

Service Invocation and Roaming in Pervasive-Computing Environments

by

Alvin Yung Chian Chin

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Masters of Applied Science

in

Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2004

©Alvin Yung Chian Chin 2004

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Emerging mobile-communication protocols aim to disassociate computing platforms from the location and the medium the user is accessing services. Moreover, a user should be able to access his/her personalized services any time, anywhere, on any type of device and with any type of wireless technology in a non-obtrusive environment. The complexities and problems associated with a mobile-wireless environment should be insulated from the user. This new type of computing is what we call pervasive computing.

In this thesis, a framework for accessing location-aware services (service invocation) and the protocols involved for a user to be able to access services across different locations and networks (service roaming) are presented. Service advertisement, service registration, and service discovery need to be performed before service invocation. The proposed framework uses Web services to achieve these objectives. Specifically, Web Services Description Language (WSDL) and Universal Description, Discovery and Integration (UDDI) are used for service advertisement and registration, UDDI is used for service discovery, and Simple Object Access Protocol (SOAP) is used for service execution. In order to determine the particular services that a user is accessing and maintain information related to the task that the user is performing, a stateful protocol is required. In this context, in order to implement service-roaming capabilities the framework uses a proxy-based architecture. By doing so, the client, location, and transaction details can be kept and processed while the user roams in different locations, and accesses different services.

Acknowledgements

There are so many people out there that I want to acknowledge. First of all, I want to thank the Lord God Almighty, for His wisdom and telling me to do my Masters degree. It was great planning that while I was working full time, I decided to apply for the Masters program at Waterloo. It was an even greater wonder that during October 2001 when the economy and job market were on the decline, and the dot-com boom had bust, the company I worked for folded. Wondering whether to find another job or pursue my Masters (and having already been accepted into the program), I decided to go for my Masters. And thus that's where this had all begun.

Second of all, I must give my sincere thanks to my supervisor, Dr. Kostas Kontogiannis for accepting me into the Software Engineering (SWEN) group and allowing me to start my Masters one term ahead of my proposed start (and allowing me from not having to worry about what to do during the 6 months lapse had I started later!). Also, thank you Kostas for providing me the financial support with the research funding, the guidance, the advice (both academic and personal) and the opportunity for me to choose and work on the topic of my Masters thesis. Thank you for giving me the academic and financial support to allow me to do my NSERC Industrial Post-Graduate Scholarship at Bell exCITE!, a group within Bell Canada and part of the Bell University Labs. Also, thanks for giving me the opportunity to present and publicize my work at various conferences. Despite the heavy workload and hectic schedule between school and going to conferences and meeting with industry partners, I am indebted for your patience, your willingness to coach me during these two years, and being there whenever I needed your counsel.

Third of all, I need to give thanks to Bell University Labs (BUL), ICR and the Bell

exCITE! team and especially Angela van Damme and Konstantine Liris for providing me a "home" to work as part of my NSERC IPS. Thanks for providing a great environment with great machines and a friendly atmosphere (with the couches!). The whiteboards were of very much use, especially that was where I did my best thinking when designing the architecture and protocols for my thesis. I would like to acknowledge Mike Milton for initiating the arrangements to work at Bell exCITE! and providing opportunities to allow me to participate in BUL meetings between Waterloo academic researchers and Bell Canada. Sincere thanks to Jean Webster and Vic DiCiccio of ICR for providing me the opportunity to participate in BUL related events.

I cannot forget to thank my family, especially my parents who have supported my decision to return back to school. Even though there have been hard times, my parents have been there for me and instilling the values that have made me the person I am today.

To my thesis reviewers (Professors Paul Ward and Jay Black), thank you for taking time out of your busy schedules to read and critique my thesis. Your suggestions and feedback will definitely help me in my studies as a PhD student. Also, thank you to the Electrical and Computer Engineering Department at the University of Waterloo for providing me academic and administrative support.

Lastly, but not least, thanks to my classmates at the SWEN lab. You have provided help for me whenever I have asked whether it was personal, technical or academic. It was great working with you at the lab (even though sometimes I can be a bit annoying and a pain!). Thanks for putting up with me and providing me the comic relief whenever I had Java coding bugs or have felt down for not being able to

solve a particular thesis-related problem (yes, you know who you are!).

I am sure there are others out there that I have forgotten. But the entire collection of all these people have contributed to this thesis that you now read in your hands.

Doing this Masters thesis has been a tremendous learning experience. I have learned so many new technologies, different research issues and problems, have gotten industry and public exposure, and have witnessed the entire software development lifecycle. Starting from the grains of an idea, gathering requirements, conducting research and literature surveys, proposing a design, implementing the design, and documenting the entire work, it has been a very rewarding journey. So rewarding that I have decided to pursue a PhD in pervasive computing.

Thank you all!

Contents

1	Introduction	1
1.1	Problem Description	2
1.2	Major Thesis Contributions	3
1.3	Case Study	4
1.3.1	Checking Conference Agenda	4
1.3.2	Instant Messaging and Checking the Weather	5
1.4	Thesis Organization	6
2	Background and Related Work	7
2.1	Pervasive Computing	8
2.2	Service Invocation	9
2.2.1	Web Services for Service Invocation	10
2.2.2	Service Invocation Used in the Thesis	12
2.3	Service Roaming	12
2.3.1	Physical-layer roaming	13
2.3.2	Network-layer roaming	13
2.3.3	Session-layer roaming	14

2.3.4	Application-layer roaming	14
2.3.5	Service Roaming Used in the Thesis	17
2.4	Location Awareness	17
2.4.1	Determining location	18
2.4.2	Location-based Services Research Issues	19
2.4.3	Location-based-Services Architectures	21
2.5	Summary and Conclusion	23
3	System Architecture and Design	25
3.1	System Architecture for Service Roaming	26
3.1.1	Layered Architecture	26
3.2	m-Roam Architecture	29
3.2.1	Proxy-based Architecture for Service Roaming	30
3.2.2	Components of the m-Roam architecture framework	32
3.2.3	Service-Invocation Framework	38
3.2.4	Distributed Directory of Services	40
3.3	Detailed Design of m-Roam	40
3.3.1	Session	40
3.3.2	Proxy	42
3.4	Complete Architecture for Service Roaming	53
3.5	Summary and Conclusion	56
4	Messaging Protocols and Use Cases	57
4.1	No roaming and no disconnection	58
4.1.1	Login	58

4.1.2	Service access	59
4.2	Disconnection and Crash Recovery	63
4.2.1	Client disconnection recovery	64
4.2.2	Client crash recovery	66
4.3	Roaming	66
4.3.1	Client-Migration Protocol	68
4.3.2	Roaming use cases	70
4.4	Summary and Conclusion	78
5	Experiments	80
5.1	Experimental Setup	81
5.2	Time Performance and Space Requirements	83
5.2.1	Time Experiments	83
5.2.2	Space Experiments	86
5.3	Summary and Conclusion	88
6	Conclusions	89
6.1	Thesis Overview and Findings	89
6.2	Future Work	91
A	Case Study Screen Shots	92
A.1	Checking Conference Agenda	93
A.2	Instant Messaging and Checking the Weather	96
	Bibliography	102

List of Figures

2.1	Service-Oriented Architecture	10
3.1	System architecture for service roaming	26
3.2	Architecture of m-Roam	32
3.3	Design architecture of front-end content	33
3.4	Design architecture of Service-Roaming Framework	36
3.5	Session finite state machine	41
3.6	Proxy-message finite state machine	44
3.7	Proxy message queues	45
3.8	Proxy operation for no disconnections or crashes	46
3.9	Proxy operation for client reconnection	47
3.10	Proxy operation for client request after reconnection	47
3.11	Proxy instance finite state machine	50
3.12	System architecture for service roaming	54
4.1	Message sequence diagram for service access	60
4.2	Message sequence diagram for client disconnection recovery	65
4.3	Message sequence diagram for client crash recovery	67

4.4	Message sequence diagram for client migration protocol	69
4.5	Message sequence diagram for roaming	71
4.6	Message sequence diagram for roaming - query type	73
4.7	Message sequence diagram for roaming, location-sensitive query . . .	76
4.8	Message sequence diagram for roaming, location-independent query .	78
5.1	Time to access service for last user without UDDI	84
5.2	Time to access service for last user with UDDI	85
5.3	Space required for N number of users	87
A.1	Logging into conference web site	93
A.2	Finding all services in Second Cup	93
A.3	List of services in Second Cup	94
A.4	Selecting the local news service	94
A.5	Accessing the conference agenda	95
A.6	Accessing previous disconnected session	95
A.7	Continuing existing conference agenda	96
A.8	Cynthia logging into conference site from Starbucks	96
A.9	Find all services in Starbucks	97
A.10	List of services available in Starbucks	97
A.11	Selecting the instant messaging service	98
A.12	Starting a chat with Jennifer	98
A.13	Chat response from Jennifer	99
A.14	Selecting the weather service	99
A.15	Getting the weather for Dartmouth	100

A.16 Cynthia logging into the conference site from Second Cup	100
A.17 Existing session open from Starbucks for Cynthia	101
A.18 Updated weather from the system for Halifax after roaming	101

List of Tables

5.1	Experimental Setup Environment	82
-----	--	----

Chapter 1

Introduction

Computing has evolved over the past couple of decades. First, there was the mainframe-computing era. Then the personal-computing era, client-server era, distributed-computing era, and mobile-computing era. Today there are many mobile devices such as PDAs and cell phones yet there does not exist a uniform protocol to enable these types of devices to access services on an enterprise network, such as an e-mail or a sales-force application. Nevertheless, there are special mobile application servers which allow mobile devices to hook into an enterprise environment to access corporate resources like e-mail and RDBMSes. However, these application servers are proprietary and not easily customizable to an enterprise's requirements. Users have to install special software on each device and configure the mobile application server to recognize and communicate with that device. Users should be able to easily access a service within an enterprise or have services delivered to them when entering a particular area like a meeting room.

Even though many definitions exist, for this thesis, the ability to obtain services

and information from an environment anytime, anywhere we call pervasive computing. However, pervasive computing itself is complex, especially when many devices and configurations are involved. If users wish to access their own personalized services, they should be able to do so on any type of device and with any type of wireless technology. In mobile computing, wireless networks are unreliable and operate slower in limited bandwidth than wired LANs. As well, devices and users are not stationary. They are highly mobile. Furthermore, devices and networks are prone to failure and disconnection when migrating to different networks or different protocols. Therefore existing approaches in the distributed-computing arena that operate in a wired environment cannot be applicable in the pervasive-computing environment. The complexities and problems within a mobile wireless environment should be insulated from the user. In addition, the user should also be able to access services in a location and continue to use those equivalent services without interruption when moving to another location or different wireless network, provided the service is accessible in this new location. The computing environment must adapt to user mobility, insulate the complexity from the user, and blend and disappear into the human environment, thus becoming ubiquitous.

1.1 Problem Description

From the above, the problem with pervasive computing is that in order to perform a particular task suited towards the user, information needs to be obtained from the underlying network. The root of this information is provided by services. Therefore, a service-driven paradigm is what is needed and involves having to perform service

invocation and roaming.

The objective of this thesis is to develop a framework of this service paradigm that facilitates a user to perform a particular task to get access to services in a seamless fashion, tailored to their preferences and to the location that they are in. Services should be accessible wherever a user is, and users should be able to execute their services without exposing the failures inherent in wireless networks. The cellular phone has now become a pervasive device due to its high-mobility factor because cellular users can make calls and continue their calls while moving to different cellular base stations. However, data devices have not reached that level. There does not exist in practice an integrated architecture that allows a user to roam for services at the application level, due to the lack of a business model for deployment.

1.2 Major Thesis Contributions

This thesis makes the following major contributions:

- 1) service roaming is achieved at a higher level independent of the lower levels;
- 2) an architecture is designed and a framework is developed for integrating enterprise services with the addition of service invocation and roaming in a mobile wireless environment; and
- 3) service roaming can be deployed in an enterprise environment relatively easily with existing technologies.

1.3 Case Study

A case study is used to illustrate an application of our service invocation and roaming architecture and framework. The case study is in the context of a conference where services are provided to users with mobile devices in wireless hotspot locations. This allows conference attendees to check out special conference services while during the conference or away from it at their own convenience. The specific conference that we use is the Precarn 2003 conference from Halifax, Nova Scotia [25]. Two locations are selected for this case study, a Second Cup coffee shop in Halifax that has a wireless 802.11b hotspot (where the conference is), and a Starbucks coffee shop in Dartmouth with Bluetooth wireless connectivity. Two scenarios are demonstrated in this case study. The first involves checking the conference agenda, and the second involves using an instant messaging service and checking the weather. The screen shots used to demonstrate the case study for the scenarios can be found in Appendix A.

1.3.1 Checking Conference Agenda

Alvin is chief technology officer of a company. There is a break in-between sessions at the Precarn conference, and he decides to go outside the conference venue to have a coffee to drink. Nearby there is a Second Cup coffee shop, which advertises that it is an 802.11b hotspot. So, he goes there, orders a coffee and takes out his Palm m505 handheld. Alvin wishes to get the updated conference agenda so he knows the next session that he wants to attend after the break. A connection is made to the conference web site using an 802.11b module attached to the Palm m505. Alvin then gets interrupted by his cell phone, so he goes outside Second

Cup to receive the call, taking his Palm m505 with him. Upon his return back, the Palm m505 has lost the 802.11b wireless connection and along with it the updated conference agenda. However, he re-establishes connection and is able to retrieve back his updated conference agenda. He decides to attend the session on "Robotics and Artificial Intelligence". This scenario is illustrated with the screen shots from his Palm in Section A.1 of Appendix A.

1.3.2 Instant Messaging and Checking the Weather

Cynthia is a marketing director attending the conference. She wants to organize a night on the town in Dartmouth with a couple of friends from the conference. On her way out from the hotel, she goes to the Starbucks coffee shop in Dartmouth to grab a coffee. She discovers that there is a trial of Bluetooth at Starbucks, and since she has a Bluetooth module for her Palm Vx, decides to try it out. She discovers there is an instant messaging service and Jennifer is online at the conference. Cynthia tells her that she is organizing a night on the town in Dartmouth, and asks Jennifer whether she wants to go. Jennifer suggests Halifax instead. Jennifer tells Cynthia that they will discuss further when she arrives at the conference. Cynthia checks the weather in Dartmouth to see whether she will need to grab a coat when she returns. Cynthia then arrives at the conference, and meets with Jennifer for a session break at Second Cup. Knowing that the planned event is now going to take place in Halifax instead, she checks the weather here to figure out what kind of clothes to wear and they both plan their night on the town. This scenario is illustrated with the screen shots from her Palm in Section A.2 of Appendix A.

1.4 Thesis Organization

The thesis is organized as follows. Chapter 2 presents the background and related work required for the understanding of the thesis. Chapter 3 discusses and illustrates the design and architecture of the proposed system which was implemented in a prototype for our conference case study. Chapter 4 presents the messaging protocols and use cases for the architecture related to the two scenarios presented in the case study. Experimental results are presented in Chapter 5. Chapter 6 draws conclusions and discusses future work. Finally, Appendix A provides the screen shots of the user interface for our case study that are displayed to the user, resulting from the underlying architecture and protocols presented in chapters 3 and 4.

Chapter 2

Background and Related Work

The proliferation of mobile devices and their penetration into every day life in recent years has generated new computing paradigms in this emerging and fast growing area of business. In this chapter we provide the background information in this area and discuss the issues pertaining to the invocation and mobility of pervasive services. By surveying related approaches that address these issues we focus on related work in four major problems: how to invoke services, how to address mobility issues such as disconnections and crashes for services in an unreliable wireless network, how to seamlessly access services when migrating to another location and to a different network, and how to enable location awareness of services and resources.

The chapter is broken down into six sections. Section 2.1 describes the vision and goal of pervasive computing, of which this thesis forms a part. Section 2.2 explains about invoking services in a pervasive-computing environment. Service roaming and mobility is described in Section 2.3. Section 2.4 deals with location awareness which is an important context in pervasive computing. Finally, Section 2.5 provides a con-

clusion and summary to the chapter.

2.1 Pervasive Computing

According to Mark Weiser, pervasive computing is “a new way of thinking about computers, one that takes into account the human world and allows the computers themselves to vanish into the background ” [50]. Users are presented with information from services which can be accessed anytime, anywhere and on any device tailored to their environment. The computers and technology are embedded into the human environment such that they blend into the background and appear to become invisible to the user. In this way computing adapts to the user’s requirements and needs, rather than the current way which is to have users conform to the computers.

Many of the early pervasive-computing scenarios were considered futuristic, something that would come out of science-fiction and from academic research. However, the infrastructure, hardware, network, and software to enable pervasive computing are available today. In industry, the vision of pervasive computing articulated by Weiser is quickly becoming a commercial and viable computing initiative. IBM has a division that deals solely with pervasive computing and has their Websphere Everywhere [24] family of software based on it [22]. HP Labs has a technology called CoolTown which is viewed as an enabler for pervasive-computing applications [30]. Pervasive computing is considered no more as a “cool” and farfetched vision, but rather it is now invading the enterprise [44].

However, the challenges involve integrating all the hardware, software, and networks into a system that can be utilized by common folk in a non-obtrusive and

invisible environment, as explained by Satyanarayanan [41]. According to Weiser, “The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it” [50]. More research needs to be done in order to solve the above challenges.

This thesis concentrates on solving the problem of how to execute services in an enterprise network in response to a particular task or query initiated by the user, and compensate for lost connectivity and interruption of that task as a result of disconnection or migration.

2.2 Service Invocation

Users perform tasks in their environment and these tasks require access to services to carry out the task. Service invocation refers to executing operations on a service in order to perform a particular task. We define a service as an entity that provides a group of related functions with a specific purpose and is exposed as a well-defined interface. Before services can be invoked, they need to be described to other applications or services, advertised on the network, and discovered by other applications or services.

A service-oriented architecture (SOA) is used for describing the above and is illustrated in Figure 2.1 [5].

From Figure 2.1, the *service provider* is a network entity which provides the service and its capabilities. The service provider creates a service description document containing features of the service, a set of inputs, methods and operations, how to invoke the service, and a set of outputs. This process is known as *service description*.

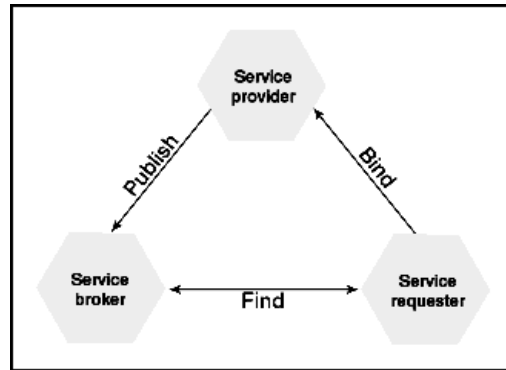


Figure 2.1: Service-Oriented Architecture

The service description document is published to the *service broker* which is a registry or directory during *service registration*. A service broker can be centralized (central registry) or can be distributed (with multiple registries on the network). A service broker functions like a Yellow Pages directory. A client looks for a particular service in a location by querying the service broker. We call this *service localization*. The client that is finding a service to satisfy specific requirements, is called a *service requester*. Once the service requester locates the service, then it establishes a binding to the service provider and executes operations on that service. This is called *service invocation*. Communication then occurs directly between the service requester and the service. Service invocation involves having to execute procedures or methods on the service, either remotely or locally by retrieving a copy of the service interface and executing the method as if it were a local method.

2.2.1 Web Services for Service Invocation

There are many service discovery protocols that implement SOA like Jini ([28], [49]), UPnP [48], Salutation [40] and Bluetooth SDP [35]. Richard [39] explains them in

more detail. The problem with these service discovery protocols is that they are proprietary and disparate, they do not interoperate well with each other, and they are not commonly used. Additional software protocol stacks need to be installed on clients, and this is not desirable. As a result we decide to use web services in the thesis for the following reasons. First, it allows services to talk to other services in a seamless fashion through HTTP (Hyper-Text Transfer Protocol) and XML (Extensible Markup Language). Since the web has become a ubiquitous communications medium with HTTP as the standard web protocol, it makes sense to leverage on existing proven technology. XML (Extensible Markup Language) is the data protocol used for encapsulating service-related information, and it is agnostic of programming language, implementation and device. Second, web services separate content and data from the display allowing services to be delivered to any mobile device. Third, since mobile devices have user interfaces that have different form factors, screen size, support of colors and so on, using web services permits the display of the services to be tailored to the specific device. This is done using transcoding technologies that take the content and format it according to the device's constraints [18].

Web services have three protocols that do service description, service registration and discovery, and service invocation: WSDL (Web Services Description Language), UDDI (Universal Description Discovery and Integration) and SOAP (Simple Object Access Protocol) . Existing enterprise services can be wrapped into web services, which permit them to be accessed from any HTTP client. Services are described using WSDL, published by the service provider to the service broker using UDDI, discovered by the service requester using UDDI, and invoked by the service requester using SOAP. Curbera explains the underlying details of WSDL, UDDI and SOAP [9].

Web services can then be extended to semantically describe and search for services, to improve on the accuracy of existing service discovery protocols by adding ontologies. An ontology attempts to capture and represent as much knowledge as required about services using relationships. This is performed by developing semantic markup that sits at the application level above WSDL. This is the premise for semantic web services [14] and DAML-S [7] is a technology to accomplish this.

2.2.2 Service Invocation Used in the Thesis

The thesis uses web services for service invocation for the reasons mentioned above. Specifically, we base our service invocation on Cheung's Masters thesis, with support for the ECA (Event Condition Action) paradigm [5] as described in the next chapter to execute the services in order to perform a particular task.

2.3 Service Roaming

Roaming is the ability to move from one coverage area to another without any interruption in service or loss in connectivity. It is analogous to cellular roaming, in which a cell phone user can originate a call from Toronto and continue the conversation in Waterloo, without the call being dropped. The concept of roaming is certainly not new, and has been implemented in different ways, and at different layers of the classic network OSI reference model [47]. Service roaming means being able to start a service in one area and continue to use that service in another area.

There are four methods to accomplish roaming: *physical-layer roaming*, *network-layer roaming*, *session-layer roaming* and *application-layer roaming*.

2.3.1 Physical-layer roaming

Physical-layer roaming involves measuring the strength of the radio signal from the access point or base station (in cellular terminology) to the device. If the signal strength from another access point is stronger then, with high likelihood, the device is moving towards that location. The transfer from one access point to another is known as handoff.

2.3.2 Network-layer roaming

Network-layer roaming occurs at the network layer using a signalling protocol and the addition of network components. It uses the concepts of a *home network* to indicate the network from which the mobile device is associated with and originates from, and a *foreign network* to indicate the network the mobile device is in when it is away from the home network. This is similar to roaming from the telecom cellular network [10]. A Wireless Internetworking Gateway (WING) [13] can be used to perform handover between one network to another. For IP networks, *MobileIP* is a network-layer roaming protocol designed to maintain a seamless network connection. It introduces a *home agent* and a *foreign agent* in a home and foreign network respectively, to route and forward IP packets to the mobile device [37]. Base stations with beacons act as foreign agents [45] to detect the roaming of mobile devices. Extensions to MobileIP ([16], [51]) have been proposed to enhance roaming. Current developments and trends for handover design in IP wireless networks include Reverse Address Translation (RAT), multicast-based handover, HAWAII, and Cellular IP [17]. A MobileIP-like protocol by Sung [46] uses handover agents for the client and

for the server, and uses beacons to detect the presence of the mobile device within the vicinity of the access point. WiFi Bridge [8] is a network-layer roaming framework for wireless LAN (local area network) that extends Cellular IP and uses GPRS as a temporary relay between disjunct wireless LAN domains.

2.3.3 Session-layer roaming

Above the network layer is the session layer upon which roaming can be achieved through the introduction of a session. An example of session-layer roaming is *Session Initiation Protocol* (SIP). A session is established for real-time communications [42] and SIP provides the protocol for roaming of data, in a similar fashion to the protocol used for the roaming of voice calls in telecommunications voice networks. A session identifier is used to indicate that a mobile user wishes to continue previous data communication after the user moves to a new location and obtains a new temporary address [17].

2.3.4 Application-layer roaming

Application-layer roaming uses middleware and application-level signalling to detect that the client is roaming from one location to another. Roaming at the application layer involves using the underlying lower layers in the network OSI model as is. Some research work has been conducted in this area, but not to the vast extent like network-layer and physical-layer roaming. For example, a *directory* can be used to store user information so that when the user logs in and is authenticated, the system can determine if that user has moved. Corbi et al. [6] use this to design a roaming

model based on the integration of a RADIUS (Remote Authentication Dial In User Service) server (for authorization and authentication) with an LDAP (Lightweight Directory Access Protocol) directory which contains the users' profiles.

2.3.4.1 Proxy

A *proxy* can be used which acts as the intermediary to hide migrations and disconnections from the servers. To compensate for lost connectivity in wireless environments in which clients can be frequently disconnected or can even crash, existing recovery techniques for distributed systems can be used. A recovery proxy for wireless applications, devised by Bin Yao and W. Kent Fuchs [53], uses the proxy to intercept and cache client requests and server responses. When the client reconnects, it is able to retrieve the previous client requests or server responses from the proxy. An extension to Yao and Fuchs' work is a distributed proxy server system [11]. Here, the proxy concept is extended into a framework that supports geographically dispersed locations and client roaming, through the introduction of a distributed proxy server associated with a geographic location served by an access point. The proxy server performs handover, and caches the client requests and server responses to recover from network disconnections.

An extension and enhancement to the proxy-server system is a *shadow proxy*, as implemented in the SOMA middleware platform [3]. The shadow proxy represents the mobile device and keeps track of its state in a user session. The SOMA platform introduces additional middleware components (device-specific clients, portable device lookup service, and profile manager service) in order to accomplish service roaming.

2.3.4.2 Session

To address the state of the client which is compromised during migrations and disconnections, a *system-session abstraction* can be used. Here, a session data construct is employed which encapsulates all the information about the state of the user's interaction with the system. Network disconnections are handled as session continuations, and connection migrations are handled as session migrations along with a TCP migration protocol in the *Migrate* architecture [43].

When a user moves to another location and may use another different mobile device, the user needs to be able to continue from where he or she left off. *iMASH* [12] accomplishes this through the concept of application session handoff, similar to handoff in other layers mentioned in this section. This involves using application state to capture what a user is doing (data and services), and then transfer that over to continue that task and finish it on a possibly different target platform. The work uses proxies, content adaptation and client-awareness.

2.3.4.3 Web services

A framework for mobile web services [4] uses mobile agents that act on behalf of mobile users and are integrated with web services in a mobile agents platform. Service agents are used to process the user's request, find the service using UDDI, and invoke on the web service using SOAP. Location information is added to web services through the use of JAIN PARLAY APIs ([26], [36]) which obtain the location maintained by the mobile network operator.

2.3.5 Service Roaming Used in the Thesis

Physical-layer, network-layer and session-layer roaming all address roaming at different layers and uniquely solve the problems that encompass roaming in each layer. However, this does not solve the problem of service roaming. Service roaming works at the application layer, where users access services and data and can continue doing so using a different mobile device in a different location on a different network. Application layer roaming is agnostic of the type of wireless network compared to physical-layer roaming. It does not require the complexity of routing and home and foreign agents like at the network level with MobileIP, and it does not disrupt the existing network infrastructure because there is no need to modify existing network entities. Only middleware components need to be added which plug into the network. Therefore, application-layer roaming is the correct roaming approach. For the thesis, application-layer roaming with proxies, sessions and web services is used and is described in chapters 3 and 4.

2.4 Location Awareness

One of the important driving forces behind pervasive computing is the ability to obtain services tailored to the location of the user. Many services have to deal with location because it provides the bridge between human context and the physical environment. Thus in service invocation and roaming, location needs to be described and specified explicitly or implicitly before a client can access information. Second, once location has been determined, then research issues such as how the system determines location, how location can be addressed in a client query, and how location

can be added to services, need to be addressed in the design of a location-based-services infrastructure. Finally, a location-based services-architecture needs to be set in place to allow the users to query for services. Even though the thesis does not deal with extensive research into location-based services and we do not make contribution to this area, it uses it and therefore we explain our selection of solution in these research issues.

2.4.1 Determining location

In this thesis, location from the point of view of the mobile unit, is described as the surrounding area where the mobile unit is in, permeated by the access point. For a service, location is the area where the service resides, or where it is offered. The underlying location-based services-architecture needs to determine the location of the client and the location of services. In that way, customized services can be provided to the user (who is identified as the person using the mobile unit) based on knowledge of the user's location.

Since the thesis uses web services for service invocation, we use semantic location embedded in a URL for issuing a client query. Semantic location is a subset of symbolic location in which the location of a place is represented by a URI (Uniform Resource Identifier) which can include other attributes associated with that place including physical location [38], of which Cooltown is an example that illustrates this [30]. Symbolic location describes location space using logical, real world entities that humans can understand [31]. For services, we include a location attribute in the WSDL description of the web service, which is published to the UDDI registry. We do not concern ourselves with the determination of location because we assume that

location can easily be obtained by previous methods like GPS (Global Positioning System), triangulation or automatic identification mechanism [20]. Hence depending on the granularity of location, any of the previous methods can be employed to determine location without affecting our architecture and protocol.

2.4.2 Location-based Services Research Issues

There are various research issues that need to be considered when designing and implementing a location-based services (LBS) architecture. They are *query types*, *data placement*, *query scheduling*, and *data caching* [31]. The thesis does not make any contribution to these problems but merely selects a possible solution from those that exist.

2.4.2.1 Query types

Service localization is accomplished in the thesis through client queries. The thesis supports both location-dependent and location-independent queries by virtue of whether a location is indicated in the query. Local and non-local queries are supported through the indication of location in the query. Local queries are bound to a user's current location (for example, find the local weather), whereas non-local queries are bound to locations other than the user's current location (for example, find the weather in New York City). Location is indicated in a client query by embedding it as a parameter in an HTTP URL (eg. `http:// website?location=loc1`). Here, `loc1` is the location value to be matched upon for services in this query. This follows the DATA-MAN project approach [1]. Simple queries deal with matching functional attributes of the services with the queried values specified (for example, find the restaurant that

serves Chinese food). General queries handle complex conditions on the underlying data (for example, book the conference which involves making reservations for the airplane, hotel accommodations, and booking a taxi). Clients issue simple queries by specifying service attributes in the query, while general queries are supported by breaking down the query into multiple service queries that are invoked on the ECA framework [5].

2.4.2.2 Data placement

To support service localization, one needs to consider the placement of location-dependent data for the services in the LBS architecture. This issue is not so much a research issue than a design issue, however it battles down as to whether the data should be centralized or distributed. Centralized data placement means using a central database to store service data like a central UDDI repository for all services. Distributed data placement involves either associating the data with the service in the location where the service resides or placing this data in a remote repository but not centralized. The debate between centralized and distributed data placement is a data management and distributed systems problem which we do not address. Suffice to say, the thesis is not dependent on the placement of data for the service, whether it be centralized or distributed, because the framework that we create for service invocation in the next chapter can handle both.

2.4.2.3 Query scheduling

When performing service localization, the system needs to determine how to process the client query, called query scheduling. If the query is a local query dependent on

location, then when the user moves to another location, the results of the local query may become invalid because they return data that is relevant to the previous location. In addition, the services invoked from the local query may not reside or be valid to access in this new location. As a result, the local-query response must properly reflect the location change. This may involve having to reprocess the query for the client's current location. The thesis takes this approach by reissuing the previous query but changing the location attribute in the query URL to be this new location.

2.4.2.4 Data caching

Service roaming deals with seamlessly continuing the service from where a client has left off before the migration. Data caching is used in order to achieve this seamless transition. The client can cache the data obtained from the server (client-data caching), the server can cache the data from the client (server-data caching), or some combination of the two can cache. The problem with these approaches is that it requires an additional protocol and state for maintaining this and that the client (for client-data caching) or server (for server-data caching) be robust in the midst of disconnections and crashes. Another method is to perform proxy-data caching by having an intermediary proxy cache the data from both the client and server. This thesis uses proxy-data caching. In this case, the proxy hides disconnections and crashes from both the client and the server, as will be illustrated in the next chapter.

2.4.3 Location-based-Services Architectures

Finally from determining the location and the LBS research issues, the LBS architecture can be created which this thesis is based upon. Many LBS architectures exist in

the research literature but only a subset will be discussed here. One such architecture is AROUND [29] where symbolic location is used as the location context for service scope and includes the AROUND service for determining location context governed by an AROUND server, a contextualisation process that determines the appropriate location context for the current location of a mobile device, and a name service which is a resolver for location context names to specific AROUND servers. However, a significant disadvantage with AROUND is that relationships for location context need to be planned and created before hand. On the other hand, these relationships provide for an ontology that facilitates accurate query results due to the propagation of the query through those context relationships.

Another similar LBS architecture is the Rover system [15]. Location awareness is enabled through tracking the location of users, and detecting them by the Rover controllers. One advantage of Rover is this addition of the controller which performs the management of client requests, schedules them, and filters content according to current location. Location is determined based on certain properties of the wireless infrastructure. However doing this is a disadvantage because users usually express location in terms of symbolic or semantic meaning as opposed to physical location.

Yet another approach is to breakdown a location-based query into spatial data (data pertaining to location) and non-spatial data (data not pertaining to location) to create an architecture that exploits this separation [32]. Li's Masters thesis introduces geometry managers and kiosks which are used to process spatial and non-spatial data respectively, and clients make query requests to a resolver which separates the spatial and non-spatial constraints. The addition of geometry managers and kiosks which form a tree hierarchy, can become quite complex and significant planning is required

to organize and place them. The way that they are distributed and how they are related will affect the processing, scheduling, performance, and accuracy of the query. On the upside, this method can provide more accurate and relevant results because of the semantic relationships.

Our thesis is similar to the AROUND service in that location is determined in the system based on the controller that the client connects to. The controller governs the location and manages client requests just like the Rover controller. A location manager in our system maps location names to specific DPCs (Dispatcher Proxy Controller) which are explained in the next chapter, similar to the name service in AROUND. However, we do not create relationships for location context as is done with AROUND. The thesis differs from Rover in that we do not use physical location but rather symbolic location as achieved in AROUND. Our thesis differs from Li's approach in that we do not break down the location-based query into spatial and non-spatial data in order to avoid the complexity required to manage and process the query.

2.5 Summary and Conclusion

This chapter has presented the background and related research both in industry and academia, that relate to this thesis. A list of concepts and topics have been discussed, which by no means is complete. First, the concept and vision of pervasive computing were described and elaborated on. Second, web services were presented as the appropriate technology for performing service invocation as compared to others. Third, various approaches were discussed for performing service roaming, upon which we

decided to use application roaming with proxies and sessions. Finally, we discussed about location awareness and provided background for determining location, the research issues that need to be addressed, the architectures that exist for LBS, and explained the decisions to the above that the thesis used.

The next chapter will present a system architecture for performing location-aware service invocation and roaming.

Chapter 3

System Architecture and Design

This chapter presents the system architecture for performing service invocation and roaming in a pervasive-computing environment, whereas the next chapter will explain the protocols required in doing so. The system architecture is broken down into subsystems. Each subsystem is described in terms of its design, functionality, and relationships with other subsystems in the architecture.

The chapter is organized as follows. Section 3.1 presents the high-level system architecture for service invocation and roaming and applies it to the case study. Based from the background and related work in the last chapter and the system architecture, we develop m-Roam which is our contribution and framework for performing service roaming based on proxies and sessions, and is integrated with enterprise services. We describe its architecture in Section 3.2 and its design and framework in Section 3.3. The architecture and framework is applied in Section 3.4 to the two scenarios presented in Section 1.3 to demonstrate its functionality and operation in a service roaming scenario. Finally, the last section, Section 3.5 summarizes and concludes the

chapter.

3.1 System Architecture for Service Roaming

Common mobile applications use typical multi-tier architectures. The architecture of our system is divided into four tiers which is an extension of the classic three-tier model of presentation layer/tier, application or middle layer/tier, and data or back-end layer/tier. These four tiers are the presentation layer, content layer, application layer, and back-end data and service layer. The complete system architecture diagram is illustrated below, and is the basis for the case study.

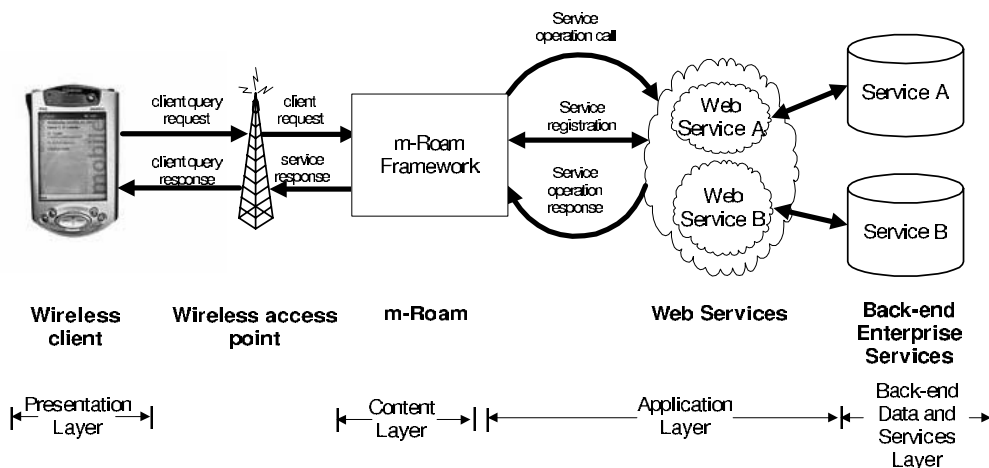


Figure 3.1: System architecture for service roaming

3.1.1 Layered Architecture

Each layer receives data and control information from the previous layer, performs the specific functions in its layer, and then provides the results to the next layer.

3.1.1.1 Presentation layer

The *presentation layer* is responsible for presenting information and services to the end-user in a user interface using a mobile device that is agnostic of the operating system (Windows CE, Palm OS, Linux, or Symbian OS) and wireless interface (Bluetooth, GPRS, 802.11, or CDMA for example). We call this a wireless client. The mobile device connects to the network through a wireless access point (indicated in Figure 3.1) that acts as a network bridge between the wireless network and the standard wired network, using standard network authentication protocols. The client sends client query requests looking for a particular service using a web browser and then invokes operations on that service by sending it to the content layer. For the case study, the mobile devices used in the presentation layer of the two scenarios from Section 1.3 were the Palm m505 with 802.11b module for an 802.11b network in the Second Cup coffee shop and the Palm Vx with an external Bluetooth module in the Starbucks coffee shop. In both scenarios, the user interface was an HTTP web browser and the figures of screen shots from the Palm in Appendix A illustrate this.

3.1.1.2 Content layer

The *content layer* provides the formatting of the service requests and responses and delivers them to the wireless client. It relays the service requests from the wireless client to the application layer. Service responses from the enterprise that reside in the back-end data and service layer (explained further down) are processed by the application layer, formatted appropriately for the wireless client and the task at hand, and then presented to the wireless client in the presentation layer. In our case study,

the content layer used a web-based architecture [54] (that the Precarn conference IT group would have already deployed) in which the content repository was a collection of static and dynamic HTML web pages served by and hosted on an HTTP web server, and the content that was displayed to the Palm handheld devices (m505 and Vx) was an HTTP web browser.

3.1.1.3 Application layer

The *application layer* processes the service requests and clients log into the system as in standard web applications. Once logged in, the application layer provides seamless roaming of services for clients, discovers available services pertaining to the client service request query and delivers them, insulates client disconnections, and performs crash recovery. From our case study, the first scenario which involved discovering the conference agenda service for Alvin at Second Cup (Figure A.5) allowed Alvin to continue using it after he got disconnected from his cell phone (Figure A.6 and Figure A.7, was achieved in the background at the application layer. For the second scenario, the application layer enabled Cynthia to discover the instant messaging service at Starbucks (Figure A.10 and Figure A.11) and chat with Jennifer (Figure A.12 and Figure A.13), as well as allow her to automatically find the weather in Halifax at Second Cup after moving from Starbucks in Dartmouth (figures A.15 to A.18. All of this was hosted on an application server typical in an EAI (Enterprise Application Integration) architecture.

3.1.1.4 Back-end data and services layer

The *back-end data and services layer* is where all the databases and services that access those databases reside. The back-end enterprise services are wrapped into interfaces that are exposed to the application layer. This is the termination point from where the data that is presented to the wireless client resides. In our case study, for our two scenarios with Alvin and Cynthia, the conference services such as the conference agenda service, the instant messaging service and the weather service resided in this layer. Services were described in WSDL and registered in a UDDI registry during service registration. When Alvin and Cynthia performed a query for a particular service or a list of services in the Starbucks or Second Cup location, this translated to a service discovery using UDDI on the UDDI registry that resided in this layer and was stored on the conference web site. Alvin and Cynthia were able to select the services and then use those services by invoking on them in this layer using SOAP, by having m-Roam make service operation calls directly on the corresponding web services.

3.2 m-Roam Architecture

From the system architecture in Figure 3.1, the central glue to enabling service invocation and roaming and which is our contribution, is m-Roam. The m-Roam architecture lies in between the content and the application layers. It is responsible for processing client requests originating from client queries, performs the control logic behind service invocation and roaming, hides the disconnections and crashes, and delivers service responses to wireless clients.

3.2.1 Proxy-based Architecture for Service Roaming

In m-Roam, a middleware recovery and roaming proxy is used as explained in Section 2.3.4.1. A proxy is assigned to each client in a particular location and tracks the service requests and responses for the duration that the client is in that location. A proxy server is used to send HTTP requests and retrieve HTTP responses from the back-end enterprise services. Our architecture makes use of the distributed-proxy-server system framework developed by Kim et. al. [11] and explained in Section 2.3.4.1, though it differs in other respects.

First, it differs in that the functionality of an HTTP proxy server in our approach is not extended to include transcoding, proxy handoff (migration), cache management, request message handling, and handling HTTP requests and HTTP responses. Rather, these extensions are distributed into three separate components: *Dispatcher Process*, *Dispatcher Proxy Unit* and *Proxy Server*. In this way, this leads to separation of concerns where each component is responsible for a particular function and leads to a clear and easy to deploy architecture. This is the third contribution of the thesis, which states that service roaming can be done relatively easy with existing technologies. Second, our design does not incorporate transcoding or distilled data. Instead, our thesis is concentrated on creating the logic behind service roaming and does not address distilled data which we assume is taken care of by existing transcoding methods. Therefore, it deals only with simple service requests where the results are processed at the back-end service. Third, the architecture does not use a client agent and does not require additional code to be installed on the client. The functionality of a client agent is handled in the Front-End Content subsystem of Section

3.2.2.1 using servlets. Again, the reason for this design choice is to facilitate easy deployment on any wireless client as long as an HTTP web browser is installed, thus supporting our third thesis contribution. Finally, handoff in the Client-Migration Protocol (Section 4.3.1) is done between proxies and not proxy servers. The concept of a proxy server by Kim et. al. [11] incorporates our concept of a proxy and proxy server together, whereas we split this functionality.

From Section 2.3.4.1, Bellavista et. al. [3] have implemented middleware similar to our design. However instead of having the proxy be responsible for maintaining the session, we decided to maintain the session and proxy separately, allowing for cleaner implementation, easier understanding, and separation of concerns. This is a design decision to support our second contribution of the thesis which is to develop and design a framework for integrating enterprise services with service invocation and roaming. We adopt a system-session abstraction. The session is responsible for keeping track of client interaction with various services during the duration that the client is performing tasks in the system. On the other hand, the proxy is responsible for keeping track of the proxy-server requests and responses to and from the backend services. During the session, a client can use multiple proxies in different locations.

From our case study, a proxy was associated each for Alvin and Cynthia to individually track their service requests to the system and maintained service responses from the conference agenda, instant messaging and weather web services. A session was created each for Alvin and Cynthia when they first accessed m-Roam from Second Cup and Starbucks respectively (Figure A.1 and Figure A.8). The session contained the flow control logic of the list of services and service operations invoked, not the actual service requests and responses.

When roaming, the user gets disconnected from the old location and reconnects to the new location with possibly updated features or service state. This is accomplished by performing a migration from the old proxy to the new proxy. From the case study, Cynthia experienced this when she went from Starbucks in Dartmouth to Second Cup in Halifax to check the weather. The details are encompassed in the Client-Migration Protocol and the client roaming use case, explained in sections 4.3.1 and 4.3.2. Our m-Roam architecture and framework is illustrated in Figure 3.2 and the following sections explain the subsystems in the architecture. The underlying details to these procedures embodied in the message protocols and use cases, are discussed in the next chapter.

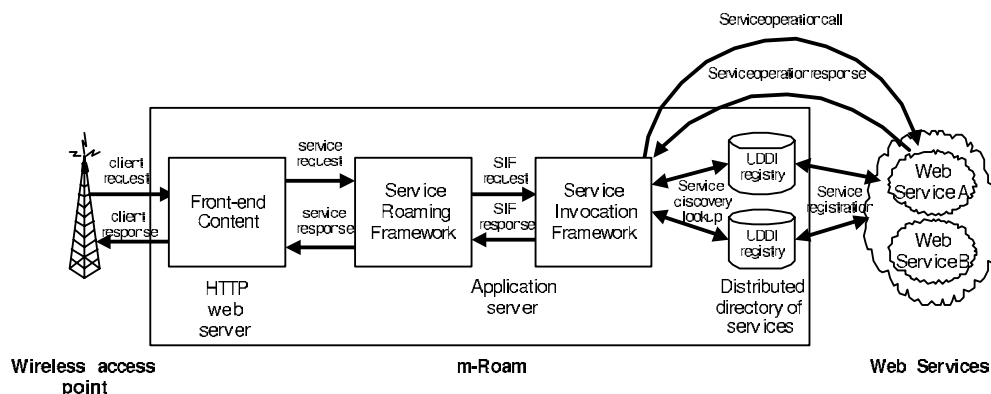


Figure 3.2: Architecture of m-Roam

3.2.2 Components of the m-Roam architecture framework

The m-Roam architecture framework consists of four subsystems: *Front-end Content*, *Service-Roaming Framework*, *Service-Invocation Framework*, and *Distributed Directory of Services*.

3.2.2.1 Front-end Content

The Front-end Content subsystem processed Alvin's and Cynthia's query requests from the scenarios, which could be to select a service (Alvin selected the conference agenda service while taking a break from the conference at Second Cup) or to invoke a service operation request (Cynthia chatted with Jennifer using the instant messaging service while she was at Starbucks). The architecture is illustrated in Figure 3.3. Appendix A shows the web pages that Alvin and Cynthia viewed from Second Cup and Starbucks.

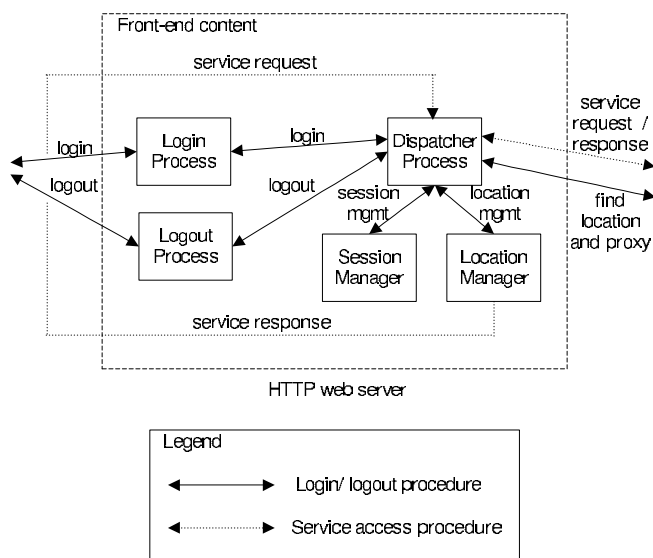


Figure 3.3: Design architecture of front-end content

The *Login Process* allowed Alvin and Cynthia to log into the system, and determined whether Alvin and Cynthia were new or existing clients so that either a new query was entered, or previous content before the interruption in service was delivered (figures A.1 and A.8). A session was created and maintained for Alvin and Cynthia by the *Session Manager*. Alvin and Cynthia's query requests were processed

by the *Dispatcher Process*, and then sent to the Service-Roaming Framework. Service responses were received, the responses were formatted, and presented back to the wireless clients' web browsers. Once Alvin and Cynthia were finished with m-Roam, the *Logout Process* handled the logout procedure. Our contributions to the Front-end Content are the Dispatcher Process, Session Manager and Location Manager.

Dispatcher Process. All requests that Alvin and Cynthia issued to the system were through HTTP GET URLs processed by the Dispatcher Process. The Dispatcher Process uses the Session Manager to manage sessions for the duration that all clients are logged in, and the Location Manager to process the location. Depending on the request, the Dispatcher Process redirects it to the Login Process for login requests, Logout Process for logout requests, or the Service Roaming Framework for client query requests. Service responses are returned back to the Dispatcher Process, formatted appropriately to the client, and delivered to the client.

Session Manager. Each scenario that we described for Alvin and Cynthia from Section 1.3.1 and Section 1.3.2, is encompassed in a session. The Session Manager creates new sessions, detects existing sessions, loads existing sessions to recover from disconnections or crashes, saves sessions, and removes sessions when logging out. We describe the details of a session in the detailed design of m-Roam in Section 3.3.

Location Manager. The Location Manager is used to determine the appropriate location server to service this client, based on the user's location during login and roaming. From our case study, it found out that Alvin was in Second Cup and Cynthia was in Starbucks. A location server called *Dispatcher Proxy Controller* was assigned

to Second Cup and Starbucks. The Location Manager stores the locations and URLs of their corresponding Dispatcher Proxy Controllers in a table which is used for the detection of roaming when the user moves to a different location.

3.2.2.2 Service-Roaming Framework

The Service-Roaming Framework is the key component and main contribution of the thesis. It is responsible for providing the seamless invocation and interaction of services without any interruptions. Interruptions can include client disconnections, network disconnections, location changes and migration to different networks, and client crashes. The Service-Roaming Framework was responsible for allowing Alvin to continue accessing the conference schedule agenda after he got interrupted with a cell phone call, and for Cynthia to continue accessing the weather service after she moved from Starbucks in Dartmouth to Second Cup in Halifax.

The Service-Roaming Framework consists of the logic needed to perform roaming, brokers client requests and server responses, and insulates client disconnections, crashes, and migrations from the back-end server. Our major contributions are the architecture and design, and the operation of the proxy using adaptation of various fault tolerance principles for mobile wireless devices that apply to service roaming. The design architecture for the Service-Roaming Framework is illustrated in Figure 3.4 and consists of the *Dispatcher Proxy Controller (DPC)*, *Dispatcher Proxy Unit (DPU)* and *Proxy Server*.

Dispatcher Proxy Controller (DPC). The function of the Dispatcher Proxy Controller is to localize the proxy allocated for the client in a particular location.

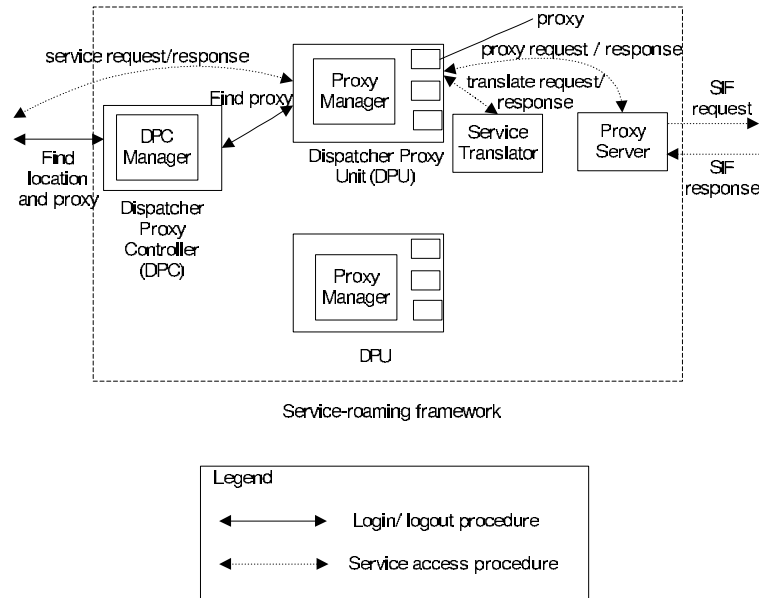


Figure 3.4: Design architecture of Service-Roaming Framework

Each location area (designated by a wireless access point) has an associated DPC. For our case study there were 2 DPCs, one each for serving Starbucks and Second Cup. The DPC can be seen as similar to that of an AROUND server in the AROUND service architecture [29] or a location server in the Rover system [15], as described in Section 2.4.3. It performs two functions: finding an available proxy (in the DPU which we explain later) for this client, and performing handover for a migration request if the client roams to a different location. For Alvin, the DPC associated with Second Cup allocated a DPU for the duration that Alvin was in Second Cup, and similarly for Cynthia when she was in Starbucks. When Cynthia moved to Second Cup, she was assigned a different DPU because she connected to a different DPC. The central unit of the DPC is the *DPC Manager*.

DPC Manager. Each DPC has a DPC Manager which assigns an available Dispatcher Proxy Unit (DPU) from the collection of DPUs to the user's session. Every user in a location is assigned a DPU. The DPC Manager is also involved in coordinating the handoff from this location to another, as explained in the Client Migration Protocol of the next chapter.

Dispatcher Proxy Unit (DPU). The Dispatcher Proxy Unit is the entity that contains a list of proxies that are maintained by a Proxy Manager. It is not required to have a DPU, but for the thesis by aggregating the collection of proxies into a DPU, we have added scalability, easy maintainability, and fault tolerance which are important features for software components in an enterprise environment. When Alvin first logged into Second Cup as in Figure A.1, the DPC here sent a "find proxy" request to the DPU in Second Cup to find an available proxy for Alvin. This was performed similarly for Cynthia in Figure A.8, except in Starbucks.

Proxy Manager. The Proxy Manager keeps track of all proxies in the DPU, as well as the proxy that is allocated to a particular client. When Alvin and Cynthia logged in, the Proxy Manager assigned a proxy for them. The Proxy Manager keeps track of client state for all proxies, sends client-query requests to the Proxy Server, sends server responses from the Proxy Server via the Dispatcher Process to the client, and recovers from disconnections and crashes.

Service Translator. The Service Translator is responsible for translating HTTP service requests sent by the Proxy Manager into HTTP requests compatible for input to the Service-Invocation Framework (called SIF requests in Figure 3.4), as well as

HTTP responses sent from the Service-Invocation Framework (called SIF responses in Figure 3.4) that are returned back to the Proxy Manager. We use existing XML stylesheets and parsers to make this transformation.

Proxy Server. The Proxy Server is a standard proxy server that creates sockets to establish HTTP proxy connections from the client to the Service-Invocation Framework and vice versa. A Proxy Server can be assigned to each DPU or multiple DPUs in a location. The assignment and distribution of proxy servers is a scalability and load balancing issue which is not discussed here.

3.2.3 Service-Invocation Framework

The Service-Invocation Framework is responsible for processing the client-service requests relayed from the Service-Roaming Framework, and discovering available services that satisfy that request based on authorization, location, quality of service and user profile. It is based on Cheung's Masters thesis [5] where services are offered to clients as web services, and can be composed using a workflow engine. Our framework is very much related to the Cheng's framework for mobile web services [4] in which web services are used and are invoked with SOAP. Our thesis is not dependent on Cheung's Masters thesis work for service invocation, and in fact service invocation can be performed directly on the web services. The reason we use his work in our Service-Invocation Framework is to support workflows. A workflow denotes the sequence of particular services that need to be performed in order to achieve the desired client task. The components of the Service-Invocation Framework are adapted from Cheung's thesis [5] and we do not discuss them any further here.

In our case study, each service (conference agenda, instant messaging and weather service) in the system was described using WSDL to indicate the service's characteristics and service operations. For example, the instant messaging service that Cynthia used had a service operation called "send message" which allowed her to send a chat message to Jennifer as in Figure A.12. The WSDL documents for the services were registered into the Distributed Directory of Services using the existing UDDI protocol for web services. Services in the conference were then deployed by publishing to the Distributed Directory of Services using the existing UDDI publish APIs, and we added the existing service name, service URL and location as attributes. When Alvin sent a query request to m-Roam to find all services in this location in Second Cup (Figure A.2), the Service-Invocation Framework issued a UDDI query to the Distributed Directory of Services to search for services that have their location attributes match the location of "Second Cup". In general, the client query is "find a service or list of services that provide features $A = a, B = b, C = c, \dots$ in location L ", where (A, B, C, \dots) are the features or attributes and (a, b, c, \dots) are the values of those attributes. The values of these features are matched to any services' attributes that have these values in the Distributed Directory of Services.

Once the web service is found from the service search query, operations on the web service can be executed. In our architecture, we performed service invocation using ECA rules and SOAP. ECA rules model workflows [5], and indicate what services (called actions) to execute based on events that have occurred according to satisfied conditions. The actions which invoke on the services are issued using SOAP requests, which essentially is remote process communication (RPC) over HTTP using XML.

3.2.4 Distributed Directory of Services

The Distributed Directory of Services is where the information regarding the services in the system resides for service localization. In our architecture, the Distributed Directory of Services consists of the UDDI registry and the web services. When a client query is performed that pertains to a service operation, the web service is executed directly which accesses its corresponding enterprise service. The Distributed Directory of Services can be thought of as a yellow pages catalog of back-end enterprise services. This provides the repository for the Service-Invocation Framework to discover and invoke services.

3.3 Detailed Design of m-Roam

In this section, we describe the two fundamental concepts which allow for service invocation and roaming. They are session and proxy which were introduced earlier in Sections 2.3.4.2 and 2.3.4.1.

3.3.1 Session

When interacting with m-Roam, the system needs to associate certain activities for a specific client so that when disconnections, crashes or migrations occur, the last activity performed can be continued without interruption. In our case study, m-Roam had to distinguish and keep track of service and user activity for Alvin and Cynthia. In m-Roam, a session is defined as user activity from the time that the user first logs onto the system until the time that the user successfully logs off. Users

can save existing sessions and then resume sessions at a future time to continue their saved work. This was not demonstrated in our case study. The detecting and loading of existing sessions is crucial to service roaming since this allows the client to continue the interrupted query. The session involves saving state information pertaining to the service, proxy associated with the client, and the location. Sessions are implemented using standard HTTP sessions, and follow a session finite state machine as depicted in Figure 3.5.

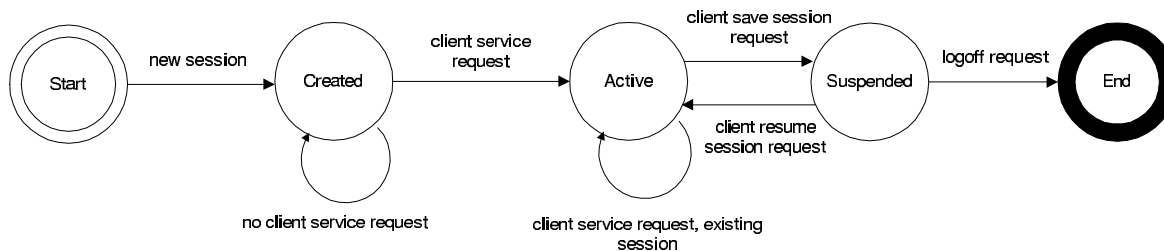


Figure 3.5: Session finite state machine

When a new user logs in and obtains a new session from the Session Manager, the session becomes *Created*. After, the session becomes *Active* when client service requests are issued. The client can save sessions to continue work later so the session becomes *Suspended*. When the client logs back in and selects to resume the session, the session returns to an *Active* state. Finally after the client has finished and logs off, then the session is removed and it has reached the *End* state. Our concept of a session encapsulates state therefore our contribution to the architecture and framework for service invocation and roaming, is the addition of stateful information to the stateless protocol for HTTP.

3.3.2 Proxy

The proxy forms the fundamental component in the Service-Roaming Framework. Every service request from the client and response from the service is intercepted, processed, and cached by the proxy. This is in contrast to caching at the client as explained in Section 2.4.2.4. We choose to cache requests and responses at the proxy because (i) this facilitates movement of the user to a different device while maintaining the same session and (ii) data is not lost because of a client crash. Our assumptions here is that the proxy is reliable and exists on the network where standard distributed systems and fault tolerance principles are implemented. The proxy's functions are two-fold. First, it acts as a cache manager as described by Kim et. al. [11], and second, it acts similarly as a shadow proxy [3] except that it keeps track of the user's service requests and responses. Every user that is logged into m-Roam gets assigned a proxy for the duration that the user is in this location. For example, Alvin and Cynthia were each allocated a proxy that cached each service request sent, and each service response received.

Our proxy design architecture is based on proxy-based recovery work by Bin Yao and W. Kent Fuchs [53] which we explained in Section 2.3.4.1. Here, the authors introduced a proxy finite state machine with messages and queues. Our design is a variation of the proxy finite state machine presented by Yao and Fuchs in that it eliminates an unneeded Buffered message. As well, Yao and Fuchs do not specify a finite state machine to track the proxy's state. Therefore, our contribution to the proxy for service roaming is our version of the proxy finite state machine and its protocols.

3.3.2.1 Proxy Finite State Machine

Every proxy has an associated finite state machine (FSM) [33] which details the logic for the operation of the proxy. The FSM is the fundamental basis for addressing disconnection and crash recovery, and roaming. The proxy design uses two FSMs which form our contribution: a *proxy-message FSM* and a *proxy-instance FSM*. Each client query request (or service request) and its associated service response is a message to the proxy, which progresses to various states in the proxy message FSM. Queues are maintained for the client and server that store the requests and responses. We use the term server to denote any remote service that is offered in the system for clients to use. Along with the queues, each proxy also maintains a state of service execution for the client which is encompassed in the proxy instance FSM. It is important to be aware that the correctness of the FSM is bound by the scenarios which we have illustrated. It does not claim that the FSM is complete and will solve every possible disruption. Doing so would warrant rigorous fault tolerance and is another thesis in its entirety. Nonetheless, both types of FSMs are explained in the following sections.

3.3.2.2 Proxy-Message Finite State Machine

Every message that comes into the proxy goes through the proxy-message FSM, which is illustrated in Figure 3.6.

The proxy-message FSM shows the lifetime of a message processed by the proxy. There are two proxy-message FSMs, one for each destination of either client or server where server denotes the remote service. In the case study, there was a client proxy-message FSM each for Alvin and Cynthia. A server proxy-message FSM was allocated

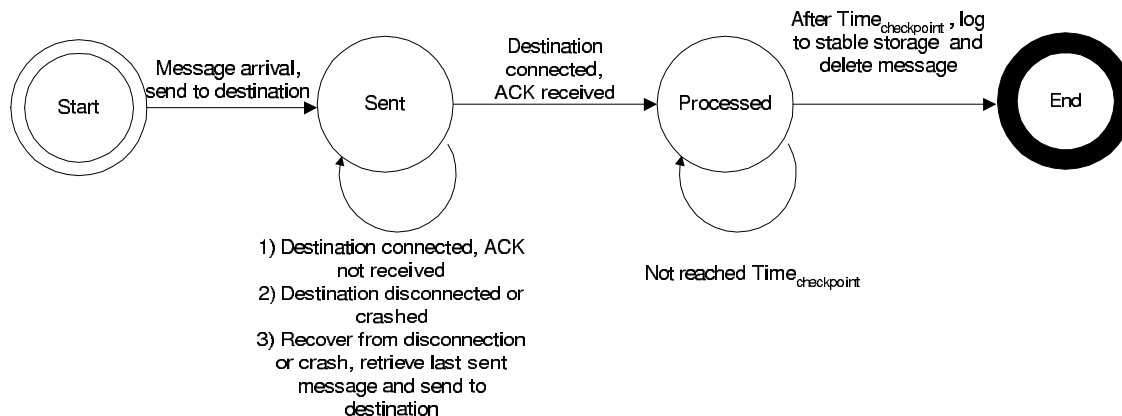


Figure 3.6: Proxy-message finite state machine

to handle service responses from the conference agenda service for Alvin. For Cynthia, a server proxy-message FSM was used to handle service responses for the chat web service in Starbucks, and another server proxy-message FSM for the weather web service in Second Cup. A message can be a client service request (query or operation) like for example Alvin selects the local news service, or a server response to a client service request like the conference news and Halifax news from the local news service. Messages are sent to the Proxy Server which then sends it to the client (if messages came from the service), or to the service (if messages came from the client). If the client sends a service request, this message is processed by the client proxy-message FSM. If the service sends a service response, this message is processed by the server proxy-message FSM. In either case, the FSM logic for both remains the same and we create an application protocol using four states in the FSM: *Start*, *Sent*, *Processed* and *End*. Queues are created for each of the *Sent* and *Processed* message states for both the client proxy-message FSM and server proxy-message FSM. We refer to the queue used for the client proxy-message FSM and the queue for the server proxy-

message FSM as client queue and server queue respectively. Figure 3.7 illustrates the queues used for the client and server, denoted as client request and server response.

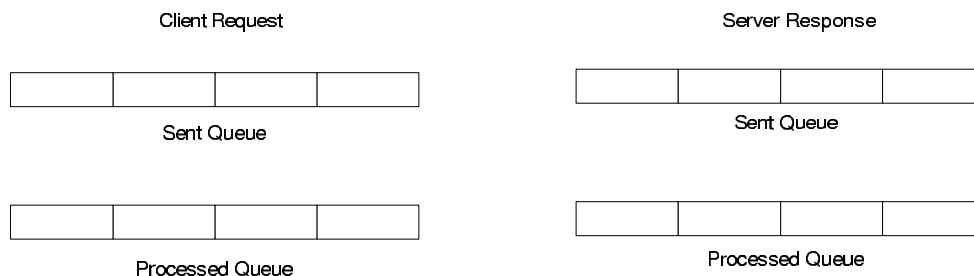


Figure 3.7: Proxy message queues

Regular operation. An example will illustrate the states that messages go through during a scenario using our proxy-message FSM from Figure 3.6. In the scenario that Alvin was in Second Cup, he selected to view the conference news from the local news service offered there (Figure A.4). This request got sent to the proxy allocated to Alvin and was received by its client proxy-message FSM. This message began in the *Start* state. When the message arrived at the proxy, the state of the message became *Sent* and was placed in the client's Sent queue as illustrated in Figure 3.8. The proxy then sent the message to the local news service via the Proxy Server. At this point, the proxy was waiting for an acknowledgement (ACK) from the local news service to indicate that the message was successfully sent. Here the service could have sent a separate ACK to the proxy to indicate it had received it. However, if we examine that the service receives subsequent messages from the next client query requests, then the next message that the proxy receives from the client can serve as the ACK for the previous client query request. In other words, ACKs are piggybacked onto subsequent messages. So, the service responded and sent back an HTML page of the

conference schedule agenda (Figure A.5) which got cached at the proxy in the server's Sent queue to be sent back to Alvin. This is also illustrated in Figure 3.8.

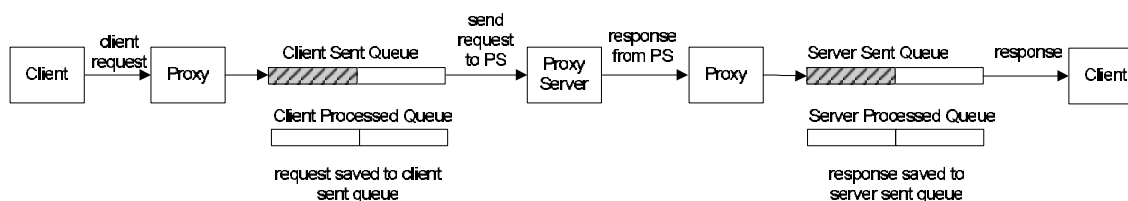


Figure 3.8: Proxy operation for no disconnections or crashes

Disconnection. When Alvin got interrupted by his cell phone, he got disconnected from the network. m-Roam was unaware so the service response remained in the Sent state and was stored in the server's Sent queue. Upon Alvin's return, he logged back into the conference web site and continued where he had left off (Figure A.6). m-Roam then recovered from disconnection and retrieved the last message in the server's Sent queue as illustrated in Figure 3.9, which was the HTML page of the conference schedule agenda displayed in Figure A.7. In fact, m-Roam did not detect disconnection in that a radio disconnect or a network disconnect had occurred. It knew to recover from disconnection because when Alvin logged back in, m-Roam detected that his session still existed and he was in the same location (Second Cup) as before. Therefore m-Roam made the assumption that Alvin would have wanted to have his last saved service response sent back to him from the proxy, which was a reasonable assumption. Even if this was not a reasonable assumption, Alvin was able to confirm this by selecting whether he wanted to continue the existing session or start a new session (Figure A.6). If Alvin's Palm had crashed or was disconnected, the above procedure would have obtained the correct service response because we

make the assumption that requests and responses are received in order. This means that every request will be followed by a corresponding response before a subsequent request is sent.

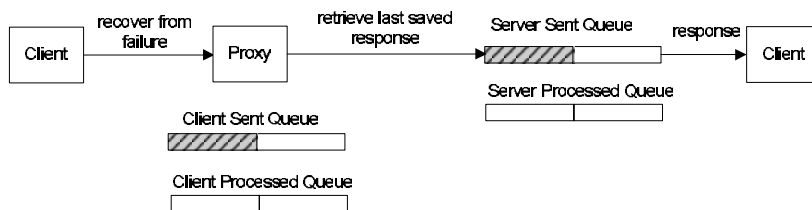


Figure 3.9: Proxy operation for client reconnection

Request after reconnection. If Alvin then wanted to go back to the list of services to find another service, this request would have been sent to his proxy and the message would have been saved to his proxy’s client Sent queue. Since m-Roam had received the next subsequent request, there was no need to save the “select conference news” request. Therefore, the “select conference news” request was removed from his proxy’s client Sent queue, and became *Processed* by saving it to the client Processed queue. This is illustrated in Figure 3.10.

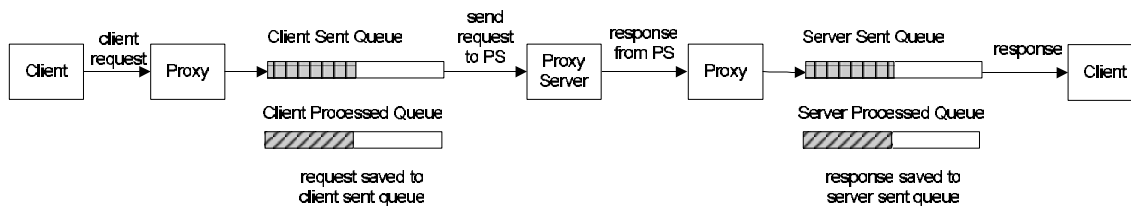


Figure 3.10: Proxy operation for client request after reconnection

Checkpoints and proxy crashes. If we wish to make a log of all Processed requests and responses from both the client queue and server queue, then we can use

standard checkpointing techniques from the database area to store them to stable storage. This can facilitate a history of Sent requests and responses for Alvin which may be useful for analysis and for rollback if the proxy crashes. However we do not consider proxy crashes in this thesis but only include this for completeness, and therefore do not discuss this any further. Once the Processed message is written to stable storage, then the message in the proxy comes to an end which is designated by the *End* state.

Roaming. We now illustrate the operation of the proxy-message FSM for the scenario when Cynthia moved from Starbucks to Second Cup. While Cynthia was checking the weather in Dartmouth at Starbucks (Figure A.15), she already obtained a proxy and her service requests sent to m-Roam and service responses received from m-Roam returned to her, following the regular proxy operation as that for Alvin. At Second Cup, m-Roam assigned a new proxy in this location (Figure A.16). m-Roam then initiated a handover to transfer all the messages from the old proxy in her previous location of Starbucks. All messages were removed from the client's and server's Sent and Processed queues from her old proxy, and added to the client's and server's Sent and Processed queues in her new proxy. As a result when Cynthia told m-Roam she wished to continue her existing session from before (Figure A.17), m-Roam was able to retrieve the last saved proxy response which was the weather service (Figure A.18).

Design analysis of proxy-message finite state machine. The proxy-message finite state machine presented in Figure 3.6 is a modification of the one presented by

Bin Yao and W. Kent Fuchs [52] where a message state and queue called Buffered is added. All messages that arrive at the proxy but have not been sent to the destination, are in the Buffered state. Buffered messages become Sent when the source is connected to the proxy and the Buffered messages are sent to the destination. This Buffered state and associated queue is not really required because once the message arrives at the proxy, it can be sent to the destination right away and stored in the Sent queue. If the destination gets disconnected before the Buffered message is sent, then the message remains Buffered. In our proxy-message FSM, destination disconnection keeps the message in the Sent state because the message has already been sent. In Yao and Fuchs' proxy-message FSM, they use an extra queue and state to process arrived but not Sent messages because they do not send the arrived message right away. The benefit of our approach is that it eliminates an unnecessary Buffered message, thus reducing the number of proxy queues and simplifying the proxy operation.

3.3.2.3 Proxy-Instance Finite State Machine

In addition to each proxy having a proxy-message FSM to process incoming messages (client requests and server responses), each proxy also has an associated FSM that tracks the lifecycle of the proxy instance throughout the lifetime of the client session. The FSM for this is shown in Figure 3.11:

The proxy-instance FSM consists of seven states: *Initialize*, *Started*, *Running*, *Recovering*, *Waiting*, *Migrating* and *Available*. When the DPU is first created during system initialization, the proxies are all created and their states are set to *Initialize*. We explain the operation of the proxy-instance FSM through the illustration of our scenarios with Alvin and Cynthia.

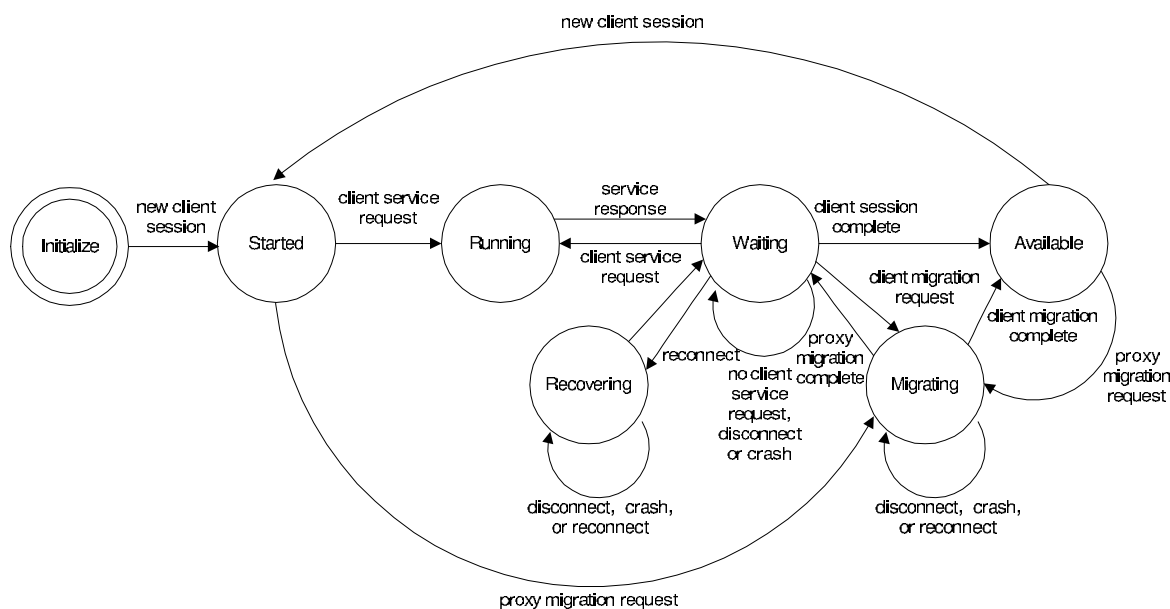


Figure 3.11: Proxy instance finite state machine

Regular operation. When Alvin logged into Second Cup (Figure A.1), a new proxy was assigned to him and the proxy entered the *Started* state. He then selected to find all services in this location (Figure A.2). This request message was received by his proxy and transferred the proxy from the *Started* state to the *Running* state. The message was then sent to the Proxy Server. Upon receiving a server response which was a list of all services available in Second Cup (Figure A.3), his client request was completed and the proxy entered the *Waiting* state, to wait for more requests from Alvin. Alvin then selected the local news service (Figure A.4), which generated another client request and transferred his proxy state from *Waiting* to *Running*. This alternation between *Waiting* and *Running* continued for Alvin's proxy while he selected the local news service (Figure A.3) and then the conference news (Figure A.4).

Disconnection. When Alvin got interrupted by his cell phone, he got disconnected from the network. His proxy did not know this therefore his proxy remained in the Waiting state when he received the conference agenda (Figure A.5). We assume that server responses will have returned back at the proxy while he was still disconnected, therefore the proxy will have always remained in the Waiting state and never the Running state. After he logged back in and reconnected to the network (Figure A.1), his proxy moved into *Recovering* mode where it had to recover from the failure by retrieving the last saved response which was the conference agenda (Figure A.7). If Alvin's Palm had crashed and he reconnected back to the network, his proxy would also have been in the Recovering state. After the proxy was recovered, it returned back to the Waiting state waiting for Alvin's next client request. When Alvin finished the session, a "client session complete" message was sent to his proxy which made his proxy *Available* for other clients in this location to use. When an available proxy becomes allocated to another client, it goes into the Started state and the above procedure is repeated. It is important to remember that the proxy never detects an actual disconnection or crash, it only receives a "Reconnect" message indicating that the client is continuing a previous session which transfers the proxy's current state to the Recovering state.

Roaming. For roaming, we explain the proxy-instance FSM using the scenario with Cynthia. Cynthia's proxy operated in Starbucks in the same fashion as Alvin's for regular operation. When Cynthia logged into Second Cup (Figure A.16), she obtained a new proxy in this new location. Since m-Roam determined the location had changed from the location parameter in the URL at login, it had to perform a mi-

gration. Cynthia's new proxy could have been an existing proxy (which was used to service previous clients) in which case she would have received a proxy in the Available state, or it could have been a proxy which was never used before in the Started state. Either way, her new proxy received a "proxy migration" request and entered the *Migrating* state. In this state, her new proxy in Second Cup notified her old proxy from Starbucks to transfer all proxy messages from the proxy queues over. For her old proxy, we assumed that the service response was received by the proxy just before she migrated to Second Cup. Hence its state moved from Waiting to Migrating, whereas for her new proxy its state moved from either Started or Available to Migrating. During the migration process, Cynthia could have disconnected, crashed or recovered from those failures. In either case, the proxy would have remained in the Migrating state since it would have to finish the migration process. Once the migration completed, the old proxy was made available for other clients to use so it entered the Available state. The new proxy returned the updated weather service to her as in Figure A.18. The new proxy was able to service additional requests for Cynthia in Second Cup, therefore it entered the Waiting state.

It is important to note that in the proxy-instance FSM there is no End state, since the proxy continues to operate according to the FSM. Only when the DPU is shut down (either gracefully or it has crashed), does the proxy disappear. Information in all proxies are then lost, unless they are logged and saved to stable storage for restore and boot up.

3.4 Complete Architecture for Service Roaming

Having been exposed to each architectural component in m-Roam, we now explain the complete architecture for service roaming. The architecture for service roaming involves multiple instantiations of the Service-Roaming Framework as illustrated in Figure 3.4 (one for each designated location), combined with the system architecture for one location illustrated in Figure 3.2. This results in the system architecture for service roaming, shown in Figure 3.12. We explain this architecture with our scenario with Cynthia in Starbucks and then moving to Second Cup. From the figure, location A refers to Starbucks and location B refers to Second Cup in our scenario.

When Cynthia arrived at Starbucks, she logged in to the conference web server (1) as shown in Figure A.8. The system located the appropriate DPC and assigned a proxy for Cynthia in the DPU (2). Cynthia then got successfully logged in (3). She performed a client request to find all services in Starbucks (Figure A.9) which was intercepted by her proxy and sent to the Service Translator (4). The Service Translator translated the request into a SIF request which was sent by the Proxy Server to the Service-Invocation Framework (5). The Service-Invocation Framework located all the services offered in Starbucks by querying the Distributed Directory of Services with the location attribute matching Starbucks and then invoked it at the Back-end Enterprise Services (6). The list of all services in Starbucks was returned and sent back to the Proxy Server as a SIF response (7). The Proxy Manager received the SIF response and the Service Translator converted the SIF response into an HTTP service response, which got stored to Cynthia's proxy (8). The list of all services in Starbucks was then displayed on the client (9) as shown in Figure A.10. She then

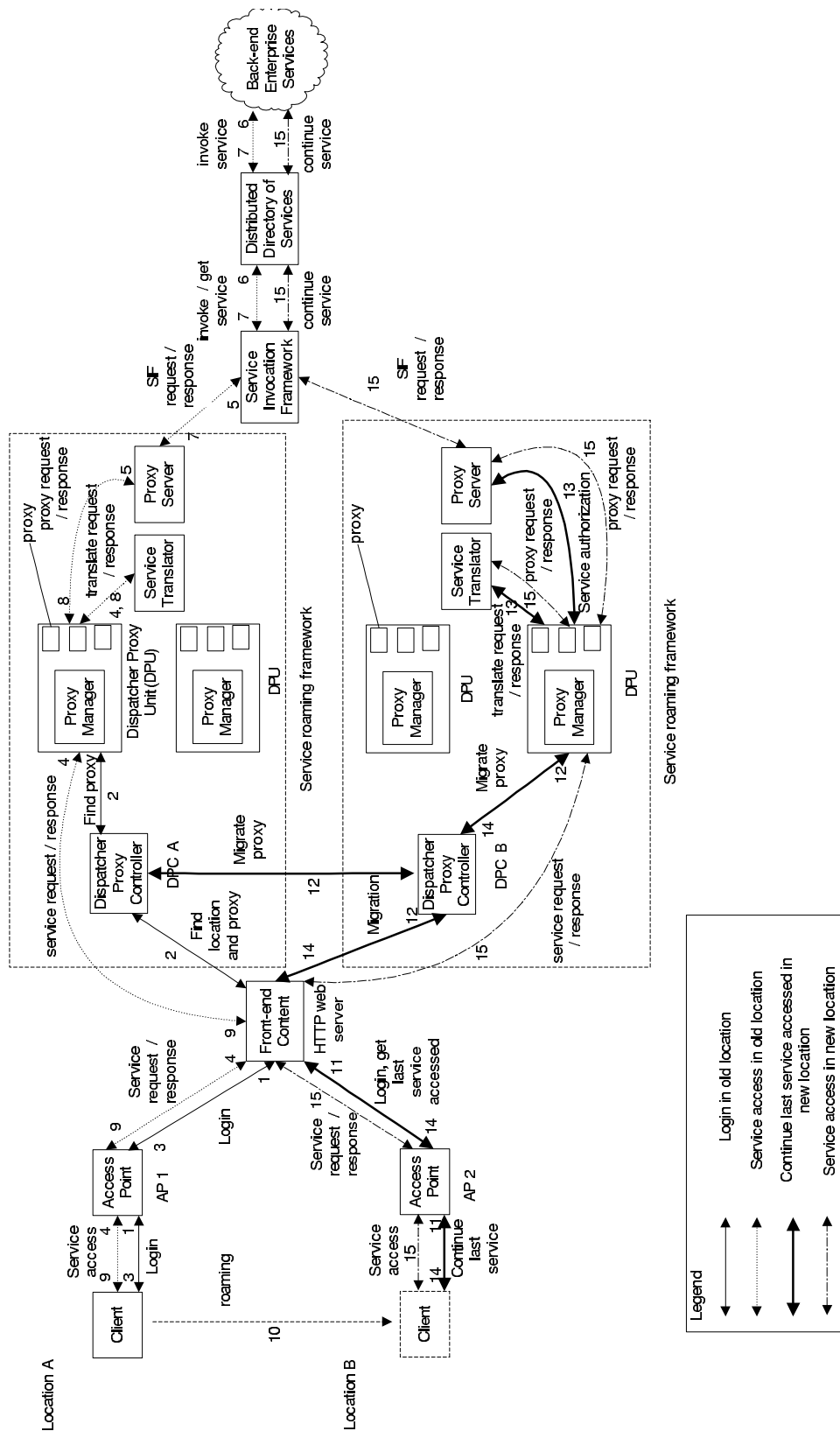


Figure 3.12: System architecture for service roaming

proceeded to select the instant messaging service (Figure A.11), then chatted with Jennifer (figures A.12 and A.13), selected the weather service (Figure A.14), and located the weather information for Dartmouth (Figure A.15). Each of these requests followed steps (4) through (9).

Cynthia then logged out of Starbucks and went to the conference. She met with Jennifer at Second Cup then logged back in (10) as in Figure A.16. The Dispatcher Process in the Front-end Content discovered that Cynthia was in a different location based from the session (11) as in Figure A.17, and invoked the Client Migration Protocol (12), detailed in the next chapter. A new proxy was located on the DPU in Second Cup by DPC B, and information and data from her old proxy was transferred to her new proxy (12). Once the handover was complete, the system had to determine if Cynthia was authorized to access the interrupted service (which was the weather service) in Second Cup by making a service-authorization request (13). m-Roam discovered that the weather service was authorized for her to use in Second Cup, then the weather information was retrieved from the proxy. However, before it was returned to her the information in the weather had changed since she had last accessed it from Starbucks in Dartmouth. Therefore, the request for the current weather had to be reissued to the weather web service so that the response returned was valid, following steps (4) through (8). The weather for Halifax was displayed to Cynthia (14) as in Figure A.18. Cynthia could now continue performing operations on a previously running service without any interruption (15) in the same manner as in (4) through (9).

3.5 Summary and Conclusion

This chapter has presented the high-level system and detailed architecture of our system for supporting service invocation and roaming. The architecture for m-Roam is divided into four components: Front-end Content, Service-Roaming Framework, Service-Invocation Framework, and Distributed Directory of Services. We described in detail the architecture of the first three components using our two scenarios with Alvin and Cynthia, of which our major contribution is the Service-Roaming Framework. The Front-end Content dispatches client-query requests and presents the service responses back to the client. The Service-Roaming Framework processes these requests based on location and hides client disconnections, migrations and crashes. Service discovery and execution of the client-query requests are performed by the Service-Invocation Framework using web services found from the UDDI registry in the Distributed Directory of Services. We then described the details of our design of the proxy and the proxy FSMs, namely the proxy-message FSM and the proxy-instance FSM which are responsible for the service invocation and roaming. Finally, we combined all the architecture elements together to present a complete architecture to explain a service roaming scenario from our case study with Cynthia.

The proposed architecture shows how existing client recovery methods, location-based services, web services, stateful sessions, and proxies can all be integrated together in order to create a wireless, pervasive computing environment conducive for accessing services using wireless devices. The next chapter will describe the interactions of the architectural components in the context of use case scenarios using messaging protocols.

Chapter 4

Messaging Protocols and Use Cases

The previous chapter presented the proposed architecture for service invocation and roaming. This chapter provides the details behind the interactions of the subsystems in the system architecture. Here, we present use cases for the proposed system based from the case study scenarios described in Section 1.3 and its associated messaging protocols. Each use case illustrates the use of the proposed messaging protocols that dictate and govern the communication between the various modules in the subsystems of our architecture presented in the previous chapter. Message sequence diagrams are used to explain the messaging protocols behind each use case. A message sequence diagram is a sequential flow of messages and events that occur among and between the components of the architecture. Many of the use cases and protocols presented here are not new, however, our contribution is the creation of protocols that integrate our various architectural components into a cohesive infrastructure for supporting our case study scenarios and other similar ones.

This chapter is organized into four sections. The use cases are grouped into three

categories. Section 4.1 explains the *no roaming* use case in which the user does not experience any disconnection, crash, or migration. When a disconnection or crash occurs, the user needs to recover from it. Section 4.2 illustrates how this is done in the *disconnection and crash recovery* use case. If the user moves to another location, the user is roaming and Section 4.3 details the *roaming* use case and the Client Migration Protocol which is used to perform the migration. Section 4.4 provides a summary and conclusion to the chapter.

4.1 No roaming and no disconnection

The first use case category deals with the situation where the mobile client is connected only in one location. Furthermore, the assumption is that there is no disconnection or crash on the client. We use the scenario with Alvin in Second Cup to illustrate this. The scenario with Cynthia in Starbucks follows the same protocol. The figures that we refer to (denoted as Figure A.x) can be found in Appendix A.

4.1.1 Login

Alvin logged into the system as in Figure A.1 using standard login procedures. A session was created for Alvin by the Session Manager and the Dispatcher Proxy Controller (DPC) was found for Second Cup. The DPC Manager located an available DPU for Alvin, then the Proxy Manager assigned an available proxy which created the necessary proxy message queues from which the proxy started. Once the login was complete and a proxy was assigned, Alvin received a URL to the services start page as shown in Figure A.2.

4.1.2 Service access

Our contribution is the addition of state to the proxy and the client session in order to make it become a stateful protocol. In order to execute a service in our protocol, we divide service invocation into three phases in which a user needs to: 1) *perform service query* to find a particular service, 2) *select service* if multiple services are returned from the query, and then 3) *invoke service*. The queries here are considered simple queries because they place simple equality conditions on the underlying data [31]. In other words, the values of attributes embedded in the query are matched to services that have those attribute values. In our scenario, Alvin wanted to find the conference schedule while at Second Cup. Figure 4.1 shows the message sequence diagram that was used in the system to accomplish this.

4.1.2.1 Perform service query

In performing a service query shown in Figure 4.1, the user can select two different types of queries. One query is to search for all services in this location using a local or simple query, and the other is to search for specific services according to user defined criteria or service categories using local, nonlocal, simple or general queries. Alvin selected to find the service that offered the conference schedule from browsing through the list of all services in Second Cup. He generated a *perform service query* by issuing a "Find all services in this location" query (1) from Figure A.2. The Dispatcher Process received his search query request (2) and sent it to the appropriate DPU found from the login procedure in Section 4.1.1. The Proxy Manager in this DPU saved this request to his proxy's client Sent queue (3). The proxy moved into the

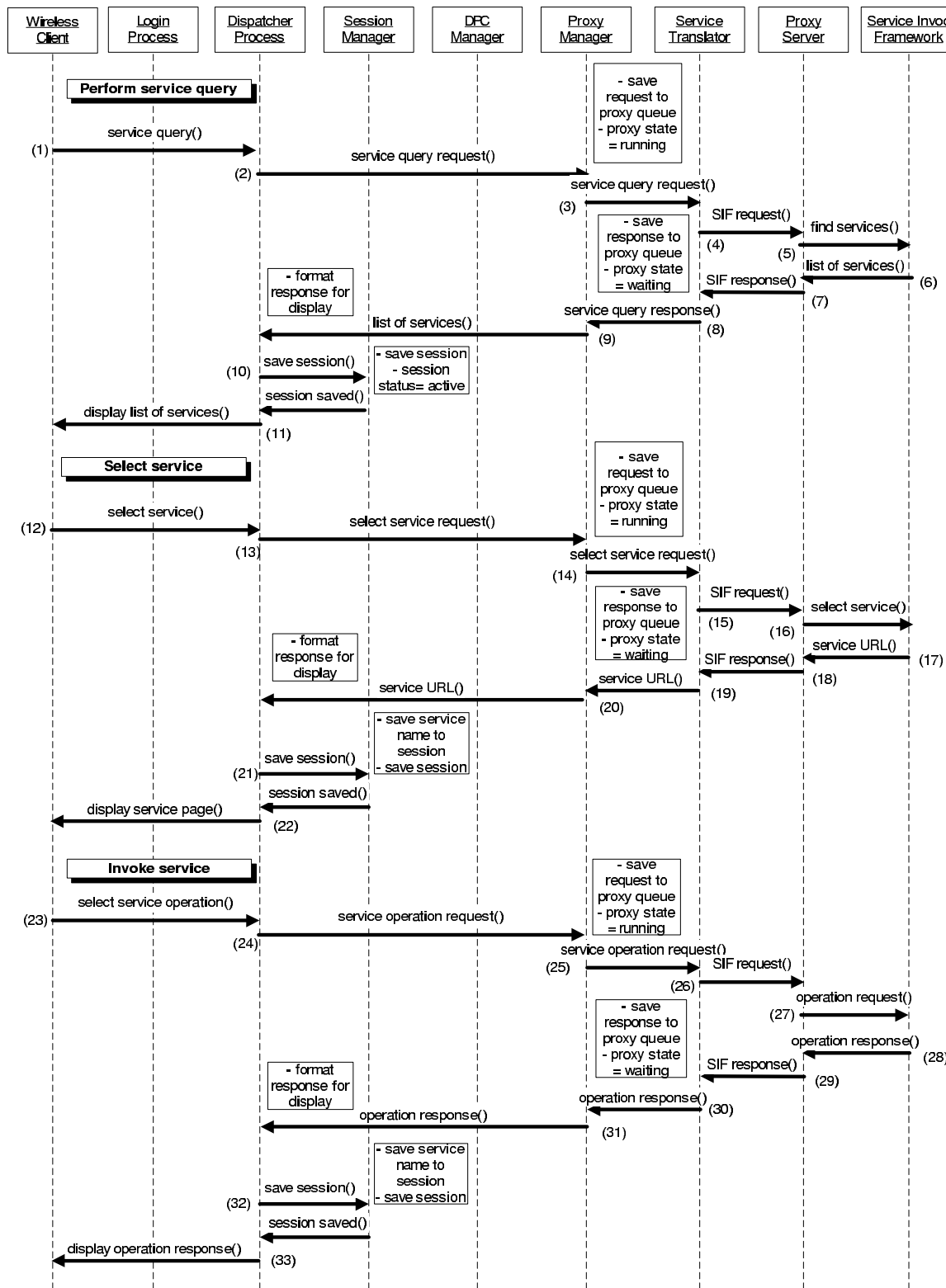


Figure 4.1: Message sequence diagram for service access

Running state. The query request was translated by the Service Translator into a SIF request and delivered to the Service-Invocation Framework via the Proxy Server (5). The Service-Invocation Framework then sent a UDDI search query to perform service discovery on the UDDI registry in the Distributed Directory of Services to find services whose location attribute had the value "Starbucks" or services that were location-independent.

After the UDDI registry answered the service search request, the list of services was returned to the Service-Invocation Framework and to the Proxy Server (6) which delivered it to the Service Translator (7), and the Proxy Manager saved it to the proxy's server Sent queue. The proxy went into the Waiting state (8), then the Proxy Manager relayed the response to the Dispatcher Process (9). The session was updated to Active and saved (10). The list of services was returned and displayed to Alvin (11), as shown in Figure A.3.

4.1.2.2 Select service

Next, Alvin selected the Local news service from Figure A.3 which issued a *select service* query (12) in Figure 4.1. The Dispatcher Process received this and forwarded it to his DPU (13). The Proxy Manager saved this query request to his proxy's client Sent queue changing the proxy state from Waiting to Running (14). The request was translated by the Service Translator into a SIF request (15) and sent to the Service-Invocation Framework via the Proxy Server (16). A UDDI request was issued to the UDDI registry on the Distributed Directory of Services in order to find the service that offers local news. Upon finding that service, it returned a list of service operations as service URLs in an HTTP response which was received by his proxy

via the Service-Invocation Framework (17), Proxy Server (18) and Service Translator (19). His proxy saved this response to the server's Sent queue, entered the Waiting state from the Running state (20), and the Session Manager saved the session (21). The response was formatted for display by the Dispatcher Process, and the conference and Halifax news were displayed to Alvin as illustrated in Figure A.4.

4.1.2.3 Invoke service

From Figure A.4, Alvin then wanted to find the conference agenda so he selected the Conference news from the Local news service to invoke it, which issued a *select-service operation* request (23) in Figure 4.1. This request was received by the Dispatcher Process (24) and saved to his proxy's client Sent queue by the Proxy Manager on his DPU (25). The proxy entered the Running state, and the Service Translator translated the request into a SIF request (26) which was sent via the Proxy Server to the Service-Invocation Framework (27). This request got executed in the Service-Invocation Framework which generated a SOAP request to the Local news web service in the Distributed Directory of Services. The Local news service executed this request and produced the conference agenda which was returned as a SIF response (with an embedded SOAP response) to the Service Translator (29) via the Service-Invocation Framework and the Proxy Server (28). The proxy stored the response in the server's Sent queue, and updated its state to Waiting (30). The response was formatted by the Dispatcher Process (31), and the session was updated and saved (32). Alvin received the conference agenda as shown in Figure A.5.

4.2 Disconnection and Crash Recovery

This section deals with the scenario when the client reconnects back to the system after it has been disconnected or it has crashed in the current location. We do not consider every possible failure that can happen in the system because we do not do fault tolerance. Rather, every failure that does happen (crash and disconnection) involves having the user continue the service from where he left off, thus abstracting the failure and forming our contribution. To hide disconnections and crashes, we divide the client to server (or service) connection into two connections separated by a proxy. We assume the proxy to server connection is fairly robust with a very low probability of disconnection that we consider negligible, whereas client disconnections are more probable in wireless environments. Therefore, we do not address server or proxy crashes but consider only client to proxy connections. Our decision to abstract client disconnection and client crash combined with low probabilities of proxy and server crashes, allow for easy integration of existing services into our architecture. In this way, services do not have to add fault tolerance or recovery mechanisms because failure on a client is not exposed to the service since it is intercepted by the proxy. Our position on failure recovery is that of a dumb approach. We assume a failure has occurred because when the user logs in, m-Roam will have detected that he has logged in before, yet we do not figure out the type of failure.

The following explains the use cases for *client disconnection recovery* and *client crash recovery*.

4.2.1 Client disconnection recovery

To solve the problem of client disconnection, we ask the user to re-login to the system and the system retrieves the last saved message, which is nothing new. Therefore when Alvin got disconnected and he was interrupted by his cell phone, he was forced to re-login in order to retrieve his last saved response which was the conference agenda that he last accessed before the disconnection. Thus, this prevented other users to use his Palm to connect and retrieve his data. Asking the user to re-login is a simple solution to client disconnection and is our contribution. This was accomplished by following the message sequence diagram for client-disconnection recovery shown in Figure 4.2.

When Alvin logged in as in Figure A.1, the Login Process found out that Alvin had logged in from before (1). m-Roam notified Alvin that he had a previous session still active as shown from Figure A.6. Alvin selected to continue the existing session which was received by the Dispatcher Process (3), and the Session Manager retrieved the existing session (4). The Dispatcher Process issued a "recover from disconnection" request to the Proxy Manager on his DPU (6) to find the last saved response stored in his proxy's server Sent queue, which sent his proxy into the Recovering state. Once his last saved response was obtained (7), it was sent back to the Dispatcher Process. The session was saved (8), the last saved response was formatted for display, and Alvin received the conference agenda (9) as shown in Figure A.7 just like before the disconnection.

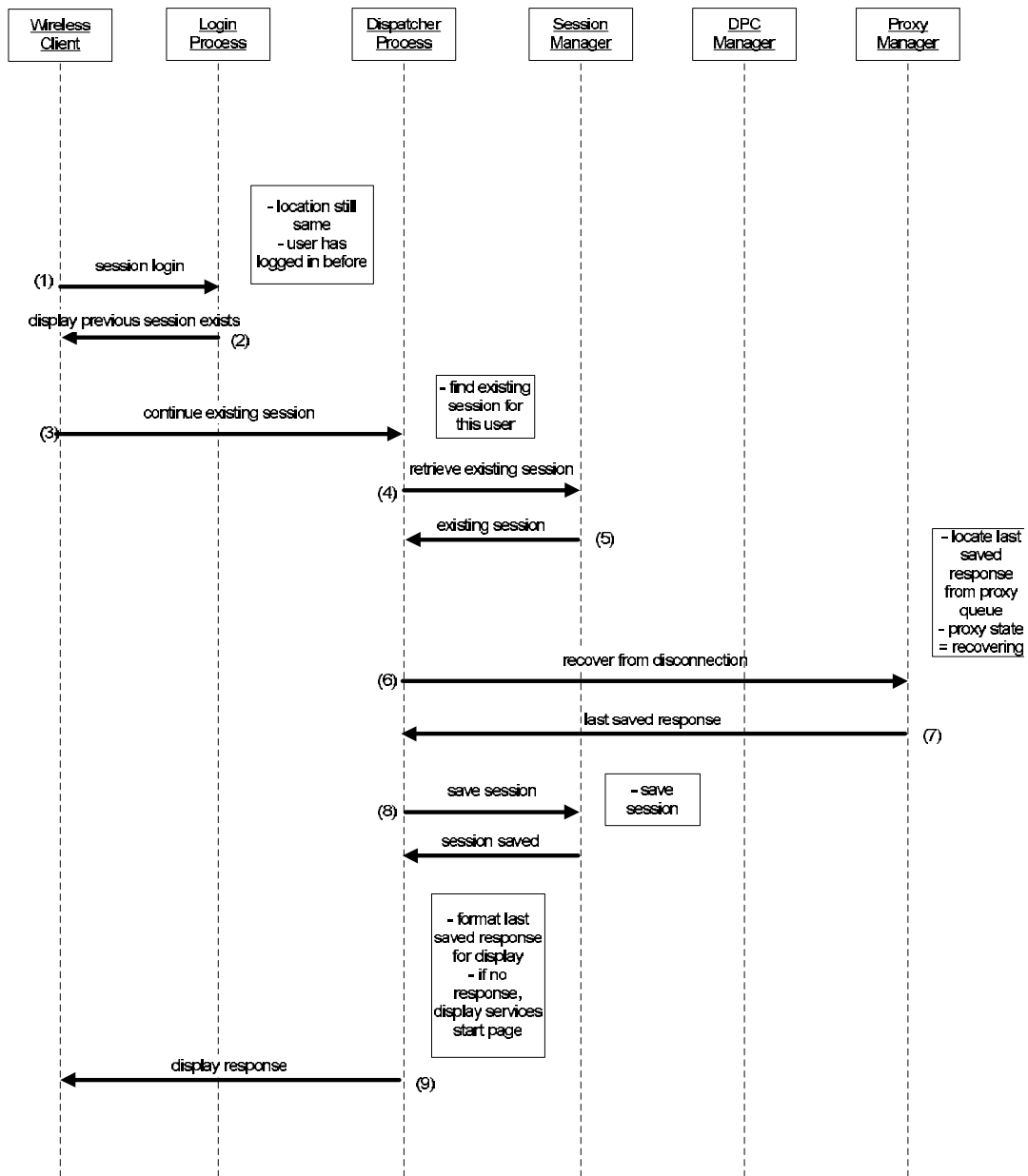


Figure 4.2: Message sequence diagram for client disconnection recovery

4.2.2 Client crash recovery

Recovering from a client crash means that the client needs to return back to its original state before the crash. In m-Roam, we infer that a client has crashed based on whether the current HTTP session exists. If the HTTP session does not exist, then the HTTP session has either timed out or we make the assumption that the client crashed because opening a new web browser instance creates a new HTTP session. The current HTTP session cached at the client is lost when the client's web browser is closed. We contribute to solving client crashes by performing recovery at the application layer using HTTP sessions. m-Roam does not distinguish between a client crash or a client disconnection, the recovery process for both is identical. The message sequence diagram is illustrated in Figure 4.3.

4.3 Roaming

A high-level overview of roaming was presented in Section 3.4. Here, our contribution is the application protocol for handover using interprocess communication between our architectural components. We use our scenario with Cynthia moving from Starbucks in Dartmouth to Second Cup outside the conference in Halifax, to illustrate this with roaming use cases. The handoff procedure is described in the *Client-Migration Protocol*.

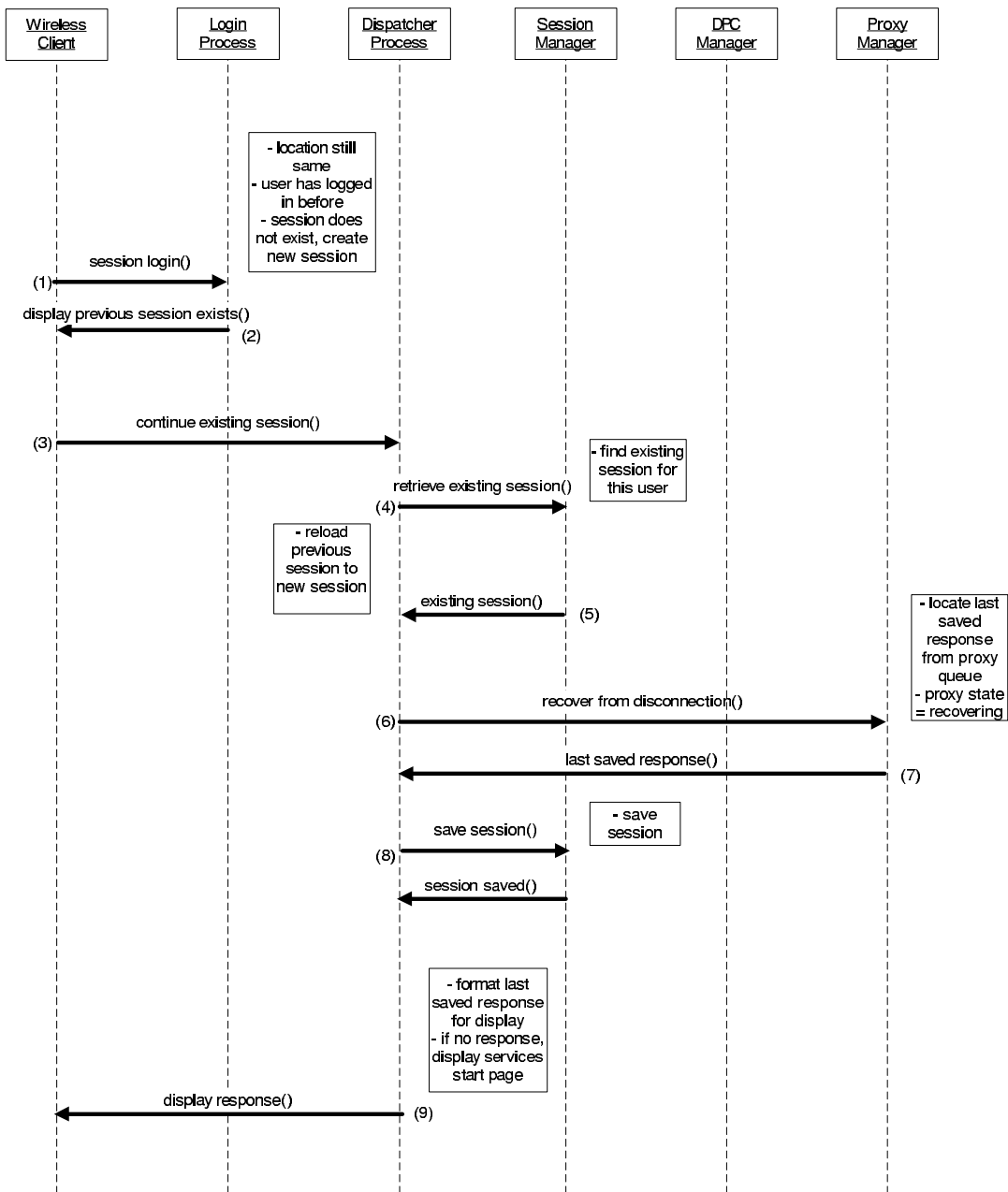


Figure 4.3: Message sequence diagram for client crash recovery

4.3.1 Client-Migration Protocol

The Client-Migration Protocol performs the handover from the previous location that the user was in, to the new location that the user has now moved to. It is responsible for transferring service state from the previous location to this new location, by migrating proxy information. The Client-Migration Protocol is similar to the signalling performed in MobileIP of network layer roaming (Section 2.3.2) and handover in physical layer roaming (Section 2.3.1), except that the handover is done at the application layer. The message sequence diagram in Figure 4.4 illustrates this.

We assume that when the client moves to another location say from A to B, the client will perform some network activity in B and will not suddenly move to a different location C. As such, only one migration happens at once. Another assumption is that the time to perform the Client-Migration Protocol will be fast enough such that if the client moves out of B, then all processing at B will have completed. In other words, there will not be any partial processing of queries at B and then a new migration occurs at C.

When Cynthia logged into Second Cup (Figure A.16) after coming from Starbucks, the Dispatcher Process realized that her location had changed and started to invoke the Client-Migration Protocol. We use the terms “new” and “old” here from Figure 4.4 to refer to Second Cup and Starbucks respectively. The Dispatcher Process sent a “handover request” message to DPC_{new} (1) which sent a “find new proxy” message to DPU_{new} (2). The Proxy Manager in DPU_{new} located a free proxy for Cynthia, set her new proxy’s state from Initialize (if the proxy was not used before) or Available (if the proxy was used before) to Migrating (3).

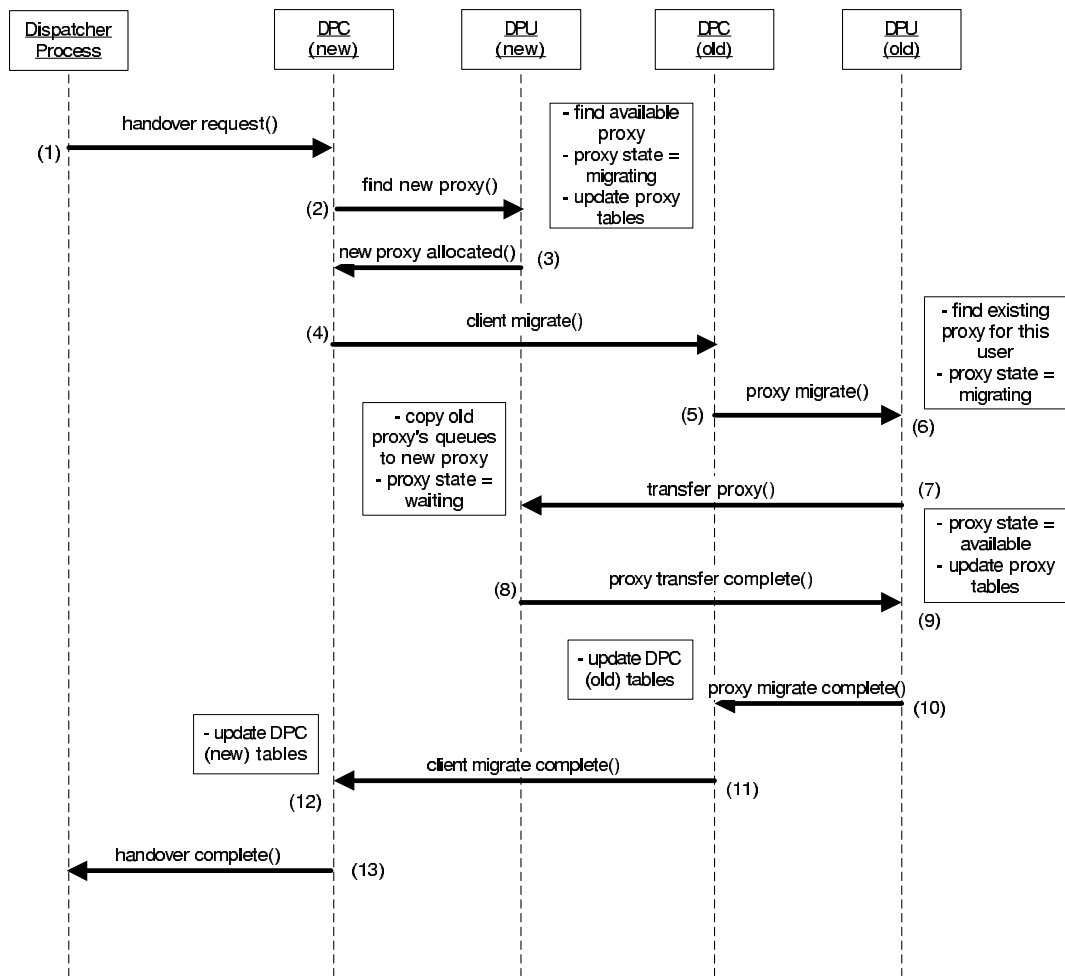


Figure 4.4: Message sequence diagram for client migration protocol

Once a new proxy was allocated, DPC_{new} notified Starbucks' DPC (4) (DPC_{old}) to transfer its proxy data to the new location of Second Cup through a "client migrate" message. DPC_{old} then sent a "proxy migrate" message to DPU_{old} , which was the DPU associated with Cynthia from Starbucks (5). The Proxy Manager in DPU_{old} located the existing proxy that was assigned to Cynthia and set its state to Migrating (6). It transferred the old proxy's messages from the client's and server's Sent and Processed queues to the new proxy in DPU_{new} (7). When the proxy transfer completed, the new proxy updated its state to Waiting (8) and updated its proxy tables accordingly. The old proxy in Starbucks updated its state to be Available and updated its proxy tables.

When the proxy transfer had completed, the proxy migration was complete and DPU_{old} notified DPC_{old} (10). The DPC Manager removed Cynthia from its table for the DPU_{old} entry. DPC_{old} sent a "client migrate complete" message to DPC_{new} (11) to indicate client migration was complete. The DPC Manager in DPC_{new} added Cynthia to its table and DPC_{new} sent a "handover complete" message to indicate Cynthia was successfully migrated over (13).

4.3.2 Roaming use cases

Several use case scenarios can be used to demonstrate the types of events that can occur after the user has moved to a new location. These use cases happen after the client has successfully migrated from the Client-Migration Protocol explained above. There are two interesting use cases to consider: (i) the query is *location-sensitive* that is dependent on the location or (ii) the query is *location-independent* that is not location-sensitive. These are referred to as local and nonlocal queries respectively

[31]. The query can be of three types: *service query*, *select service* or *select service operation*, as explained from Section 4.1.2. The reason for distinguishing between these two use cases is because the query response returned in the previous location may become invalid for this new location (as discussed in query scheduling of Section 2.4.2.3). Figure 4.5 illustrates this protocol using the roaming scenario with Cynthia.

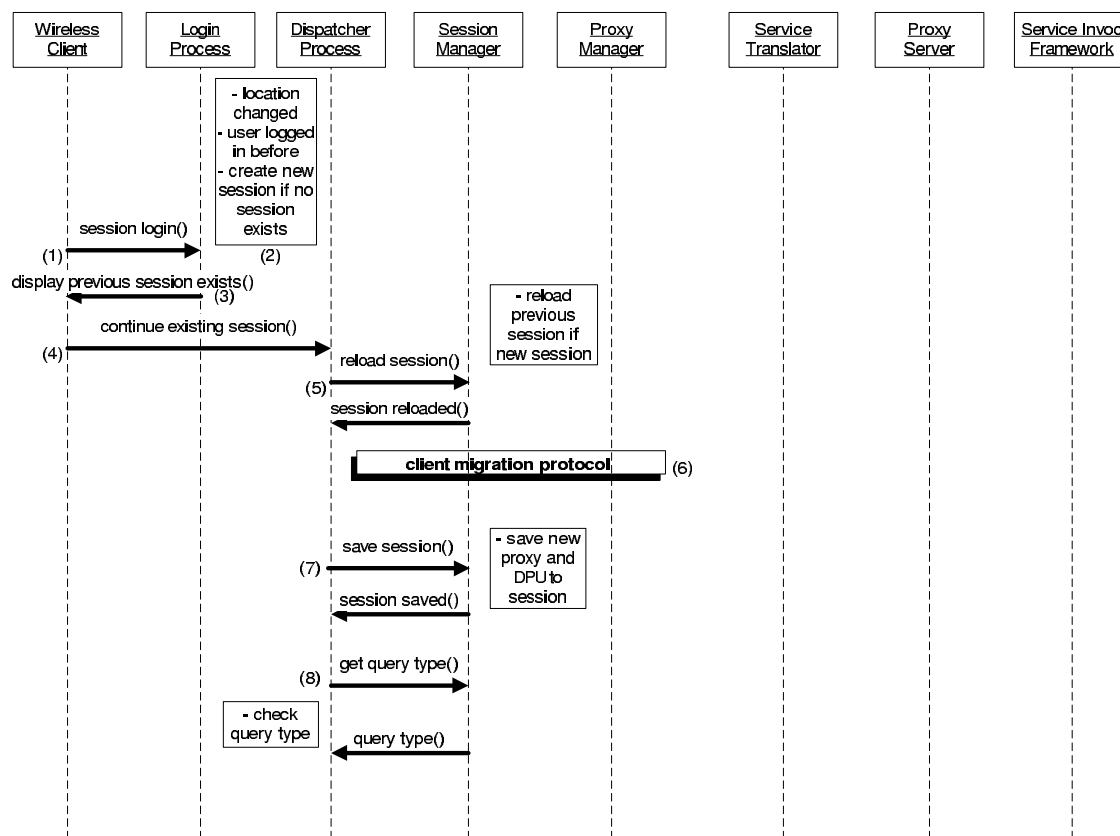


Figure 4.5: Message sequence diagram for roaming

When Cynthia arrived at Second Cup, she logged in as shown in Figure A.16 which was sent as a session login (1). The Login Process detected that Cynthia had an existing session from Starbucks (2) and created a new HTTP session for her. The Location Manager found the DPC that Cynthia was in from her location of Second

Cup. Roaming was detected by determining her current location was different from her previous location of Starbucks. Cynthia got notified that a previous session exists (3) as shown in the display of Figure A.17. Cynthia wanted to continue where she had left off from Starbucks and selected to continue the existing session (4). Since Cynthia moved, she obtained a new session so her previous session had to be reloaded to her new session (5). After, the Dispatcher Process started the Client-Migration Protocol (Figure 4.4) (6). The session was updated with the new proxy and DPU and saved to the session by the Session Manager (7). The query type was determined from the type of the last client-query request stored in the session.

4.3.2.1 Determining query type

m-Roam handles the last client-query request differently depending on the query type as Figure 4.6 shows. The query type can be either a *service query* or a *select service or select service operation*. Once the query type was found, the system had to determine whether Cynthia's last issued query was *location-sensitive* or *location-independent*. Figure 4.6 illustrates the protocol that was used to determine this.

Service query. A service query is defined as a query which involves a service search of either “find all services” or “find a specific service”. If the query type is a service query, then this is retrieved from the new proxy through a “recover from disconnection” message (9, 10) in Figure 4.6, just like steps (7) and (8) from the client disconnection recovery of Figure 4.2. The service query is parsed to find the location and if it exists, then the query is a *location-sensitive query* (11). The location-sensitive query procedure is carried out as in Figure 4.7 except that steps (1) and (2) are omit-

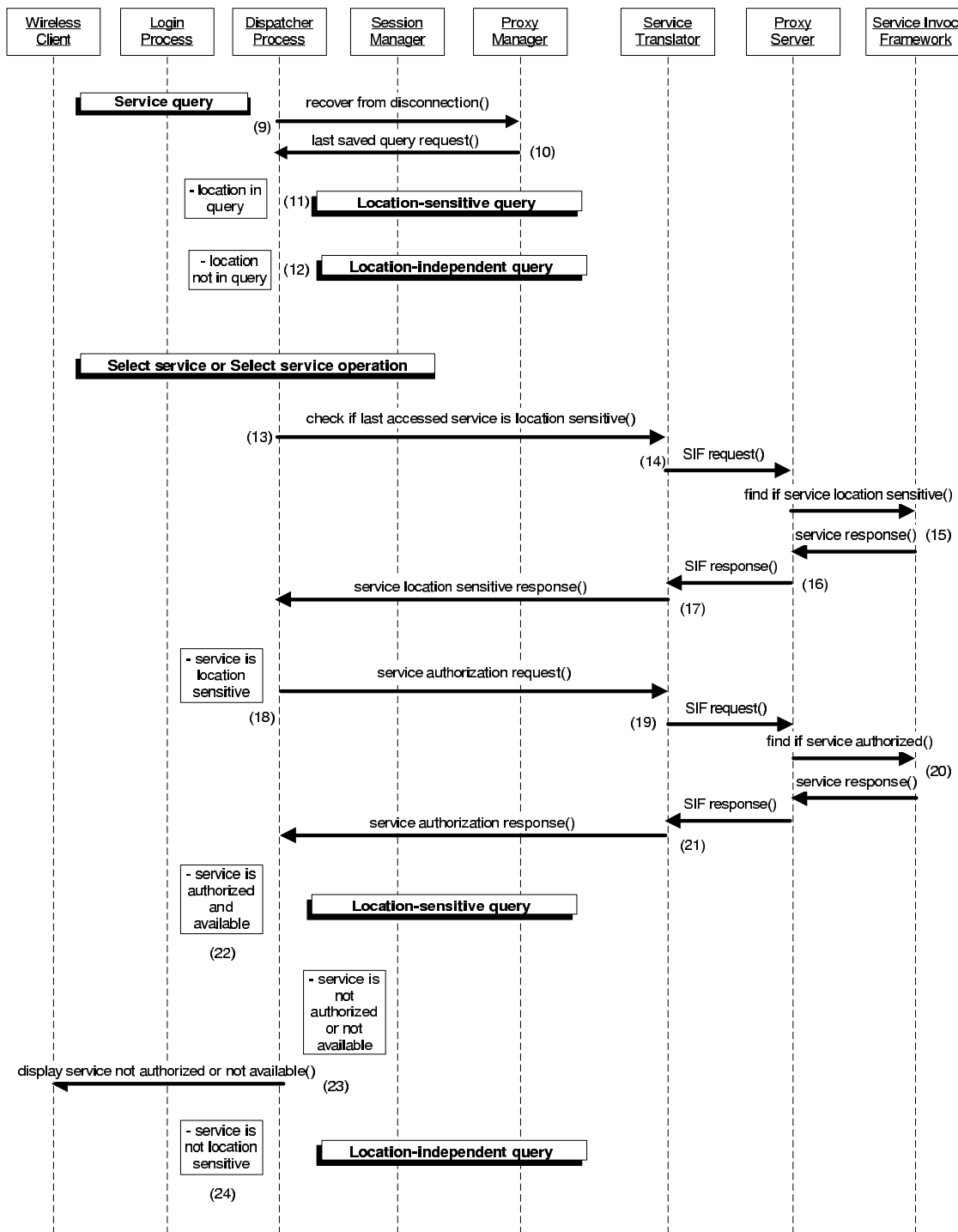


Figure 4.6: Message sequence diagram for roaming - query type

ted, because the recover-from-disconnection procedure has already been performed. However, if the location parameter is not found in the service query, then the query is location-independent and the *location-independent query* procedure is followed as in Section 4.3.2.3 (12). For Cynthia, her last query request was not a service query. m-Roam determined that her last query request was to find the weather in her current location, and this corresponded to a select service operation.

Select service or select service operation. Before allowing her to continue her last service operation, m-Roam figured out whether she was authorized to access the weather service at Second Cup. We do this for personal security purposes such that others trying to impersonate her will not have access to her data and services. We do not discuss security any further because we do not make any contributions here, and we assume that existing security practices and measures are followed. The Dispatcher Process checked to see if the service was location-sensitive (using the *Check if service is location-sensitive* protocol) and whether Cynthia was authorized to use the weather service in her new location of Second Cup (using the *Service authorization* protocol).

Check if service is location-sensitive. m-Roam determined that the weather service was location-dependent because she had asked for the current weather in her location. The Dispatcher Process sent a message to the Service Translator to check if the last accessed service (weather for Cynthia) was location-sensitive or location-independent (13). This was determined by querying the service using the Service-Invocation Framework with a “service-location-sensitive” SIF request (14). The weather service communicated to the Service Invocation Framework that it was

location-sensitive (15). After translation by the Service Translator (16), this response was returned to the Dispatcher Process (17). Next m-Roam determined whether Cynthia had permission to use the weather service in Second Cup by performing service authorization. If m-Roam had discovered that the service was not location-sensitive, it would have followed the location-independent query procedure in Figure 4.8 (24).

Service authorization. Since the weather service was location-sensitive, the Dispatcher Process sent a "service authorization" request to determine if she was authorized to use the weather service in Second Cup (18) which was sent to the service via the Service-Invocation Framework (19). The weather service found out that she had permission and returned a "service authorized" message in the "service authorization" response back to the Dispatcher Process from the Service-Invocation Framework (21). The *location-sensitive query* protocol in Figure 4.7 was then followed. However if Cynthia had no permission to use the weather service, she would have been notified by the Dispatcher Process (23).

4.3.2.2 Location-sensitive query

A query is *location-sensitive* if the result of the query request depends on the client's location. The request or service that the user invoked from the previous location, may have become outdated. As a result, the user needs to receive an updated handler for the service that was executed in the previous location before the roaming resumes [31]. This is because the last-saved proxy response may be invalid in this location. For our scenario with Cynthia, her last query from Starbucks was to find the weather for her current location. The service responded with the weather for Dartmouth,

however this was not valid in Halifax where she had moved to. m-Roam addressed and resolved this problem using the message-sequence diagram of Figure 4.7.

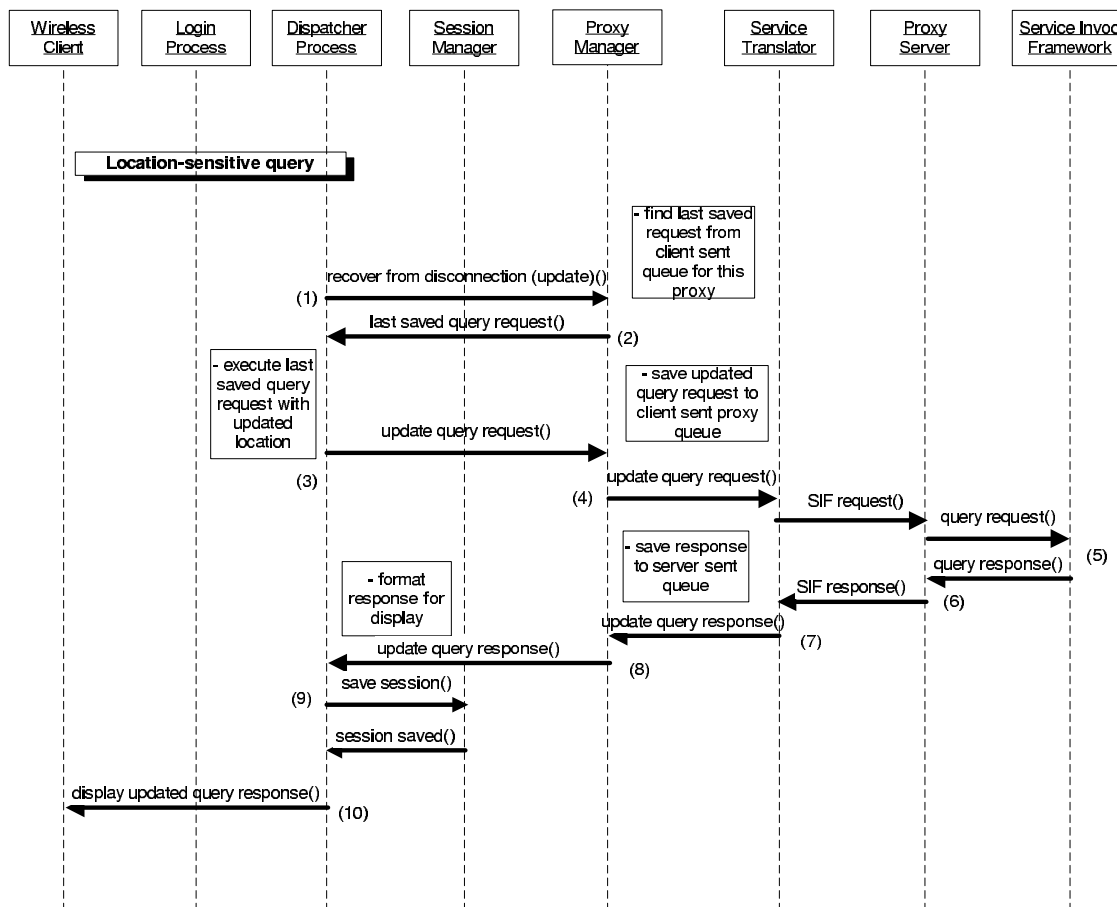


Figure 4.7: Message sequence diagram for roaming, location-sensitive query

Essentially, this involved recovering from a disconnection except that the response was not the last-saved query response from the new proxy because this response was incorrect (1) which was obtained from the Client-Migration Protocol. Instead, her query request that was sent in Starbucks before the migration, was executed again in her new location of Second Cup in order to retrieve the correct updated query response. Therefore, the last-saved query request was found from the client’s Sent

queue of the new proxy (2). We assume that we have pairwise synchronized requests and responses so that the last-saved query response corresponds to the last-saved query request. This request was modified by the Dispatcher Process to change the location parameter in the HTTP GET URL to the new location of Second Cup (3). An “update-query” request (4) was sent to the Service-Invocation Framework (5), was executed on the weather web service, and the “updated query” response (weather for Halifax) was sent back to the new proxy via the Service-Invocation Framework, Proxy Server, Service Translator, and Proxy Manager (6). The Proxy Manager saved this “updated query” response to the server’s Sent queue in the new proxy (7), it was returned to the Dispatcher Process for formatting to the display, and the session was updated and saved by the Session Manager (9). Finally, Cynthia received the updated current weather for Halifax, as shown in Figure A.18, from the Dispatcher Process.

4.3.2.3 Location-independent query

A query is *location independent* if the result of the query request does not depend on the user’s current location. This means that when the user moves, the service that the user had accessed in the previous location, is available subject to the condition that the user has permission to access it in this new location. The details of this scenario are shown in the message sequence diagram of Figure 4.8.

Once the user is authorized, then all that is required is to retrieve the last- saved service response before the roaming. This is essentially the recovery from disconnection use case of Figure 4.2 except there is no login. A “recover from disconnection” message is sent to the Proxy Manager for the new DPU (1), and the last-saved service response from the server’s Sent queue for this new proxy is retrieved (2) and

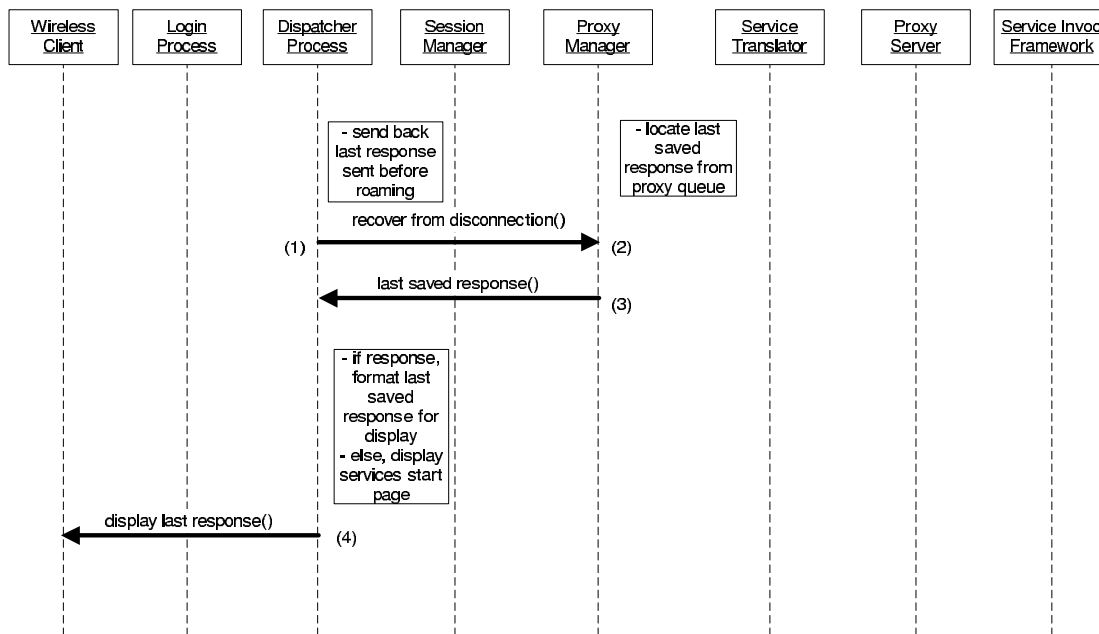


Figure 4.8: Message sequence diagram for roaming, location-independent query

returned to the Dispatcher Process (3). The Dispatcher Process formats the response for display to the user (4).

4.4 Summary and Conclusion

This chapter has presented the use cases and their associated messaging protocols to support service invocation and roaming, in the context of our 2 case study scenarios. The use cases fall into three categories: no roaming, disconnection and crash recovery, and roaming. Each use case was explained using message sequence diagrams, which illustrated the interactions between the Front-end Content, Service-Roaming Framework and Service-Invocation Framework (as presented in Chapter 3).

The no-roaming category of use cases show the scenario where the client stays in

one location, experiences no disconnection or crash, and does not move to another location, while continually accessing services. Service query, selection and execution form the process for accessing services. If the client becomes disconnected from the network or crashes, then upon recovery the client follows the disconnection- and-crash recovery category of use cases. In the client-disconnection-recovery use case, the client HTTP session still remains intact and services are accessed from before the disconnection. In the client-crash-recovery use case, the client's HTTP session is lost and needs to be reloaded from the system, before the previous service can be continued. When the client moves to another location, the client follows the roaming category of use cases, and the Client-Migration Protocol forms the core part of this. Depending on the type of client query, the appropriate use case is followed (service query, or select service or select service operation). If the query is location-based, then the location-sensitive use case is used, otherwise the location-independent use case is followed.

The next chapter will explain about the experiments that we conducted to validate and demonstrate our m-Roam architecture.

Chapter 5

Experiments

In the last two chapters, the design, messaging protocols, and use cases were explained. This chapter presents test cases using our proposed architecture and protocols. In our experiments, client queries from the non-roaming use cases of Section 4.1 are used to test and validate that our m-Roam architecture and service invocation protocol works as described. We do not test the service roaming protocol. Specifically, we measure the time performance and space requirements that our architecture uses. We conduct these experiments in order to determine how existing users in the system affect time for the next user to access a service, and whether the system works. The architecture and protocols are implemented using the Java programming language with Java 2 SDK 1.3 and 1.4.

This chapter is organized into three sections. Section 5.1 details the hardware and software environment we used to conduct our experiments. Section 5.2 explains the experiments to test the scalability and performance of our architecture using time and space, and the number of users and services. Finally, Section 5.3 provides a brief

summary of our experimental results.

5.1 Experimental Setup

Our experiments are conducted using the experimental setup in Table 5.1. This shows that our architectural components can be distributed and demonstrates how they would be deployed in a practical enterprise environment. In addition, the setup follows what the architecture suggests where we have one machine dedicated to each architectural component. By performing this allocation, we have low coupling between components and create a flexible architecture because the components do not have to be located in a central location. Thus, this exercises good software engineering practices.

The system architecture is implemented in the Java programming language using Java 2 SDK 1.3 and 1.4. All of the machines in Table 5.1 are connected in a 100 Mbps Ethernet network on the same subnet. We do not worry about the client having a wireless connection because we are just testing our protocols to ensure that they function well. Each component of the system architecture is assigned to one machine or URL. The Front-end Content is located on *Skiros* and consists of Java servlets [19] installed on a web application server using Apache Tomcat 4.0.3 [2]. *Skiros* also contains the web pages that display the services and their operations, as well as the Service-Invocation Framework for issuing client queries to the UDDI registry. The web services toolkit included in the IBM Emerging Technologies Toolkit (ETTK) [21] is used in the Service-Invocation Framework. The Service-Roaming Framework consists of 1 DPC which is comprised of 10 DPUs, and each DPU has 1000 proxies to

Table 5.1: Experimental Setup Environment

Machine/URL	Hardware Configuration	Software Configuration	Architecture Component
Samos	Sun UltraSparc-III 440 MHz, 256MB RAM	Solaris 8, Java 2 SDK 1.4.0-01, JBoss 3.0.4	Service-Roaming Framework (1 DPC,10 DPUs, 1000 proxies per DPU)
Skiros	Sun UltraSparc-III 270 MHz, 256MB RAM	SunOS 5.6, Java 2 SDK 1.4.0-01, Apache Tomcat 4.0.3, IBM ETTK 1.0	Front-end Content, Service-Invocation Framework
Anafi	IBM NetVista PC - Pentium III 1GHz, 384 MB RAM	Microsoft Windows 2000 Professional, Service Pack 4, Java 2 SDK 1.3.0-02	Service Roaming Framework - Proxy Server
Tinos	Sun UltraSparc-III 440 MHz, 256MB RAM	Solaris 8, Java 2 SDK 1.4.0-01	Test Client
IBM UDDI Business Test Registry	N/A	N/A	Distributed Directory of Services - UDDI Registry

accommodate a total of 10000 users. It is installed on *Samos*. The DPC and DPUs are implemented as Enterprise JavaBeans (EJBs) [34] and are deployed on an EJB application server using JBoss 3.0.4 [27].

The Proxy Server in the Service-Roaming Framework is installed on *Anafi*, and is a Java class which accepts socket connections for clients and then sends the request on the network to the Service-Invocation Framework. *Tinos* is used as the test client for conducting the experiments. For the UDDI registry, the IBM UDDI Business Test Registry [23] is used.

5.2 Time Performance and Space Requirements

In this section, we determine if our architecture works and performs well using time and space parameters. In order to do this, three sets of experiments have been devised that test the design of our architecture. We conduct our experiments using the experimental setup listed in Table 5.1. The experiments are broken down into two categories, time and space, to give a total of three experiments.

5.2.1 Time Experiments

Time is used to measure the performance of accessing services by issuing a service query without any client disconnection or client crash. The objective of these experiments is to determine how the designed architecture in the Service-Roaming Framework affects the time to obtain a service response from the service query. Two sets of experiments are conducted, one where the service is accessed locally and not through a remote UDDI registry, and one through UDDI. We repeat each experiment five times and then take the average of them as our time. Before the beginning of each experiment, the proxies and associated tables are deleted then newly created.

5.2.1.1 Time to access service for last user without UDDI

For this experiment, we measure the time that it takes to issue a service query where the service is stored locally in the Service-Invocation Framework, and not in a UDDI registry, versus the total number of users in the system. This time is calculated from the beginning of the service query to when the response is delivered from the Front-end Content back to the client. The service query that is issued is a URL to a service

where the request is the word “Hello”. The service then responds with an echo of the request (which in this case is “Hello”). This service query is issued after the last user has been created in the system, and $N-1$ users have already been created (simulating new client login). The time that we calculate is used to determine the overhead of having a Distributed Directory of Services layer. The graph that illustrates the results of this experiment is illustrated in Figure 5.1.

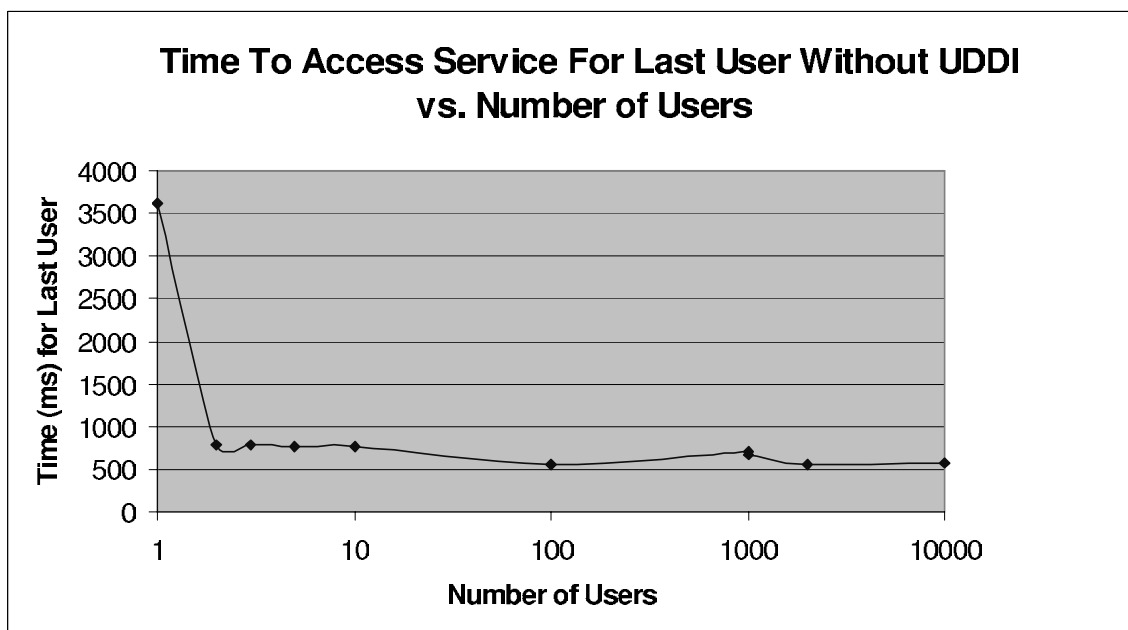


Figure 5.1: Time to access service for last user without UDDI

From Figure 5.1, it can be seen that there is no significant impact of the number of users in the system on the time to access a service. Overall the the time is pretty constant for the last user in the system to access a service. This shows that the number of DPUs and proxies in our system for a particular DPC is independent of the time for a user to access a service. We experienced some overhead when the first user obtained a DPU and proxy.

5.2.1.2 Time to access service for last user with UDDI

This experiment is similar to the previous one except that the service is located on the IBM UDDI Test Business Registry with varying number of services in UDDI. The number of services in the UDDI registry are set to 3, 9, and 22. The number 22 is used because that is the limit of the total number of services that can be saved to the IBM UDDI Test Business Registry. The service query that is issued by the client is a URL to a servlet to find a service in the SWEN lab location which provides a chat feature. The objective of this experiment is to determine the additional overhead involved for having the user access a service using UDDI. Figure 5.2 shows the graph for this.

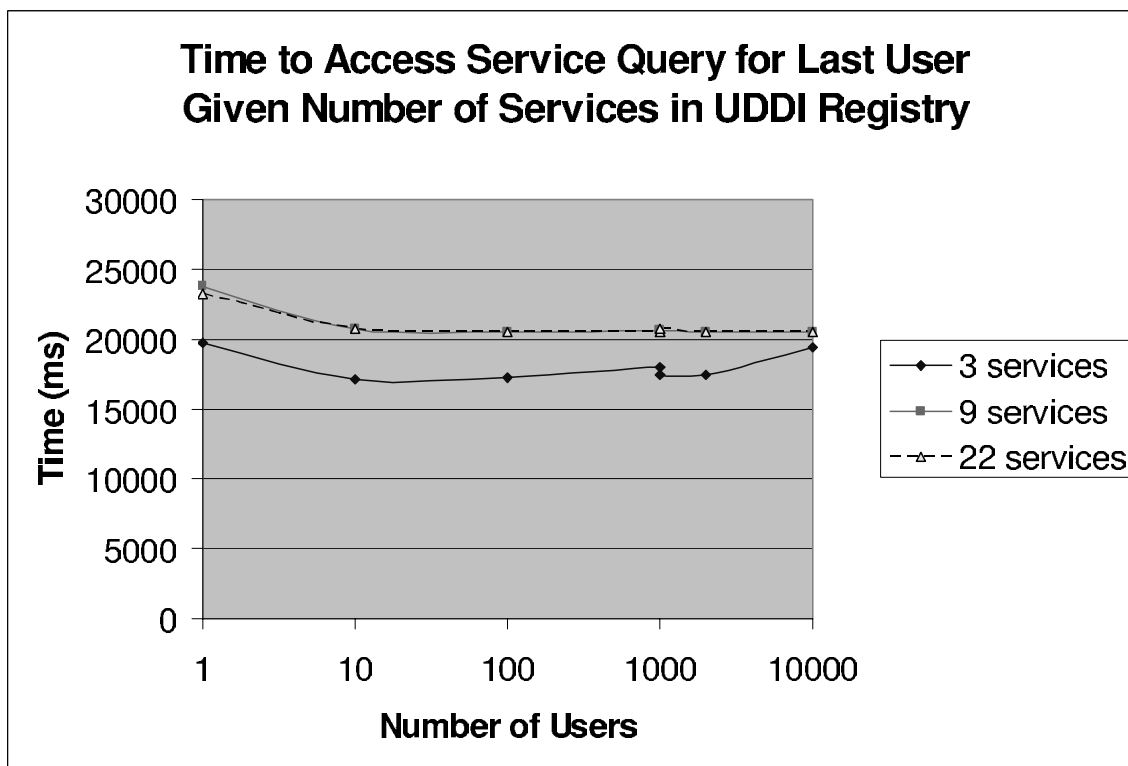


Figure 5.2: Time to access service for last user with UDDI

From Figure 5.2, it can be seen that the time for the last user to access a service is independent of the total number of users in the system as shown by the fairly straight horizontal line. As the number of services in the UDDI registry increases, the time to access the service query by the last user also increases. The time to access a service is nearly the same for 9 services and 22 services in UDDI. We can see that it takes more time to access the service with 1 user in the system due to the initial overhead in setting up the tables. Compared to Figure 5.1 with accessing a service without UDDI, the difference is around 40 fold, which is extremely large. This means that there is an overhead in accessing the UDDI registry remotely. In the future, it would be advantageous to investigate having a local UDDI registry for each location to determine the savings in time compared to a central UDDI registry implementation.

5.2.2 Space Experiments

The next category of experiments involves measuring the space required in the implementation of our architecture to accommodate the users in the system. The tables in our architecture are the Proxy Table, Client-Proxy Table, DPC Table and User-DPU Table. The tables are implemented using hash tables in the Java programming language. These results are an average of five trials.

5.2.2.1 Size of proxy tables

For estimating space requirements, we calculate the size of the tables in our system architecture. Each user is assigned a DPU in the DPC Table from the list of available DPUs from the User-DPU Table (which decreases the size of the User-DPU Table).

From the DPU, the Proxy Manager assigns an available proxy from the Proxy Table (which decreases the size of the table) and records this in the Client-Proxy Table for that user. The space required for N users is then the sum of all the sizes of the four tables. The graph of the space required against the number of users is illustrated in Figure 5.3.

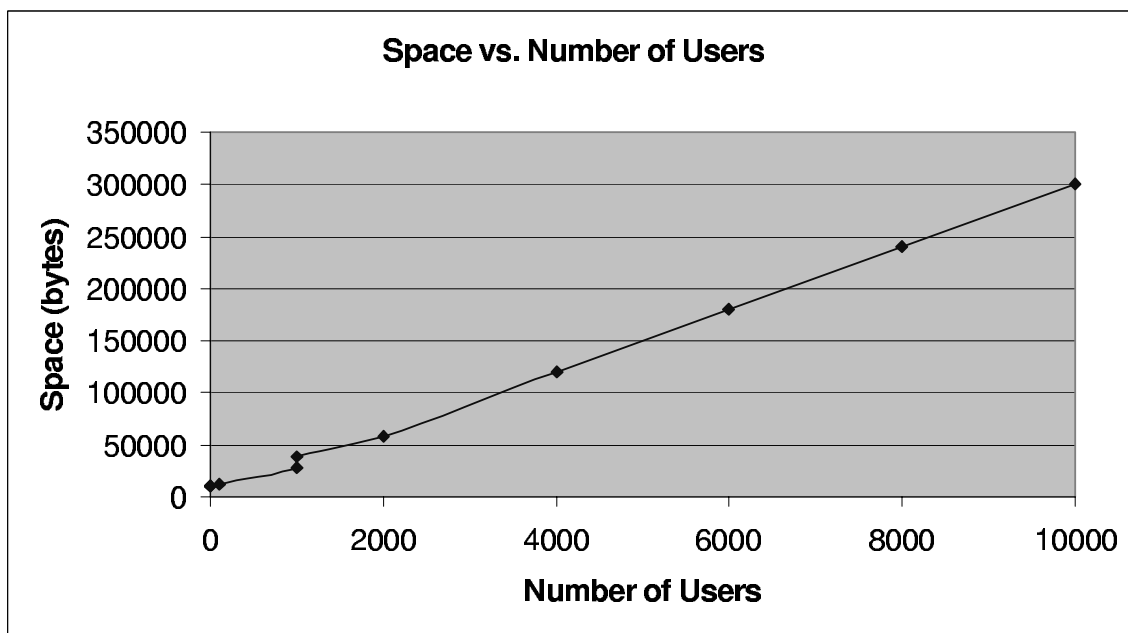


Figure 5.3: Space required for N number of users

From the graph, the size of the tables grows linearly with the number of users which is expected. The maximum size is 300K of table space. The size of the Client-Proxy Table for each DPU of 1000 users is 17151 bytes, and for the Proxy Table it is 1156 bytes. We did not calculate the size of the proxy queues as part of the space requirements. This is because these are dependent on the size of the Sent and Processed messages which are variable depending on the query request and service response.

5.3 Summary and Conclusion

This chapter has presented the experiments to test the performance of our architecture. The experiments were divided into time and space. For the time experiments, we determined how long it took to access a service query with UDDI and without UDDI, depending on the total number of users in the system. It was found that the time to access a service is independent of the number of users. There is some overhead involved in using a remote UDDI registry for accessing services. Space increases linearly with the number of users, as one would expect.

The next chapter is Appendix A which illustrates the screen shots that were used for our case study scenarios.

Chapter 6

Conclusions

One of the challenges of today's pervasive-computing environments, is for a user to be able to continue accessing services from where that user left off, in a different location and on a different wireless network, without having to start the service over. This thesis has presented an architecture, framework and protocols that contribute towards addressing some of these challenges.

6.1 Thesis Overview and Findings

This thesis proposed an end-to-end system architecture that integrates existing components and work in the distributed-and-mobile-computing areas, to support mobile users in a pervasive computing environment. The proposed architecture aimed for ease of use, while maintained a high degree of scalability and flexibility. Our contributions were the following:

- 1) service roaming was achieved at a higher level independent of the lower levels;

2) an architecture was designed and a framework was developed for integrating enterprise services with the addition of service invocation and roaming in a mobile wireless environment; and

3) service roaming can be deployed in an enterprise environment relatively easily with existing technologies.

The heart of the system architecture is m-Roam, our service-invocation-and-roaming framework, which hides the failures, disconnections and migrations from the user. The DPCs and DPUs form the roaming infrastructure, whereas it is the proxy with the proxy FSM and proxy-instance FSM, that is responsible for the insulation of those disruptions.

Use cases were designed to illustrate the usage of our architecture and framework driven from the enactment of the user. Messaging protocols were developed to show the underlying communications and interactions among and between the various components of our design architecture. Experiments were conducted to test the performance of our architecture and framework. The results showed that our system can support many users without much performance degradation, has a linear relationship between space for the proxy tables and number of users, and validates that our architecture works. Our case study was incorporated into the architecture, framework, messaging protocols, and use cases in order to illustrate the applicability to a real-life scenario, which in this case was a conference environment.

6.2 Future Work

Currently, the system only allows for keyword-based search queries for services. One area for future work is to extend our system for more intelligent search queries that involve a natural language query, and use DAML-S and ontologies to add these semantics. In this case, UDDI would have to be extended to allow for DAML-S ontologies to be added. This would involve having to map DAML-S attributes to UDDI records, and creating a DAML-S matchmaking engine that would locate the appropriate service directly using a DAML-S query or by translating the DAML-S query into a UDDI query on the UDDI registry.

Another area for future work would be to support user profiles which would have user preferences like a user's quality of service for a particular service, and the types of services that a user would want to access based on the context of where the user is located and what the user is doing. These user profiles can be integrated into an enterprise-wide network through the existing use of LDAP (Lightweight Directory Access Protocol). Security in our architecture and framework is very simple through the use of user authentication with user name and password, and access to services through a simple service authorization table. RADIUS can be employed to support user authorization and authentication for enterprise security.

Appendix A

Case Study Screen Shots

This appendix illustrates the screen shots of the browser interface that a typical user would see and interact with for the case study that was introduced in Section 1.3. The framework is implemented using the Java programming language and consists of two DPCs. The first DPC is located in Second Cup and has two DPUs, with each DPU having 5 proxies. The second DPC is located in Starbucks and has one DPU with 5 proxies.

The screen shots are also used to provide the visual output to explain the architecture and the use cases as described in chapters 3 and 4 respectively. From the scenarios in sections 1.3.1 and 1.3.2, we describe the steps that Alvin and Cynthia take in order to accomplish their particular tasks. Finally, we provide some concluding remarks from the screen shots of our case study.

A.1 Checking Conference Agenda

First, Alvin logs into the conference site using his username and password, and selects the **Sign In** button as shown in Figure A.1.



Figure A.1: Logging into conference web site

Once successfully logged in, Alvin clicks the **Find all services in this location** button as shown in Figure A.2 which locates all the services in Second Cup.



Figure A.2: Finding all services in Second Cup

The **E-mail**, **Local news** and **Local weather** services are all available in Second Cup as illustrated in Figure A.3.



Figure A.3: List of services in Second Cup

From here, Alvin selects the **Local news** service and a list of Conference news and Halifax news is displayed as in Figure A.4.



Figure A.4: Selecting the local news service

Alvin selects the conference news by choosing the **Conference news** button and the conference agenda is displayed to him as in Figure A.5.



Figure A.5: Accessing the conference agenda

Alvin gets interrupted by his cell phone. Upon his return, the Palm m505 has lost the 802.11b wireless connection and his session has timed out. Alvin logs back in as in Figure A.1 and he has a previous session open as shown in Figure A.6.



Figure A.6: Accessing previous disconnected session

He wants to continue accessing the conference agenda from where he left off, so he selects **Existing session**. From here, the system recovers from his disconnection and displays the conference agenda to him, as illustrated in Figure A.7.



Figure A.7: Continuing existing conference agenda

A.2 Instant Messaging and Checking the Weather

On the way to the conference, Cynthia finds a Starbucks coffee shop and uses her Bluetooth on her Palm Vx, to try out the new Bluetooth network there. She logs into the conference site with her username and password as shown in Figure A.8.



Figure A.8: Cynthia logging into conference site from Starbucks

She then proceeds to find a list of all services available in Starbucks and selects **Find all services in this location** from the page displayed in Figure A.9.



Figure A.9: Find all services in Starbucks

A list of all services in Starbucks which are **E-mail**, **Local traffic**, **Instant messaging**, and **Local weather** are returned from the system and displayed to her as shown in Figure A.10.



Figure A.10: List of services available in Starbucks

From here, Cynthia wanting to talk to one of her friends at the conference, realizes there is a **Instant messaging** service and selects that, as displayed in Figure A.11.



Figure A.11: Selecting the instant messaging service

The instant-messaging service starts and Cynthia realizes Jennifer is at the conference and is online. She selects **Jennifer** from the users' list, and writes a message asking whether she wants to go out tonight in Dartmouth, then clicks the **Send** button. This is shown in Figure A.12.



Figure A.12: Starting a chat with Jennifer

The instant-messaging service sends the request to Jennifer, who replies back to Cynthia as indicated in Figure A.13.

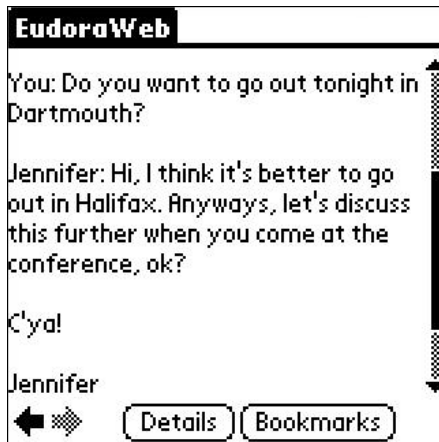


Figure A.13: Chat response from Jennifer

Cynthia then clicks **Back to services**, giving her the list of services (Figure A.10), then selects the **Local weather** service. The screen appears as in Figure A.14.



Figure A.14: Selecting the weather service

Cynthia wants to see the weather for Dartmouth so she selects the **Weather for here** button and receives the weather forecast for Dartmouth. This is displayed in her browser as shown in Figure A.15. From here, she can go back to the weather service, or back to all services.



Figure A.15: Getting the weather for Dartmouth

Cynthia arrives at the conference, and meets with Jennifer for a session break at Second Cup. Knowing that the planned event is now going to take place in Halifax instead, she wishes to check the weather there. She logs into the conference site as shown in Figure A.16 with the 802.11b module for her Palm Vx.



Figure A.16: Cynthia logging into the conference site from Second Cup

The system realizes that she has an existing session from Starbucks as in Figure A.17 and displays this to her.



Figure A.17: Existing session open from Starbucks for Cynthia

She clicks on **Existing session** to continue where she left off. Since the location has changed from Starbucks in Dartmouth to Second Cup in Halifax, her last query (which was weather for here) has to be resent to the system because the last service response (weather for Dartmouth) is now invalid here. The result of this resent query is the updated weather for Halifax as shown in Figure A.18.

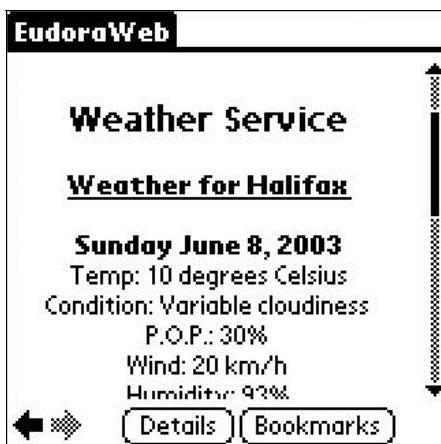


Figure A.18: Updated weather from the system for Halifax after roaming

Bibliography

- [1] Arup Acharya, T. Imielinski, and B. R. Badrinath. DATAMAN Project: Towards a Mosaic-like Location Dependant Information Service for Mobile Clients. Technical Report TR-320, Department of Computer Science, Rutgers University, 1994.
- [2] Apache Tomcat. The Apache Jakarta Project, Apache Software Foundation, <http://jakarta.apache.org/tomcat/>.
- [3] Paolo Bellavista, Antonio Corradi, and Cesare Stefanelli. The Ubiquitous Provisioning of Internet Services to Portable Devices. *IEEE Pervasive Computing*, 1(3):81–97, July-September 2002.
- [4] Sheng-Tzong Cheng, Jian-Pei Liu, Jian-Lun Kao, and Chia-Mei Chen. A New Framework for Mobile Web Services. In *Proc. 2002 Symposium on Applications and the Internet (SAINT) Workshops*, pages 218–222. IEEE, January 28-February 1, 2002.
- [5] Kelvin H. Cheung. A Customizable Web Services Integration Environment. Master’s thesis, University of Waterloo, 2002.

- [6] Cecilia Corbi and Giuseppe Sisto. A Directory Enabled Solution for Internet Roaming. In *Proc. IEEE International Symposium on Computers and Communications*, pages 39–45. IEEE, 1999.
- [7] DAML Services. DAML Coalition, <http://www.daml.org/services/>.
- [8] A. Calvagna et al. WiFi Bridge: Wireless Mobility Framework Supporting Session Continuity. In *Proceedings of the First International Conference on Pervasive Computing and Communications (PerCom'03)*, pages 1–8. IEEE, 2003.
- [9] Francisco Curbera et. al. Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–92, March-April 2002.
- [10] Hiroshi Sawada et al. Inter-Network Roaming Based on Personal Digital Cellular Standards. In *Proc. GLOBECOM '93*, volume 3, pages 1944–1949. IEEE, December 1993.
- [11] Kisup Kim et al. A Distributed Proxy Server System for Wireless Mobile Web Service. In *Proc. 15th International Conference on Information Networking (ICOIN'01)*, pages 749–754. IEEE, 2001.
- [12] R. Bagrodia et al. iMASH: Interactive Mobile Application Session Handoff. In *Proceedings of MobiSys 2003: The First International Conference on Mobile Systems, Applications, and Services*, pages 259–272. The USENIX Association, 2003.

- [13] Ronald Beaubrun et al. Global Roaming Management in Next-Generation Wireless Systems. In *Proc. IEEE International Conference on Communications 2002*, volume 4, pages 2070–2074. IEEE, 2002.
- [14] Sheila A. McIlraith et al. Semantic Web Services. *IEEE Intelligent Systems*, pages 46–53, March-April 2001.
- [15] Suman Banerjee et al. Rover: Scalable Location-Aware Computing. *IEEE Computer*, 35(10):46–53, October 2002.
- [16] Theodore B. Zahariadis et al. Global Roaming in Next-Generation Networks. *IEEE Communications Magazine*, 40(2):145–151, February 2002.
- [17] A. Festag, H. Karl, and G. Schafer. Current Developments and Trends in Handover Design for ALL-IP Wireless Networks. Technical Report TKN-00-007, Telecommunication Networks Group, Technical University Berlin, August 18, 2000.
- [18] Kyle Gabhart and Jason Gordon. Wireless Web Services with J2ME. *Web Services Journal*, pages 44–48, January 2002.
- [19] Marty Hall. Core Servlets and JavaServer Pages. <http://www.coreservlets.com>.
- [20] J. Hightower and G. Borriello. Location Systems for Ubiquitous Computing. *IEEE Computer*, 34(8):57–66, August 2001.
- [21] IBM Emerging Technologies Toolkit (ETTK). IBM Corporation, <http://www.alphaworks.ibm.com/tech/ettk>.

- [22] IBM Pervasive Computing. IBM Corporation, <http://www-3.ibm.com/software/pervasive/index.shtml>.
- [23] IBM UDDI Business Test Registry. IBM Corporation, <https://uddi.ibm.com/testregistry>.
- [24] IBM Websphere Everyplace. IBM Corporation, <http://www-3.ibm.com/software/pervasive/products/index.shtml>.
- [25] IS 2003: 13th Annual Canadian Conference on Intelligent Systems. Precarn Inc., http://oldwww.tomoye.com/simplify/precarn/ev.php?URL_ID=1986&URL_DO=DO_%TOPIC&URL_SECTION=201&reload=1008704170.
- [26] JAIN API. Sun Microsystems, <http://java.sun.com/products/JAIN>.
- [27] JBoss 3.0.4. The JBoss Group, LLC, <http://www.jboss.org>.
- [28] Jini Network Technology Specification v1.2. Sun Microsystems, <http://www.sun.com/software/jini/specs/index.html>.
- [29] Rui Jose, Adriano Moreira, Filipe Meneses, and Geoff Coulson. An Open Architecture for Developing Mobile Location-Based Applications over the Internet. In *Proc. 6th IEEE Symposium on Computers and Communications*, pages 500–505. IEEE, July 3-5, 2001.
- [30] Tim Kindberg and John Barton. A Web-based Nomadic Computing System. HP Cooltown whitepaper, Hewlett-Packard Laboratories, <http://www.cooltown.hp.com/dev/wpapers/nomadic/nomadic.asp>.

- [31] Dik Lun Lee, Jianliang Xu, Baihua Zheng, and Wang-Chien Lee. Data Management in Location-Dependent Information Services. *IEEE Pervasive Computing*, 1(3):65–71, July-September 2002.
- [32] Qiyang Li. An Architecture for Geographically-Oriented Service Discovery on the Internet. Master’s thesis, University of Waterloo, 2002.
- [33] Finite State Machine. National Institute of Standards and Technology. <http://www.nist.gov/dads/HTML/finiteStateMachine.html>.
- [34] Richard Monson-Haefel. *Enterprise JavaBeans*. O’Reilly and Associates, 2001.
- [35] Nathan J. Muller. *Bluetooth Demystified*. Mc-Graw Hill, 2001.
- [36] Parlay API. The Parlay Group, <http://www.parlay.org>.
- [37] Charles E. Perkins. Mobile Networking Through Mobile IP. IEEE Internet Computing Online, <http://www.computer.org/internet/v2n1/perkins.htm>.
- [38] Salil Pradhan. Semantic Location. HP Cooltown whitepaper, Hewlett-Packard Laboratories, <http://www.cooltown.hp.com/dev/wpapers/semantic/semantic.asp>.
- [39] Golden G. Richard. Service Advertisement and Discovery: Enabling Universal Device Cooperation. *IEEE Internet Computing*, 4(5):18–26, September-October 2000.
- [40] Salutation Consortium. <http://www.salutation.org>.

- [41] M. Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, 8(4):10–17, August 2001.
- [42] Henning Schulzrinne and Elin Wedlund. Application-Layer Mobility Using SIP. *Mobile Computing and Communications Review*, 1(2):1–9, July 2000.
- [43] Mark Alexander Connell Snoeren. *A Session-Based Architecture for Internet Mobility*. PhD thesis, Massachusetts Institute of Technology, February 2003.
- [44] Vince Stanford. Pervasive Computing Goes To Work: Interfacing to the Enterprise. *IEEE Pervasive Computing*, pages 6–12, July-Sept 2002.
- [45] Mark Stemm and Randy H. Katz. Vertical Handoffs in Wireless Overlay Networks. *ACM Mobile Networks and Applications*, 3(4):335–350, 1999.
- [46] Edmund Sung and Arkady Zaslavsky. Software Assisted Handover of Mobile Clients in Heterogeneous Wireless Computing Environments. In *Proceedings of APSEC '97 and ICSC '97*, pages 527–528. IEEE, December 1997.
- [47] Andrew S. Tanenbaum. *Computer Networks*, page 29. Prentice-Hall, Third edition, 1996.
- [48] UPnP Forum. <http://www.upnp.org>.
- [49] Jim Waldo. Alive and Well: Jini Technology Today. *IEEE Computer*, 33(6):107–109, June 2000.
- [50] Mark Weiser. The Computer for the 21st Century. *IEEE Pervasive Computing*, 1(1):18–25, January-March 2002.

- [51] Jon Chiung-Shien Wu, Chieh-Wen Cheng, Nen-Fu Huang, and Gin-Kou Ma. Intelligent Handoff for Mobile Wireless Internet. *ACM Mobile Networks and Applications*, 6(1):67–79, 2001.
- [52] Bin Yao and W. Kent Fuchs. Recovery Proxy for Wireless Applications. In *Proc. 12th International Symposium on Software Reliability Engineering (ISSRE 2001)*, pages 112–119. IEEE, 2001.
- [53] Bin Yao and W. Kent Fuchs. Proxy-based Recovery for Applications on Wireless Hand-held Devices. In *Proc. 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, pages 2–10. IEEE, October 16-18, 2000.
- [54] Ying Zou and Kostas Kontogiannis. Migrating and Specifying Services for Web Integration. In *Engineering Distributed Objects, Second International Workshop, EDO 2000, Davis, CA, USA, November 2-3, 2000, Revised Papers*, volume 1999, pages 253–270. Springer-Verlag Heidelberg, 2001.