation of mobile communication. In 3G systems data rates should be higher and access ubiquitous. On the path to third generation, mobile systems are seeing extensions to second generation standards. Because they fall between second and third generation, these standards are sometimes referred to as 2.5G standards. The most commonly deployed 2.5G standards are High Speed Circuit Switched Data (HSCSD), General Packet Radio Service (GPRS)[8] and Enhanced Data Rates for GSM evolution (EDGE)[14]. HSCSD is particularly well suited for streaming data while GPRS and EDGE both are better suited for asynchronous data, due to the bursty nature of their performance. HSCSD and GPRS can be implemented with only small changes to base station software while EDGE also requires changes in hardware.

EDGE and GPRS provide users with always-on capability. The mobile device is always connected to the Internet, and billing is based on the amount of communication rather than the actual online time. This is achieved through packet switching of the cellular channels. Rather than dividing the channels into fixed channels for circuit switching, all the GPRS or EDGE devices in a cell share the same channels. When a device has data to send, it sends it as free packages to the base station. Likewise, when packages arrive at the base station destined for the mobile device, they are sent as soon as a channel is free. This approach allows much better use of the channels for data transfer, but if everyone wants to send at the same time, it can get congested.

The split of the community has lead to two major organizations working on standardizing third generation mobile systems: 3G Partnership Project for Wide-band CDMA standards based on backward compatibility with GSM and IS-136(3GPP) and 3G Partnership Project for Wide-band cdma2000 standards based on backwards compatibility with IS-95(3GPP2).

Third generation mobile networks will be based on ubiquitous, high bandwidth, always on technology. In other words: whenever a user needs the network, it is instantly available. The bandwidth is sufficient for watching movies or video-chatting with a friend or even a group of friends through small handset screens. The network is equally available whether the user is seated at his office desk, or traveling at 80 km/h in his car.

In Europe, the third generation standard is called Universal Mobile Telecommunications System (UMTS)[1]. UMTS shows a sixfold increase in spectrum efficiency over GSM, and is designed to handle data and voice efficiently. One of the central design issues in UMTS is the cell layout. UMTS has a mixed cell layout. Some cells cover large areas, and other cover smaller ones. A large cell will also typically have several smaller ones inside it. At any time, a UMTS mobile device can be in range of several cells of different size. Regular UMTS cells offer bandwidth og about 300 kb/s, and the smallest cells in UMTS also called pico cells, offer up to 2 Mb/s. Adaptive software in the UMTS device selects which of the available cells to use. This behavior is similar to that of a wireless overlay network.(see section 3.3)

The mixed cell layout allows UMTS operators to provide services even in very congested in-building environments. In-building coverage has been a weak-point for all the cellular technologies so far. A buildings outer walls blocks a lot of radio signals from propagating and the quality of the radio link worsens. Because cells now can overlap, UMTS allows the possibility to install UMTS base stations inside office buildings to allow for premium service in-building.

Japan and Korea are currently leading the way in 3G deployment. Telecom companies in Italy and in the UK recently took on their first UMTS subscriptions.

3.1.2 Short-range wireless networks

Recently, the deployment of short-range wireless networks have exploded. More and more companies install wireless network access to office employees and visitors. The most widely used technology is Wireless Local Area Network (WLAN)[25]. This is a physical and link layer protocol defined by the IEEE 802.11 standard. Communication in a WLAN network can either be managed by base stations, or managed by the computers using it. In this latter mode, computers are able to communicate directly without resorting to a base station at all. This sort of communication is often referred to as Peer-to-Peer (P2P) networking. WLAN is a typical example of a short-ranged high bandwidth wireless network. Currently, WLAN is offering bandwidth up to 11 Mb/s, but future systems aim for 70 Mb/s. Another example of a short-ranged wireless network technology is BlueTooth[6, 15]. BlueTooth is a wireless technology design for low-cost communication up to 10 meters and offers bandwidth in the area of 1 Mb/s. It is currently under standardization at the IEEE with project number IEEE 802.15.1.

WLAN and BlueTooth both utilize an unlicensed band in the spectrum. This means that anyone can buy a WLAN or BlueTooth network base station or network card and plug it in, or even design their own wireless hardware that utilizes this band. More and more networks in private homes and in office areas are based on WLAN or BlueTooth technology.

3.1.3 The chaos of providers

Currently most users of wireless services use only one or maximum two different technologies to access the Internet. In order to utilize fully the available WLAN, BlueTooth and cellular systems, the user has to manually configure the new network each time he wants to switch between technologies. Protocols like Dynamic Host Configuration Protocol(DHCP)[11] aid the user in this configuration, but much is still left for the user to do. Developing hardware and software which allows a mobile device to seamlessly switch between network technologies is currently the focus of research all over the world, and is also a core issue in this thesis.

3.2 The vision of tomorrow

In the near future virtually every company, store, gas station and home will have their own wireless networks. Users will own and carry around computers communicating wirelessly with each other. Telecom companies would like this world to be dominated by UMTS or other third generation cellular systems, but because anyone and everyone can install their own wireless network, the wireless world of tomorrow is likely to heterogeneous. The grocer could have a fairly old WLAN base station running, while the newly started coffee bar on the corner has state of the art equipment. UMTS is likely to be expensive to use, since telecom companies have paid enormous amounts of money to gain access to the spectrum needed to implement UMTS. Enterprises like coffee bars or gas stations can probably benefit from letting their customers access the Internet cheaply or for free. Imagine the followin scenario.

"Darling, Ill swing by the gas station to download Turbonegro's latest album"
"Cool. While you are there, bring some sweats home, will you?"

In some areas, there will be no local grocery or gas station to provide access to the net. In these areas, we fall back on second and third generation systems such as GSM or UMTS to provide access to the network.

As a mobile device moves around, it is likely to encounter several different kinds of networks. The UMTS link would always be there, but WLAN or BlueTooth links may come and go as devices move around. It is a challenge for the mobile systems of tomorrow to be able to at all times use the network or networks that currently provide the best service.

This kind of heterogeneous network environment is sometimes referred to as 4G, or fourth generation mobile networking. A lot of research is currently being done to determine what direction mobile systems will take after third generation systems. One of the most important research forums in the field is Wireless World Research Forum (WWRF)[52]. WWRF is a collaboration of industrial and academic partners on fourth generation networking. It was founded jointly by Siemens, Alcatel, Nokia and Ericson, and now has members from all over the world. WWRF issued a "Book of Visions"[17] in 2001 which includes many considerations and research efforts needed on the path towards 4G.

WWRF refer to 4G technology as “The Wireless World”. They predict that between two and five years from now, the number of users connected to the Internet through wireless means will outnumber the ones connected through wired connections. They base this claim on the fact that users are getting more and more mobile. A user in the future will not be satisfied with using network resources seated down at his desk, but he will want to use the resources in the Internet all the time, no matter what else he is doing at the moment. Additionally, in underdeveloped countries it is highly likely that Internet access in new areas will be based on wireless technologies rather than wired connections. It is much cheaper to install a UMTS base station, than it is to draw a wire to each of the users in a town.

3.3 Heterogeneous Networks

As previously stated, the wireless network of tomorrow is likely to be a conglomeration of different network technologies. Among the first researchers to state that a computer will benefit from seamlessly exploiting more than one of these technologies were Katz and Brewer[33, 7] at Berkeley in California, USA. In 1996 they looked at the case of wireless overlay networks on a project called BARWAN. This is a scenario where a computer has access to several network interfaces. Some short-ranged like IR, others more wide-ranged like cellular systems or even satellite based systems. Katz and Brewer organize the wireless networks in levels based on spatial extension.

At the short range side they have room/building area networks, progressively going through larger area networks to regional area networks at the top. The main contribution of Katz and Brewers work is the concept of vertical hand-over. Vertical hand-over is hand-over between different network technologies, for example from WLAN to GPRS. We believe that this is the first attempt by a research group to develop automatic systems for vertical hand-over. Horizontal hand-over, on the other side, is hand-over between cells in a given mobile technology. For example hand-over between cells in GSM. The solution proposed by Katz and Brewer involves extensions to almost every layer in the protocol stack. Most of the new code is located at the network layer.

In order for a device to be able to connect to more than one wireless network, it obviously needs more than one network interface. Having more than one network interface means having more than one network protocol stack. A central design issue when designing network software for heterogeneous networks, is to determine on which layer the network stacks should be joined. Katz and Brewer proposes to join the stacks at the network layer. In order for the different network technologies to be able to do hand-off from each other, Katz and Brewer believes that they all need to implement a new network layer protocol. We believe that if all participating computers need a new network layer, it will be hard if not impossible to implement BARWAN on a large scale. The ideas presented in the BARWAN projects stand as a foundation of newer research done in this field.

3.4 Mobility

Being able to be connected to the Internet with a mobile device when moving around requirers some sort of mobility scheme. In this section we will cover some of the technologies that allow this.

3.4.1 Classic mobility

Classic mobility handles mobility of a mobile device with up to one live network link at a given point in time. A laptop using a wired connection while docked and WLAN when undocked, or the same laptop moving from one WLAN network to another are examples of classic mobility. Even a stationary computer that is carried from one office with a wired link to another can be thought of as of classic mobility.

The most basic form of mobility handling is using domain name system(DNS)[37] services. Whenever the computer is moved from one IP address to another, the DNS registers concerning its host-name are updated, and packages are henceforth forwarded to the new address. Due to the caching of DNS data in the network, it typically takes up to 48 hours after an update to DNS is mirrored in packages coming to the new location. This delay was acceptable in the wired world, where moving a computer was a rare event. In the current world, a mobility system with 48 hours of latency is clearly not satisfactory.

Mobile IP[26, 40] is the current standard as defined by Internet Engineering Task Force (IETF)[28] for handling mobility. It defines an extension to IP[27] that allows computers to transparently move between points of access.

Other computers can always reach the mobile host without actually knowing where it is located, or without even knowing that the mobile host is not a stationary computer.

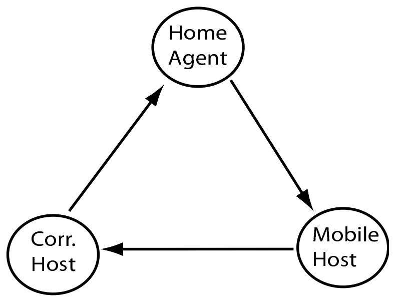

Figure 3.1: Mobile IP triangular routing. Packages from the mobile host go directly to the correspondent host, but packages from the correspondent host travel via the home agent.

In the basic form of Mobile IP, the system consists of a *Home Agent*, a *Foreign Agent* and the mobile host itself. When a mobile host enters into a new network zone, it locates and registers with a resident foreign agent. The mobile host uses multi-cast to discover a foreign agent. The foreign agent assigns a care-of address for the mobile host. The care-of address is the IP address the mobile host can be reached through in the new network zone. The foreign agent sends this care-of address to the home agent. The home agent is reachable through a fixed, global IP address that is associated with the mobile host.Packages sent by correspondent hosts destined for the mobile host are sent to the home agent. The home agent keeps track of the current care-of address of the mobile host. When the home agent receives packages destined for the mobile host, it forwards them to the current care-of address.

When the mobile host sends a package to a correspondent host, it puts the IP address of the home agent in the from field of the IP header. To the correspondent host it looks as if the IP package came from the home agent. When the correspondent host wishes to reply to the package it sends packages to the home agent, believing this is where the original package came from. Packages arriving at the home agent are forwarded to the foreign agent, who in turn forwards it to the mobile host. In other words, packages from the mobile host to the correspondent host are sent directly, while packages from the correspondent host to the mobile host are sent via the home and foreign agent(see figure 3.1).

Much of the success of Mobile IP derives from the fact that only the mobile host and the network the mobile host visits have to support it. The correspondent host acts as if the mobile host was permanently attached to the network at its home address. However, because all the packages have to travel via the home agent to reach the mobile host, there is a potentially large overhead associated with Mobile IP. Route optimizing[31] alleviates some of this overhead. Route optimizing allows the correspondent host to cache the current care-of address of the mobile host. Instead of sending packages via the home agent, the correspondent host is able to send packages directly to the mobile host. Should a packet bounce, the correspondent host may ask the home agent what the current address of the mobile host is. This approach puts more strain on the correspondent host, and an upgrade to the network layer in the correspondent host is necessary.

In Internet Protocol version 6(IPv6)[10], the next generation of IP, mobility as per mobile IP with route optimization is included. Here a correspondent host or a router can cache the current address of the mobile host, and forward packages directly rather than through the home agent. Implementing route optimizing in IPv6 holds more merit than in IPv4(The current version on IP, IPv4 is commonly just referred to as IP) because it can be included in the standard right from the beginning. Deploying a brand new IP version across the Internet is tedious, and it is currently an open question if IPv6 ever gets widely deployed.

Improvements to DNS has been proposed by Snoeren and Balakrishnan[43] which improves DNS for better mobility support. They claim that their approach outperforms Mobile IP and that performance is in the same order of magnitude as Mobile IP with route optimization. They take advantage of secure updates to DNS servers. This update method insures that no host may cache data about the host associated with this particular DNS-address. When the host changes IP address, the change message is propagated to all the DNS servers. In addition, Snoeren and Balakrishnan add functionality in TCP/IP to enable keeping transport session alive across IP changes.

Other approaches to mobility support also exists. Some approaches focus on extending Mobile IP to further improve its performance[16][22].

Teraoka, Yokote and Tokoro focus on making routers in the net more mobility aware. In their proposed design[19], routers can cache home address care-of address pairs, and forward packages addressed to the home address of a mobile host directly to the care-of address. This approach demands changes to the network layer of all routers in the Internet.

Some focus on mobility on the transport layer[36] or application layer[23]. The advantage of moving mobility support higher up in the network stack is to allow the higher level protocols to adapt to the changes in mobility. When a hand-off is initiated, the properties of the network may well have changed dramatically. There may be a wholly different bandwidth, and latency may well have increased or decreased.

3.4.2 Mobility in Heterogeneous Networks

As the use of wireless technology expands, access to different kinds of wireless networks will increase. A natural effect of this is the movement from cell-based networks to overlay networks. In other words, where there used to be only one base-station available at a single point, there will now be any number of base stations covering the same turf. In order for wireless devices to take advantage of this situation, it has been predicted that mobile devices in the future will each carry multiple network interfaces[33].

Motivation for Multiple Network Interfaces

There are a number of advantages to using multiple network interfaces:

1. **Smoother Hand-offs**
   The initialization of a new network such as acquiring network address and registering with entities needed for the mobility scheme used can be done before any hand-off is initiated, thus reducing the hand-off time. It is also possible to start using the new interface before shutting down the old one. If properly programmed, this will further reduce the latency of hand-offs.

2. **Different Network Characteristics**
   The different networks a mobile host with multiple network interfaces is connected to, can have widely varying characteristics. The quality of service, the cost of use and the security schemes available are all things that make these networks different. With access to several networks, the mobile host can choose dynamically which one that best fits its requirements.

3. **Multiple Simultaneously Active Network Interfaces**
   When the mobile host is connected through several network interfaces, it is possible to use them all at the same time. This obviously increases the network quality as compared to only being able to use the best network available. If the demand for network Quality-of-Service can't be met by any one of the networks the mobile host has available, it may be met by combining two or more of the networks.

Heterogeneous network architecture presents a tremendous challenge to mobility software. Not only does the network software have to keep track of where the mobile host is, but it also has to make a decision on which of the several network interfaces of the mobile host it should send data to. In the same manner, software on the mobile host needs to make decisions on which of its several network interfaces to use when it has data to send.

MosquitoNet

The Wireless Computing Group at Stanford University has developed a mechanism that enables a mobile host to make use of multiple active network interfaces at the same time[53]. They have based their research on an extension of Mobile IP(see section 3.4.1). This mechanism is a part of a network architecture called MosquitoNet.

In MosquitoNet each mobile host is associated with several care-of addresses, one for each network interface it has currently connected to a network. When the mobile host wants to open a connection to a correspondent host, it selects one of the active interfaces. Using this interface, it sends an update to its home agent requesting that all packages originating from a specified port on the correspondent host be routed through the selected interface.

When a package arrives at the home agent bound for the mobile host, the home agent checks its routing tables and decides which care-of address to direct the package to. If there is no entry for the correspondent host address/port combination, the package is directed at the default care-of address.

This scheme is not compatible with the route optimization for Mobile IP. All packages from a correspondent host need to go through the home agent to be routed to the appropriate care-of address. Routing thus have to be either triangular, with the mobile host sending its packages directly to the correspondent host, or bi-directionally tunneled, where traffic both to and from the mobile host goes through the home agent.

3.5 Service Centric networking

Classic network applications work by opening a socket from a client to a server. The application has to deal with the protocol on this socket itself. Remote Procedure Call(RPC)[5], Remote Method Invocation(RMI)[47, 38] and Simple Object Access Protocol (SOAP)[44] are software modules which enables the possibility of remotely calling procedures in a server. Using these technologies, a server application may export a method interface which clients can use. Middleware in the client and server handles the actual sockets used to move instructions to and from the server.

In time, software developers grew accustomed to thinking about a server as a place to invoke methods rather than as a place to connect sockets. This design pattern opened up multiple possibilities for the design of distributed applications. Combined with object-oriented programming, an interface on a remote server could be encapsulated into an object, which the application could use as if it was a locally stored object. From the server-side it was a single object that exported its interface to the outside world.

The next step on the evolutionary ladder is code mobility. In a system with code mobility a server may send a fully functional object complete with both data fields and code to the client. The client could then use this object as if it was a local object in the client. This mobile object would typically have a binding with the backbone object in the server. In some occasions it might forward procedure calls to the server, other times the procedure could be local to the object. Fuggetta et al.[3] gives a nice overview of the field of mobile code. In the Java[46] programming language, serialization[49] and reflection[48] may be used to enable mobile code.

It is popular to call an interface provided by a backbone object a *service.* The server exporting the interface is referred to as a *service provider.* Service centric networking is when a client wishes to use a service, but don't mind what actual computer the service is located on. No more is the IP or DNS address important to the user, merely the type of service he wants to use matter. Several research efforts have been taken to allow users to locate such services as they are needed. The first approaches to the problem was the naming services of RMI systems like CORBA[38] and JavaRMI[47]. These naming services allow a server which wishes to export a service to store a link to the service along with a name that can be used to look it up. A client can ask the naming service to find a service with a specific name. For example, a printer service might export an interface called printer/queue in a naming server. When a client asks the naming server for a service with the name *printer/queue*, the naming service returns a remote link to the service. The client needs to know beforehand where to look for the naming service, and it is only possible to search on service names, not service types. A client could not ask the Java RMI naming server to return all the printer queue interfaces it has stored.

To solve this problem, JINI[30] was introduced. JINI offers a service directory called the lookup service. Using code mobility, a service provider may send a proxy object to the lookup service. This proxy implements the type of the service the service provider wishes to offer. A proxy object is an object that can be used to access the service. This proxy object may either handle all procedure calls all by itself, or it may communicate to the backbone service object. The service provider may attach attributes to the proxy describing various service characteristics and QoS issues. These attributes are called *entries*.

When a client wish to find a service of a certain type, for example a printer service, it asks the lookup service to return all the services whos proxy implements the *printer* interface. It is also possible to limit the search to proxies by dmanding that certain entries are attached to the proxy. The client may also register in the lookup service and receive events each time a service the client is interested in becomes available, becomes unavailable, or otherwise changes state. Lookup services are discovered using multi-cast, much like the way foreign agents are discovered by the mobile host in Mobile IP (section 3.4.1). This means that whenever a client using JINI enters a new network, it does a multi-cast and asks if there are any lookup services available. If one is available, it responds to the client, and the client can now lookup the services it needs. When a lookup service is initiated, it broadcasts to the network that it is present and ready, and clients wishing to use it may connect to it.

Other research groups have worked in the same direction of JINI. HAVi[24] and UPnP[50] are two other systems which offer support for service centric networking.

3.6 Sessions and Session Migration

According to the ISO/IEC Session Layer Definition[29], a session is a durable, ordered and long-time data connection between two computers. As communication increasingly is between objects rather than computers, a session evolved to being durable, ordered and long-time data connection between two objects. Rendering both these objects mobile increases the complexity of a session. Keeping a session alive across end-point mobility is referred to as session migration.

Snoeren proposes in his PhD thesis[2] a system called Migrate which implements this scheme. In Migrate, a session is a data connection between two network end-points that is kept alive even across attachment changes of the end-points. Migrate is built on top of TESLA[41], a framework which supports a flow-based abstraction on the session layer. The creators of TESLA claim that it is able to provide easy-to-use programming paradigms for multiple session layer functionalities including encryption, application-controlled routing, and flow migration. Migrate is a design of the flow-migration part of TESLA. Migrate uses virtualization and rebinding to support migration. Virtualization introduces an abstraction level on the session layer which tracks mobility of the host and the correspondent host. Whenever there is a change of attachment point, the software transparently takes down the old transport connection and opens a new one. The session is then resumed using the new connection. Rebinding with Migrate includes modifications to the TCP protocol to support mobility on the transport layer.

Suspending a session and resuming it at a later point in time requires some sort of resource management. When a session is suspended, the service provider has to decide if it should keep the resource allotments for the session, and if so, for how long. Snoeren proposes a scheme called session continuations to store all the state information in the session itself. When the service user at a later point in time locates and connects to the service provider, the session object generates all the state necessary to continue the session.

Session Migration is perhaps best know in enterprise servers. In the context of enterprise servers, session migration occurs when the server cluster moves a session from one server to another. This can be because the original server crashes, or for load balancing reasons. Almost all applications servers with enterprise Java beans support include this functionality, e.g. are BEA Weblogic[4] and Oracle Application Server[39]. The definition for enterprise Java beans servers can be found at: [13].

3.7 Adaptation

It is a widely accepted[12, 34, 17] that applications in a mobile environment benefit from adapting to the changing environments. As stated earlier, the network of tomorrow is likely to be heterogeneous. A mobile device will have multiple network interfaces, and for popular services there will be several service providers for the application to choose from. In addition, each of the service providers may be reached through two or more of the network interfaces. All these aspects add to the complexity of choosing which service to use.

One approach is to gather as much information as possible and forward it to the application layer. The application layer makes the choice on which network resource to use and instructs lower layers to initiate the connection. The information can for example be gathered with estimation and network sampling, and the service providers may also contribute data. This scenario can relatively easily be implemented using JINI or another service centric software and some sort of estimation scheme[35].

A similar approach has been taken by Inouye, Binkley and Walpole[32]. They propose a system called Physical Media Independence (PMI). PMI uses six general true/false characteristics to describe a network interface(See table 3.1). All these characteristics have to be true for a network interface to be available.

PMI introduces software into the protocol stack which monitors these characteristics. Whenever there is a change to one of them, the system

Characteristic	Description
Present	Both the hardware and the software device driver is present.
Connected	There is an active link connection.
NetNamed	The device is associated with an IP address
Powered	The device has sufficient power to function properly.
Affordable	The monetary cost of using the interface is within budget.
Enabled	The user has enabled the device.

Table 3.1: Device characteristics in PMI

makes sure that all the layers adapt to the new resource situation. An event propagates through the system and lets each layer in turn know that the resource in question has changed. PMI leaves it to the application layer to deal with issues like quality of service of the different network interfaces, and merely keeps track of which interfaces are up and running. If combined with other monitoring and estimation schemes to figure out the cost of use of a network, the bandwidth available, the latency encountered etc., PMI can assist the application developer in designing adaptive network software that supports multiple network interfaces. However, as PMI requires great changes to the link, network and transport layers in the network protocol stack, the deployment of the system is likely to be slow.

As the device is moved around, the available service providers and networks links will change — Some will get better, some worse, some will disappear, and others will emerge. All this adds to the complexity of handling the adaptation. Session migration may be used to move a session from one service provider to another as done in EJB systems, but the actual decision on when to do the migration still has to be taken. As the number of available service providers grows, both from heterogeneous networks and from the growing number of service providers, decisions about whether or not to switch to another service provider has to be taken more and more often.

Policy languages

Policy languages are sometimes used to separate the policies which govern a desicion from the code that implements it.

A policy is a rule that defines a choice in the behavior of a system[9]. Using policies, it is possible for users to define the behavior of a complex system without detailed knowledge on how the system is implemented. It is also possible to use a common policy language to harness the behavior of differently implemented systems.

In [51] a research group from Berkeley shows a system that uses policies to configure hand-offs across heterogeneous networks. They study the case of a device which has network links to several different networks. They define some policy parameters, as cost, power consumption and quality of service. The user or application level software, is allowed to assign weights to each parameter. The system uses these parameters to define which network is *the best* to use, and guides an underlying Mobile IP system to hand-off to this network. The research group from Berkeley postulates that the power requirements of using more than one network at the same time are too large, and focuses on the single active link paradigm.

Policy systems are used in a wide range of applications today. Most prominent are the policy systems for security control and network management. As networks grow more heterogeneous, policy systems will have growing importance. Through policy systems, the task of determining the policies for network usage can be separated from the task of implementing these policies. You don't have to know the details of how a specific router or base-station works if you are a manager, you just define how the network should act using some common policy language. The manufacturers of network entities have to implement management schemes where the entity keeps itself updated on the policies from some central or distributed policy repository.

The policy system as defined in [45] focuses on centralizing the storage of rules rather than centralizing the implementation of these rules. Each entity or group of entities in the distributed system can manage themselves according to the rules laid down in the central repository. The manager only needs to know the language used to define these rules.

This effectively separates the policy makers from the policy enforcers. As shown in [42], this facilitates dynamic change of behavior of a distributed system.

Observing the arrival of these and other policy systems, the IETF and DMTF in tandem has developed a core information model for policies as an extension to the Core Information Model called Policy Core Information Model[18].

3.8 Summary

Although many of the technologies presented in this chapter go a long way towards providing flexibile support for mobility, service centric networking and adaptability, no single system exists which supports policy controlled service migration in a mobile setting. We therefore can not base our research merely on available software packages.

Several of the technologies presented here provide us with a foundation on which to build upon. In the following chapters we will present a design and implementation of Embouchure which is introduced in this thesis to demonstrate policy controlled session migration in highly dynamic, mobile environments.

Chapter 4

Discussion of Design Possibilities for Embouchure

In this chapter, we will discuss various decisions made during the design of Embouchure. We will present these in comparison with existing technology which we presented in chapter 3 and argue why new software is needed. We will also present JINI in more detail. Service centric networking stands at the very core of Embouchure and is an absolute necessity for Embouchure to work. Another service centric architecture could well be used in place on JINI, but we choose to use JINI because it is the architecture we have had most experience with.

We start this chapter with a presentation of JINI and service centric networking in general. As all other software systems, choices had to be made on how to design Embouchure and on how to organize the system. In section 4.2 we cover these choices.

4.1 JINI and Service Centric Networking

As briefly covered in section 3.5, service centric networking moves the focus of networking away from hosts and towards the service. Rather than focusing on the host address to locate a resource, one instead focus on the service required. For example, when you need to print a document you locate the nearest printer and uses that instead of first finding the printer, then getting its host address and then use that address to allow your computer to locate the printer.

We believe that this manner of network usage will be the prevailing one in the future, a future promising great diversity for mobile communication. Today, anyone can start his own wireless network merely by inserting a

WLAN card into his computer. In the future we believe that new technology will make this kind of networking even more accessible. We also believe that the Internet as we know it today will prevail as a backbone for the emerging wireless networks. For some services that are located at fixed points in the Internet, it can be profitable to still think of them as attached to the net at predetermined network addresses. However, we believe that many services will be located either on mobile devices which constantly change connection point or as mobile software migrating from fixed server to fixed server. The old paradigm of finding services by using its IP address or DNS name will be outdated. In its place, we believe service centric lookup will be important.

4.1.1 Overview of JINI

JINI is a general architecture which allows service providers to deploy services which service users can locate through a lookup service. When entering a new network, the service user uses multi-cast to locate a lookup service. This lookup service can be queried by the service user to locate services the user is interested in. The query is based on the type of the service and entries which describe it.

JINI is written in Java. When a service provider wishes to deploy a service, it generates a proxy object. This object is mobile and can be downloaded to the lookup service and from there to the service user. This object may contain protocols for communication with the service provider, or it may provide the whole service itself.

The proxy object thus becomes a local object in the service user which acts as a stand-in for the service provider.

The service provider may attach entries to the proxy object before it is shipped to the lookup service. These entries can contain any information the service provider wishes to give about its service. The entries can be used by the service user to limit the lookup search when it uses the lookup service to locate suitable services.

4.1.2 The discovery and lookup of a service

When a service user enters a new network, it uses a JINI mechanism called *discovery* to locate a lookup service. When a lookup service has been contacted, the service user downloads the lookup service's proxy object. Using the proxy, the service user can use the *lookup* functionality in the lookup service to find service proxies which match its needs. The service user has to tell the lookup service which Java *interfaces* service proxies must implement. In addition, the service user may specify that certain entries should be present in the proxy and possibly that the entries should have certain value. If service proxies in the lookup service match these requirements, they are sent to the service user. The service user may also instruct the lookup service to send events should a service proxy matching the requirements appear at a later point in time.

4.2 Design Choices

Embouchure aim to improve the end user's perceived quality of service in a mobile, heterogeneous network environment. This should be achieved with a minimum of complexity for the application developer, but without hiding functionality the application developer needs to use.

Embouchure uses service centric networking as a basis for the distributed programming.

This aim presents several challenges. We group them into four sections:

1. How to design for multiple network interfaces.
2. How to design for maximum mobility of service providers and service users.
3. How Embouchure should adapt to the changes in available resources, and how the application layer should be involved in this adaptation.
4. How Embouchure may conserve the sparse resources of networking and power on the mobile device.

4.2.1 Multiple Network Interfaces

In classic networking each computer has a single network stack. Going from bottom to the top, each layer introduces new protocols and abstractions and allows each layer to use the network without caring for the complexity internal to the layers below it. The theoretical network stack has seven layers: physical, link, network, transport, session, presentation and application. The Internet as we know it today only uses the bottom four. The layers above the transport layer are optional.

Inserting more than one network interface into a computer introduces more than one access point at the link layer. Each network interface is associated with its own physical layer technology and the link layer technology used on that physical link. For example, a typical WLAN card uses wireless transfer of data on the physical layer, and Ethernet as its link layer protocol.

In other words, when we have more than one network interface, we also have more than one network protocol stack. In order for an application to be able to use more than one network interface, these network stacks have to be joined at some point.

How others do it Today, some computers already have more than one network interface. The most commonly known of these are the hubs and routers of the Internet. A hub is a device which joins multiple physical connections on a single link layer connection. In other words, it joins the network stacks on the link layer. A router joins the stacks on the network layer. This way it uses network layer protocols and functionality to send the packages out on the correct link connection. However, in a router there is no transport layer, nor any other layer above the network layer. Network traffic propagates through the physical and link layers up to the network layer, where routing decisions are made before it propagates back into another protocol stack's link and physical layers.

As we covered in chapter 3, some technologies exists which make use of multiple network interfaces. MosquitoNet(section 3.4.2) joins the protocol stacks at the network layer. Being an extension to Mobile IP, MosquitoNet hides the complexity of multiple network interfaces from the transport layer. This allows regular transport layer protocols to act as if the connection was on a classic network interface. However, the network interfaces can have widely varying performance. Hiding the fact that hand-over between network interfaces from the layers above result in a network connection which varies widely in performance without prior notice. BARWAN(section 3.3) too joins the protocol stacks at the network layer.

How we believe it should be done

Selecting which network interface to use is not a simple matter. It is not enough to simply select the network interface with the best performance, one also has to make sure it is affordable. The selected network interface should not consume more power than the device can spare. These are just a few of the considerations to make. In addition, different applications will have different definitions of what is considered good performance, and they will have different threshold on when a network connection gets too expensive to use.

Should we select to do the protocol stack joining at the network layer, as do BARWAN and MosquitoNet, the network layer would need to be updated — only in the home agent and the mobile host in MosquitoNet, but in every single router and participating computer in BARWAN. Updates to the network layer is tedious, and generally means changes to IP. Changes to IP require a large community to agree before it can be tried out at a large scale. Using this approach would therefore cause deployment of the system to be hard and slow.

Due to the service centric nature of Embouchure , decisions on which service to use also need to be made. Some services may be reached through multiple network connections, others may be reached only through a single connection. The characteristics of the network connections should be taken into account along with the information provided in the entries by the service provider when selecting which service is the best suited to use.

Service centric architecture is generally implemented at the session layer. This means that information about services will be available at the session layer and not below it.

After taking all these facts into consideration, we have decided that the session layer is the proper place to join the network protocol stacks.

Multiple network interfaces and its effect on service proxy objects

Some services may be accessible through more than one network interface. Embouchure therefore has to decide both which service to use and which network interface to reach it through.

One possibility would be to use a single proxy for a service and use options to the method calls to determine which network interface to use.

Another possibility is to generate a separate proxy object for each service/network interface pair. In this manner, if a service is reachable through two different network interfaces, it is represented by two different proxy objects. This creates the illusion that there are two different services available.

We have chosen the latter variant. We believe that this variant is easier to design and implement. JINI and the other service centric networking systems are all based on single interface networking. If we use multiple service proxies, each proxy only has to deal with a single network protocol stack. Furtherservice arguments in favor of this approach are presented in the adaptation section of this chapter.

4.2.2 Mobility Management

Classic mobility management deals with keeping a network connection alive across changes in attachment point to the Internet. As we covered in section 3.4, numerous schemes exists to tackle this problem.

However, by deciding to join the network stacks of a device with multiple network interfaces at the session layer, we commit to solving the mobility problem at the session layer or higher. A mobility system like Mobile IP, which is a network layer mobility system, can be used in the network layer of one of the stacks, but if mobility which allows moving a session from one network interface to another should be supported, the mobility management needs to be done on the session layer or higher.

In Embouchure the atomic network component is the service. We don't think about hosts or network layer addressing, we simply treat the network as a means to access *services*. When the application layer wishes to use a service to provide some functionality, we open a *session* with a service which can provide this functionality.

One example of a service session is the video streamer. When the application layer wishes to stream data from a video streamer, Embouchure locates a service which can provide video streaming functionality, and opens a session with that service.

If for some reason the service we are using to provide a session is lost, Embouchure will try to continue the session on another service. This should be done without the application layer even noticing it.

In other words, in Embouchure mobility is about session migration from service to service. When one of the network interfaces of a device looses its link, or the link quality deteriorates below a predetermined threshold, the session should be migrated to another service.

Session Migration

In order to move a session from one service to another, there has to be a way to transfer the state from one service to another. If a new video streamer should be able to take over for a lost one, the new video streamer needs to know which video to stream, and where to start in the video. There is no point in starting all over again, because the user has already watched part of the video.

The size and form of state variables are likely to vary widely from service type to service type. For a video streamer, a mere number telling it at which byte to continue streaming may be enough to resume the session. For a stock trading service the requirements might be quite different. The service user will need to be authenticated and go through a series of security procedures.

Because state transfer is so diverse, we have decided to require that service proxies implement mechanisms for transferring state between service providers. Sometimes state can be transfered directly from service provider to service provider, other times the service user has to instruct a new service provider of the state it is supposed to be in. Embouchure provides an interface design of the proxy object which includes methods for session migration from one service to another. The service providers need implement these methods for the system to work.

4.2.3 Adaptation and application interaction

Two essential assumptions that lie at the core of Embouchure are:

1. Mobile applications have to adapt to the changing network environment in order to fully utilize the available services.

2. Adaptation is application dependent. Session layer software needs application input in order to adapt in the manner best suited for an application.

As services are lost and new ones emerge, it may be profitable for the user to change from one service to another. For example, consider a user who is currently downloading his email using a service proxy which is connected to his email service through a slow and expensive network. As he moves into a fast and cheap network, he discovers that his email service is now also reachable through the new network. It is then highly likely that it is profitable for him to migrate the mail downloading session to the new network.

The second assumption comes from the fact that different applications have different assessments of what constitutes a good service and what does not. An application which wants to download and play music which has been encoded at 128 kb/s, does not want to pay big bucks for a 10 mb/s connection if a cheap 300 kb/s connection is available. On the other side, a video chat application could be able to use all that bandwidth, and if it is important enough, say for example a video chat with an important customer, the application may be willing to pay the big bucks.

In conclusion, Embouchure should include software to adapt to changer in the mobile network and service environment. Embouchure needs input from the application layer on how this adaptation should be done.

Application layer interaction

Deciding how the application layer should access the functionality in Embouchure is a central decision in the design. We strive to create an interface which is both rich and simple at the same time. The "simple" application who do not need customization should not have to specify how the adaptation is done, and the "advanced" application who needs a highly customized solution should be able to get that. This constitutes the classic trade-off between simplicity and expressiveness.

There are several possible solutions to this problem. One is to provide the application layer with a rich API which also includes default methods that the "simple" application may use. This API could include methods to allow the application to get events whenever there is a change in the available services, and allow the application to make the choice on which service to use by itself. There could be methods that allows the application to set rules for how the selection process should be done. This interface would have to be extended for each service type the application wishes to interact with. The parameters which define how the selection between the service are made will change drastically from service type to service type. When using a printer service, the application could be interested in the location of the printer as well as the resolution available. When using a video streaming service, the location does not matter, but the available bandwidth is important.

We believe that an API based approach would be hard to implement and extend for each service type. It would require the "advanced" application wanting to do much customization to have detailed knowledge of the inner workings of Embouchure .

Another approach is to allow the application developer to subclass the class which defines how the service selection is done. The service selection super class would define methods that are called when there are changes in the available services, and the subclass would define how the service selection is done. This approach has the same weaknesses as the API based one. It requires the application developer to have extensive knowledge on the inner workings of the service selection module.

The solution we believe is best suited to allow application layer interaction in the service selection process is to employ service selection policies. By using a separate policy language the application can define how it wants the selection process to be done without considering how the actual process is implemented. In this manner, the application layer could define how much bandwidth it needs, and how much it is interested to pay for it. When a new kind of service is introduced, the application layer can make policies which take into account the entries used to describe this service without the middleware having to know on beforehand what these entries are.

It would also be possible for service providers to bundle default policies with their products. When they introduce a new entry, they could send a policy on how this entry should be used to grade services to the service user, who could decide to use it or not.

We believe that if policy language and entries are made as simple as possible, the threshold for using them would be lower than the threshold for using the other methods described here.

We propose to use a policy language to allow applications to define how service selection is done. This allows us to offer a very simple interface to the application layer. A single method can be used to initiate a session. This method would return a session object which in turn can be used to access the actual session. The session initiation method would take a policy as an optional parameter. Application which wish to provide their own customized policy would include one, application which are satisfied with the default policy would simply call session initiation without any parameters.

Using the policy approach we have a single method interface between application layer and Embouchure. We believe that this will make it easier to make future upgrades to the system.

4.2.4 Resource Management

A mobile device will typically have less resources available to it than a fixed computer. Two resources, power and network will be of particular interest in this thesis. Other resources like CPU and memory could also be sparse, but we assume here that we have enough of those.

Power is limited on a mobile device. One of the most power consuming tasks of the mobile device is the use of a wireless network. A central point in Embouchure is to use multiple wireless network interface simultaneously. When we do this, the power requirements will obviously grow — Two active interfaces use more power than one. We will not cover power saving schemes in depth. We will limit power conservation in Embouchure by limiting the amount of necessary network traffic.

When an application signals its interest in a particular service type, there could be hundreds of service providers available which provide the service needed. Only one of these will be used to initiate the session, so downloading a fully functional and specialized proxy from each and every one of the available service providers is clearly a waste of power and network bandwidth.

To address this issue, we propose to use two kinds of proxies: One very generalized one and one specific to the service type. In this manner, the service user only have to download a small, generalized proxy from all the possible service providers. This small proxy could then be used to download the bigger service specific proxy of the service provider that is chosen to provide the service.

We call the small proxy the *Core Proxy* and the service specific proxy the *Service Proxy.*

Chapter 5

The Design of Embouchure

In this chapter we present the design of Embouchure. Embouchure is a system for session migration in a mobile, heterogeneous network environment, and it offers a simple yet rich interface to the application layer. We start this chapter by presenting an overview of the whole Embouchure architecture.

5.1 Overview of Embouchure

Figure 5.1 shows a high-level overview of Embouchure . Embouchure consists

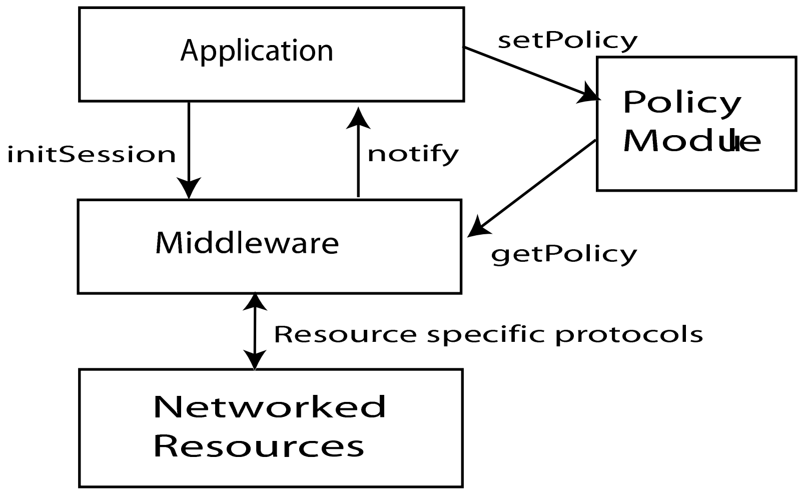

Figure 5.1: A high-level overview of Embouchure

of two modules: a middleware module and a policy module. An application uses the middleware module to start a session with a desired service type through the initSession method call. The application may also define policies for session migration and store these in the policy module using the setPolicy method call. Default session migration policies will be used should the application chose not to define its own. The middleware module obtains up to date information about resources such as available services and network links from lower-level middleware (depicted as the Networked Resources module in figure 5.1). The middleware module constantly monitors changes in the available resources and based on policy rules may decide to initiate the migration of sessions. The application can subscribe to events in the middleware. The delivery of these events are depicted by the "notify" arrow in figure 5.1

A service type is represented by an object interface. An interface in this context is a list of the public methods that are available in objects which implement it.

After the application has told the middleware that it is interested in a session, the middleware uses a service location scheme to locate all matching services, and to build a list of possible services. The policy associated with the session is now used to rank these services, and the best service is used to initiate the session. If the application provided a policy, this one is used. If the application did not provide a policy, a default policy is downloaded from one of the matching services.

After the session has been initiated, the middleware continues to monitor it. If changes occur in the available services, the middleware may choose to migrate the session from one service to another. Typical changes would be the arrival of a new service, the loss of an existing service or the change of an existing service. The middleware uses the policy which the application layer provided to determine whether or not to migrate the session.

The initSession method returns a Session object to the application layer. This object includes functionality to change the policy associated with the session, and it has a method the application layer uses to get the service type specific session object. A service specific session object that is associated with printer services could have *enqueue* and *dequeue* methods, while a video streamer session object would have *getStream* and, possibly, *setDataRate* methods. The point being that each service type would provide a different service specific session object to the application. These service specific objects implement the service type the application specified when initiating the session.

5.1.1 Overview of the Networked Resources

Figure 5.2 shows the middleware modules of Embouchure with the networked resources box from figure 5.1expanded. The networked resources consists of service providers that represents their offerings through proxy objects. These

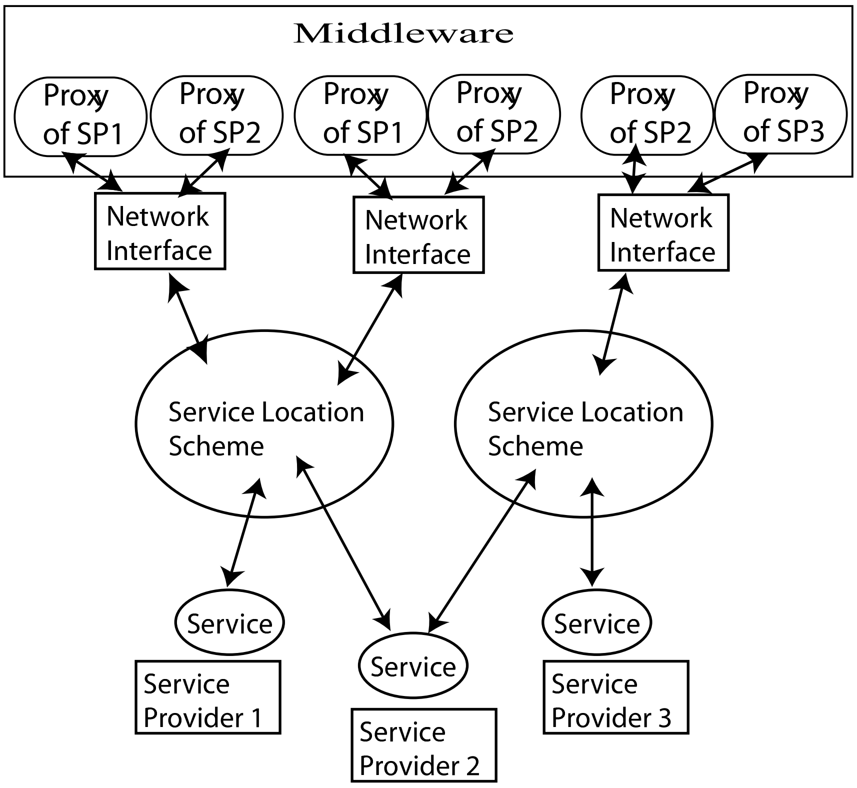

Figure 5.2: Overview of the networked resources used by Embouchure . Notice how each service provider/network interface pair have its own proxy associated with it.

proxy objects are made available to the middleware through a service location module.Figure 5.2 shows a configuration of three service providers and two service location modules that are access by the middleware modules through three network interfaces.

Notice how service provider 2 is reachable through two of the network interface of the mobile device. It is therefore represented by two different proxies in the middleware. The proxy objects originate at the service provider and travels through the service location systems and the individual network interfaces to the middleware. The service provider attaches entries which describe the service before it is sent. These entries can contain any sort of information the service provider deems interesting for the service user. Typical entries could describe the quality of the service provided and the cost of the service.

As the proxy travels towards the middleware, other entities may add entries to it. This could be a network firewall which adds an entry showing that network usage through this firewall has a cost attached to it, it could also be a software module in the mobile device which monitors the network link the proxy arrives on. This monitoring module could attach entries which describe the bandwidth and latency of that particular network interface, or the power usage needed to transmit data using that network interface. When the proxy arrives in the middleware, it has a number of entries attached to it which describes the service provider represented by this proxy, the cost of its use or other data the service provider or an entity on the path to the mobile device thinks is interesting for making the choice of which service to use.

5.1.2 Overview of the Middleware of Embouchure

Figure 5.3 shows an overview of the Embouchure middleware module. The

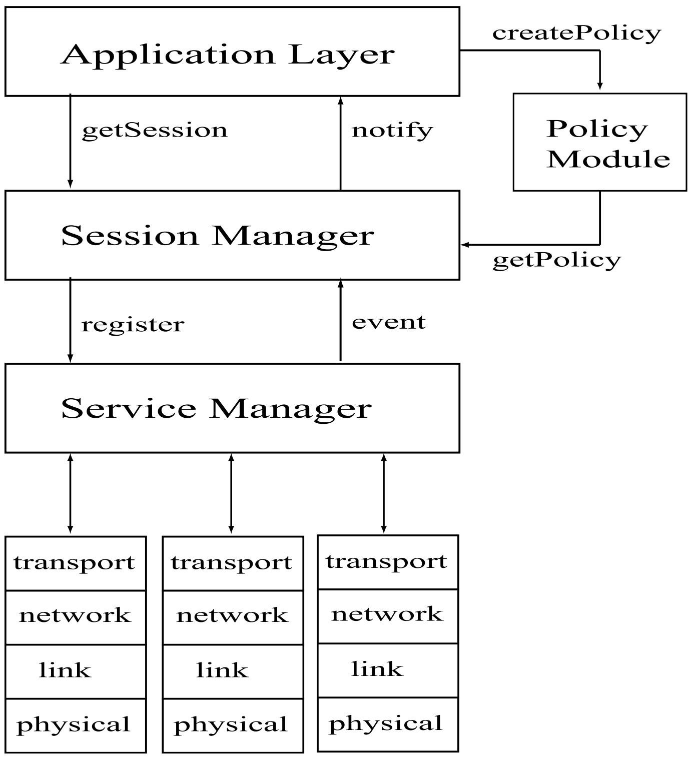

Figure 5.3: Protocol stack overview of Embouchure on the mobile device.

figure shows all parts of Embouchure that are located in the mobile device. Note that figure 5.3 also contains expanded versions of the the network interface boxes from figure 5.2 to illustrate how the ServiceManager and the Session Manager fit into the overall network protocol stack.

This figure identifies two new software modules: The Session Manager and the Service Manager. The Session Manager uses the Service Manager to provide a list of all the services which match a certain service type. The Service Manager provides a simple subscriber/publisher interface to the Session Manager. Through this interface, the individual session in the Session Manager register its interest in services of a certain kind. As long as at least one session is interested in a service type, the Service Manager keeps updated on which services of that type are available, and their current state. Whenever there is a change to the list of available services or to the entries of one of the available services, an event is sent to the sessions subscribing to this service type.

We will not focus on the design of the Service Manager in this thesis since it can be based on well-known service location systems such as Jini. The focus will be on the Session Manager, the Session objects and the Policy module. We will also discuss in some detail the design of the service proxy object.

We will, however, give a brief overview of the interface the ServiceManager offers to the rest of the system.

Overview of the Session Manager and the Session objects

The Session Manager is essentially a factory for Session objects. It receives initSession method calls from the application layer and generates a session object for each call. This session object registers in the Service Manager. When the Service Manager has data about the service type associated with this session, it sends events directly to the session object. The session object also accesses the policy module to get the policies specified by the application. Alternatively, if the application did not provide a policy, the session object uses one of the service proxies returned by the Service Manager to get a default policy.

The session object uses the policy module and the service manager to gather as much information about the service providers as possible. Based on this information, the Session object selects which service provider to initiate the session with and which service provider to migrate to.

We will now present a more detailed view of the design of Embouchure . The different classes will be described in the sections below. The entry definitions and the policy language in section 5.2. These are highly intertwined, as the policy language uses the entries attached to each service proxy to rank the proxies. In this section we also present the design of the policy module of Embouchure . In section 5.3 we define the proxies, in section 5.4 we describe the interface of the service manager, and in section 5.5 we cover the session management module.

5.2 The entry definition and the policy language

As we discussed in the previous chapter, we have chosen XML to define both policies and entries. We strive to keep the entry and policy language as simple as possible. We are not trying to design a fully fletched policy language here, we merely want to try out the concept of using policies to control adaptation in the session layer.

5.2.1 The Entry Definition

An entry in Embouchure is a simple name/value pair. Both name and value are strings, and the entry is written as a single XML tag:

<ENTRY NAME=<name> VALUE=<value> />

Additionally, the *<ENTRYLIST>* tag is used to denominate a list of entry tags.

The service provider attaches a string containing all the entry information to the core proxy of the service, and any entity on the path from service provider to service user may add entries to this string. No-one may remove entries from the string, however.

This approach has a severe security risk, as an intermediate entity could add entries that are faulty or malignant. This security risk could be addressed with future works projects of Embouchure . We choose to use this approach here because it is very simple, and it illustrates a way of gathering information along the connection path which is very important to assess the different characteristics of heterogeneous networks. We simply assume that all parts of the system behaves themselves for now.

As discussed in section 4.2.3, we assume that the proxies which are delivered from the service manager to the individual session objects have entries attached to them which fully describe the quality of service and performance associated with each proxy. In other words, the string containing the entries of a proxy includes data from all the interesting points along the path from service provider to service user.

A entry string associated with a printer service could look like this when it reaches the service user:
<ENTRYLIST NAME="PService">

```
 <ENTRY NAME="BW" VALUE="300"/>
 <ENTRY NAME="LA" VALUE="50"/>
 <ENTRY NAME="PService_RES" VALUE="680*1024"/>
 <ENTRY NAME="PService_COLOR" VALUE="TRUE"/>
</ENTRYLIST>
```

This design could possibly be expanded to allow a service provider to provide several quality of service choices to the service user. The printer service could for example provide both color and monochrome printing at two different prices.

5.2.2 The Policy Language

Like the entry definition, the policy language is extremely simple. We base the language on simple boolean logic and a couple built in functions. The XML file used to define a policy has only two tags:

<POLICY>
and

<DEF NAME=<name> IF=<float expression> VALUE=<float expression> />

The POLICY tag is used to encapsulate the whole policy file.

The DEF statement defines a variable NAME which is set to the value defined by VALUE. Once defined, a variable may not change its value. If the expression in the IF field is less than 0.5, the whole statement is discarded, if its more than 0.5, the statement is executed. A value of exactly 0.5 is invalid.

A float expression has three possible atomic parts: The name of a previously defined variable, a floating point number or the name of a built-in function, possibly with parameters which in turn are float expressions. The atomic parts can be joined with the regular floating point operators, $+$, $-$, $*$ and $/$.

Every policy file has to define the variables *POLICY_VALUE* and *MIGRATION_THRESHOLD*. *POLICY_VALUE* is returned to the session object as the desirability of this proxy. *MIGRATION_THRESHOLD* determines when migration from one service to another should be done. If the *POLICY_VALUE* of the new service proxy is more than

*MIGRATION_THRESHOLD * POLICY_VALUE*

of the old proxy, migration is initiated.

A sample policy definition could look like this:

```
 <POLICY>
  <DEF NAME="BW" IF="1" VALUE="minMaxLinear(100,10000,ENTRY_BW)"/>
  <DEF NAME="LA" IF="1" VALUE="1 - minMaxLinear(75,150,ENTRY_LA)"/>
  <DEF NAME="POLICY_VALUE" IF="1" VALUE="LA * BW"/>
  <DEF NAME="MIGRATION_THRESHOLD" IF="1" VALUE="1.5"
</POLICY>
```

Note that all entry data has the ENTRY_ string added to the name of the entry. This is to avoid having internal variable names which conflict with entry names. A service provider or other entry-adding entity could at any time decide to add a new entry to a proxy.

Policy values are limited to the range 0 to 1. 1 is the best, 0 is non-acceptable. the minMaxLinear(min, max, val) built-in function normalizes the data entered in the val field by returning 0 if val<min, 1 if val>=max and $(val - min)/(max - min)$ if $min <= val < max$. Essentially this is a linear function of val for values between min and max. The result is a number describing val that is between 0 and 1.

This approach enables us to easily compare widely varying data. In the example above, we use the minMaxLinear function to normalize bandwidth and latency in order to compare them. The first DEF statement generates a variable, BW, which is 0 if the bandwidth is below 100kbs, 1 if the bandwidth is above 10mbs and if the bandwidth is between the maximum and minimum, BW is a linear function of the bandwidth.

The second DEF statement defines a variable, LA, which describes latency on a scale from 0 to 1. If the latency is above 150ms, LA is 0, if it is below 50ms, LA is 1, and if the latency is between the maximum and minimum values, LA is a linear function of the latency.

5.2.3 The Policy Module

Table 5.1 shows the UML class diagram representation of an Entry object. An entry object is generated by providing the a string containing the <ENTRY> tag to the constructor. Note the static method makeEntryArray. This method takes a whole file of <ENTRY> tags and generates an array of Entry objects to match. The Entry object is just a read-only structure for holding the name and the value of an entry for use in the Embouchure system.

Entry
- *name:String*   - *value:String*
+ *Entry(xmlString:String)*   + *String getName()*   + *String getValue()*   + *static Entry[] makeEntryArray(xmlString:String)*

Table 5.1: The UML Class Diagram of the Entry Class

The core of the Policy Module is the Policy object. The UML class diagram representation of the interesting public variables of the Policy object is shown in table 5.2.

Policy
+ *Policy(XML_input:String)*   + *eval(entries:Entry[])*

Table 5.2: The UML Class Diagram of the Policy Class

Whenever a policy rule is specified to be used by a session, it is represented as an XML stream. This XML stream can either be retrieved from a policy storage as specified by the application calling the initSession method, or it can be provided by a random core proxy object if the application layer does not provide a policy to use.

When provided with an XML stream, the constructor parses the XML data and generates code that is able to evaluate the policy value and return the migration threshold. The policy object compiles the XML stream into an object which can calculate policy values and return migration thresholds.

When the eval method is called with an array of Entry objects, a variable in the policy module is created for each entry. The name used is the name of the entry with "ENTRY_" added at the front. The value stored is the one specified in the entry description.

After the entries have been store in the policy module, the evaluation code is executed. The result is returned as a PolicyValue object, which is nothing more than a pairing of the POLICY_VALUE and MIGRATION_THRESHOLD variables.

The UML class diagram for the PolicyValue object is shown in table 5.3

PolicyValue
+ PolicyValue(val:float, mig_thr:float) *+ float getValue()* *+ float getMigrationThreshold()*

Table 5.3: The UML Class Diagram of the PolicyValue Class

5.3 The Proxies

Embouchure uses two kinds of proxies to allow communication between the service providers and the service user:

- **The Core Proxy**
  The core proxy is a stripped down version of a proxy. It contains methods for accessing the entries associated with the service, for accessing the default policy for the service type and for accessing the service proxy. The CoreProxy is bound to the service provider through a remote object in the service provider. This remote object has to implement the ServiceBackend interface.

- **The Service Proxy**
  The service proxy contains methods to initiate and shutdown a session, methods to support migration from one service to another, and a method to access the service specific session object that is returned to the application layer.

5.3.1 The Core Proxy

The Core Proxy is a stripped down proxy. The service provider developer is supposed to use the CoreProxy object as it is defined here and not subclass it. By doing this, the service user will never have to download the code for the CoreProxy, thus saving bandwidth and time when a new service provider is located.

The UML class diagram of the CoreProxy is shown in table 5.4. The core proxy has only three major methods: getEntries, getServiceProxy and getDefaultPolicy.

CoreProxy
+ CoreProxy(backend:ServiceBackend)
+ Entry[] getEntries()
+ ServiceProxy getServiceProxy()
+ inputStream getDefaultPolicy()
+ void poll()

Table 5.4: The UML Class Diagram of the CoreProxy Class

getEntries returns an stream of data containing the XML data for the entries associated with that particular service. As previously covered, these entries originate both from the service provider and from other entities along the path from the service provider to the service user.

getDefaultPolicy returns a stream of data containing the XML data for the default policy definition for this service type. The service provider decides if he wants to ship this data with the core proxy, or if he prefers to have the core proxy do a call-back to the service provider and get the data if needed. Because the core proxy is supposed to have a minimal footprint, the latter solution is probably the best choice.

getServiceProxy returns the service proxy of the service.

The poll method is used to check if that particular service is still available. If the method call returns without incident, the service provider is still reachable, if an exception occurs, it probably is not available.

The CoreProxy is initialized in the service provider with a reference to an object implementing the ServiceBackend interface. The ServiceBackend object is a remote object in the service provider and used by the CoreProxy to download the information the service user requires. The UML class diagram of the ServiceBackend interface is shown in table 5.5

***interface* ServiceBackend**
+ CoreProxy(backend:ServiceBackend)
+ Entry[] getEntries()
+ ServiceProxy getServiceProxy()
+ inputStream getDefaultPolicy()
+ void poll()

Table 5.5: The UML Class Diagram of the ServiceBackend interface

5.3.2 The Service Proxy

ServiceProxy is an interface the service provider developer has to implement when designing a service provider. The object implementing this interface represents the backend in the service provider associated with a single session. For example, for a streaming service the service offer would represent the methods needed to access the stream and possible to shut it down or change the data rate.

The methods defined in the ServiceProxy interface deals solely with session initiation and migration. The implementation of ServiceProxy also has to implement service specific methods that are needed to provide the service. For a streamer service this could be methods like setDataRate or getStream. The UML class diagram of the ServiceProxy interface is shown in table 5.6

Service Proxy
+ ServiceSession initSession() *+ ServiceState getState()* *+ void setState(state:ServiceState)* *+ shutdown()*

Table 5.6: The UML Class Diagram of the ServiceProxy Class

ServiceProxy has four methods:

- **initSession**
  initSession starts a new session with this service, and returns a ServiceSession object. The service provider is supposed to subclass the ServiceSession object to provide the functionality specific to the service type. This object has to be of the same type as the service type specified by the application layer when the session was initialized.

- **getState**
  Returns the state of the session provided as a parameter. The state is returned as a ServiceState object, which the service is supposed to subclass to contain the relevant state variables for that particular type of service. This method is generally only used by the target service to get the state of the source service during service migration.

- **setState**
  This method is called to set the state in the service provider represented by this service proxy so that it can take over a session which is at that

state. When this method call returns the ServiceProxy is ready to resume the responsibility for the session.

- **shutdown**
  This method causes the ServiceSession to shutdown gracefully. It should cause the service provider backend to also shutdown and release any resources being used.

5.4 The Service Manager

The task of the service manager is to locate and monitor the service providers the session module is interested in. The UML class diagram shown in table 5.7 shows the interface of the service manager towards the session module.

***interface* ServiceManager**

+ addServiceListener(listener:ServiceListener)
+ removeServiceListener(listener:ServiceListener)

Table 5.7: The UML Class Diagram of the ServiceManager interface

The Session object has to implement the ServiceListener interface. This is a simple interface which allows the object to receive events from the service manager. When there is a change in the list of available service, the service manager generates a ServiceEvent and sends this to all the ServiceListeners which have subscribed to this information. The UML class diagram for the ServiceEvent class is shown in table 5.8.

ServiceEvent
ServiceOffer[] getServiceOffers()

Table 5.8: The UML Class Diagram of the ServiceEvent class

The service manager encapsulates information about one CoreProxy in a ServiceOffer object. This object can store the entry information from the CoreProxy to enable the Session object to retrieve the entries without having to use the CoreProxy. Using the CoreProxy to get the entries causes the CoreProxy to call back to the service provider, and has much higher overhead than simply returning a list of entries stored in a local object.

The ServiceOffer object can also be used by the Session object to store policy information associated with this CoreProxy.

The UML class diagram representation for the ServiceOffer class is shown in table 5.9.

ServiceOffer
+ Entry[] getEntries() *+ CoreProxy getProxy()* *+ PolicyValue getPolicyValue()* *+ void setPolicyValue()* *+ float getValue()* *+ float getMigrationThreshold()* *+ int compareTo(o:Object)*

Table 5.9: The UML Class Diagram of the ServiceOffer class

The getEntries and getProxy methods are used to access the entries and CoreProxy stored in the ServiceOffer. The rest of the methods are use by the Session object to set and read policy information in the ServiceOffer object.

5.4.1 Summary of ServiceManager

To recapitulate the ServiceManager:

The service manager monitors the available service providers. Each service provider is represented by a CoreProxy, and the service manager stores these CoreProxy objects in ServiceOffer objects. When there is a change to the available ServiceOffer objects, a ServiceEvent is sent to every ServiceListener that has requested information. The ServiceListener will typically be a Session object.

5.5 The Session Management Module

The Session Management module of Embouchure is responsible for initiating and keeping alive session that the application layer requires. The application layer provides a service type to open the session to as well as a policy on how to select the best service and when to migrate from one service to another.

The Session Management Module consists of the Session Manager, Session objects and ServiceSession objects.

5.5.1 The Session Manager

Table shows the UML class diagram of the Session Manager. The Session Manager is a factory for Session objects. It provides the application layer interface of Embouchure . The Session Manager has a single method: initSession. An initSession method call always includes a service type parameter and may as well include a reference to a data stream containing a policy declaration.

SessionManager
+ Session initSession([policyRef:String,] type:Class)

Table 5.10: The UML Class Diagram of the ServiceManager interface

The service type parameter defines what kind of service the application layer wishes to open a session with. This is a interface including all the methods the application layer wishes that the required service implements.

5.5.2 The Session Interface

The UML class diagram of the Session interface is shown in table 5.11. Each implementation of Embouchure has to provide an implementation of the Session interface.

The Session interface has the following methods:

- **getServiceSession**
  This method returns the ServiceSession object of the service that has been chosen to provide the session. This object is obtained from the ServiceProxy that is used to initiate the Session.

- **addSessionListener/removeSessionListener**
  These methods allows the application to subscribe and unsubscribe to events in the Session object. In future versions of Embouchure there will probably be possible to subscribe to only the events the application layer is interested in, but for now the application layer has to either get them all or none at all.

- **abort**
  This method aborts and shuts down the Session gracefully.

interface Session

+ ServiceSession getServiceSession()
+ void abort()
+ addSessionListener(listener:SessionListener)
+ removeSessionListener(listener:SessionListener)

Table 5.11: The UML Class Diagram of the Session interface

5.5.3 The ServiceSession interface

An object implementing the ServiceSession interface is returned to the application layer when getServiceSession is called in the Session object. The ServiceSession object uses a ServiceProxy to provide the actual service to the application layer. The ServiceSession has to be able to stop using one ServiceProxy and to continue the session using another ServiceProxy object. The UML class diagram representation of the ServiceSession interface is shown in table 5.12

interface ServiceSession
+ void suspend() *+ void resume()* *+ void setServiceProxy(proxy:ServiceProxy)*

Table 5.12: The UML Class Diagram of the ServiceSession interface

The ServiceSession object is initiated by a ServiceProxy instantiation and initially bound to the ServiceProxy that created it. When suspend() is called, the ServiceSession object has to make sure the the ServiceProxy is stopped by calling the shutdown() method in ServiceProxy. When the suspend() method of the ServiceSession returns, the caller can be sure that both the ServiceSession and the active ServiceProxy have been stabilized.

When suspended, the ServiceSession can receive a call to setServiceProxy(). This call instructs the ServiceSession to henceforth use another ServiceProxy to provide the service for the application layer.

When resume is called, the ServiceSession resumes the service to the application layer using the new ServiceProxy.

5.5.4 Session migration

The implementation of the Session object uses all the information it has gathered from the service manager and the policy module to determine when session migration should be done. Exactly how session migration is implemented is left to the implementation of the Session object. We provide one possible implementation of the Session object in the implementation chapter (6.3).

5.6 Example of usage of Embouchure

In this section we provide an example of usage of Embouchure .

An application decides that it wants to open a session with a service type. It calls the initSession method in the SessionManager.

When the SessionManager receives this call, it creates a session object and instantiates it with the policy and service type information. The SessionManager is now done with all its work on this session.

Upon being initialized, the Session object registers with the Service Manager. It tells the Service Manager to start monitoring service of the type required by the application layer. Hopefully after a while the Service Layer sends a ServiceEvent object containing an array of ServiceOffers to the Session object.

If the application layer provided a policy reference to use with this session, this stream is used to initialize the Policy module, if not, a default policy is downloaded from the CoreProxy object of one of the ServiceOffers.

Using the Policy.eval() method, each ServiceOffer is assigned a PolicyValue object and ranked due to its policy value. The one with the highest ranking is chosen, getProxy is used to get the CoreProxy object associated with the ServiceOffer, and getServiceProxy is used on that core proxy. Using the service proxy, the session is initiated.

The getServiceSession method in the Session object blocks until this point, but now the initSession method in the service proxy object is used to generate a ServiceSession object which in turn is returned to the application layer.

The ServiceSession implements the original service type the application layer defined when initiating the session. At this point, the application layer can use these methods to initiate the actual session with the selected service provider.

The Service Manager continues to provide the Session with information about the different services. If at some point another ServiceOffer is ranked higher than the rank of the one currently used times the migration threshold, session migration is initiated.

The ServiceOffer which is now the best suited one is used to access the CoreProxy associated with it. This CoreProxy is used to download the ServiceSession. At this point, the policy values are compared again to make sure that migration is still needed. If migration is still needed, the ServiceSession is told to use the new ServiceProxy to provide the service for the application.

Exactly how session migration is done is left to the implementation of the Session object.

5.7 Summary

In this chapter we have presented the design of Embouchure . We focus on the session management and policy modules, but also provided a rudimentary design of the service manager as well as some requirements of the service provider backend.

In the next chapter we will provide a sample implementation of the Embouchure middleware using the Java programming language. Following that we will implement a sample service provider.

Chapter 6

Implementation of Embouchure

In this chapter we present a sample implementation of parts of Embouchure . As in the design chapter, we focus on the service user software in general, session and policy management in detail.

In order for the focus parts of the system to work, we implement a simplified version of the service manager. This service manager can then be used to test the other parts of the system.

We start this chapter by giving an overview of the implementation. In section 6.2 we describe the implementation of the policy module and in section 6.3 we cover the implementation of the session management module. Finally, in section 6.4 we provide the implementation of the simple service manager.

The Java code for this implementation of Embouchure is aproximately 3500 lines long. Instructions for downloading code and API documentation can be found in appendix A

6.1 Overview of the implementation

The implementation of Embouchure is done using the Java programming language. We have chosen to use Java because it is well suited for distributed, mobile systems. Java has support for strong code mobility, which means that it is possible to move objects from computer to computer which both include state variables and methods which operate on these variables. The effect of this is that the target computer need not be aware of how a particular method is implemented, only the interface of the method.

Using the terms of Embouchure , the service provider could upload a proxy object to the service user which contains all the necessary code for communication with the service provider. Java makes this possible through its serialization and reflection packages.

6.1.1 The structure of Embouchure

We have divided the implementation of Embouchure into three Java packages:

- **embouchure.servicelayer**
  This package includes all the code for the service manager.

- **embouchure.sessionlayer**
  This package includes all the session management code.

- **embouchure.policy**
  This package includes all the policy management code.

6.2 The implementation of the Policy module

The policy module has two main tasks: To compile the XML code used to describe the policy into runnable code, and to use this code to generate the PolicyValue object which is returned by the eval call to the Policy object.

The Policy class is at the core of both these tasks. It acts as the interface with the other packages. An object of the Policy class is generated each time a new policy stream or file is introduced. In other words, each time a session is initiated by the application layer, or when the application layer uses the methods in the Session object to change the policy associated with that session.

We have chosen to simply use the UNIX file system as a policy repository. When an application wishes to define a new policy, it generates a policy file which is stored in the file system. Both initSession in the SessionManager and changePolicy in the Session object requires a string describing the policy as a parameter. In this implementation we merely use the path to the policy file to describe the policy.

6.2.1 The Policy Compiler

The policy compiler parses the policy file and turns it into a *StatementSet*. A StatementSet is an ordered collection of statements. Currently there is only one kind of statement available in the policy language, the *DefineStatement*. This statement is represented by the <DEF> tag. An example of a <DEF>

tag is shown below:
<DEF NAME="BW" IF="1" VALUE="minMaxLinear(100,10000,ENTRY_BW)"/>

A DefineStatement has three parts:

1. **NAME**
   The name of the variable the value is stored in.

2. **IF**
   If this expression evaluates to >0, the statement is executed.

3. **VALUE**
   The value that is stored if the expression is evaluated.

The expression

In Embouchure we have three types of atomic expressions:

1. *A floating point number*
   A floating number, for example 5.

2. *An variable*
   The name of a previously defined variable, for example ENTRY_BW. The value stored in this variable is substituted when the expression is evaluated. Only floatin point variables are available in Embouchure .

3. *A built-in method*
   A method call to a built-in method, for example minMaxLinear(100,10000,ENTRY_BW). The method call may in turn have parameters which are expressions.

In addition it is possible to join any two atomic expressions into a composite expression by using the floating point operators: $+, -, *$ and $/$.

The boolean operators $<, >$ and $=$ may also be used. They result in a floating point expression where the result is 0 if the boolean expression evaluates to false and 1 if it evaluates to true. Both these numbers are far enough from the threshold value at 0.5 to insure proper functionality.

The expression in the IF or VALUE fields of the <DEF> tag can contain either a single atomic expression, or two atomic expressions joined by an operator.

If more elaborate expressions are needed, successive <DEF> statements have to be used.

Evaluation of the policy

After the policy language has been compiled into a set of statements, it can be evaluated. The eval() method of the Policy class is used for this. Here a EntrySet is given as a parameter. The EntrySet object includes all the entries provided to the Session by the ServiceManager.

The first task of the Policy object which receives an eval() call is to propagate through the Entries and create Variables for each one.

In this implementation of Embouchure , only entries which has a floating point number as its value will be considered.

Variables in Embouchure are stored in a Hashtable in the Policy object. As the policy object gets a call to eval() with a list of entries, it propagates through the entries and creates a Variable for each entry that has a numeric value. In order to separate the entries from the internal variables specified in the policy language, the string "Entry_" is added to the front of each entry name when stored in the variables Hashtable.

After the entries have been converted into variables, the first statement of the policy is executed. If this is a <DEF> statement, and the IF expression evaluates to >0, a variable is created with the name defined by the statement. This variable is given the value returned from the VALUE expression, and inserted into the variables Hashtable.

In turn, all the statements are executed.

After all the statements have been executed, the policy object retrieves the values stored in the POLICY_VALUE and MIGRATION_THRESHOLD variables, and uses them to generate the PolicyValue object which is returned to the caller.

At this point, the policy object is ready to receive another call to the eval() method.

The current implementation do not support the parallel evaluation of multiple EntrySets associated with a single policy. The evaluations have to be done in succession.

6.3 The implementation of the Session management module

The core of the session management module is the *Session* objects. These objects implement the Session interface and are the application's interface to Embouchure . These objects subscribe to service information from the service manager, and uses the policy module to rate the service offers it gets.

The application uses the *Session* objects to access the *ServiceSession* objects which provide the methods the application uses to access the service.

We have provided a skelton implementation of the various interfaces.

The application viewpoint From the application layers point of the view, the important classes are the SimpleSessionManager, the SimpleSession and the ServiceSession.

The SimpleSessionManager is an implementation of the SessionManager interface, and SimpleSession is an implementation of the Session interface. Neither SimpleSessionManager nor SimpleSession have any public methods that do not derive from their interfaces. The SessionManager interface can be reviewed in table 5.10 and the Session interface in table 5.11.

The application uses the SimpleSessionManager to initate the session with the initSession() method. The SimpleSessionManager creates a SimpleSession object and generates a new Thread for the new object. The new SimpleSession object is the returned to the application layer.

The application layer may now decide to subscribe to events in the SimpleSession object by using the addSessionListener method.

When it is ready, the application uses the getServiceSession method to start the interaction with the service. This methods blocks until the SimpleSession object has chosen a service provider and initated the session with it.

When the getServiceSession method returns a ServiceSession object, the application uses this object until it is done with the service.

The SimpleSession object The SimpleSession object is at the core of the session module. This object decides which service provider to use and when to migrate to a new service provider.

The first thing a new SimpleSession object does when it is created, is telling the ServiceManager which service types it is interested in. The SimpleSessionManager tells the SimpleSession where to find the ServiceManager through the contructor of the SimpleSession.

The next step is to parse the policy file provided by the application and generate a Policy object to evaluate the service offers.

After some time, the service manager has hopefully returned some service offers containing the core proxy and a list of entries associated with each service provider. The Policy object previously created is now used to evaluate these service offers. If one of the service offers is found to be satisfactory, its core proxy is used to retrived its service proxy.

The service proxy is in turn used to generate a ServiceSession object. This ServiceSession object is stored and returned to the application when the application requests it using the getServiceSession method.

Selecting how long to wait before processing the service offers for the first time is tricky. If the SimpleSession waits too long, the overhead of starting a new session might get too large. If the SimpleSession do not wait long enough, the best service offer might not have arrived yet. One possibility is to include a tag in the policy language to allow the application to decide how long the SimpleSession object should wait before starting the session.

Selecting when to migrate The SimpleSession object maintains a list of all the available service offers. If at any time the best offer is better than the one currently being used, service migration may be initated. The SimpleSession object compares the policy value of the active service offer with the best one. If the best suited service offer has a policy value which is more than he active service offer times the migration threshold of the session, migration is initiated. The actual migration is handled by the ServiceSession object in consort with the service offers involved.

The migration process When the SimpleSession object decide its time to migrate from one service offer to another, the following events occur:

1. The SimpleSession uses the CoreProxy in the target ServiceOffer to download a new ServiceProxy.
2. The SimpleSession calls the suspend() method in the ServiceSession, causing it to stop using the active ServiceProxy.
3. The SimpleSession calls the suspend() method in the active Service-Proxy. This causes the ServiceProxy to stabilize its state so that a future getState() method calls returns a deterministic result.
4. The SimpleSession object retrieves the state of the active ServiceProxy with the getState method, and uses the setState method in the target ServiceProxy to put it in the same state.
5. The SimpleSession calls teh setServiceProxy method of the ServiceSession with the target ServiceProxy as parameter.
6. The SimpleSession object calls the resume method in the ServiceSession.
7. The SimpleSession object calls the shutdown method of the old ServiceProxy.

6.4 The implementation of the Service Manager

We have designed and implemented a very simple service manager for use in Embouchure . The service manager presented here uses Java RMI name servers to locate the service providers. The service manager is told on startup where to look for these name services and which service types correspond whith which RMI name URLs. In this manner, when a session requests information about a service type, the service manager looks up in its service type to RMI name URL table and connects to the corresponding service providers.

This approach is static and differs from the service Embouchure is meant to provide, but it provides a foundation for implementation and testing of the other parts of Embouchure . A new service manager which supports dynamic service discovery and lookup, for example using JINI or UPnP is needed if Embouchure is to be turned into a working piece of software.

Entry propagation In the fullscale design of Embouchure , any enitity on the way from the service provider to the session object in the service user may add entries as the core proxy passes through. In this manner, network service providers could add a price tag to all proxies passing through, or monitoring software could add information about network bandwidth or latency. In this implementation of Embouchure the focus is on the session and policy control and not on the entry propagation software. We have chosen to only allow the service provider to enter entries for now. This is a simplified approach, but in the same manner as the simplified service manager, it allows us to make a foundation for testing the session and policy software.

Chapter 7

Case study

The focus of thesis is session migration and policy controlled adaptation. In this chapter we implement a sample service provider to show that these functionalities work in Embouchure .

We have chosen a simple stream service as our example. A stream service streams data to the service user. This could be video or audio data, or other kinds of real-time data streams. We have chosen to use a stream server which streams incrementing integer numbers, because it will be easy to detect both missing data and data that are received more than one time.

It is also very easy to make a state object for this service, all that is needed is what integer was received last.

We have called the service the IntegerService.

We will implement the IntegerService using the implementation of Embouchure that was presented in chapter 6. When we have presented the whole implementation of the IntegerService, we will provide a simple use-case description of all the method calls, threads and objects that are parts of a normal session between a service user and a service provider. This use-case will also show how session migration is done.

Appendix A contains information about where the source code and API documentation of the case study can be downloaded.

7.1 Overview of IntegerStreamer

Implementing a service provider for Embouchure can be divided in four tasks:

1. **Making the actual service provider back-end and the type interface that is used to represent it to the application layer in the service user.**

For the IntegerService this means making software in the service provider which will generate the numbers and stream them through a socket to the ServiceProxy. This is defined by the IntegerStreamer interface, and implemented by the IntegerStreamerImpl class. In addition we have to define the IntegerService interface, which is the type interface of the IntegerService.

2. **Making the service lookup back-end system.** In the simplified version of Embouchure , this means making a ProxyHandle and binding it in a RMI name service. For IntegerService we have to make sure that this ProxyHandle returns a CoreProxy which is bound to the proper ServiceBackend.

3. **Making the back-end support for the CoreProxy: Implementing the ServiceBackend interface.**

4. **Making the modules which will be downloaded to the service user: Implementations of ServiceProxy, ServiceState and ServiceSession.**
   In IntegerService this means defining the class: IntegerServiceProxy, IntegerServiceState and IntegerServiceSession.

Remember that the service provider is not allowed to provide its own CoreProxy, it has to use the standard one.

7.2 Type interface and service provider backend

We start out by defining the type interface that represents the IntegerService. The UML code for this interface is shown in table 7.1

***interface* IntegerService**
+ InputStream getStream()

Table 7.1: The UML Class Diagram of the IntegerService Interface

The IntegerService interface has a single method which allows the application to gain access to the stream of data. The data is represented as a InputStream.

The IntegerStreamer has to be able to generate data and stream these to the service user through a IntegerServiceProxy. We have decided to stream data on a regular socket, and to use Java RMI to allow the IntegerService-Proxy to access IntegerStreamer.

IntegerStreamer is a remote interface, in other words, it extends java.rmi.Remote. Any class which implements a Remote interface can be used as a remote object. The easiest way to do this is to subclass java.rmi.server.UnicastRemoteObject, and this is the way we have done it with IntegerStreamerImpl. IntegerStreamer is the interface the IntegerServiceProxy uses to get access to the socket containing the integers, while IntegerStreamerImpl is the actual class which implements this functionality.

The IntegerStreamer interface's UML definition is shown in table 7.2.

***interface* IntegerStreamer**
void initSocket(startAt:int) *void shutdown*

Table 7.2: The UML Class Diagram of the IntegerStreamer Interface

IntegerStreamer extends the java.rmi.Remote interface. java.rmi.Remote has no methods, but merely acts as a marker interface to mark all classes which implement it as remote classes.

The initSocket method is used by the IntegerServiceProxy to set the correct state in IntegerStreamer. This makes it possible to start streaming integers using one service provider and switch to another who takes over where the first one left off.

The shutdown method is used to gracefully shut the IntegerStreamer down.

The UML class diagram of IntegerStreamerImpl is shown in table 7.2

IntegerStreamerImpl
void initSocket(startAt:int) *void shutdown* *int getPort()* *void run()*

Table 7.3: The UML Class Diagram of the IntegerStreamerImpl Class

IntegerStreamerImpl implements IntegerStreamer, so it contains the initSocket and shutdown methods of IntegerStreamer. IntegerStreamerImpl is a subclass of java.rmi.server.UnicastRemoteObject.

IntegerStreamerImpl also implements the java.lang.Runnable interface.

The getPort method is used to access the port number the IntegerStreamerImpl is streaming data to.

Every IntegerStreamerImpl objects run in their own thread. This assures that a single IntegerService service provider can provide service to several service users at the same time. When constructed, the IntegerStreamerImpl object selects a random port and tries to bind to it. If it is successful, it starts to listen for connections to that port. If someone connects to the selected port, IntegerStreamerImpl starts to stream integers through the resulting socket.

Each IntegerStreamerImpl object is associated with a single IntegerServiceProxy object. IntegerServiceBackend objects are responsible for creating both IntegerServiceProxy objects as well as IntegerSTreamerImpl objects and for binding each proxy/streamer pair together.

7.3 The RMI name service bindings

When talking of RMI it is constructive to talk of the server and the client. The server allows clients to call methods in one of its objects. We call this object the remote object. This remote object implements a predefined interface which describes which methods that can be called remotely. After the initial connections between the server and the client has been made, a stand-in object has been created in the client. This stand-in object can then be used by the client as if it where the remote object. This stand-in object is often referred to as a stub. When a method call is made in the stub, the call is propagated all the way back to the remote object in the server. Here the method call is executed, and any return value is sent back to the stub which in turn returns the value to the application.

When using RMI one of the main problems is how to locate the server and the remote object in the first place. The simplest solution is to use an application called rmiregistry. This is a service which allows applications that want to export remote objects, that is applications that wish to be rmi servers, to store a remote referense to their remote objects. In addition, each referense thus stored is associated with a name.

When a client wish to use a remote object, it contacts a rmiregistry and identifies the remote object it is interested in by the name. If a remote object is stored in the rmiregistry using that name, the referense is returned to the client. This referense is in turn used to download the stub object from the server and to instantiate it so that method calls to it are redirected to the correct remote object.

In IntegerService as in any service provider in the current version of Embouchure , we have store an implementation of the ProxyHandle interface in a rmiregistry for the service users to lookup. The purpose of the ProxyHandle is to provide functionality for initializing a CoreProxy to be able to communicate with the ServiceBackend of the service provider.

In IntegerService, the implementation of ProxyHandle is called IntegerProxyHandle.

As we covered in the design chapter, ProxyHandle has a single method. The UML class diagram of ProxyHandle is shown in table 7.4.

***interface* ProxyHandle**
CoreProxy getCoreProxy()

Table 7.4: The UML Class Diagram of the ProxyHandle Interface

After the initial lookup and initialization of the RMI connection with the ProxyHandle remote object, the service user can use the getCoreProxy method to download the CoreProxy object the is associated with the ProxyHandle. CoreProxy is a final class and known to the service user on beforehand, so all that is actually downloaded is the content of the data fields in the CoreProxy. These fields, however, bind the CoreProxy to a particular instance of the ServiceBackend interface. In the case of IntegerService, this is a IntegerServiceBackend object.

IntegerServiceBackend is also a remote object, but because the referense is passed to the service user through a CoreProxy object, there is no need to register the back-end object in a rmiregistry.

Recalling from the design chapter the UML code for a ServiceBackend is shown in table 7.5

The IntegerServiceBackend implements these methods and provides two additional methods. Table 7.6 shows the UML class diagram of IntegerServiceBackend.

The IntegerServiceBackend is much the main administrator of the service provider. The class has several tasks:

- It is responsible for acting as a service provider back-end for the CoreProxies that are associated with this IntegerService service provider.

interface **ServiceBackend**
void addRemoteServiceListener(rsl:RemoteServiceListener) *void removeRemoteServiceListener(rsl:RemoteServiceListener)* *ServiceProxy getServiceProxy()* *String getEntryString()* *boolean poll()*;

Table 7.5: The UML Class Diagram of the ServiceBackend Interface

interface **IntegerServiceBackend**
void addRemoteServiceListener(rsl:RemoteServiceListener) *void removeRemoteServiceListener(rsl:RemoteServiceListener)* *ServiceProxy getServiceProxy()* *String getEntryString()* *boolean poll()* *JFrame getFrame()* *void setEntries(entries:String)*

Table 7.6: The UML Class Diagram of the IntegerServiceBackend Interface

- It is responsible for sending updates to all interested parties when there are changes to the entries which describe the service provider. Interested parties register with the CoreProxy the have downloaded, and this CoreProxy uses the addRemoteServiceListener method in the IntegerServiceBackend to register for events. When changes to the entries are made, a RemoteServiceEvent is sent to everyone that has registered.

- It is responsible for creating and returning IntegerServiceProxy objects when a service user calls the getServiceProxy method in the CoreProxy associated with this IntegerServiceBackend. When a IntegerServiceProxy is required, the IntegerServiceBackend object is also responsible for creating a IntegerStreamer object that will act as remote object for that particular IntegerServiceProxy.

The getFrame method returns a window which can be used to change the entries associated with the service provider. This window is used later when we run tests on the system.

The setEntries method can be used to change the entries associated with the service provider.

7.4 The downloaded classes

Finally we will cover the classes that the service user do not know on beforehand. The code contained in these classes will have to be downloaded to the service user or retained in other ways. This code will not necessarily be present in the service user before the session with the IntegerService is initiated.

These classes are IntegerServiceSession, IntegerServiceState and IntegerServiceProxy. In addition, IntegerServiceSession has a embedded class called IntegerStream.

7.4.1 IntegerServiceProxy

We start out by examining the IntegerServiceProxy. As we have described earlier in this chapter, IntegerServiceProxy objects are generated by the IntegerServiceBackend object when the service user calls getServiceProxy in the CoreProxy of a service provider. The IntegerServiceProxy implements the ServiceProxy interface. Recalling from the design chapter the UML class diagram of the ServiceSession interface is shown in table 7.7.

These methods provide functionality needed to support session migration. In addition to this functionality, IntegerServiceProxy needs a method

interface **ServiceProxy**
ServiceState getState() *void setState()* *ServiceSession initSession()* *void shutdown()*

Table 7.7: The UML Class Diagram of the ServiceProxy Interface

to access the stream of data which is originating from the IntegerStreamer that is associated with this IntegerServiceProxy.

IntegerServiceProxy has to keep track of which integers have been passed through it, and for this reason we have decided that the access method for the data is a simple:

int read()

The read method returns an integer containing the next value that is sent from the IntegerStreamer.

Initiating a new session

When a IntegerServiceProxy is used to initiate a new session, the initSession method is used to generate a IntegerServiceSession. If the IntegerServiceProxy is used to migrate a session, either setState or getState is used depending on whether it is the target or source of the migration.

Migration of an existing session

getState should only be called after shutdown has been used to stop the proxy. This is to avid race conditions, where the state may or may not include the last integers that have been read.

When setState is called on a IntegerServiceProxy, it tries to tell the IntegerStreamer that is associated with the proxy which integer it should start at. If setState returns without incident, the IntegerServiceProxy is ready to provide data through its read() method.

7.4.2 IntegerServiceState

This class is just a holder class which contains the last integer that was received.

7.4.3 IntegerServiceSession

The IntegerServiceSession has to implement both the ServiceSession interface and the IntegerService interface. An object of this class is what is returned when the application calls the getServiceSession method in Session object that resulted from the initial initSession call to the SessionManager.

The IntegerService interface

Through the IntegerService interface the IntegerServiceSession has to provide an InputStream through the getStream method. To provide this stream is can only use the read method in the IntegerServiceProxy it is currently using. In order to make an InputStream from this, the IntegerServiceSession uses an embedded class called IntegerInputStream which is a subclass of InputStream. IntegerInputStream overrides the read method of InputStream and returns the data it gets from the IntegerServiceProxy.

The ServiceSession interface and migration

Recalling from the design chapter, the UML class diagram of the ServiceSession interface is shown in table 7.8

interface **ServiceSession**
void suspend() *void resume()* *void setServiceProxy(serviceProxy:ServiceProxy)*

Table 7.8: The UML Class Diagram of the ServiceSession Interface

These three methods are used when the IntegerServiceSession should start to use another IntegerServiceProxy as the source of the data that is sent to the application layer. When suspend() is called, the IntegerServiceSession suspends its actions. If the application layer tries to read data from the IntegerInputStream when the IntegerServiceSession is suspend, the read operation will block. After setting its own state to suspended, IntegerServiceSession sends a shutdown message to the currently active IntegerServiceProxy.

At some later point in time, the setServiceProxy is called to assign a new IntegerServiceProxy the IntegerServiceSession can use to access the data it wants to send to the application layer. At this point, any referense to the old IntegerServiceProxy is discarded, and the new one placed in its stead.

When resume is called, operation resumes. If a read call to the IntegerInputStream from the application layer had been blocked, it is now unblocked and the stream of data to the application layer resumes.

7.5 A simple use-case

We will now provide a walk-through description of all method calls, objects generated and threads executed which are used to generate a session between the service user and a service provider. We will then follow that description with a step-by-step description of what happens when session migration is done.

7.5.1 Step 1: Starting the service provider

The first step in this use-case is to start the service provider. This is done via the IntegerServiceStarter class. This class starts out by generating an IntegerServiceBackend object. Following this a CoreProxy object is generated and bound to the IntegerServiceBackend object.

Now a IntegerProxyHandler object is generated and given a reference to the CoreProxy previously initiated. This IntegerProxyHandle is bound in the RMI registry server using the name "integerservice".

The service provider is now ready to receive calls from the service user.

7.5.2 Step 2: Starting the service user

The second step in the use-case is starting the service user. The SimpleServiceManager has to be told where it should look for service providers at startup, and it tries to download the CoreProxy objects associated with each of the RMI registry lookup strings it has gotten.

In this example, the SimpleServiceManager has only gotten the lookup string of a single service provider.

The SimpleServiceManagaer generates a RMIServiceWrapper object which is responsible for monitoring the service provider and to generate events to the SimpleServiceManager whenever there is a change in the service provider it monitors.

At this point, the RMIServiceWrapper tries to download a ProxyHandle from the RMI registry. After it has downloaded this proxy handle, it uses the handle to download the CoreProxy that is associated with it. After the CoreProxy has been downloaded, the RMIServiceWrapper uses it to access the entries associated with it, and to subscribe to events which occur if the entries in the service provider changes. Using the CoreProxy and the entries it has retrieved from the service provider, RMIServiceWrapper generates a ServiceOffer object and sends it to the SimpleServiceWrapper inside a WrapperServiceEvent.

Whenever there is a change in the service provider this RMIServiceWrapper monitors, it sends a WrapperServiceEvent to the SimpleServiceWrapper informing it of the change.

The SimpleServiceWrapper stores the ServiceOffer and awaits calls from the session module.

7.5.3 Step 3: Initiating the session

The third step of this use-case starts when an application in the service user calls the initSession() in the SessionManager. At this point, a SimpleSession object is generated to handle the session. This object uses the policy file provided by the initSession call to make a Policy object which is later used to rate the ServiceOffers it receives.

As we discussed in the implementation chapter, the SimpleSession object runs a separate thread. This is the thread which does migration and service selection. Additionally SimpleSession has a method that is called asynchronously. This is the method used by the SimpleServiceManager to inform the SimpleSession that there are ServiceOffers available, the serviceChanged() method.

The SimpleSession registers with the SimpleServiceManager, and starts to receive information about available ServiceOffers.

After the initial delay time has passed, SimpleSession uses the Policy object it has created to assign each of the ServiceOffers a policy value and a migration threshold value. If the ServiceOffer with the best policy value has a value that is larger than the one defined in MIN_VALUE, it is used to initiate a session.

The first step in initiating the session is to retrieve the CoreProxy from the ServiceOffer. The CoreProxy object has already been downloaded to the service user, so this is only a local call. Following this, the CoreProxy is used to download the IntegerServiceProxy object. Should something go wrong when downloading this object, the ServiceOffer is discarded and the selection process in done again. If the IntegerServiceProxy object is successfully downloaded, initSession is called on it. This method call generates a IntegerServiceSession object.

This IntegerServiceSession object can now be returned to the application layer when getServiceSession is called and the application can start to stream data from the service provider.

7.5.4 Step 4: Session migration

Before session migration can be undertaken, another service provider will have to be started. This is done by doing step 1 over again. At this point we have to assume that during step 2 we provided the SimpleServiceManager with two lookup names to use to locate the IntegerProxyHandle objects. That would result in two RMIServiceWrapper objects being created, each monitoring one of the service providers.

When we now start the second service provider, the RMIServiceWrapper object associated with it will make a ServiceOffer object and send it to the SimpleServiceManager encapsulated in a WrapperServiceEvent object. This will in turn cause the SimpleServiceManager object to send a ServiceEvent to the SimpleSession object containing both ServiceOffers. These service offers will be stored in the SimpleSession object, and the next time the SimpleSession evaluates its service offers, there will be two service offers to consider.

At this point three things might occur:

1. The ServiceOffer that is currently being used to provide the service is still the best one. No action need to be taken.

2. The ServiceOffer that is not being used is a little better than the currently active one, but not more than the product of the active ServiceOffer's policy value and the migration threshold. No actions need to be taken.

3. The ServiceOffer that is not being used is much better than the currently active one. "Much better" means that the policy value is more than the product of the active service offer and the migration threshold. In this case, migration to the second service offer has to be done.

In the design chapter we presented a seven point list to describe the migration process. SimpleSession now goes through this list. If it goes all the way through the list without any errors occurring, the session is resumed using the new service provider.

A lot of errors can perceivably occur during session migration. We will cover these in the next chapter.

Chapter 8

Testing and evaluation of Embouchure

In this chapter we present an evaluation of Embouchure . We evaluate Embouchure based on the overhead it introduces when compared to other middleware solutions, the complexity of use for both service user and service provider and the correctness of the design. Finally, we draw some conclusions to the general performance and applicability of Embouchure .

8.1 Overhead evaluation

In this section we examine if Embouchure causes overhead in the use of networked resources, and if so, how that overhead compares to existing technologies. We have not done any empirical studies of the overhead caused by Embouchure because the service manager is only partially implemented, but we will provide a number of arguments here to what kind of overhead one could expect Embouchure to show.

We define *overhead* to be the amount of resources that are used on an action beyond the minimum required for that task. The resources we will cover are time, network bandwidth, power consumption, memory usage and CPU usage.

8.1.1 Temporal overhead

In this section we will examine the overhead in Embouchure which causes actions to take longer time than they should.

Setup delay

The main cause for concern in Embouchure is the setup time of a new session. As we discussed in section 6.3, the session object has to wait until it has gotten a list of available service offers from the service manager. This waiting time is a direct setup overhead. In order to keep this overhead as small as possible, the session object should choose a service offer to initiate the session as soon as it has one that is acceptable. If the session object makes this choice too soon, the possibility that a better suited service offer may arrive shortly could be great. This in turn increases the possibility of a session migration. This session migration could in turn cause temporal overhead.

In the current version of Embouchure there is no definition of what policy value a service offer should have in order to be good enough.áHowever, since 1 is the highest possible policy value a service offer can have, as soon as a service offer with policy value 1 is returned from the service manager, it can safely be selected as good enough.

This suggests that an application which is dependent on a small setup temporal overhead should provide a policy which is more likely to return a policy value on 1. This, however, deprives the application from the possibility of distinguishing between service offers effectively.

One solution that could be used to amend this problem, would be to allow another predefined value to the policy language called STARTUP_THRESHOLD. This value could contain the policy value which the application deems *good enough* to initiate the session.

When compared to single interface mobility enabling systems like mobile IP or mobility in IPv6, Embouchure is likely to show a greater setup delay. Both the mentioned technologies show little or no setup delay beyond that which is shown by systems without mobility capabilities.

Migration delay

The second area where there might be temporal overhead is during the migration from one service provider to the other. The way Embouchure is designed at the moment, the ServiceSession has to be suspended, the ServiceProxy has to be suspended, then the state is transferred from the old ServiceProxy to the new one, who in turn sets the state in the service provider it is associated with, and when all that is done, the ServiceSession is resumed.

A lot of the functionality here is in the service provider implemented objects of ServiceSession and ServiceProxy. If the suspend, resume, setState and getState methods are badly implemented, the migration delay might get significant. In any case, there is likely to be a delay. The requirement of having both ServiceSession and ServiceProxy suspended before the state is transferred between the service providers is there to protect the integrity of the session. This might be a stringent requirement, and for some session types it might be unnecessary. Making it possible for the application to choose a less stringent approach to session migration could possibly help to reduce the migration delay.

Compared to mobile IP and IPv6 we believe that Embouchure is likely to exhibit less migration delay. In single interface mobility, the term describing migration delay is hand-over delay. Most of the hand-over delay in mobile IP stems from the fact that the network interface that is migrated has to register itself with the new network. The interface could use protocols like DHCP to get the network address and so forth. Mobile IP also needs to register with the resident foreign agent and wait until it has updated the home agent. Additionally, packages arriving at its last point of attachment might need to be sent again.

Embouchure does all this before the core proxy is even downloaded and compared for migration. At the expense of downloading core proxies which will not be used, Embouchure removes the delay in migration which originates from initializing the network layer of the protocol stack. As we saw in the example with the StreamService service specific migration delay is likely in a lot of the services that could be accessed through Embouchure .

8.1.2 Bandwidth overhead

Embouchure wastes a lot of bandwidth downloading core proxies of services which will not be used. The whole matter of downloading both core proxies and service proxies is in itself overhead. If the service user knew beforehand where the service provider was and how to access it, perhaps using sockets instead of RMI calls, there would be a much traffic on the network to provide the same service.

In the design chapter we made sure that the core proxy is as small as possible in order to minimize this overhead.

If the mobile device only has access to slow network links, this bandwidth overhead could be crippling. The whole design is based on the possibility of wasting a lot of bandwidth in favor of simplicity of development.

Compared with mobile IP or mobility in IPv6, the bandwidth overhead in Embouchure is likely to be far greater. Mobile IP wastes bandwidth in the Internet by having all the packages sent through its home agent, but route optimization alleviates this problem to a great degree. In IPv6 this problem is solved because route optimization is embedded in the protocol.

8.1.3 Power consumption overhead

One of the sparse resources on a mobile hand-held device is its battery lifetime. Reducing the power spent on network usage is therefore important. The action of keeping more than one network interface alive at the same time is a considerable drain on power. This is the very reason why [33] argues that multiple simultaneously active network links is unfeasable.

We decided early on to ignore this in Embouchure , and consider the possibilities of multiple simultaneously active network links. Also, most new mobile phones and personal digital assistants (PDAs) aimed at advanced users do offer multiple network interfaces (GSM/GPRS, WLAN, Bluetooth and IR), though operating systems may not yet support simultaneous operation of all interfaces.

The power problem remains a large obstacle for the deployment of Embouchure .

8.1.4 Memory and CPU overhead

For each core proxy the service manager downloads there has to be allocated memory to contain the references to the host where the proxy originated and various other references and data. As we stated previously, Embouchure is likely to download many core proxies which will never be used, and this constitutes a waste of memory.

For each core proxy downloaded there will be an accompanying service offer which will have to be processed to yield a policy value. This process uses CPU resources, and like bandwidth and memory CPU is likely to suffer because of all the unnecessarily downloaded core proxies.

The service manager and session manager along with the session objects and all the other parts of Embouchure also on their own constitute memory and CPU overheads. As for the other overhead issues mentioned, the overhead could be less if the application knew where to find and how to use the networked resources on beforehand.

8.1.5 Overhead conclusions

We have showed that Embouchure is likely to incur serious overheads when compared to other mobility enabling systems, but that it also alleviates some of the overhead issues of those system.

The unnecessary downloading of core proxies is the main source of overhead and reducing it to a minimum will be important. One way of doing this is to employ the *good enough* paradigm introduced in the temporal section above. If the service manager could get information that a core proxy is *good* *enough* it could cease to monitor all the other available ones until it is no longer *good enough*. This could allow the service manager to turn off network interfaces which are not being used and to keep unnecessary network traffic at a minimum. It would also increase the migration delay, because the network interfaces would have to be brought online before migration could take place.

We believe that the overheads discussed here provide a foundation for future work on Embouchure , but that none of these overheads gives us reason to immediately discard Embouchure as unusable or less than useful.

8.2 Complexity of usage

In this section we will look at the complexity of using Embouchure . We will cover the application developer's angle as well as the service provider developer's angle on the issue.

8.2.1 The complexity of the application developer

One of the aims of Embouchure is to provide a foundation on which application developers may use the power of multiple simultaneously network links without having to cope with the complexity.

At first glance, the interface offered by the session manager appears to be simple and straightforward to use. The design, however, requires that the application developer learns how to use the policy language if he wants his application to behave intelligently when adapting to the changing network environment.

In order to be able to use a certain kind of service, the application developer needs to know the interface of that service. This means that he has to find out what interface the service providers use to access their service. Additionally, he has to know what entries are used. If one service provider had one entry call PRICE_PER_MINUTE and another used the entry DOLLAR_PER_SECOND, it would be hard to write a policy which can compare them. However, compared to the complexity of manually finding services and managing migration, we feel that Embouchure has succeeded in offering a simple environment for application developers.

8.2.2 The complexity for the service provider developer

If the service provider was writing a service that would be accessed through JINI, he would have to provide a service proxy and a means for that proxy to contact its backend objects if required.

In order to develop a service provider for Embouchure , the developer would have to tackle complex issues like state and migration code. This is much more complex than the old case, and shows that Embouchure induces more complexity on the service provider developer than other existing systems do.

We believe that is should be possible to write software packages which alleviates this problem. If the service provider developer could use off-shelf software packages to handle migration and state control, much of the complexity would be removed.

8.2.3 Conclusions on complexity

The main complexity problem remains the standardization of service types and entry names. Before Embouchure can be widely used, everyone who wants to provide a printer service should agree on what interface that describes a printer service. The same holds for entries describing common features like pricing or quality of service. Some sort of standardization body would be needed. However, the problem of standardizing interfaces and entries of services is a problem shared by all service-centric computing infrastructure.

In summary, Embouchure provides the application developer with a simple programming interface at the cost of requiring the service provider developer to tackle a more complex situation.

8.3 Fault tolerance

Several error conditions might occur in Embouchure . In this section we list those conditions and determine if the conditions are handled in a consistent manner.

Much of the software in Embouchure is left for the developers of the service providers to write. We will particularly focus on what rules and restrictions apply to those sofware objects.

We look at error conditions that erupt as a result of faulty network connections or service providers and we cover error conditions that might occur during session migration.

8.3.1 Error conditions resulting from loss of contact with the service provider

This section covers the errors that occur when the service provider can no longer be reached because the service provider itself has failed or because the network link connecting the service user and the service provider has been lost.

We assume for now that the loss of connection occurred outside a migration process.

When the connection is lost two things might happen. The use of the service proxy to get data from the service provider could block or it could result in an IOException. In both these cases the ServiceProxy or the ServiceSession have to handle the exception. These are both classes which are implemented by the service provider developers and leaving them to handle this critical error situation may not be the best solution.

In the IntegerService example, the ServiceSession catches the IOException. When the exception is intercepted, the ServiceOffer that is associated with this session is reset to a policy value of 0. Next time the SimpleSession object invokes evalMigration, the session is migrated to another service provider if a satisfactory service provider is available.

IntegerService does not handle the case where the read call to the ServiceProxy simply blocks. It has no way of knowing if it has been blocked because the network is faulty or if it is because the IntegerService service provider does not send data at the moment.

One could perhaps add a time out counter to step in if the read method blocks for too long.

8.3.2 Error handling during service migration.

Migration from one ServiceProxy to another requires the Session object to suspend the ServiceSession before the state is extracted from the old ServiceProxy. Following this the state is inserted into the new ServiceProxy and the ServiceSession is resumed.

The Session object would typically be running in its own thread. The ServiceSession will operate on the thread of the application which made the initial initSession call to the ServiceManager. When these threads interact there is the possibility of deadlock.

Deadlock

It is up to the service provider developer to design and implement the ServiceSession object. The code in this object which handles suspension and resumption of the ServiceSession has to be well thought out to avoid any possibility of deadlock. Suspend is supposed to block before it returns to avoid race conditions when reading the state from the active ServiceProxy. In IntegerService, if the read() method in IntegerServiceSession blocks while downloading data from the IntegerServiceProxy, the suspend method will never return and we have deadlock.

It is very hard to make some general assessments on the chances for deadlock in a general service provider. Deadlock prevention has to be one of the issues a service provider developer has to deal with.

Partial failure

Another possible scenario for errors while executing the session migration is that the service provider associated with the target ServiceProxy becomes unavailable after the old ServiceProxy has been shut down.

In SimpleSession this merely causes a big more temporal overhead during session migration. If an exception is reported before the state has been successfully transferred to the new service provider, the old ServiceProxy is still the "active" ServiceProxy.

Another session migration can now be initiated. Central to making this possible is the fact that the state of the session at all times is available from the ServiceProxy. The ServiceProxy has to keep the state internally and be able to deliver the state without having to contact its service provider.

8.4 Conclusions

Embouchure offers a simple and easy-to-understand interface to the application developer but gives the service provider developer more complexity to handle. It incurs overhead in many areas that can possibly be crippling to the system, but it also shows some promise of alleviating the overheads shown by existing mobility systems.

Embouchure demonstrates that session migration is a possibility for the future, and shows that adapting to the changing network environment is a complex issue.

We have shown that the current design and implementation of Embouchure has some faults, but we believe that Embouchure also contributes new ideas to the area of mobile distributed computing, which is perhaps the main purpose of this thesis.

Chapter 9

Summary and conclusions

In this chapter we summarize our work in this thesis. We cover the contributions we have made to the research field and the deficiencies of our solution. We finish with a list of suggested issues for further research.

9.1 Summary of the thesis

We set out to discover if policy controlled service migration has merit in developing application for the future wireless networks. We wanted to examine if it is possible to create an architecture which makes it easy and effective to fully utilize the heterogeneous and volatile collection of service providers that are available to a mobile device.

Our research into works done by others showed that there is no available technology that can be used to confirm or reject this possibility. We found that several research efforts exists that go part of the way towards service migration and policy control of adaptation to changing envioronments, but that no system or collection of systems could be used directly.

We found that parts of the research done by others could be used to validate the research area. It seemed that there was many reasons to be interested in service migration and that research into service migration could bring the state of research for distributed systems forward.

We presented Embouchure as an example design and implementation to show that policy controlled service migration is possible to develop and use. We believe that we are among the first to use policies to control service migration in this way.

A central goal was to reduce the complexity for the application developer, and we feel that this goal has been accomplished. The complexity of using a policy language to describe what is a good service provider is very low compared to the user perceived quality of service that Embouchure can show.

While we solved some of the problems associated with using networked resources in a heterogenous mobile environment, we created new ones. We have the feeling, however, that some ideas have been presented in this thesis which might add to the pool of knowledge about distributed programming.

9.2 Contributions

In this section we try to summarize what new ideas and concepts we have contributed to the research field. We have not had any major breakthroughs, but we believe that we have contributed to the research on network usage in heterogenous mobile networks in two areas:

We have showed that session migration is a possible way to improve the user perceived quality of service in heterogenous mobile network environments. In a future world which is dominated by mobile devices which host service providers, the ability to migrate a session from one service provider to another is useful. In this manner the user of the mobile device will perceive the quality of service to be better than if the whole session had to be restarted when a service provider became unsatisfactory because the network link detoriated or for other reasons.

We have showed that using a policy language can be useful to allow the application to govern in detail how the service providers should be selected in a simple manner. We have also shown that policy control of the session migration scales well both for the application that wishes full transparency and for the application which wishes to have detailed control on how the service migration is done

In conclusion:

The service migration adds new possibilities that can be exploited, the policy control allows the application to use service migration without having to deal with the complex issues that is associated with it.

9.3 Deficiencies

The current version of Embouchure has several deficiencies. Some of these have been discussed in previous chapters, but we will summarize all of them here:

- **Complexity for the service provider developer**
  While Embouchure offers a very simple interface to the application layer in the service user, this is at the cost of added complexity for

the service provider developer. The service provider developer have to implement many interfaces and create proxies and session objects. When creating these objects, the service provider developers have to take great care to not make mistakes, because the whole system might crash if some of this code fails.

- **Service type and entry naming standardization**
  In order for a service centric architecture to fully work, the participants all have to agree on type names and entry definitions. This is a trouble of all service centric systems and not merely Embouchure . It is very hard to locate a printer if the service provider providing the printer service and the service user do not agree on the service type interface.

- **Overhead**
  As we discussed in the evaluation chapter, Embouchure has several overhead issues. The most serious of these is probably the bandwidth and power overheads. Embouchure uses a lot of bandwidth to discover service provider and keep updated on the state these service provider are in.

- **Simplifications**
  The design and implementation of Embouchure that is presented in this thesis is a very simplified policy controlled service migration system. It is hard to use the policy language to induce complex behavior, and the implementations of the Session and ServiceManager interface in particular do not provide the flexibility that is needed.

9.4 Future work

In this section we present a number of issues that can be the subject of future work on Embouchure . These issues are based on the deficiencies presented in the above section.

- **Complexity for the service provider developer**
  The complexity of the service provider developer stems from the fact that he has to develop from scratch a number of classes to support Embouchure . It should be possible to generate general purpose classes which can be used by a variety of service providers. It should also be possible to develop development environments which simplifies the service provider development. A comparison to think about is the Enterprise Java Beans(EJB) development environments. In order for

a service provider to provide EJB services, he has to implement some pretty complex interfaces and write a lot of code. To simplify this task, several development environments have been developed.

- **Overhead**
  One of the worst overheads in Embouchure stems from the fact the the mobile device is using multiple wireless network interfaces at the same time. Development of better antennas and protocols to use on wireless links could reduce this overhead. If the wireless network interfaces could cooperate and use the same antenna further reductions in the power overhead could be made.

- **Simplifications**
  In order to reduce the simplification and provide added flexibility to the application developer we have to expand the policy language and the support for this language in the Session object. One crucial expansion of the policy language would be to add the possibility of other than the application to enter policies. It should be possible for the computer administrator to set policies which apply to all the application on that computer. The human user of the computer should also be able to set policies which apply to all the application he has control over. The policy language should also be expanded with functionality to allow administrators, human users and application to influence how the discovery and monitoring of the service providers are done. Possibly, the policy language should be expanded so that the service user can influence the service providers to negotiate a service which is even better suited for the application.

Appendix A

Instructions for downloading the code and API documentation of Embouchure

The Java code of Embouchure and the sample service provider IntegerService can be downloaded from http://www.stud.ifi.uio.no/ kjetim/hf/

The API documentation which has been created using Javadoc can also be downloaded from this website.

Glossary

2

2nd Generation Mobile Systems (2G) The first generation of digital cellular systems. This is the generation most of world is using at present date., p. 5.

2nd Second Generation Mobile Systems — Improved version (2.5G) Improvements to the 2G systems used today are sometimes referred to as 2.5G system. They are thought of as improvements on the way to 3G., p. 5.

3

3rd Generation Mobile Communication (3G) The next generation of digital cellular systems. UMTS is an example of a technology which belongs in 3G. 3G is currently being deployed., p. 5.

E

Enhanced Data Rates for GSM evolution (EDGE) Further upgrade of the GPRS standard., p. 15.

F

Fourth Generation Mobile Networking (4G) The new generation of mobile networking that is expected to supersede third generation. It is based on heterogeneous networks and multiple providers rather than the monolithic homogeneous networks know from previous generations., p. 17.

G

General Packet Radio Service (GPRS) A 2.5G technology for mobile communication, p. 5.

Global System for Mobile Communication (GSM) The 2G standard for mobile communication that is most widely deployed in Europe, p. 2.

H

High Speed Circuit Switched Data (HSCSD) A 2.5G technology for mobile communications, p. 15.

I

Internet Engineering Task Force (IETF) The Internet Engineering Task Force (IETF) is a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet., p. 19.

P

Peer-to-Peer (P2P) Mode of network usage where communication is between equal partners and not between a client and a server., p. 16.

Physical Media Independence (PMI) A system which enables dynamic network reconfiguration using monitors., p. 26.

S

Simple Object Access Protocol (SOAP) XML-based protocol for simple remote method calls., p. 23.

T

Third Generation Partnership Project One (3GPP) 3G Partnership Project for Wide-band CDMA standards based on backward compatibility with GSM and IS-136, p. 15.

Third Generation Partnership Project Two (3GPP2) 3G Partnership Project for Wide-band cdma2000 standards based on backwards compatibility with IS-95(3GPP2)., p. 15.

U

Universal Mobile Telecommunications System (UMTS) A 3G cellular system., p. 5.

W

Wireless Local Area Networks (WLAN) Technology which enables anyone to setup a wireless network., p. 5.

Wireless World Research Forum (WWRF) A collaboration of academia and industry founded by Nokia, Siemens, Ericsson and Alcatel. Works on research leading towards 4G., p. 18.

Bibliography

[1] 3rd Generation Partnership Project. *Technical Specification Group Radio Access Network, Multiplexing and channel coding*, 3gpp ts 25.212, v3.4.0 edition, Sept. 2000.

[2] Snoeren. A.C. *A session-Based Architecture for Internet Mobility.* PhD thesis, Massachusetts Institute of Technology, Dec. 2002.

[3] G. Vigna A. Fuggetta, G.P. Picco. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[4] *BEA WebLogic.* http://e-docs.bea.com/platform/docs70/index.html.

[5] Andrew D. Birrel and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[6] Bluetooth. http://www.bluetooth.com.

[7] et al Brewer, Katz. A network architecture for heterogeneous mobile computing. *IEEE Personal Communications Magazine*, Oct. 1998.

[8] Goodman Cain. General packet radio service in gsm. *IEEE Communication Magazine*, pages 122–133, October 1997.

[9] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *POLICY*, pages 18–38, 2001.

[10] Stephen E. Deering and Robert M. Hinden. *Internet Protocol version 6 (IPv6) specification.* Internet Engineering Task Force, December 1998.

[11] *DHCP - Dynamic Host Configuration Protocol.* http://www.dhcp.org/.

[12] D. Duchamp. Issues in wireless mobile computing. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 2–10, Key Biscayne, FL, USA, April 1992.

[13] *Java 2 Enterprise Edition 1.4 Documentation.* http://java.sun.com/j2ee/1.4/docs/.

[14] Ericsson. http://www.ericsson.com/technology/EDGE.shtml.

[15] J. Haartsen et al. Bluetooth: Vision, goals, and architecture. In *ACM Mobile Computing and Communications Review*, pages 2(4): 38–47, Oct. 1998.

[16] Subir Das et al. Telemip: Telecommunication-enhanced mobile ip architecture for fast intradomain mobility. *IEEE Personal Communications*, pages 7(4):50–58, August 2000.

[17] Wireless World Research Forum. Book of visions 2001. December 2001.

[18] I. Framework and W. Group. Policy core information model specification rfc, 2001.

[19] Y. Yokore F. Teraoka and M. Tokoro. A network architecture providing host migration transparency. In *Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 209–220, Zürich, Switzerland, September 1991.

[20] Gnutella. http://www.gnutella.com.

[21] Global system for mobile communication. http://www.gsm.org.

[22] S. Gupta and A.L.N. Reddy. A client oriented, ip level redirection mechanism. In *Proc. IEEE Infocom*, pages 1461–1469, New York, March 1999.

[23] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. Rfc 2543: Sip: Session initiation protocol, 1999.

[24] *The HAVi Specification*, beta 1.0 edition, November 1998.

[25] IEEE Computer Society LAN MAN Standards Committee. *Wireless LAN medium access control and physical layer specifications, IEEE Standard 802.11-1997.*

[26] IETF - RFC 2002, http://www.ietf.org/internet-drafts/draft-ietf-mobileip-rfc2002-bis-08.txt. *Mobile IP.*

[27] IETF, http://www.ietf.org/rfc/rfc0791.txt. *Internet Protocol.*

[28] The Internet Engineering Task Force, http://www.ietf.org. *IETF.*

[29] *ISO/IEC 8326:1994, Session Layer Definition (OSI).*

[30] The JINI Community, http://www.jini.org. *Jini Network Technology.*

[31] D. Johnson, C. Perkins, R. in, m IP, and I. Draft. Internet engineering task force, 1996.

[32] J. Binkley J. Inouye and J. Walpole. Dynamic network reconfiguration for mobile computers. In *3rd ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '97)*, Budapest, Hungary, 1997.

[33] Randy H. Katz and Eric A. Brewer. The case for wireless overlay networks. In Tomasz Imielinski and Henry F. Korth, editors, *Mobile Computing*, pages 621–650. Kluwer Academic Publishers, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1996.

[34] R.H. Katz. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1(1):pp. 6–17, 1994.

[35] Minkyong Kim and Brian Noble. Mobile network estimation. In *ACM Conferance on Mobile Computing and Networking*, pages 298–309, 2001.

[36] David A. Maltz and Pravin Bhagwat. MSOCKS: An architecture for transport layer mobility. In *INFOCOM (3)*, pages 1037–1045, 1998.

[37] Network Working Group. *Domain Name Service RFC 1034 among others*, November 1987.

[38] Object Management Group(OMG), http://www.omg.org/corba/. *Common Object Request Broker Architecture (CORBA).*

[39] *Oracle Application Server.* http://www.oracle.com/ip/deploy/ias/.

[40] Charles E. Perkins. Mobile networking through mobile ip. *IEEE Internet Computing*, pages 58–69, January/February 1998.

[41] Jonathan Michael Salz. Tesla: A transparent, extensible session-layer framework for end-to-end network services.

[42] M. Sloman. Policy driven management for distributed systems, 1994.

[43] Snoeren and Balakrishnan. An end-to-end approach to host mobility. In *6th ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '00)*, Boston, MA, USA, 2000.

[44] Simple object access protocol. http://www.w3.org/TR/SOAP/.

[45] M. Stevens and P. Framework. et draft, draft-ietf-policy-framework, 1999.

[46] Sun Microsystems, Inc, http://java.sun.com/jdk/. *JavaTM 2 Platform.*

[47] Sun Microsystems, http://java.sun.com/products/jdk/rmi/. *Java Remote Method Invocation.*

[48] Sun Microsystems, http://java.sun.com/j2se/1.3/docs/guide/reflection/. *Reflection in Java.*

[49] Sun Microsystems, http://java.sun.com/j2se/1.4/docs/guide/serialization/. *Serialization in Java.*

[50] Universal plug and play (upnp). http://www.upnp.org.

[51] Helen J. Wang. Policy-enabled handoffs across heterogeneous wireless networks. Technical Report CSD-98-1027, Computer Science Division, Berkeley, 23, 1998.

[52] Wireless world research forum. http://www.wireless-world-research.org/.

[53] X. Zhao, C. Castelluccia, and M. Baker. Flexible network support for mobility.