NOTE TO USERS

This reproduction is the best copy available.

UMI[®]

Instant Messaging-Based Networked Service Provisioning and Access Framework

by

Liping Zhao

A thesis submitted to the Faculty of Graduate Studies
in partial fulfillment of the requirements
for the degree of

Master of Science

Ottawa-Carleton Institute for Electrical and Computer Engineering
Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6

March 2004

© 2004, Liping Zhao

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-93994-4

Our file Notre référence

ISBN: 0-612-93994-4

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

ABSTRACT

As wireless devices continue to proliferate, more and more services will be created for use in the wireless environment. Whenever a wireless device moves into a new environment, potential services, which we call the networked services, will become available. Those networked services should be accessible to users to find and use, not only statically through the existing connection, but also dynamically. Due to the limitations of wireless handheld devices, such as little memory, low processing power and small screens, the current protocols or mechanisms targeting desktop applications do not suit the wireless situation. Some of them rely on the web browser, while others have to implement a full set of protocol stack. Therefore, we need a lightweight protocol that is simple, user-friendly and efficient, in order to meet the end user's minimum requirement. With respect to the overall network architecture, to avoid reinventing the wheel we should take advantage of the current existing infrastructure, with a few modifications. This thesis proposes a generic service-provisioning framework that is based on the protocol of instant messaging and presence service, and combines a Web Services categorization mechanism coupled with a generic service deployment schema. It specifies both the network infrastructure and the relevant service presentation scheme. One of the possible solutions is given under the newly defined Instant Messaging and Presence SIP extension (SIMPLE). Through our innovative service provisioning framework, we hope to illustrate a feasible prototype that provides inspiration for further development.

ACKNOWLEDGEMENT

I would like to express my deepest gratitude to my supervisor, Professor Babak Esfandiari, for his constant guidance, patience, inspiration, advice and encouragement throughout the research work and the preparation of this thesis. I really appreciate it.

I also want to sincerely thank Dr. Raul San Martin from Eion Inc. and NCIT for trusting and supporting me during this process.

Finally, I would like to thank my family for their support, understanding and love during my study and research.

Table of Contents

CHAPTER 1 INTRODUCTION	1
1.1 Networked Service	1
1.2 Motivation.....	2
1.3 Instant Messaging – a successful communication tool	3
1.4 Contributions.....	4
1.5 Organization of the Thesis	5
 CHAPTER 2 STATE OF THE ART: RELATED WORKS AND TECHNOLOGIES.....	 6
2.1 Existing User-Oriented Service Provisioning Application.....	6
2.1.1 Overview of Relevant Solutions	6
2.1.2 User Interface	8
2.1.3 Transport Media	12
2.1.4 Service Discovery	16
2.1.5 Overall Architecture	18
2.1.6 Conclusion	20
2.2 Instant Messaging and Presence Protocols.....	22
2.2.1 RFC 2778 and RFC 2779.....	22
2.2.2 IETF Instant Messaging/Presence Protocol (IMPP)	24
2.2.3 Wireless Village.....	25
2.2.4 Jabber & XMPP.....	27
2.2.5 SIMPLE	33
2.2.6 JMS.....	37
2.2.7 SMS and MMS	38
2.2.8 Conclusion	40
2.3 Service Lookup and Location Protocols	41
2.3.1 Service Location Protocol.....	42
2.3.2 JINI.....	44
2.3.3 Web Service and UDDI/UDDIe	47
2.3.4 DAML-S.....	51
2.3.5 Others.....	54
2.3.6 Conclusion	56
2.4 Summary.....	58

CHAPTER 3 IM-BASED NETWORKED SERVICE PROVISIONING AND ACCESS FRAMEWORK.....	59
3.1 IM-Based Networked Service Provisioning Architecture	59
3.2 Networked Service Categorization and Discovery.....	63
3.3 Generic Networked Service Invocation Mechanism.....	66
3.3.1 XML and XML Schema	67
3.3.2 Networked Service Access Information Schema.....	68
3.3.3 Networked Service Access Mechanism	71
3.4 Summary.....	73
CHAPTER 4 DESIGN AND IMPLEMENTATION DETAILS	75
4.1 Requirements for Small Wireless Device Applications	75
4.2 J2ME & J2ME APIs	76
4.2.1 Profile.....	77
4.2.2 Configuration	78
4.2.3 KVM.....	79
4.2.4 Restrictions of CLDC & MIDP	79
4.3 Design and Implementation Background.....	81
4.4 Service IM Proxy Deployment.....	82
4.5 SIMPLE Client Design and API Definition in J2ME	85
4.5.1 SIMPLE Overview.....	85
4.5.2 SIMPLE Messages	89
4.5.3 Description of SIMPLE Client Functionality.....	92
4.5.4 SIMPLE Client Implementation	94
4.5.5 SIMPLE Client API.....	102
4.6 SIMPLE Server.....	111
4.7 Message Parsing and User Interface	111
4.8 Summary.....	113
CHAPTER 5 CASE STUDY	114
5.1 UDDI / UDDIe Server	114
5.2 Networked Service – Google	115

5.3	An Example of Networked Service Discovery and Access	115
5.4	Related Statistics in Case Study.....	126
5.5	Summary.....	128
CHAPTER 6 SUMMARY AND FUTURE WORK.....		129
6.1	Summary.....	129
6.2	Future Work.....	130
REFERENCES.....		132

Table of Figures

Figure 2-1 Google Access through SOAP	13
Figure 2-2 sView Architecture	18
Figure 2-3 ActiveBuddy Architecture	19
Figure 2-4 Presence Service in RFC 2778.....	23
Figure 2-5 Instant Messaging Service in RFC 2778.....	24
Figure 2-6 Wireless Village Protocol Stack	26
Figure 2-7 Jabber IM System Architecture.....	28
Figure 2-8 Jabber IM Message Flow	29
Figure 2-9 Jabber Presence Service Message Flow	30
Figure 2-10 SIMPLE IM Message Flow	34
Figure 2-11 SIMPLE Presence Message Flow.....	36
Figure 2-12 SLP Architecture.....	43
Figure 2-13 Interactions in Jini	45
Figure 2-14 UDDI Protocol Stack.....	48
Figure 2-15 XML structure defined in UDDI.....	48
Figure 2-16 Top Level of DAML-S Ontology.....	52
Figure 3-1 Structure of Service IM Proxy	61
Figure 3-2 Architecture of our framework.....	65
Figure 3-3 Interactions between IM entities.....	66
Figure 4-1 J2ME Structure	79
Figure 4-2 Service Proxy Deployment Architecture.....	84
Figure 4-3 SIMPLE Presence Service	88

Figure 4-4 SIMPLE Client Deployment Diagram	96
Figure 4-5 Main Activity Diagram of SIMPLE Client	97
Figure 4-6 Overall Class Diagram of SIMPLE Client.....	98
Figure 4-7 SIMPLE Client State Chart	101
Figure 5-1 Interactions for Google service discovery and access	116
Figure 5-2 Start GLUE UDDI server	116
Figure 5-3 <tModel> of Google service registration	117
Figure 5-4 Google service UDDI registration	118
Figure 5-5 Register UDDI proxy with SIMPLE Atrium Server.....	119
Figure 5-6 Start SIMPLE Client (left) & Subscribe to UDDI proxy (right)	120
Figure 5-7 Get Service List.....	121
Figure 5-8 Google Service Access Template	122
Figure 5-9 Display Google Service (left) & Generate GUI for Service Access.....	123
Figure 5-10 Message Details for Google Service Access.....	124
Figure 5-11 Returned result from Google proxy	125
Figure 5-12 Result received by the SIMPLE Client	126

List of Tables

Table 2-1 Relevant Work Comparison	21
Table 2-2 IM protocol comparison	40
Table 4-1 SIP/SIMPLE Message	89
Table 4-2 SIP/SIMPLE Message Headers	90
Table 4-3 SIP/SIMPLE Response Classes	91
Table 4-4 Headers Presence in SIMPLE Messages	92
Table 4-5 SIMPLE Client Functions.....	94
Table 4-6 SIMPLE client Class Description	100
Table 4-7 Description of the SIMPLE Client States	101
Table 4-8 Relevant JSRs overview	103
Table 4-9 JAIN SIMPLE API (JSR 164/165)	105
Table 4-10 SIP API for J2ME.....	107
Table 4-11 SIMPE Client API	110
Table 4-12 Comments on SIMPLE Client API Design.....	111
Table 5-1 Statistics for Case study	127

List of Acronyms

API	Abstract Programming Interfaces
CDC	Connected Device Configuration
CLDC	Connected Limited Device Configuration
CPIM	Common Presence and Instant Messaging
DA	Directory Agent
DAML-S	DARPA Agent Markup Language for Services
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DOM	Document Object Model
GSM	Global Standard for Mobiles
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IANA	Internet Assigned Numbers Authority
IM	Instant Messaging
IMPP	Instant Messaging/Presence Protocol
IMPS	Instant Messaging and Presence Service
J2EE	Java 2 Platform Enterprise Edition
J2ME	Java 2 Platform Micro Edition
J2SE	Java 2 Platform Standard Edition
JAIN	Java API for Integrated Networks
JCP	Java Community Process
JMS	Java Message Service

JSR	Java Specification Request
MIDP	Mobile Information Device Profile
MIME	Multipurpose Internet Mail Extension
MMS	Multimedia Message Service
MOM	Message-Oriented Middleware
OIL	Ontology Inference Layer
PA	Presence Agent
PSE	Personal Service Environment
PUA	Presence User Agent
QoS	Quality of Service
RDF	Resource Description Framework
RMI	Remote Method Invocation
SA	Service Agent
SASL	Simple Authentication and Security Layer
SAX	Simple API for XML
SDP	Service Discovery Protocol
SIMPLE	Instant Messaging and Presence Leverage Extensions
SIP	Session Initiation Protocol
SLP	Service Location Protocol
SMS	Short Message Service
SMSC	Short Message Service Center
SOAP	Simple Object Access Protocol
SS7	Signaling System 7

SSP	Server-to-Server Protocol
TLS	Transport Layer Security
TLS	Transport Layer Security
UA	User Agent
UA	User Agent
UAC	User Agent Client
UAS	User Agent Server
UDDI	Universal Description, Discovery and Integration
UDDIe	UDDI extension
UPnP	Universal Plug and Play
URL	Universal Resource Locator
WAP	Wireless Application Protocol
WDP	Wireless Datagram Protocol
WSDL	Web Service Description Language
WSP	Wireless Session Protocol
WTLS	Wireless Transport Layer Security
WTP	Wireless Transaction Protocol
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol

Chapter 1 Introduction

It is predicted that the next wave in the IT industry will be associated with wireless devices and their applications. In the wireless environment, it is becoming more and more demanding to discover and access the surrounding services in an efficient way. Within this context, we proceed with the work described later in this document.

This chapter will introduce some basic concepts, motivations and contributions covered in the thesis.

1.1 Networked Service

With today's rapid growth of the Internet and wireless LAN, more and more resources are breaking free from their physical locations. Resource devices are grouped according to the logical or network location and tend to be shared among different users in a distributed manner. For example, hundreds of PCs in a lab can share five printers and three scanners, as long as all the devices are properly configured. Each resource can be treated as a network node that is capable of communicating with other network peers. The term "networked service" therefore refers to the functionality supported by the hardware device through certain type of network connection. This network architecture enables maximum usage of the available resources and therefore reduces the operational cost. In such an environment, wireless devices are able to access resources almost anywhere, at any time. Due to the mobility of the wireless device, the networking service is not bound to a fixed number of users. When a mobile user moves to a new network, the associated services should be obtained dynamically.

1.2 Motivation

In order to take advantage of wireless and mobile devices, users need to know what services are available, anywhere and at any time. For example, a mobile user with appropriate authorization should be able to access an online dictionary while he is attending a conference meeting. Considering that the user may not have prior awareness of those networked services, the ability to find and use those services is becoming more and more critical. A mechanism that supports both service discovery and access in a real-time manner is preferred. For such a mechanism to be realized, the service provider requires an efficient and generic framework that is compatible with all types of services. It should not only target the services currently available, but also focus on the extensibility and adaptability of new services, so that the new services can easily be attached. For the end user, this means that more and more value-added services will be accessible through one common interface. Therefore, protocols are needed that support both the communication and the networked service discovery. Especially within the wireless environment, the limitations of the wireless device and the mobility of the users both serve to increase the complexity. It requires a fast response, efficient and user-friendly inquiry of the available services and a relatively simple mechanism as well.

So far a few attempts have been made to address this problem. (Some of them will be reviewed later, in Chapter 3.) However, none of them fits well with the above requirements. This thesis will try to propose a feasible and complete solution with respect to service provisioning and service access. In addition to this, there still exist some common concerns such as security problems, privacy issues and context learning possibilities, etc, but those issues will have to be left for future work.

1.3 Instant Messaging – a successful communication tool

Instant Messaging (IM) is increasingly becoming part of our daily lives. It is primarily used for social communication and entertainment, and is mainly centered on having fun and building a community. IM occupies a very interesting niche between e-mail and telephony, taking the best of both worlds:

➤ **Instant response**

While e-mail lacks the warmth of real-time interactions, IM links people instantaneously, providing a quick sense of presence and activity. Though as instant as a phone call, it doesn't require as much attention and can be more convenient.

➤ **Presence availability**

The presence detection feature is unavailable with telephone or e-mail communication. IM allows users to find out who is available to receive an instant message, e-mail or even a phone call.

➤ **Non-intrusive**

IM is a very convenient way to communicate quietly without disturbing others, while maintaining privacy. It has proven that it is less intrusive than phone calls, and more efficient than email.

➤ **Inexpensive**

IM, unlike phone calls, is inexpensive, as users can download the software for free from the Internet.

The future of IM is very bright. Millions of people currently use free IM software every day. By the end of 2002, AOL Instant Messenger (AIM) alone had nearly 100 million users worldwide [1]. In terms of revenue, the market is expected to grow from \$7.6

million to \$1.9 billion by 2003, according to IDC. Gartner Group expects that messaging will be the No.1 data revenue source for carriers for the next 5 years (39% in 2002 and 62% in 2005) [2].

With regard to the wireless network, more and more consumers are asking for mobile instant messaging and presence services (IMPS). “Chief among the technologies consumers are asking for is mobile IM and presence services. Research Portal.com reports IM is the No. 2 requested application after voice,” Wireless Village stated in a report [3].

With the monumental growth patterns of Short Message Service (SMS) and the fast adoption rate of desktop instant messaging, it is very likely that the wireless IMPS will capitalize on both these trends. As mobile devices become more powerful, with better CPU, larger screen and more memory, it is predicted that the IM world will center on wireless users in the near future.

1.4 Contributions

The advantages of wireless instant messaging are obvious. It takes place in an instantaneous fashion; it requires relatively short delivery latency; its notion of presence service provides authorized users with knowledge of the availability of their correspondents. In this research, we were interested in several questions. Could the main concept of instant messaging and presence service be adopted and applied to non-human networked services? How could the entities of the IM world be mapped into the service provisioning architecture? Which instant messaging protocol would fit our situation better and have the brightest future in the industry? Could networked services be viewed and categorized in the same way as Web services? Is the mechanism for accessing Web

services applicable to networked services? Are there any other considerations that need to be addressed? The thesis will focus on those issues and try to provide some acceptable answers.

The following contributions are made in the thesis:

- Compare the existing protocols for instant messaging and Web service description and discovery;
- Compare the existing Java Abstract Programming Interfaces (APIs) for the related messaging protocol and define a new set of Java APIs for the instant messaging protocol in J2ME;
- Provide an implementation of the instant messaging protocol under J2ME for wireless devices such as cell phones;
- Introduce a network architecture based on instant messaging communication and presence service;
- Propose a generic mechanism for the access and deployment of networked services.

1.5 Organization of the Thesis

The rest of the thesis is structured as follows. Chapter 2 analyzes the state of the art of our proposed framework and introduces the rationale behind it. A detailed review of the relevant literature will also be provided. Chapter 3 proposes our Instant Messaging-based service provisioning and access framework. Chapter 4 elaborates the details of our sample design and implementation, which is based on SIMPLE and J2ME. Chapter 5 provides a concrete example, showing how our proposed framework can be realized in practice. The summary and future work are given in Chapter 6.

Chapter 2 State of the Art: Related Works and Technologies

In order to tackle the problems described in the previous chapter, this chapter will review some reference approaches, as well as the most related technologies.

2.1 Existing User-Oriented Service Provisioning Application

As the focus of our research work is to help the user access services, the application must be easy to operate; in other words, it must be user-oriented. This section summarizes the related solutions in terms of the user interface, the transport media, the service discovery approach and the overall architecture. The analysis and the relevance to our work will be discussed in each section.

2.1.1 Overview of Relevant Solutions

This section will briefly introduce the background of some commercial products and research projects that are close to the topic of this thesis. The detailed analysis and comparison will be given later.

- **ActiveBuddy**

ActiveBuddy Inc., founded in March 2000, provides software for developing and hosting interactive agents. The interactive agent is a customized software application, often called a "bot," which normally interacts with users through a messaging protocol. Users are able to retrieve the desired information by talking to the bot. Currently, there are several bots available, offering services like weather forecasts or stock quotes. The BuddyScript Server is responsible for operating all the interactive agents.

- **IBM's E-mail integration with Web services**

In September of 2003, IBM proposed an architecture that uses e-mail to interact with Web services. The motivation for this is based on the fact that more and more wireless devices have started to support the e-mail exchange functionality. Similar to the traditional Web application access through browser, this architecture wraps all the necessary service invocation information in the headers/content of an e-mail. The e-mail server has to be tightly coupled with the Web service application server.

- **sView**

On top of the previous work of the Open and Adaptive Service Infrastructures group at the Swedish Institute of Computer Science, sView introduces a personalized platform for creating services and using services and a corresponding network structure. The sView system is intended to encourage users to create their own personal services and share them with each other. It consists of three components: a personal service environment (PSE), tools for creating services and a network for service sharing [4]. After creating a service, users can use and store it with PSE, through a specially designed network called "briefcase connectivity". Each user maintains a personal service briefcase, which is just like a folder containing the information of a number of services for future invocation.

- **Agent-Based Mobile Services Access**

The agent approach was previously applied to specific networks, such as the Intelligent Network and the Telecommunication Management Network. With the evolution of Web services and wireless devices, some research groups advocated the idea of using agents to carry and deliver the service called mobile service (m-service) across the network. For

example, [5] and [6] discuss two different agent-based frameworks that support mobile services.

- **Other Solutions**

With XML and SOAP, Web services technology has introduced another service access model by extending the usage of the web browser and its associated protocols. It is considered to be very valuable in the e-business process.

There are other technologies available, some of which use instant messaging similarly to ActiveBuddy. Rather than providing a framework, however, they focus more on providing extra features on existing applications. For example, Googlematic [7] and BotGoogle [8] are deployed for AOL IM and MSN to access the Google search engine.

In the following sections, we will survey the state of the art through a separate discussion of each of the issues we think are important for a user-oriented service-provisioning framework, in terms of user interface, transport media, service discovery, and overall architecture.

2.1.2 User Interface

The user interface defines how the human user interacts with the real service through the wireless device. There are three main ways for the user to communicate with the service: the Web browser, e-mail exchange, and text messaging, especially instant messaging.

- **Web Browser**

Serving as an interaction terminal, the Web browser plays a key role in Web services access. The name indicates how closely Web services are associated with the Internet and its displayable interface – the Web browser. As the information exchange during the Web services access takes place over HTTP and is carried in HTML, it is fairly

straightforward to invoke Web services and publish the related information for the human being via the interface of the Web browser. For example, the user first needs to enter the Web service's URL in the Web browser in order to locate a service on the Internet. This type of interface is typically associated with e-business applications, especially with regard to Web services. Once the Web service has been deployed and linked to a Web server, it can be accessed through a browser by using GET or POST in HTTP.

The Web browser has existed for a long time. Both standards and products are well developed and mature. People are used to interacting with Web browsers. It would be easy to promote and recognize when porting this approach to wireless devices, but it would not be realistic for wireless devices. First, a traditional Web browser, which involves a lot of graphic user interactions and multi-media applications, will cost the small device too much in terms of resources. Second, the HTML format for a small screen, as with the LCD display on a cell phone, should differ from that for a normal desktop Web browser. It may require more work to provide an extra set of compatible formats for wireless devices. For example, the Wireless Application Protocol (WAP) and Wireless Mark-up Language (WML) enabled micro-browser is one such attempt. In this case, it requires every service to maintain two sets of interfaces: one is through normal browsers; another is for micro-browsers. Besides that, WML is so strict that it puts too many constraints on the service application and loses the generality. In addition, the Web browser lacks the notation of presence information.

- **E-Mail exchange**

Given the fact that e-mail has penetrated our daily life, IBM has sought an alternative way to access Web services by extending the traditional e-mail interface [9]. This

architecture wraps all the necessary information in the headers/content of an e-mail. All the information exchanges happen through a specific protocol only between the end user and the server, including the request submission, the session management, interactions and so on.

The following example from [9] illustrates one of the scenarios with Web applications.

⇒ Server sends e-mail to end user, informing him of four options regarding the review meeting.

To:	user@abc.com	Reference ID
From:	server@ibm.com	
Subject:	Review Meeting in 10 min #349	
Body:	Your Review Meeting is in 10 minutes.	
	Options:	
	1=cancel	
	2=notify that you will be 10 min late	
	3=20 min late	
	4=reschedule	

⇒ User replies to the e-mail with option 2.

To:	server@ibm.com
From:	user@abc.com
Subject:	Re: Review Meeting in 10 min #349
Body:	2

The Web application server will route all the incoming requests to the same set of command objects, no matter what the originator and the transmission protocol. It puts extra functional requirements on the server side, since the Web application server should also act as an e-mail server to parse the headers and content. In addition to that, the main drawback comes with the message delivery mechanism in the e-mail system. Because the e-mail server normally does not support notification, there is no automatic detection of

the incoming requests. Both the end user and the server have to rely on periodic pulling or other methods of checking the new requests. I would say that the lack of the presence service and inadequacy of displayable interface for small devices might impede the further development of this proposal.

- **Text Messaging/Instant Messaging**

As instant messaging becomes more and more popular, the industry has realized its potential as a communication tool for network applications.

ActiveBuddy's agent interacts with the user through a text-based, conversational messaging interface, such as instant messaging or Web chat [10]. By sending short messages in an IM user interface, the user is able to retrieve the desired information in a self-served and private manner without knowing the underlying process. The result will be formatted to fit in a conversational window.

MSN Messenger or America Online's AIM client can access the Google search engine by messaging via the Googlematic or BotGoogle agent. After typing in a search string and sending it to googlematic@interconnected.org or botgoogle@hotmail.com, the user then waits for the returned result. Similarly, Yahoo Messenger can perform the Google query by sending an instant message to the YIMGoogle [11] agent.

The above brief review shows that text messaging is welcomed by more and more parties. This is mainly due to its simple, lightweight and user-friendly nature. Because the service agent hides all the service invocation details, it makes the service access totally transparent to the end user. Although this method of service invocation is pretty straightforward, its efficiency is still worth discussing. For instance, given a specific

service invocation parameter, different people may have different perspectives. The background server has to be able to handle various situations. It does lack a mechanism that will accelerate the invocation process and reduce the error possibility at the same time, but we still consider it to be a better candidate than the others. Some modification may be necessary for it to be adopted into real applications, however.

2.1.3 Transport Media

Regardless of the different transport protocols, the service information exchange is usually conducted within one of the following media types: SOAP and XML, MIME, plain Text and mobile agents.

- **SOAP and XML**

The Simple Object Access Protocol (SOAP) proposed by W3C is intended for message exchange. It builds on XML and uses common Internet transport protocols like HTTP to carry the data. It has gradually been recognized by players in the industry, such as Microsoft, and these days is tightly coupled with Web services. The SOAP message contains an envelope, represented by the *Envelope* XML element, a body enclosed by the *Body* XML element which contains the content of the message, and the headers carrying extra information such as authentication, transaction identifier and etc. That information is rarely the interest of the normal end user. In addition, there is a *Fault* element defined to carry the error information in SOAP messages. Under each parent element, SOAP also defines a number of child elements and associated attributes.

ActiveBuddy uses XML for message exchange. However, SOAP will provide more options for available services. Jabber, an open-source IM project, has proposed

transferring SOAP messages through its instant messaging interface [12]. The idea is to wrap and deliver the SOAP XML document within normal Jabber messages.

Figure 2-1 shows how YIMGoogle collaborates with Yahoo IM through Google SOAP API to conduct the search.

Figure 2-1 Google Access through SOAP

This messaging mechanism is simple because SOAP messages are pure text-based and encoded with XML. It is also extensible and flexible due to the nature of XML. It is language and platform independent. Despite those features, however, it does have some shortcomings that need to be addressed. SOAP is based on XML but it defines a series of very restrictive rules for encoding. The encoding section occupies almost half of the W3C's SOAP specification. There are too many rules to enumerate them here. There are about 50 supporting data types defined in SOAP specification. The details can be found at W3C's office web site [13].

To understand those rules and elements, the application needs to be equipped with a powerful SOAP parser. The parsing procedure is resource consuming. The memory size then becomes the bottleneck, especially for small devices. For instance, it costs about 4951K of memory to run a simple stock quote example on J2SE with open source project--JOXA SOAP tool [14]. For the similar example, it takes about 7569K of memory with GLUE SOAP parser [15].

Besides that, some of the data types may not be supported by wireless applications. For example, the float type is not supported by J2ME. Currently, some open source projects are working on porting SOAP parser in J2ME, such as KSOAP [16].

Because SOAP and XML were mainly designed for communication between applications, it is not targeted on the normal end users. Without technical knowledge or a translation layer, the end user is not able to handle the SOAP messages. Therefore, from the end user's point of view, SOAP and XML are not ideal. But SOAP can be supported through other ways which will be discussed as our future work.

- **MIME**

As described in section 2.1.1, IBM's solution recommends using e-mail as the vehicle to carry the service invocation data. Since e-mail is generally encapsulated with Multipurpose Internet Mail Extensions (MIME), it requires that the Web application server be capable of parsing and processing the MIME headers and types related to the service invocation, and respond in the same format. With regard to the example given in previous section, please note that the detailed MIME headers have been omitted for the sake of simplicity.

Although e-mail has been a very successful application and MIME is fairly mature regarding data presentation, the interaction mechanism of the e-mail system is its striking drawback. Under the “pull” mechanism, all the interactions happen in a passive manner, and even service discovery is made into a tough issue. The details were discussed in Section 2.1.2.

- **Plain Text**

Plain text or English is the most straightforward and easiest way for the human user to communicate. In the ActiveBuddy system, the user talks with the server in plain English to get the desired service. Users are able to retrieve the desired information in a self-served way, by sending plain English sentences. The server will translate the user’s request and forward it to the corresponding service component. For example, if a user sends a message asking, “what is today’s temperature?” to the server, the most up-to-date information about the temperature will be returned.

Although this mechanism is very convenient for the human user, it causes too many concerns on the server side. First, the problem arises from the vocabulary, the grammar and the convention. It is very difficult and expensive to implement a language-recognition and learning system. Consequently, the accuracy or the Quality of Service (QoS) will not be guaranteed. On the other hand, this mechanism may survive using simple service applications like the weather forecast, but it cannot be generalized to more sophisticated services at a large scale because it will lose its generality and extensibility.

- **Mobile Agent**

An agent is a piece of self-contained software capable of performing a programmatic process [17]. One type of agent, called a mobile agent, is designated to move around and

perform a task on behalf of the user. In the framework for mobile Web Services [6], the transportable software agent representing the mobile service is recommended to migrate to the end user's device and execute the service locally. In [5], mobile agents are integrated with Web services in order to facilitate access to Web services. These ideas offer new alternatives to the traditional agent technology.

There are three main issues worthy of discussion: the scalability, the synchronization, and the resource consumption of small devices. Given the huge number of such agents distributed across the Internet, managing them and keeping them consistent is a real challenge. For complex services, it is not appropriate to use the small device as an execution platform, if only for reasons of security and privacy.

The above analysis shows the pros and cons for each approach. Among them, SOAP and XML are more suitable for our requirement for their extensibility and flexibility. What is more, recognition and promotion from the industry will definitely brighten their future.

2.1.4 Service Discovery

Service information can usually be obtained through the following methods: prior awareness, service advertisement, and service repository inquiry.

Prior awareness of the available services means that the services can be found and accessed statically. It is acceptable only for the fixed network and its associated services.

It is totally unfeasible in a highly dynamic wireless environment.

Within a relatively small network, the service application can periodically broadcast or multicast advertisements containing its own information. The user application that catches those advertisements will then be able to locate the available services. In this case,

the user always listens and waits for the incoming advertisement in a passive manner, and too many redundant messages are involved; bandwidth and processing capacity are also wasted. For example, the sView system relies on advertisement for service discovery [18]. Similar to the DNS system, all services publish their information in a service repository with a generic format. Every user can search or query the repository according to individual preference. Since all service information is categorized, stored and managed in a structured fashion, it enables the user to find the services easily and dynamically. For Web services, the Universal Description, Discovery and Integration (UDDI) is promoting such mechanism. In [5], a special mobile agent is designed to interact with the UDDI server to obtain the desired service information. sView also adopts this concept. As discussed earlier, each user maintains a personal service briefcase containing the desired service information. Its content is serialized and synchronized to a central briefcase server that is running all the time and manages all services. The following Figure 2-2 illustrates the sView architecture.

Figure 2-2 sView Architecture

The service repository approach is widely recognized due to its dynamism, scalability and extensibility. That is why many proprietary implementations exist, including the popular UDDI for Web services. Compared to the others, this mechanism is more systematic and structured with regard to describing the services, and thus provides more information about the service access for the end user.

2.1.5 Overall Architecture

With regard to the overall network, both the legacy client/server network and the emerging peer-to-peer architectures are widely used. The choice is made based on the

analysis of requirements, cost, and relationships among network entities. Some systems might seek a hybrid of both.

In the reference applications reviewed, sView is built on the infrastructure of Project JXTA [19], which is a peer-to-peer network. The local briefcases communicate as peers by passing messages to each other, but there is a Briefcase server to manage all the instances of local briefcases.

ActiveBuddy is a typical example of a client/server model. As shown in Figure 2-3 below, the BuddyScript Server contains every component required to operate an interactive agent, including connectivity with the messaging network, message translation and formatting, message routing, data access, user and session management, and detailed tracking [20]. It manages all the interactions with the messaging networks, and makes itself compatible with all major messaging protocols, such as AIM, MSN, Jabber etc.

Figure 2-3 ActiveBuddy Architecture

The client/server architecture relies on a centralized server. It is easy to implement and manage the services residing on the server, but it is hard to expand because the server is the bottleneck in the whole system. This kind of system is fragile. Once the server goes down, every supported service will no longer be running. The peer-to-peer network, on the other hand, is flexible, robust and efficient, with more scalability. There is no single point failure within the system. With the increasing number of peers in the system, it will be excessively difficult to manage.

2.1.6 Conclusion

The previous sections discussed several key aspects of the service provisioning and access. As the brief analyses showed, every approach has its pros and cons. Regarding our considerations for this thesis, the following decisions were made. We need:

- **An instant messaging user interface**

For wireless handheld devices, the interaction space is very limited. A Web browser or e-mail system will be too large. Instant messaging is simple, small, resource-efficient and user-friendly. As instant messaging application is normally coupled with a presence service, which will provide the extra feature of monitoring the status of the designated service.

- **XML-based transport media**

Scalability and extensibility are two of the main issues in a system. Plain text leads to ambiguity when interacting with non-human agents. The mobile agent is hard to manage and requires a special execution environment and more resources. They are not good candidates for our targeted wireless applications. XML-based messages, however, will serve as a solid foundation when integrating with services in the future, which will

include more and more Web services. XML's flexibility and its self-explain feature make it more adaptive to new services.

- **Dynamic service discovery**

A prior awareness of all available services is unrealistic. We need to find the services at any place, at any time. At present, services are scattered all over the network. We need a categorization or classification mechanism so that we can filter out the unwanted services.

- **Peer-to-peer network architecture**

With more and more services being offered, the server will be easily saturated in a client/server model. For peer-to-peer network, however, the new service can simply be plugged in with proper configuration, with little impact on the existing infrastructure. Besides that, most of the IM systems are based on a hybrid peer-to-peer communication, where a server is only used for registration and status update purposes.

Table 2-1 briefly compares our proposed framework with other solutions.

	User Interface	Transport Media	Service Discovery	Overall Architecture
ActiveBuddy	Instant messaging	Plain English	Pre-configuration	Client-server
IBM's E-mail integration	E-mail	MIME	Not defined.	Client-server
sView	Self-defined Application	JXTA message (XML, HTTP)	Advertisement broadcast	Peer-to-peer
Web services	Web browser	SOAP/XML	UDDI	Client-server
Our Framework	Instant Messaging	XML	UDDI	Hybrid

Table 2-1 Relevant Work Comparison

We need to choose the concrete protocols for our framework, based on the above criteria. An overview of instant messaging and service discovery protocols will be provided, followed by comparisons and analyses next.

2.2 Instant Messaging and Presence Protocols

With the development of Instant Messaging applications, a series of protocols have been defined to offer a standard for various service providers, such as IMPP, Wireless Village, and so on. Some are targeted towards interoperability between the vendors' different proprietary standards. Among them, Jabber, SIMPLE and JMS are the three major approaches that seem most promising and well-adopted by application developers. In the following paragraphs, I will give a brief introduction to these protocols as well as their implementations on J2ME, and also evaluate their development trend as a commercial product on the wireless market.

2.2.1 RFC 2778 and RFC 2779

Early in February of 2000, two standards for the general architecture of instant messaging and presence service were proposed by the Internet Engineering Task Force (IETF). One was called "A Model for Presence and Instant Message," referred to as RFC 2778 [21]. Another was "Instant Messaging / Presence Protocol Requirements," referred to as RFC 2779 [22]. They provide a proto-model for the forthcoming instant messaging and presence protocols, such as XMPP/Jabber and SIMPLE.

RFC 2778 defines all the necessary entities in the IM community and the behaviors among those entities. The terminology is used as a common vocabulary for other IM and presence protocols. As its title shows, the model targets two services – instant messaging and presence services. The Instant Messaging service delivers short messages between two users. The model does not specify the delivery mechanism or the format of the short messages. The Presence service receives, processes and distributes the presence information, including the status, contact address, and so on.

Many definitions in this model are commonplace; for example, the status and contact address, as mentioned above. Below I will list some that are essential in understanding the model and related protocols. For the complete set, please refer to [21].

- PRESENTITY: also called Presence Entity. This offers the Presence information to the Presence service.
- WATCHER: monitors the status change of another presentity.
- SUBSCRIBER: a type of watcher who gets notified of the status change in the presence information.
- SUBSCRIPTION: the information handled by the Presence service concerning a subscriber's request to watch other presentities.
- NOTIFICATION: the message informing the subscriber about the status change of the desired presentity.
- SENDER: the source of an instant message delivered by the IM service.
- INSTANT INBOX: the destination of the delivery and the container for the instant message.

The basic interaction between the entities through IM and the Presence service will be illustrated next, in Figure 2-4 and Figure 2-5.

Figure 2-4 Presence Service in RFC 2778

Figure 2-5 Instant Messaging Service in RFC 2778

While the above model serves as a base, the IETF describes some additional requirements for the Instant Messaging and Presence services in RFC 2779. It defines a minimal set of requirements that any instant messaging and presence applications should satisfy. Indeed it is a detailed supplement to the abstract model. This document specifies the relations and rules with which the entities in the Instant Messaging and the Presence domain should comply. It defines not only a common format for instant messaging and presence information, but also how to process this information, such as the handling of the replication and caching, rejection of unauthorized third parties, and so on. It also illustrates what actions and consequences should occur upon receiving any related information, including instant messages, subscription, notification and status publication.

2.2.2 IETF Instant Messaging/Presence Protocol (IMPP)

IMPP defines the protocols and data formats necessary to build an end-user presence awareness, notification and instant messaging system. It develops an architecture for simple instant messaging and presence awareness/notification. It also specifies how authentication, message integrity, encryption and access control are integrated within the framework. The purpose of this framework is to be descriptive and universal – the model should be mapped reasonably onto all of the systems that are informally described as

presence or instant messaging systems. The model defines two services: a PRESENCE SERVICE and an INSTANT MESSAGE SERVICE. The function of the PRESENCE SERVICE is to accept information, store it, and distribute it. The function of the INSTANT MESSAGE SERVICE is to accept and deliver INSTANT MESSAGES to INSTANT INBOXES.

As mentioned above, IMPP just defines an abstract framework that is generic enough to describe any IM products on the market. Commercial applications may fall into the subset of the IMPP framework, which has its own implementation detail. The IETF has defined the Common Profile for Instant Messaging in order to fill the gap between different vendors' implementations.

2.2.3 Wireless Village

The Wireless Village specification, founded by Ericsson, Motorola, and Nokia, defines how the Instant Messaging and Presence Service (IMPS) system should interface with the existing wireless network infrastructures. The specification also provides an open interface for the existing IM communities on the Internet [23]. It enables interoperability between existing IM providers such as AOL, ICQ, Yahoo, MSN and others. It includes the Wireless Village system architecture and an open protocol stack at the IMPS application level to provide interoperable mobile IMPS services among workstations, network application servers, and mobile information appliances, such as mobile handsets, handheld computers, PDAs and other mobile devices.

IMPS includes four primary features: Presence (user status, location, availability, etc.), Instant Messaging (two-way SMS/paging), Groups (chat, group discussion), Shared

Content (information sharing within an IM or chat session). Each of the features will be modularized as a service element within the IMPS server.

To take advantage of existing Internet and Web technologies, IMPS uses XML to represent the protocol data being exchanged during an IMPS session. It is a client-server-based system, where the server is the IMPS server and the clients can be either mobile terminals, or other services/applications, or fixed PC clients. For interoperability, the IMPS servers and Gateways are connected with a Server-to-Server Protocol (SSP).

The protocol stack is shown in the following Figure 2-6 [24].

Figure 2-6 Wireless Village Protocol Stack

The Wireless Village Protocol runs at the application level, and is compliant with IETF RFC 2778, RFC 2779 and the IMPP CPIM model [25].

Next, we will survey in detail some of the concrete protocols for the IM and Presence service. These protocols will serve as the potential candidates for our implementation in this thesis.

2.2.4 Jabber & XMPP

Jabber is an open, XML-based protocol for Instant Messaging and Presence, supported by the non-profit Jabber Software Foundation [26]. With over 75,000 servers deployed, Jabber has become a very popular and world-spread protocol for instant messaging and presence. Using XML as the base, Jabber clients connect to a Jabber server to communicate seamlessly with other systems. Jabber's open XML protocol contains three top-level XML elements: <message/>, <presence/> and <iq/> (info/query). Those are used for the instant messaging and presence service.

Jabber uses a client-server model, not a peer-to-peer model as some instant messaging systems do. All Jabber messages and data from one client to another must go through the server. Each user has a local server, which receives information for them, and the servers transfer messages and presence information among themselves. There can exist any number of Jabber servers that accept connections from clients as well as communicate to other Jabber servers. Unlike AOL IM or any of the other proprietary services, Jabber does not require that messages be run through a central server. The Jabber architecture is distributed, very much like Internet e-mail [27]. Each server functions independently of the others, and maintains its own user list. Any Jabber server can talk to any other Jabber server that is accessible via the Internet.

The server handles all communication to outside systems and to other Jabber systems. The client itself only needs to connect to the Jabber server, and from then on it does not concern itself with the type of clients it is exchanging information with. Figure 2-7 gives a basic view of the Jabber system architecture based on above description.

Figure 2-7 Jabber IM System Architecture

To achieve interoperability, Jabber enables communication between Jabber's open XML format and numerous non-Jabber formats, through the use of what are called "transports," in the Jabber world. When a Jabber user sends a message to a user on a foreign system, the Jabber server then routes the data to the appropriate transport application. In essence, a transport implements the proxy pattern. Most transports contain their own small session manager, which translates Jabber XML into and out of the "foreign" (non-Jabber) protocol for presence, messaging, and (in some cases) info/query requests [26].

The following two diagrams, Figure 2-8 and 2-9 depict the happy cases for the instant messaging and presence service in Jabber.

Figure 2-8 Jabber IM Message Flow

Figure 2-9 Jabber Presence Service Message Flow

There are many commercial Jabber products deployed on J2ME for small devices, such as Streampath™ Instant Messenger (SIM) Jabber™ Edition [28], Antepo Jabber-power ACCEPT [29]. Some major telecom companies, such as Motorola, Orange, and Ericsson, have already offered the IM services on their wireless devices, supported by the Jabber J2ME edition.

In late 2002, a working group for the Extensible Messaging and Presence Protocol (XMPP) was formed within IETF. XMPP is "an open, XML-based protocol for near real-time extensible messaging and presence." It is the core protocol of the Jabber Instant

Messaging and Presence technology, which has been adopted by the industry worldwide, mainly because of its openness and full functionalities. To some extent, XMPP and Jabber seem synonymous, but as a matter of fact, XMPP serves as a basis for all instant messaging systems, including Jabber. XMPP specifies the infrastructure and the core elements for the instant messaging and presence application. IETF defines the core layer and the Jabber organization proceeds with its own extra features beyond XMPP [30].

So far, several Internet drafts have been defined by the XMPP working group. The XMPP Core [31] describes the core features of XMPP for streaming XML elements during the exchange of message and presence information. Its network architecture, a typical client-server architecture, and the basic syntax and semantics are inherited from the Jabber community. With regard to the message exchange between IM entities, two fundamental concepts are introduced: XML streams and XML stanzas. An XML stream contains all the necessary XML elements for the information exchange between any two entities over a network. It is denoted by the XML tag `<stream/>` with appropriate attributes and namespace. An XML stanza is a discrete semantic unit carrying the structured information from one entity to another within an XML stream. It is the direct child of the XML stream and is contained by the XML stream. There are three XML stanza types defined for the server to server and client to server communication. They are `<message/>`, `<presence/>`, and `<iq/>`. Essentially, those XML stanza types are taken from the XML elements defined in Jabber, and thus they carry nearly the same information as in Jabber, in the same manner. The XMPP Core also addresses the character-encoding mechanism, the namespace for Transport Layer Security (TLS) data

and Simple Authentication and Security Layer (SASL) data, and security considerations during the communication.

XMPP Instant Messaging [32] describes the specific extensions necessary to create a basic instant messaging and presence application based on the XMPP Core. It complies with RFC 2779 and doesn't include the extra functionalities defined in Jabber, such as multi-user chat and file transfer. It addresses the protocols of establishing a session, exchanging messages and presence information, managing the subscriptions and the roster, which is analogous to the buddy list, and other complementary issues for both server and client. The <iq/> stanza is mainly used for establishing the session and managing the roster, including retrieving the roster, adding a roster item, updating a roster and deleting a roster item. The <message/> stanza and its corresponding attributes and child elements are generally used for message exchange. The <presence/> stanza makes it possible to direct the presence information to the server and then broadcast it to all subscribers. It is also used for the IM user or the server to handle the subscription.

2.2.5 SIMPLE

The IETF SIMPLE protocol appears to be one of the candidates that supports interoperability with third-party vendors. With the two leading providers (AOL and MSN Messenger) in the SIMPLE camp, instant messaging interoperability is imminent.

SIMPLE, which is being developed by the IETF, stands for SIP for Instant Messaging and Presence Leveraging Extensions. It extends the existing Session Initiation Protocol (SIP) [23] standard to support presence management and to initiate and manage instant messaging sessions. The services are achieved through the extensions to the SIP method and the SIP event notification [34].

The MESSAGE method, which is newly defined in SIMPLE, allows the transfer of Instant Messages. MESSAGE requests do not themselves initiate a SIP dialog; under normal usage each Instant Message stands alone, much like pager messages. MESSAGE requests may be sent in the context of a dialog initiated by some other SIP request. Since the MESSAGE request is an extension of SIP, it inherits all the request routing and security features of that protocol [35]. The MESSAGE requests and corresponding responses are constructed and handled according to the rules in the SIP specification.

Figure 2-10 diagram describes one of the scenarios of instant messaging in SIMPLE.

Figure 2-10 SIMPLE IM Message Flow

The Presence service in SIMPLE is accomplished through a concrete instantiation of the general event notification framework defined for SIP, and, as such, makes use of the SUBSCRIBE and NOTIFY methods defined there. Subscriptions and notifications of

user presence are supported by defining an event package within the general SIP event notification framework [36]. This event package, named "presence", is based on the concept of a presence agent, which is a new logical entity that is capable of accepting subscriptions, storing subscription state, and generating notifications when there are changes in user presence. The SUBSCRIBE and NOTIFY requests are processed according to the SIP specification or the event packages.

The user's availability can be obtained any number of different ways. None of these mechanisms are mandated by SIMPLE, however; they can be SIP-based, or other. Baseline SIP defines a method that is used by all SIP clients – the REGISTER method. Since this method allows a User Agent (UA) to inform a SIP network of its current communications addresses, it could be used for publishing the availability when the Presence Agent (PA) is co-located with the SIP registrar. In this case, the combined PA/registrar is known as a presence server. Using the register information for presence is straightforward. The address of the record in the REGISTER (the To field) is able to identify the availability.

Figure 2-11 shows the main message flow in the SIMPLE presence service.

Figure 2-11 SIMPLE Presence Message Flow

Since SIMPLE is relatively new, there are few vendors who provide solutions based on it. Ubiquity [37] and MessageMachines [38] offered the first platform-independent integration of IM with SIP/SIMPLE-compatible messaging services. In February of 2002, Lotus and AOL performed a server-to-server compatibility test of Lotus' Sametime and AOL's AIM, which are the IM products based on SIMPLE.

2.2.6 JMS

The Java Message Service (JMS) is an API for enterprise messaging, created by Sun Microsystems. JMS is not a messaging system itself; it is an abstraction of the interfaces and classes for the communication between software components and applications [39]. The application-to-application or component-to-component messaging systems used in the business are normally referred to as Message-Oriented Middleware (MOM). The messages are exchanged back and forth through MOM.

JMS provides two types of messaging models, publish-and-subscribe and point-to-point queuing, which are called messaging domains in the JMS specification. The publish-and-subscribe model is designed for one-to-many message broadcast, while point-to-point is for one-to-one message exchange.

In pub/sub, one JMS client, the publisher, can send a message to more than one client through a virtual channel, called a topic. The receiving client can subscribe to a certain topic. Any messages under a topic should be delivered to the subscriber client [40].

The point-to-point messaging model allows the JMS clients to send and receive messages through the virtual channel, similarly to a queue.

Since JMS is designed to be powerful and reliable for use by companies, its requirements and quality are not suitable for small wireless devices. Its message format is also too complex.

Most of the JMS application developments are associated with EJB, based on J2EE, for transmission of business data and targeted at the area of e-commerce. While IM on mobile devices has its own special requirements, it focuses on the concept of instantaneousness. Quick transmission has top priority, although reliability and quality is

always a concern. To adapt JMS to IM in the J2ME environment, JMS needs to be trimmed and simplified. Currently, deploying the instant messaging system for personal usage based on JMS is not appropriate for wireless devices.

2.2.7 SMS and MMS

SMS (Short Message Service) appeared in 1991 and is a globally accepted wireless service that enables the transmission of messages between mobile subscribers and external systems, like email, paging, and voice mail systems. The European standard for digital wireless, now known as the Global Standard for Mobiles (GSM), included short messaging services from the outset. In North America, SMS was initially made available on digital wireless networks, built by early pioneers such as BellSouth Mobility and Nextel.

SMS provides a point-to-point communication mechanism for transmitting short messages to and from wireless handsets. The service makes use of a short message service center (SMSC), which acts as a store-and-forward system for short messages. The wireless network provides for the transport of short messages between the SMSCs and wireless handsets. In contrast to existing text message transmission services, such as alphanumeric paging, the service elements are designed to guarantee delivery of text messages to the destination and provide a two-way communication channel.

With SMS, an active mobile handset can receive or submit a short message at any time, even if a voice or data call is in progress. SMS also guarantees delivery of the short message by the network. Temporary failures due to unavailable receiving stations are identified, and the short message is stored until the destination device becomes available.

In SMS, the wireless network signaling infrastructure is based on Signaling System 7 (SS7).

Compared with other IM services such as Jabber, the pros of SMS are:

- Large existing customer base in Europe.
- No further cost for service providers—as a standard network component, the expense has been covered in the initial network investment.

The cons of SMS include:

- It is based on a legacy telecommunication standard (SS7).
- No support for the Presence Service.
- Hard to extend to provide new features (such as multimedia service).
- Difficult to interconnect/integrate with other IM services.
- Two clients cannot talk directly; the mediate SMSC is required to store and forward the information.

Multimedia Message Service (MMS) is the evolution of SMS and mobile-to-mobile messaging. MMS can transmit messages containing text, graphics, photographic images, and so on. Both SMS and MMS are store-and-forward systems, not real-time. Unlike SMS, MMS can make use of user profiles to determine when content should be delivered – for example, a user might choose to receive certain messages after working hours. It is said that the SMS will be replaced by MMS soon.

Because the infrastructure of the telecom network is different in North America, SMS is not as popular here as in Europe.

2.2.8 Conclusion

As introduced earlier in this section, RFC 2778, RFC 2779 and IMPP serve as the abstract model for all concrete instant messaging protocols. Wireless Village defines a whole protocol stack mainly for the interoperability of different network infrastructures between the existing IM providers. Table 2-2 will briefly summarize the concrete IM and Presence service protocols.

	Jabber & XMPP	SIMPLE	JMS	SMS& MMS
Scope	End-to-end IM	End-to-end IM	Between applications	End-to-end messaging
Underlying infrastructure	Jabber proprietary network	IP telephony network	N/A	SS7
Presence service support	Yes	Yes	No	No
Network architecture	Client-server	Peer-to-peer	N/A	Client-server
Availability in North America	Yes	Yes	N/A	No
Industry support	Yes	Yes	Yes	Yes

Table 2-2 IM protocol comparison

Based on the above analysis, SIMPLE and Jabber are more suitable for the IM application in the wireless world. While JMS defines the most advanced and complicated messaging exchanging mechanism, it is too big to be integrated into the environment of a small wireless device. With regard to SMS, it is hard to make it compatible with other IM systems, from both a hardware and software point of view. In addition, because SMS lacks instantaneous response and the presence service, in my opinion it is less competitive than the fully-fledged IM technologies. At the current time, there are some commercial Jabber implementations based on J2ME. Most of those implementations are dedicated to supporting a specific IM application, such as Yahoo Messenger and MSN

Messenger. Regarding SIMPLE, there are few commercial implementations as yet. Therefore, an implementation that can integrate these two protocols can greatly improve the adaptability and extendibility for IM applications.

Between Jabber and SIMPLE, we propose SIMPLE as the underlying communication protocol for this thesis. This is not only because SIMPLE is simple and lightweight, but also because this kind of extension is better grounded and provides more flexibility. Considering that SIP has been recognized by IP telephony as a signaling protocol in the lower layer of the stack for a relatively long time, it would be easier to make extensions for SIMPLE and thus adopt our framework to an already existing infrastructure. In addition to that, it would be more cost-efficient for the IP service provider to deploy and integrate more services based on our framework. In other words, based on SIMPLE and SIP, our framework suggests an easy and economical method for both service provider and end user.

2.3 Service Lookup and Location Protocols

Within the domain of the wireless network, the devices and the applications are essentially heterogeneous and non-monolithic. When users move from one place to another, they should be aware of the available services in the new environment and therefore make use of the services around. Since the services are provided by different parties and distributed everywhere, we need a mechanism that will facilitate the discovery for both the service user and the provider. In the following section, some key protocols for service discovery will be reviewed. Please note that although the use of context awareness for service discovery provides more options, it is not within the scope of this thesis.

2.3.1 Service Location Protocol

The Service Location Protocol (SLP) is an IETF standard for dynamic service discovery. SLP is intended to function within networks that are under cooperative administrative control. It may not necessarily scale to wide-area service discovery throughout the global Internet, or in networks where there are many clients or a lot of services [41]. It works in the same way as the Yellow Pages: users do not have to know the details of a service; rather, the user is able to locate the service by submitting the desired service type and the descriptive attributes of the service. Services can be grouped into scopes, or universally. The concept of scope provides scalability, thereby limiting the network coverage of a request and the number of replies [42]. Three entities are defined in SLP:

- **Service Agent (SA)**

An SA is the provider of the actual services. Since any resource available in the network is potentially a service, an SA could be any host that contains some network resources. The SA is allowed to register its services with a Directory Agent for service lookup and service call by other hosts on the network.

- **Directory Agent (DA)**

A DA is the repository of the information about various services. It catches and caches all the service advertisements from the SA, which will later be queried by the User Agents.

- **User Agent (UA)**

A UA is the consumer of the service. The UA retrieves the information about a specific service from the DA or SA and makes use of that service.

SLP defines a series of rules for the interactions between the above three entities. In most cases, SAs discover a DA and register their services with the DA. UAs first find the DA

in a certain domain and then inquire about the services from the DA. After receiving enough information about the service invocation, UAs communicate with SAs and ask for the services directly. Figure 2-12 gives an illustration of this typical case. Alternatively, in the absence of a DA, UAs broadcast a request containing the description of the services wanted. Any SA that matches the request responds with the necessary information for UAs to invoke the actual services.

Figure 2-12 SLP Architecture

SLP is on top of TCP/UDP. It makes the best usage of the multicast and broadcast for its discovery functionalities. For example, a DA multicasts its presence periodically, waiting for the registration of SAs. Once an SA is available in the network, it listens only for the DA's advertisement. When the SA catches the multicast advertisement message, it can register its service with the DA. In a similar way, the SA can multicast or broadcast a registration request message. Only the responsible DA replies to that and therefore the service registration is set up. This is the so-called active or passive discovery of the DA. In SLP, any networked service is described in the Service URL. The Service URL is used for service registration and service location. It includes all the necessary information about what the service does and how to use the service. The Service URL is in the form of "service: "<servicetype>": //" <addrspec>. <addrspec> is a hostname or dotted decimal notation for a hostname, followed by an optional ' : ' and port number.

<servicetype> is a data structure defined by a Service Template [43]. It specifies the service name, attributes, values and protocol behaviors, and describes them according to formal syntax rules. The service type and attributes are defined by a Naming Authority, for instance, IANA (Internet Assigned Numbers Authority), by default. A service can simply be discovered by type or scope. In addition to that, the attributes of a given service can be retrieved by supplying a Service URL or for an entire service type. In the latter case, it gives the UA the flexibility to select the service with the desired features.

SLP is a lightweight, and extensible protocol for service discovery. It proposes but does not require centralized administration. The service invocation detail is not mentioned.

2.3.2 JINI

Jini is an extension to the Java library, which is built on the top of the Java Remote Method Invocation (RMI) technology. It provides a solution for the communication between the service providers and the end users by delivering the service objects across the network.

There are three basic entities in the Jini model. There is a service provider, such as a printer. There is a client, which is the user of the service. Third, there is a lookup service which functions as the broker agent between the service provider and the client. Code is moved around the network through the object serialization. The service object, called the proxy in Jini, is a Java Interface specifying the parameters and the signature of the service. The client who wishes to use a service can locate it and call it to perform tasks. Figure 2-13 shows the cooperation between those entities.

Figure 2-13 Interactions in Jini

Three protocols are defined in Jini: the discovery protocol, the join protocol and the lookup protocol. The discovery protocol is used between service providers and the clients to find the lookup service. The join and lookup protocol are not network communication protocols. They define the rules with which the client and lookup service should comply. As described in [44], there are three related discovery protocols that can be used to discover the lookup service under different situations:

- The *multicast request protocol* is employed by entities and service providers that wish to locate nearby lookup services. The lookup service listens on a well-known port. Upon receiving a valid request, it replies with the *ServiceRegistrar* proxy for further communication through unicast. It can also be used to support browsing of local lookup services.
- The *multicast announcement protocol* is provided to allow lookup services to advertise their existence. This protocol is useful in two situations: when a new lookup service is started, it might need to announce its availability to potential clients and service providers. Also, if a network failure occurs and clients lose track of a lookup

service, this protocol can be used to make them aware of its availability after networked service has been restored.

- The *unicast discovery protocol* makes it possible for an entity to communicate with a specific lookup service directly. It is mostly used when the location of the lookup service is already known before start-up.

The discovery protocols require support for multicast or restricted-scope broadcast, along with support for reliable unicast delivery, in the transport layer. The discovery protocols make use of the Java platform's object serialization to exchange information in a platform-independent manner [44].

After the lookup service is found through one of the above methods, the Join Protocol is used by the service provider while the Lookup Protocol is used by the client. The service provider uses the *register()* method of the *ServiceRegistrar* proxy obtained from the discovery procedure to register its own service with the lookup service. The service registered should contain the service proxy and other information about the service, such as the attributes. The client calls the *lookup()* method of the *ServiceRegistrar* proxy to get the desired service proxy. Afterwards, the client communicates with the service provider through the service proxy. During the lookup procedure, the services are compared and located, either by the type of Java interface or the name-value pair of the attributes. The client submits the service lookup criteria within a service template. The lookup protocol also specifies how the lookup service matches the service template with its registration entries.

Since Jini runs on RMI, a RMI activation daemon is required to act as a server and provide the services. Jini also provides the supporting APIs for the transaction operation.

2.3.3 Web Service and UDDI/UDDIe

The Universal Description, Discovery and Integration (UDDI) is a specification that defines how to describe, publish, discover and integrate Web services. It is also a publicly accessible set of implementations of the specification that allow a business to register its Web services and search for the Web services it needs, based on a distributed registry infrastructure. Detailed descriptions about the Web service are defined in XML format.

The information about the Web services are represented in a master directory called the Universal Business Registry (UBR), which is open to the public. UBR works like a huge information repository for Web services. Applications and programmers use UBR to locate information about the services and invoke the services through the Internet. Conceptually, a business can register the following three types of information in terms of functionality:

- **White pages**

Basic information about a business (company), including name, address, contact information, etc. This type of information enables Web service discovery according to business identification.

- **Yellow Pages**

Information about the Web services is categorized as per the industrial standard taxonomies. The Web service lookup can be conducted by means of this categorization.

- **Green Pages**

The technical information about the Web service is exposed by the business, including references to specifications of interfaces for web services as well as support for pointers to various file and URL-based discovery mechanisms.

Theoretically, the services could be indexed as per these three types of information. In the same way as phone books are used, the service can be found when the proper inquiry is utilized.

UDDI adopts the use of XML and SOAP and defines the next layer up. The protocol stack is shown in Figure 2-14.

Service Related Protocols
Universal Description, Discovery and Integration (UDDI)
Simple Object Access Protocol (SOAP)
Extensible Markup Language (XML)
Internet Protocol (HTTP, TCP/IP)

Figure 2-14 UDDI Protocol Stack

Based on the above architecture, UDDI defines an XML structure for SOAP messages with four core types of information, which are necessary and mandatory for the technical person to find and use the web service. These are: businessEntity, businessService, bindingTemplate and tModel. As shown in [45], Figure 2-15 illustrates the hierarchy of the XML definitions in UDDI.

Figure 2-15 XML structure defined in UDDI

- **businessEntity**

This is the top level XML element in a UDDI Business Registration. It contains the business information to support service publishing and discovery, such as company name, location, contact, and the industry or product category, etc. The overall businessEntity supports the “yellow page” taxonomy, to search for the business by industry or product category, or geographic location. The “green page” data are defined within it.

- **businessService**

This is the logical intermediate structure between businessEntity and bindingTemplate. It contains the descriptive information about the business service, which is grouped by the business process or the service category, such as shipping service. At this stage, the service can be searched by category.

- **bindingTemplate**

Within the businessService element, the technical description is offered here. It contains the data relevant to connect to and invoke the remote web services. This information includes the address or URL of the Web service, the references to interface specifications, and other supporting information.

- **tModel**

In case the bindingTemplate doesn't provide enough technical information, the tModel is used to represent the technical specification with which the real service is consistent. It is metadata about the specifications, containing the name, organization and URL pointers to the detailed specifications. With the help of the tModel, the customers can ascertain not only the location of the service, but also the proper document format, the appropriate protocols, and more.

Besides the above-mentioned descriptive information, UDDI also provides a series of programming APIs for accessing the UDDI registry. The APIs actually fall into two sets: Inquiry API and Publication API. As indicated by its name, the Publication API is used to interact directly with a UDDI registry, managing the information published. The Inquiry API enables the discovery and location of a candidate business, or a Web service, or specifications. For example, *find_xx* will return the registration data based on the search criteria, and *get_xx* will provide the user with the data he specified.

Similar to the DNS architecture, the UDDI business registry is a logically centralized, physically distributed service with multiple root nodes that replicate data with each other on a regular basis [45]. Once the service information has been registered with UBR, it is shared globally and is freely available to all customers. Recently, UDDI version 3 has defined three models of registry: fully public, semi-private, and fully private, according to the publishing domain. It controls the way in which information is distributed.

UDDIe [46] is an extension to the existing UDDI specification. It has inherited all the characteristics from UDDI and made some enhancements.

Since UDDI does not cover how to locate the service using the quality-related properties (e.g. the price and the response time), UDDIe tries to address this issue by extending UDDI's XML definition and APIs. In addition to UDDI's functionalities – white pages, yellow pages, and green pages, UDDIe supports a *blue pages* search. The blue pages contain the information about the service properties, such as Quality of Service (QoS). It enables the customers to look up a candidate service based on some quantitative criteria. There have been slight changes and extensions to the UDDI data structure and API that improve and maximize the usage of the registry [46]. In the *businessService* structure,

UDDIe adds another entity, service property, which realizes the blue page function and makes the service discovery more efficient.

Along with the enhanced search capability, some new features, such as service leasing and private domain usage, have been introduced in UDDIe.

2.3.4 DAML-S

DAML-S stands for DARPA Agent Markup Language for Services [47]. It is a part of the DAML program, which has the objective developing a language and tools to facilitate the use of the Semantic Web. DAML-S provides users and software agents with ways to discover, invoke, compose and monitor Web services.

Before going through how DAML-S is structured, let us first take a glance at the program overall. The whole DAML program is based on the concept of ontology. Ontology provides a way of capturing a shared understanding of terms that can be used by humans and programs to aid in information exchange [48]. Since the ontology needs to be specified in some language, DAML+OIL [47] (ontology inference layer) presents a layered approach to a standard representation language. It is like a logic language rather than a programming language. DAML+OIL is RDF (Resource Description Framework) compatible and includes precise semantics for describing the metadata. It is essentially designed for the user to describe and interchange the information on the Web. Considering the flexibility of Extensible Markup Language (XML) and the industry support, the service and its properties are defined in XML according to the rules specified in RDF.

The ontology of services in DAML-S consists of three sub-ontologies (or models), which are ServiceProfile, ServiceModel and ServiceGrounding. As shown in [49], Figure 2-16 sketches the relationships between the related entities. Details will be described later.

Figure 2-16 Top Level of DAML-S Ontology

Generally speaking, the ServiceProfile provides the information an agent needs to discover a service. Taken together, the ServiceModel and ServiceGrounding objects, associated with a service, provide enough information for an agent to make use of the service [49]. All the classes and the properties within the above three models are defined in XML according to DAML+OIL specifications.

- **Service Profile**

The service profile describes what the service does. By reading the service profile, service seekers should be able to determine whether the service meets their requirements. The description of the service profile is intended for the registry of the Web service. The service profile is not restricted to any form of registry; therefore, it can be used in any form, such as UDDI.

DAML-S Profile depicts the service in terms of three basic types of information: the information on the organization offering the service, the functionality of the service, and

the characteristics or properties of the service. For example, the category or the geographic radius of a given service will help the service requester to find the services, while the quality rating will give some hints for evaluating the service.

- **Service Model**

The service model explains how the service works. Since it deliberately describes what happens when the service is called, the service seeker could obtain some in-depth knowledge about the service, and thus act accordingly. There are two main components within the service model: the *process model* and the *process control model*. The *process model* describes the service in terms of input, output, precondition and effects. It also enables the service composition or decomposition, and the service interoperation as well. The *process control model* provides a way to monitor the execution of a service. So far, only the former has been addressed in the current DAML-S.

- **Service Grounding**

Service grounding provides the details of how the service is accessed. Typically the service grounding will specify the underlying communication protocol, the message exchange format, and so on. Grounding can be viewed as a mapping from an abstract specification to a concrete implementation. In DAML-S, both the ServiceProfile and the ServiceModel are thought of as abstract representations; only the ServiceGrounding deals with the concrete level of the specification [49]. The service grounding uses the Web Service Description Language (WSDL) to form its concrete messages.

Web Services Description Language (WSDL) “is an XML format for describing networked services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are

described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate” [50]. It seems obvious that DAML-S’ concept of grounding is essentially compliant with WSDL’s binding. DAML+OIL definitions can be used as the abstract types of message parts declared in WSDL, and then the current WSDL binding, such as SOAP binding, can be extended for the DAML-S formatting of messages. In this way, the abstract types in the Service Profile and Service Model can be interpreted into real, concrete SOAP messages.

2.3.5 Others

Besides the above-mentioned protocols, there are other solutions that focus on different aspects of the service location and discovery. Microsoft promotes Universal Plug and Play (UPnP) [51], which takes advantage of the World Wide Web infrastructure to find services on the network. Salutation [52], developed by the Salutation Consortium, intends to build a reliable architecture for resource discovery without depending on the network transport layer.

Next, I will give a brief review of DHCP options for Service Location Protocol and Bluetooth Service Discovery Protocol.

2.3.5.1 DHCP Options for Service Location Protocol

The Dynamic Host Configuration Protocol (DHCP) is an IETF standard which provides a framework for passing configuration information to hosts on a TCP/IP network [53]. In

SLP, knowing the location of the DA is one of the most important issues. There are two options for the DHCP messages. The SLP Directory Agent option is used to provide User Agents and Service Agents with the location of Directory Agents in the network. The SLP Scope Option is used to set up the default and static scope configurations of User and Service agents.

For example, when a new device (SA) is plugged into the network, a DHCP request can be broadcast or multicast to look for the DHCP server. The DHCP server that supports SLP can provide not only the configuration parameters for the device, but also the information about the current available DAs as well. Thereafter, the device can communicate with the corresponding DAs.

2.3.5.2 Bluetooth Service Discovery Protocol

Unlike other protocols, the Service Discovery Protocol (SDP) is only useable by Bluetooth devices in wireless communication. It defines how a Bluetooth client's application discovers available Bluetooth servers' services and service attributes. It primarily addresses the service discovery problem. It doesn't provide access to services, brokering of services, service advertisements, or service registration [54]. SDP supports search by service class and by service attribute. A client can browse all the services offered if nothing about the service is pre-known.

SDP is a simple protocol. It realizes only basic methods for searching and recognition of networked services. It suits small environments, such as the pico net in Bluetooth.

2.3.6 Conclusion

From the above review of the current technologies used for service discovery, we can see that every protocol addresses the problem from a different perspective and solves it in different ways. However, the most common issues are where the service information is stored, how to find the service information and how the service is described.

- **Architecture**

With regard to architecture, all the protocols follow the same idea to some extent. Most of them define a central agent that accepts the service registration and is capable of conducting search work. For instance, the service provider, the lookup service and the client in Jini are analogous to SA, DA and UA in SLP. The UDDI server maintains the information about Web services.

- **Discovery Mechanism**

Regarding how to find the service information, some of the protocols make use of multicast and broadcast for service requests, such as the DA discovery in SLP and three discovery protocols in Jini. On the other hand, the service can advertise itself on the network and let users know its availability. Others, such as the UDDI server, support a query-style search.

- **Service Description**

The service is described not only by its name and type, but also by its attributes, its access method, and so on. SLP adopts the Service Template to describe the service while Jini uses service proxy to represent the real service. Both DAML-S and UDDI/UDDIe describe Web services in XML. The description scheme is well defined and more general.

Similar to the mobile agent, the service proxy will be loaded on the end user's side in Jini. It is too heavy for small devices, as mentioned earlier. SLP relies on low-level multicast and broadcast and the service information is not as structured as XML. Therefore, DAML-S and UDDI/UDDIe are better candidates.

Compared to DAML-S, UDDI/UDDIe is lightweight and less abstract, but has a different focus.

DAML-S is a systematic language for describing the Web service through logic concepts. The service is defined in terms of the ontology in DAML-S and described in modules. The information on the service is carried in an abstract language DAML+OIL and the actual implementation is done with WSDL binding. DAML-S puts its efforts into providing a description mechanism that tries to cover the whole range of services. Considering the knowledge required by RDF and DAML+OIL, it is too complicated under the current architecture of Web services. With regard to the service registry, DAML-S does not address that.

UDDI provides a total solution from the service description to the service discovery and location, including the abstract data type, to concrete API. It also specifies how to manage the business registry. Although the XML scheme for the service description is relatively simple, it conveys the adequate information clearly and is easy to understand. With the introduction of UDDIe, the service discovery has been improved, to be more powerful and efficient.

From the above analysis, we will see that UDDI/UDDIe is more suitable for lightweight network architecture over a simple communication protocol, such as the Instant Messaging protocol.

2.4 Summary

This chapter covered some important technologies related to service discovery, service description, service access and the network platform. Each topic was surveyed in detail, and the solution that fits best has been proposed separately.

In conclusion, our system should be built on an IM platform, giving preference to SIMPLE, communicate in a peer-to-peer manner, describe networked services in XML format and locate them through UDDI/UDDIe. In the next chapter, the framework will be introduced in detail.

Chapter 3 IM-Based Networked Service Provisioning and Access Framework

This chapter outlines a framework that can be applied to any networked service, including network devices, Web services, Internet applications and so on. The whole architecture is built on an IM platform. It integrates the IM and Presence service and UDDI Web service categorization mechanism with proper modifications and extensions. A generic representation schema for the deployment of the networked service on the proposed platform will also be given. Finally, a user-oriented service access mechanism will be suggested. A sample design based on SIMPLE will be described in detail in Chapter 4.

3.1 IM-Based Networked Service Provisioning Architecture

Although networked services seem non-monolithic, they can naturally be treated and categorized in a unified way within a certain community. In our architecture, we expand the traditional IM platform such that all networked services can be generalized as IM entities. In our IM domain, every service is presented as a single entity, just like another IM peer. In other words, the networked service is wrapped by an IM proxy, which appears to be the same as a common IM user. The only difference is that the service IM proxy represents a non-human entity in lieu of a normal human user. By talking to the desired service IM proxy, end users can find and access the service they want. The service IM proxy communicates with the end users through the regular IM message format in a peer-to-peer manner. However, this service IM proxy acts differently in the following aspects:

- It has two interfaces. One interacts with other normal IM users. The other one communicates with the actual networked service in the background. It transfers the service invocation information between the service provider and the end users who are actually the service consumers. Instead of just a simple message relay, the service IM proxy knows how to customize the information between the two interfaces, i.e. parse and construct proper messages for both the service provider and the end user.
- It is intended to work not only passively but also actively. This means that both a pull and a push mechanism can be used here. Usually the service IM proxy does not look for other normal IM users. It just stays there and waits for incoming requests, and then handles those requests accordingly. In short, the service IM proxy interacts with the actual service upon the request of the end user and on behalf of the end user. On the other hand, the service proxy can take full advantage of the subscription and notification mechanism provided through the presence service. It is possible to add a specific IM end user into the service proxy's contact list. Whenever the corresponding events occur, the desired information can be pushed to the IM user. For example, a stock watcher service IM proxy can inform the end users when the index changes beyond a certain range.
- Each individual service IM proxy specifies a customized service invocation format to the end user. Because the service invocation differs from the input/output parameter, the interaction approach and so on, the interaction with each service IM proxy should be specific to each service. Due to the limited power and processing capability, it is unrealistic for end user devices to handle all the service invocation

details locally. Moreover, the end users themselves would probably prefer several lines of interaction, rather than having to know how to invoke the services technically. It is the individual service provider's responsibility to offer a simplified format so that the normal IM user can access more complicated services. It hides all the logics of the service call and makes the detailed implementation transparent to end users.

- It keeps its status synchronized with the background service. As anyone who has used IM may have seen, each normal IM user shows its availability by specifying a status, such as “online”, or “busy”. In our system, the availability of the service IM proxy is consistent with the real service.

This architecture consists of service IM proxies, IM end users, and the underlying networked services. The following Figure 3-1 depicts how the service IM proxy works.

Figure 3-1 Structure of Service IM Proxy

The communications between the end user and the service IM proxy are through the IM protocol, while the communications between service IM proxies and the underlying networked services follow their proprietary protocol, which only depends on the service provider's preference. The service IM proxy filters the information exchanged and passes

the minimum requirements to the IM end users for the service invocation. The existing IM end users are offered more accessible services with a simple and user friendly interface similar to their previous messaging applications. It is user-oriented and is easily accepted by existing IM users, both technically and psychologically.

In this architecture, it is easy to attach any new service without interfering with the existing functionalities. The end user will benefit from this kind of extension by gaining more and more services without extra configuration. On the other hand, since IM is usually coupled with the Presence service, the end user can get the latest status of the desired service, which is delegated by the corresponding IM proxy. As the Presence service is normally offered through a subscription/notification mechanism, the end user can subscribe to the desired service IM proxies and monitor their availability status at all times. Once the desired service shows as active, the end user will be notified and can then communicate back and forth with it later, in IM format. In this way, the service availability can be obtained not only through the “pull” mechanism but also through the “push” mechanism, by taking advantage of the Presence service. It will reinforce the whole architecture and make service access more efficient.

Since every service uses its own input and output parameters, the content carried in the instant messages will differ slightly according to the service invoked. The end user needs to obtain the service invocation format from a specific service IM proxy. All information exchanges always follow the same IM and Presence service protocol (which is SIMPLE, in our case).

Given that all the communication between IM entities, including the IM end user and the service IM proxy, are peer-to-peer (P2P), finding the service IM proxy peers previously

unknown in a new environment is another key issue in our proposed framework. First the end user needs to know all the possible services available in the environment. The next step is to subscribe to the desired services and use them if they are active.

The next section addresses the discovery of the networked services in a community.

3.2 Networked Service Categorization and Discovery

Because all services are different, a few attempts have been made to describe and categorize them in a general format. UDDI defines a specification with which Web services information can be interpreted in an XML template. Some implementations are currently available, which allow service registration and inquiry based on a distributed infrastructure. The UDDI service categorization and discovery mechanism does not just perform a simple search function, such as search by keyword. It provides a systematic methodology to describe services and manage the related information. The user is granted more flexibility to search by name, by service type, etc. The UDDI search is more dynamic, and always reflects the latest information about the services.

Although UDDI was initially designed for Web services, it can be generalized to any networked service as long as the networked service can be identified by a unique attribute. For example, in the SIMPLE IM domain, the networked service and its IM proxy are represented by a SIP address. Each proxy should register itself to the UDDI server using its unique IM identity, which will be used to identify it as a service proxy. In this way, our service proxy is compatible with other UDDI registrations. The service registration entry in UDDI is enough to depict the core part of a particular service. The auxiliary information may be handled in UDDIe or a more sophisticated mechanism. A detailed

discussion on UDDI is beyond the scope of this thesis; we merely want to use UDDI to illustrate the concept of the service repository and service discovery.

UDDI itself is presented as a networked service that manages the information of other services. Mapped into our proposed architecture, there exists a special service IM proxy for the UDDI server, called the UDDI IM proxy. The UDDI IM proxy meets all the general requirements for a service IM proxy, as stated in the previous section. Through this proxy, the end user can search and retrieve the service information stored in the distributed UDDI server, such as the service type, contact information, URL (or SIP address, probably), etc. The end user can then subscribe to the desired service and access those services by communicating with the service IM proxy. The UDDI IM proxy serves as an entry point for end users to access other networked services; the only pre-condition is that the end user knows how to contact the UDDI IM proxy. After that, all the communication issues will be restricted within the IM and the presence service domain. Since UDDI is essentially a networked service, the introduction of the UDDI IM proxy actually extends the generality of our framework. Moreover, the UDDI IM proxy can filter the services that are designated for IM users by executing the proper search on its own. Figure 3-2 illustrates the architecture of our framework.

Figure 3-2 Architecture of our framework

The framework is built completely on the existing infrastructure, with an extended scope to networked services. By integrating the service discovery mechanism, it offers more usability, more flexibility and more generality, at little cost. Figure 3-3 shows how the messages flow among the basic entities.

Figure 3-3 Interactions between IM entities

3.3 Generic Networked Service Invocation Mechanism

As described above, in short, every networked service can be represented and configured as a service IM proxy in the framework. After the IM identity (e.g. the address) of the desired service IM proxy has been located through the above mentioned method, the end user is finally going to invoke the actual service. At this phase, the end user has to know the service invocation format, such as the input parameter, the parameter type and so on. The service invocation format should contain the minimum information required to execute a service call. Upon receiving such information, the end user should just need to

follow the format to feed the input and wait for the output. However, interactions with different service IM proxies may vary a great deal in terms of the input and output parameters. The service invocation format should accommodate these variations, while remaining structured for ease of processing.

Although the UDDI service can provide a detailed description of how to access a specific service, the normal user cannot be expected to find out how to do that by reading a technical description or an XML file. Therefore, a general service invocation mechanism is necessary. This mechanism should specify how the end user interacts with the background service via the corresponding IM service proxy in the form of IM messages.

In our proposed solution, the service invocation mechanism contains three main parts. The first part is how to present the required service invocation parameters to the end user. The second part is how to make those parameters understood by the end user. The third is how to inform the end user of the corresponding outcomes. All three questions will be addressed with the help of XML.

We chose XML not only because of its current popularity in various areas, but also because of its flexibility and extensibility, and, most importantly, its nature, which is to exchange data. Before proceeding to the details, a brief introduction to XML and XML schema will be given.

3.3.1 XML and XML Schema

Extensible Markup Language (XML) defines a standard mechanism for describing, carrying and exchanging data. It can be viewed as a meta-language to describe data. While following the syntax similar to HTML, it does not have any fixed tags or semantics. This property allows users to define their own structured data with appropriate tags and

inter-relationships. Based on XML, XML Schema was first proposed by Microsoft, but became an official W3C recommendation in 2001. It specifies shared vocabularies and allows machines to carry out rules made by people [55]. The detailed structure and supported data types, including the built-in data types, are described in the specifications. For the full set of XML Schema specifications, please refer to W3C's documents. One of the main usages of XML Schema is to define the format of data for the elements in an XML file. Together with the flexibility and extensibility inherited from XML, XML Schema has become a very powerful and popular tool in information exchange.

3.3.2 Networked Service Access Information Schema

XML Schema is used to carry the structure of the service invocation parameters. Consequently, XML is used to transfer the values for service invocation and the returned result when necessary.

XML Schema fits our situation well. Since every service has a different number and types of input parameters, XML Schema is able to provide a generic method to describe any set of parameters systematically. Because XML is text-based and self-explanatory, the information expressed by XML Schema is easy to understand and operate. Since XML Schema already defines its own elements and the supporting data types, we adopt them directly for the purpose of simplicity. Any further extension is possible and will be left for the future discussion. `<xs:complexType>` and `<xs:sequence>` defined in XML Schema will be used as the wrapper layers for the detailed service invocation information. `<xs:element>` will carry the actual information through its attributes "name" and "type" which give the name and the type of the required parameters. To keep the generality, the data types defined in XML Schema such as `positiveInteger` and `string` will be used. This

structure is totally based on the XML Schema standard and is easy to implement by the individual service provider. The further exploration of the use of XML Schema will definitely improve the flexibility and extensibility.

The following example shows a simple XML Schema that describes the basic access information required for invoking the Google search service.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="GoogleSearch">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="SearchString" type="xs:string"/>
        <xs:element name="NumOfReturnedResult" type="xs:positiveInteger"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

This XML Schema file, possibly named *GoogleSearch.xsd*, can be translated as follows:

Service name: GoogleSearch

Name of input parameter 1: SearchString

Type of input parameter 1: string

Name of input parameter 2: NumOfReturnedResult

Type of input parameter 2: positiveInteger

From the above example, we can see that using XML Schema to describe the service invocation information is simple and straightforward. The end user just needs to provide the proper value for each required parameter. Later, an XML document containing those values embedded in proper tags based on the XML Schema, such as *< SearchString>* and *< NumOfReturnedResult>*, will be generated accordingly. If a result is expected from the service call, the outcome will be encapsulated by the XML tag *<data>*. A complete example will be given in the next section.

In our framework, the service IM proxy is responsible for translating the human user's parameter inputs, and conducts the service call based on these inputs. Conversely, the same service IM proxy will later process the returned result and turn it into a human-readable format. The service IM proxy can carry out this bi-directional translation procedure easily as long as the necessary information can be fetched accurately from the end user or the back end service. The rest will just follow the rules set for each specific service IM proxy. The critical point is how to inform the end users and guide them to enter the required service invocation parameters on handheld devices.

Here I would like to bring up SOAP, since it was built in XML and has been adopted by Web services. SOAP is a transport protocol that cooperates with XML to provide better functionality. For example, the XML Schema file can be transferred within SOAP messages. SOAP messages follow a strict syntax with a lot of extra information specified in its headers, such as authentication and payment, which are rarely of interest to the end user. In addition, the small handheld device is not supposed to maintain powerful parsing and handling capability. In this situation, we need a flexible and simple mechanism that will describe the basic structure of the information. In some sense, completeness must be

compromised for complexity, especially for small devices. For simplicity, SOAP is not specifically supported by our prototype, but it could and should evolve towards SOAP in the future. One possible solution is to use the service IM proxy to wrap the service invocation in SOAP.

3.3.3 Networked Service Access Mechanism

In this section, we propose a mechanism that allows the end user to provide the values for service invocation parameters through the IM interface. Considering the size of the interaction screen and the resources on the wireless device, a fancy and complex graphic user interface is not practical. Interacting with a form will reduce complexity, not only on the end user's side but also on the service IM proxy's side. Instead of prompting the end user to enter the instant message and the recipient, a simple form will be generated according to the service invocation XML Schema obtained from the targeted service IM proxy. This form maps the service invocation XML Schema into the corresponding text fields. The end user just needs to provide answers for each field.

The wireless device should understand this schema. This means that the application on the wireless handheld device is responsible for translating such scheme into a form and then sending the user's answers to the contacted IM service proxy in XML format, via instant messages. The service IM proxy knows how to parse the XML document and construct the service call to the background server on behalf of the end user. The outcome of the service invocation is wrapped in XML format by the service IM proxy and forwarded to the end user. The end user device then parses it and displays the user-relevant content of the XML document. Note that the entire information exchange should be carried within an IM message body.

The IM mobile device only needs a simple version of an XML parser to access various kinds of networked services. Let us consider an example of how to query the Google search engine under the described mechanism.

1. The Google service IM proxy sends out a message requiring 2 parameters: Search String and Number of Return.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="GoogleSearch">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="SearchString" type="xs:string"/>
        <xs:element name="NumofReturnedResult" type="xs:positiveInteger"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

2. The wireless device application translates the above XML file into a small form and displays it on the screen. Then it waits for the end user to input the value. In this example, the end user enters "Microsoft" and "2" as answers.

GoogleSearch	
Search String	
<input type="text" value="Microsoft"/>	
Number of Returned Result	
<input type="text" value="2"/>	

3. The wireless device application encapsulates the answers with an XML file. The parameter's name is used to construct an element containing its relevant value. For example, the `<SearchString>` element contains the user input value of "Microsoft".

```
<?xml version="1.0"?>
<GoogleSearch xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SearchString>Microsoft</SearchString>
  <NumOfReturnedResult>2</NumOfReturnedResult>
</GoogleSearch>
```

4. Upon receiving the above message, the Google service IM proxy parses it and forwards the required parameter to the Google server, then returns the result in XML format. The result is embedded within the `<data>` element. The value of `<data>` will be parsed and displayed on the screen by the wireless device application.

```
<?xml version="1.0"?>
<GoogleSearch>
  <data>http://www.microsoft.com</data>
  <data>windowsupdate.microsoft.com</data>
</GoogleSearch>
```

3.4 Summary

As described earlier in this chapter, our proposed framework and its corresponding service access mechanism have the following outstanding features:

- The whole framework is built on top of the existing IM infrastructure. No major modification is needed to the overall network. In other words, it comes at little cost and minimum risk, compared to establishing a brand new infrastructure.

- It defines a middle layer, composed of service IM proxies, between the end user and the networked service. This layer makes the service invocation totally transparent to the end user.
- The Presence service allows the end users to monitor the status of the desired service at all time. It provides a complimentary tool to locate the service.
- It is user-oriented and user-friendly.
- It is easy to attach any new service to the overall network, since the service provider just needs to plug in the corresponding service IM proxy.
- Our XML-based generic service access mechanism is language independent and platform independent; therefore the problem of interoperability can be solved.
- With increasing industrial support for XML and UDDI, our framework will be suitable for further exploration and extension.
- A huge number of IM users are potential customers for networked services. Time to market will be reduced dramatically.

Chapter 4 Design and Implementation Details

Small, handheld device applications are different to regular desktop applications with respect to resource usage, functionality and execution environment. First we will analyze the special requirements for this type of application, and then the design and implementation of our proposed framework will be given.

4.1 Requirements for Small Wireless Device Applications

Compared to traditional computational devices, wireless devices, such as cell phones and PDAs, are much smaller in terms of physical size, processing power, memory usage, display area and storage space. For example, a game targeted to the cell phone should be dramatically different to one designed for PC users. Therefore, some special requirements are imposed on applications running on mobile devices.

- The application should be easy to use and lightweight with regard to file size, library calls and complexity, because it is either stored in a small-sized device or downloaded on the air. This requires a mini version of the supporting library, a simple mechanism to synchronize concurrency, a necessary but not luxurious graphical interface, and so on.
- It should be efficient regarding the available resources. Since the resources on a mobile device are very limited and precious, maximizing their usage without sacrificing the desired functionalities is crucial to a successful application. Some technologies have already tried to integrate the execution platform with the chip on the mobile device. This effort will definitely improve the performance of high-level applications.

- It is not necessary to require application developers to handle low-level hardware functions. The application developers should be able to write the programs in languages that are familiar to them, without caring about the embedded part of the mobile device.
- The underlying platform should be based on an open standard recognized by the industry. If every provider develops the applications according to its own standard, interoperability will be lost, tremendous problems will arise, and the product might even fail.

Currently, there are several operating systems or platforms designed for the mobile devices. Some of them have existed for a long time, such as Palm and Symbian. Others, like BREW and .NET, are still relatively new. Java 2 platform Micro Edition (J2ME) maintains the characteristics of the object-oriented language with special APIs defined for small devices.

Since every technology has its pros and cons, the choice should be made based on budget, capability of the hardware, scalability, maintainability, and user preference. In this thesis, we chose J2ME simply because it is free and platform independent. However, the whole framework will not be compromised if any programming language is used. Next, a brief review of J2ME will be given.

4.2 J2ME & J2ME APIs

Java 2 platform Micro Edition (J2ME) is aimed at machines with as little as 128KB of RAM and with processors a lot less powerful than those used on typical desktop and server machines [56]. It has the following features:

- It contains a subset of the standard Java class libraries to enable Java applications to run on small computing devices.
- It defines a new Virtual Machine called the KVM, which is optimized for use on small devices, such as PDAs and cell phones.
- Recognizing that small devices vary greatly in their nature, it defines Profiles (application-specific) and Configurations (baseline).

4.2.1 Profile

J2ME consists of a set of profiles. Each profile is defined for a particular type of device – cell phones, PDAs, microwave ovens, etc. The industry benefits from this extensive flexibility in that users are able to define only what they need for a class of device specifically. The profile consists of a minimum set of class libraries required for the particular type of device and a specification of the Java virtual machine required to support the device. The virtual machine specified in any profile is not necessarily the same as the virtual machine used in Java 2 Standard Edition (J2SE) and Java 2 Enterprise Edition (J2EE). For example, the profile for a Palm OS device application is actually a subset of the Java Virtual Machine for J2SE [57].

To date, Sun has released two profiles: the Foundation Profile – a profile for next-generation consumer electronic and embedded devices; and the Mobile Information Device Profile (MIDP) – a profile for mobile information devices, such as cellular phones, two-way pagers, and PDAs.

Profiles are defined through the Java Community Process (JCP), and may be initiated by industries without Sun's direct involvement. Tailored Java environments can be extended

to diverse device types and industries not addressable by a single company or group of companies.

Since each profile defines a different set of Java class libraries, a Java application written for one profile cannot run on a machine that supports another profile. Likewise, an application written for Java 2 Standard Edition (J2SE) or Java 2 Enterprise Edition (J2EE) will not work on a machine that only supports J2ME. The only thing we can use is the Java classes provided in the Java class library, specified in the target device's profile.

A profile itself does not do anything; it just defines the specification. Profiles are implemented through a configuration.

4.2.2 Configuration

A configuration is comprised of a virtual machine, core libraries, classes and APIs. Currently, there are two J2ME configurations: the Connected Device Configuration (CDC) and the Connected Limited Device Configuration (CLDC). CDC is the implementation of the Foundation Profile, while CLDC is the implementation of the MIDP. CDC is designed for next-generation devices with more robust resources. Typically, these devices run on a 32-bit CPU and have 2 Mbytes or more of memory available for the Java platform and applications.

CLDC is designed for devices with constrained CPU and memory resources. Typically, these devices run on either a 16- or 32-bit CPU and have 512 Kbytes or less memory available for the Java platform and applications [58].

CLDC is the foundation of the Java runtime environment that targets small, resource-constrained devices, such as mobile phones, mainstream personal digital assistants, and

small retail payment terminals [59]. A profile targeted at the wireless market and utilizing CLDC can retain a very small footprint, consume little power, and provide as much capability as is needed for handheld devices. Devices that require more capable environments can receive profiles that provide additional functionality and are based on CDC. All profiles share a J2ME base, as well as the ability to safely download code onto a device and configure the Java environment.

The following Figure 4-1 shows the main structure of J2ME.

Figure 4-1 J2ME Structure

4.2.3 KVM

At the heart of CLDC is Sun's K Virtual Machine (KVM), a virtual machine designed from the ground up with the constraints of inexpensive mobile devices in mind. Named to reflect that its size is measured in the tens of kilobytes, KVM is suitable for devices with 16/32-bit RISC/CISC microprocessors/controllers, and with as little as 160 KB of total memory available – 128 KB of which is for the storage of the actual virtual machine and libraries themselves.

4.2.4 Restrictions of CLDC & MIDP

As described above, CLDC and MIDP provide a small footprint for wireless applications in Java. In CLDC, the following Java libraries are defined.

- *java.io - standard java input/output library*
Includes: InputStream, OutputStream, etc.
- *java.lang - standard java extended language library*
Includes: String, Byte, Thread, etc.
- *java.util - standard java utilities library*
Includes: Vector, Random, etc.
- *javax.microedition.io - CLDC-specific input/output library*
Includes: HttpURLConnection, etc.

MIDP, which is built on top of CLDC, provides extra APIs for defining and controlling applications, displaying text and graphics, and local persistent data storage [61]. It gives a common API across different devices and different networks. It provides the following Java libraries:

- *javax.microedition.lcdgui - LCD user interface library*
Includes: Screen, Graphics, Textbox, CommandListener, etc.
- *javax.microedition.midlet - Library for MIDP applications*
Includes: MIDlet
- *javax.microedition.rms - persistent data storage*
Includes: RecordStore, etc.

We can see that both CLDC and MIDP are designed to fit into resource-limited devices.

The trade-off comes with some functional restrictions, however:

- One key issue is that the CLDC class library lacks many of the classes found in the J2SE class library, especially the lack of access to UI elements – buttons, menus, and graphics. To compensate for this, MIDP provides wireless applications with some

high-level APIs for the user interface, such as Forms, Items, and TextBox etc. They are very simple and plain. The underlying CLDC APIs are used to handle the strings, integers, and objects [60].

- KVM doesn't support floating point numbers. Therefore, there are no float / double data types in CLDC.
- Due to the limitation of the devices, the error handling is reduced.
- There is no object finalization, since it complicates the Garbage Collector.

4.3 Design and Implementation Background

As shown in our proposed architecture, there are four types of entities in our framework: the end user, the service IM proxy, the IM & Presence Server and the background service residing on the specific server. The communication protocol involved is an IM and Presence service, for which we choose SIMPLE, as discussed earlier. Among those entities, our design and implementation will only cover the service IM proxy and the end user. The reasons for this are as follows:

- Background service

Since we do not modify any part of the background service, it will remain untouched in our implementation. It will only be used for the case study to illustrate the concrete example.

- IM & Presence / SIMPLE Server

There are a number of commercial IM & Presence servers available on the market, including SIMPLE servers. The implementation complies with the corresponding specification. Because our framework is above the IM & Presence service layer, it will

not interfere with the way the server works. Therefore, the SIMPLE server is kept as it is. We just use it to manage the subscriptions and process messages.

- Service IM proxy

The service IM proxy plays a key role in our framework. Although the detailed implementation for each service IM proxy varies, we tried to come up with a general framework for all service IM proxies.

- End user: client

The end user is referred to as a client technically. Besides its functionality as an IM client, it also has the ability to access networked services. In our framework, the end user will be capable of handling simple XML documents, so we would like to provide a sample version of such an end user in the case study.

4.4 Service IM Proxy Deployment

This section specifies the general form for service IM proxy implementation. No matter what the service is, the implementation will always follow this format, plus its own extra features.

From the functionality's point of view, the service IM proxy consists of three parts: IM communication with the end user, private communication with the background server, and information for translation and handling. We will analyze each of them next.

- Communication with the end user

The choice of the IM protocol between the service IM proxy and the end user depends solely on the end user's preference. If the end user is implemented in SIMPLE, the service IM proxy should support SIMPLE too, and vice versa. In more advanced cases, the service IM proxy could provide multiple interfaces for different IM protocols.

Therefore, we will not specify any specific protocol here and leave the choice to the service provider and the service IM proxy developer.

- Communication with the backend service

This kind of communication happens privately. It is hidden from the outside end users and other service IM proxies. The protocol is defined only by the service provider, so we will not discuss it here.

- Information translation and handling

This is the core part of the service IM proxy. The service IM proxy should maintain an XML parser to understand and construct the proper XML documents, as described in Chapter 3. As to the service call, each service IM proxy has to maintain some basic methods for the end user. As shown in Figure 4-1, every individual service IM proxy will inherit from the abstract class – Service Proxy. This super abstract class specifies the common attributes and operations for all service proxies. In addition to those, each individual IM service proxy offers extra information about the detailed service and service invocation. The service IM proxy stands as an adapter for the real service API.

Figure 4-2 Service Proxy Deployment Architecture

In our proposal, the SIP address serves as a unique identification for the service proxies. The end user can locate and communicate with the service proxy through this attribute. The “name” and “description” gives more information about the service proxy and its background services. The `Get_Service_info()` method is used to fetch the service invocation information which is describe by XML Schema as mentioned earlier. The `Get_Service()` method will execute the service call on behalf of the end user. The extra methods may be provided in each concrete service proxy to offer other value added features. This architecture exploits the polymorphism provided by object oriented design. By following it, the service provider will be able to be both general and flexible. The service IM proxy stands as an adapter for the real service.

4.5 SIMPLE Client Design and API Definition in J2ME

This section describes the design and implementation details for a SIMPLE client in J2ME. As stated in previous chapters, so far there has been no such implementation available. By taking on this challenge, we hope to offer some experimental information for future exploration.

4.5.1 SIMPLE Overview

SIMPLE is based on SIP, which is a signaling protocol for Internet telephony defined by IETF. Since SIP works independently of the underlying transport protocols, SIMPLE runs on top of either UDP or TCP. The default SIP port is 5060 for UDP and TCP, and 5061 for Transport Layer Security (TLS) over TCP.

In SIP, there are some important entities, including registrar server, proxy server, user agent, and method, etc. For example, the registrar server allows the user to upload its information while the proxy server helps to route the messages on behalf of the user. As SIMPLE is an extension to SIP, it makes use of those entities and extends some of them. Since we are only interested in the SIMPLE client here, the functions and behaviors of the servers will not be discussed. Although only a few elements are closely involved with SIMPLE client implementation, the definitions of the following entities in [33] should be given first, in order to avoid misunderstandings.

- SIP Message: The data sent between SIP users and servers. It is either a request or a response.
- SIP Request: A SIP message sent from a client to a server to invoke a particular operation.

- SIP Response: A SIP message sent from a server to a client, to indicate the status of the outcome of a request previously received.
- SIP Method: The method is the primary function that a request is meant to invoke on a server. The method is carried in the SIP request itself.
- User Agent Client (UAC): A user agent client is a logical entity that works as a network client. It creates and sends out the SIP requests and receives the SIP responses.
- User Agent Server (UAS): A user agent server is a logical entity that works as a network server. It receives the SIP request and generates a response corresponding to the request.
- User Agent (UA): A logical entity that can act as both a user agent client (UAC) and user agent server (UAS).
- Presence User Agent (PUA): A logical entity that can manipulate presence information for a user agent.
- SIP URI: A type of uniform resource identifier that is used to represent a community resource's identity in SIP. It is in the general form of *sip:user:password@host:port;uri-parameters?headers*, with some optional parts.

Under the IM environment, every user should be able to send and receive messages; users in this project are therefore all referred to as User Agents. In other words, our SIMPLE client works as both a UAC and a UAS. Regarding the presence service, our SIMPLE client also performs as a PUA.

To make the instant message exchange possible, SIMPLE extends the methods already defined in SIP by introducing three new methods: MESSAGE, SUBSCRIBE and

NOTIFY. They all inherit the features of SIP Request. MESSAGE is used for instant messaging while PRESENCE and NOTIFY are for the presence service. The message exchange in SIP/SIMPLE is similar to HTTP. It uses the same request/response model and similar response code. There are six classes of SIP response, five of which are borrowed from HTTP. A brief overview of the operations will be given next.

When one User Agent (UA) wants to send an instant message to another, the sender creates a SIP MESSAGE Request carrying the short message as the content with the receiver's SIP URI. Before the request arrives at the destination, it may travel through many SIP proxies, which will rewrite the header of the request and record the routing information, as in Section 16 of [33]. Upon receiving the message, the final recipient will generate a corresponding response and return it to the sender. If the message is properly accepted, a 200 OK response will be sent back to the original sender. Otherwise, an error code will be generated. A 200 OK response only means that the message has been successfully delivered to the final destination; it is not guaranteed that the recipient has read the instant message. Since all the routing information is recorded in the SIP Request's header, the SIP Response simply needs to copy it and route back to the sender. The MESSAGE request and response are constructed and processed according to the rules defined in Section 8 of [33].

In SIMPLE, the mechanism for the Presence service is specified through the SUBSCRIBE method, the NOTIFY method and the REGISTER method, which is already defined in SIP. Through the REGISTER method, the user agent is able to upload its presence information and let the presence server store it. With SUBSCRIBE and NOTIFY, the SIMPLE client can subscribe to the presence status of other users and be

notified when there is a status change. When a user, as a subscriber, wants to know about another user's presence information, it should create a SUBSCRIBE SIP request addressed to the SIMPLE presence server. This request contains the SIP URI of the desired user agent and is delivered as a normal SIP request. After receiving the subscription, the SIMPLE presence server will authenticate and authorize it. If authorized, a 200 OK response will be returned to the subscriber. If the authorization cannot be granted, a 202 Pending or 489 Rejected response will be returned instead. In both cases, an immediate NOTIFY message showing the status of the subscription will be sent out. Later, if the state of the subscription changes, a NOTIFY message will be sent to all the related subscribers because the presence server maintains the information of all valid subscriptions. Figure 4-2 briefly illustrates the above procedure.

Figure 4-3 SIMPLE Presence Service

4.5.2 SIMPLE Messages

SIP is a text-based protocol. Since SIMPLE is based on SIP, all SIMPLE messages, including MESSAGE, SUBSCRIBE and NOTIFY, are potentially SIP messages. Thus the message format and the headers defined in SIP are applicable to and valid for SIMPLE messages. A SIP/SIMPLE message is either a request or a response. They both consist of a start-line, one or more header fields, an empty line indicating the end of the header fields, and an optional message-body. As mentioned above, the SIP/SIMPLE message syntax and semantics are very similar to HTTP. Every request has to be matched by a specific response. A request is sent from a UAC to a UAS while a response is from a UAS to a UAC. Table 4-1 shows the details of the SIP/SIMPLE message format. A brief review of SIP headers, which are mostly used in SIMPLE, is given in Table 4-2. Some of them are borrowed from HTTP [62].

	SIP/SIMPLE Request	SIP/SIMPLE Response
Start-line	Request Line: Method Request-URI SIP-Version *Request-URI is the user or service to which this request is being addressed. (example: MESSAGE sip:A@domain.com SIP/2.0)	Status-Line: SIP-Version Status-Code Reason-Phrase *Status-Code represents a certain type of response uniquely. *Reason-Phrase is for human reading only. (example: SIP/2.0 200 OK)
Header	header-name: header-value (;parameter-name=parameter-value) (example: From: A@domain.com; tag=1234)	The same.
Body (optional)	Body type and body length are specified in the Content-Type and Content-Length header field. The interpretation of the body depends on the request method.	It may contain a body depending on the request method and status code.

Table 4-1 SIP/SIMPLE Message

Header	Description	Default Value or Comments
Accept	Acceptable message Internet media types as defined in HTTP.	Default value: application/sdp
Accept-Encoding	Acceptable message body encoding schemes as defined in HTTP.	Default value: identity encoding
Accept-Language	Language preference.	All languages are acceptable.
Allow	Set of methods supported by UA.	No information on supported methods.
Allow-Events	The event packages supported by client or server.	For SIP event notification.
Call-ID	Uniquely identify a call between two user agents in SIP.	Mandatory in all SIP requests.
Content-Length	Size of a message body in decimal number of octets.	0 means no message body.
Content-Type	Media type of the message-body.	application/sdp
CSeq	Sequence number of a request command.	Mandatory in all SIP requests.
Date	Date and time of a request or a response.	GMT time zone.
Expires	Valid time of a request or message content.	Integral number of seconds.
Event	Specify the event type, e.g. NOTIFY	Identified by event ID.
From	Initiator of the request.	Usually in SIP URI.
Max-Forwards	Limit the number of proxies or gateways when forwarding the request.	An integer in the range of 0-255.
Proxy-Authenticate	Contains an authentication challenge.	Authentication mechanism dependent.
Subject	Subject or nature of a call.	
Subscription-State	The status of the subscription.	Matches with the subscription.
To	Recipient of the request.	Usually in SIP URI.
Unsupported	Features not supported by UAS.	
Via	Record the SIP route taken by the request so far and the route to send the response.	Contains protocol name and version, transport.
WWW-Authenticate	Contains an authentication challenge for UAC to generate credentials in Proxy-Authenticate header in a subsequent request.	HTTP basic and digest authentication, and PGP.

Table 4-2 SIP/SIMPLE Message Headers

Since every SIP/SIMPLE request is replied by a response, the SIP/SIMPLE response carries the outcome or the effect of the request. It is generated by a UAS corresponding to

a request from a UAC. It may be a confirmation to a receiving request or contain additional information needed by the UAC. SIP/SIMPLE responses are classified into six categories. Five classes are taken from HTTP and one class is created for SIP. The classes are described in Table 4-3, with some examples. For a complete list of the detailed status codes, please refer to [33], [34] and [62]. The classification mechanism provides a generic understanding of the responses. In case a particular SIP response code is not understood by a UAC, the interpretation for the whole class will be used instead.

Class Code	Class Name	Description
1xx	Provisional/ informational	The server contacted is performing some further action and does not yet have a definitive response, e.g. 100 Trying, 180 Ringing
2xx	Successful	The request was completed successfully, e.g. 200 OK
3xx	Redirection	This gives information about the user's new location, or about alternative services that might be able to satisfy the call, e.g. 301 Moved Permanently, 302 Moved Temporarily
4xx	Client Error	Means that the request cannot be fulfilled because of an error with the client. The client SHOULD NOT retry the same request without modification, e.g. 400 Bad Request, 401 Unauthorized
5xx	Server Error	Indicates that the request cannot be properly handled because of an error within the server, e.g. 500 Server Internal Error, 502 Bad Gateway
6xx	Global Error	Means that a server has definitive information that the request will fail wherever it is tried, e.g. 600 Busy Everywhere

Table 4-3 SIP/SIMPLE Response Classes

Since SIMPLE is an extension to SIP, it uses the pre-defined SIP headers to form its own messages. In Table 4-4, the presence of headers will be shown according to different types of requests and corresponding responses. The “where” column describes the message type with which the header field can be used. The numerical value represents the response with the corresponding status code. The next three columns show the existence

of the header in each method. *Expected* means that the header field should be expected, but no guarantee is given.

Header	Where	MESSAGE	SUBSCRIBE	NOTIFY
Accept	415 Response	Expected	Optional	Optional
Accept-Encoding	415 Response	Expected	Optional	Optional
Accept-Language	415 Response	Expected	Optional	Optional
Allow	405 Response	Mandatory	Mandatory	Mandatory
Call-ID	Request/Response	Mandatory	Mandatory	Mandatory
Contact	2xx Response	Optional	Mandatory	Optional
Contact	3xx Response	Optional	Mandatory	Mandatory
Content-Length	Request/Response	Expected	Expected	Expected
Content-Type	Request	Expected if the message body is not empty	Expected if the message body is not empty	Expected if the message body is not empty
CSeq	Request/Response	Mandatory	Mandatory	Mandatory
Expires	2xx Response	Optional	Mandatory	Not Applicable
From	Request/Response	Mandatory	Mandatory	Mandatory
Max-Forwards	Request	Mandatory	Mandatory	Mandatory
Min-Expires	423 Response	Not Applicable	Mandatory	Not Applicable
Proxy-Authenticate	407 Response	Mandatory	Mandatory	Mandatory
To	Request/Response	Mandatory	Mandatory	Mandatory
Via	Request/Response	Mandatory	Mandatory	Mandatory
WWW-Authenticate	401 Response	Mandatory	Mandatory	Mandatory

Table 4-4 Headers Presence in SIMPLE Messages

4.5.3 Description of SIMPLE Client Functionality

Our SIMPLE Client is fully compliant with the IETF standard. However, it only implements minimum features and the extra features to support networked service access. All other features can easily be extended from this design. Table 4-5 gives an overview of the basic functionalities.

ID	Functionalities	Implementation
1	Send out normal messages (MESSAGE)	Deliver a MESSAGE request containing a short text message through a TCP socket connection. The default port is 5060. SIMPLE client works as UAC.
2	Compose a normal outgoing message (MESSAGE)	Fetch the text message from the input device, such as the keypad, and construct a MESSAGE request by adding the necessary headers, according to Table 4-4.
3	Receive Response	After sending out a request, the client listens and receives a response through the socket connection.
4	Process Response	After receiving a response, the client processes it and takes actions depending on the status information of the response. If there is more than one Via header in a response, it should be discarded. The client should be able to processes all six classes of responses as defined in [33].
4	Receive normal incoming messages (MESSAGE)	Messages are received on a socket connection listening on port 5060. The client works as UAS. It inspects the method, the headers, and processes the content as an incoming instant message.
5	Display an incoming message	If the human user chooses to read the incoming text message, it will be displayed in the message screen.
6	Subscribe another user's presence information (SUBSCRIBE)	Send out SUBSCRIBE request to the presence server through the socket connection on port 5060.
7	Compose a subscription/unsubscription	Fetch the desired presence information, and construct a SUBSCRIBE request by adding the necessary headers, according to Table 4-4.
8	Receive notification of a user's presentity (NOTIFY)	Messages are received on a socket connection listening on port 5060. It inspects the method, the headers, and processes the content as an incoming notification.
9	Prompt the notice for a notification	After a notification has been received, the human user should be informed.
10	Update the status of the contact list	Upon receiving the notification, the contact list will be updated to show the most recent presence information.
11	Edit contact list	The human user should be able to view and edit the contact list at any time.
12	Quit the application	Before shutting down the application, it should be deregistered from the Presence server by sending out a deregister request.
13	Start the application	The application is started.
14	Register with SIMPLE server	Send out a REGISTER request to the server and upload its current location.

15	Generate the provisional and final responses	After receiving any request, it should be able to generate a proper response according to the outcome of the income request. The format should comply with Table 4-4.
16	Send out responses	After the response generation, the response should be sent out through the socket connection, following the route of the incoming request.
17	Refresh the subscription	The subscription can be refreshed by sending out a newly created SUBSCRIBE request.
18	Get networked service list for UDDI Proxy	Get the networked service list registered in UDDI server through UDDI IM proxy.
19	Get networked service information	Get the service access schema in XML.
20	Invoke the networked service	Invoke the service by feeding values for the required parameter.

Table 4-5 SIMPLE Client Functions

4.5.4 SIMPLE Client Implementation

It is important to mention that all messages to the SIMPLE client are actually routed by proxies. Please note that the SIMPLE proxy is used only for relay purposes. It is not the same as the service IM proxy proposed in our framework. The proxy works on behalf of the client and is capable of transporting the messages between LANs. It does not have to be separated, as shown in Figure 4-4. Sometimes, the proxy and the client can be bound together physically or logically. With regard to the instant messaging, clients talk to each other through peer-to-peer communication. The message from user1 to user2 travels in two ways, as shown in Figure 4-4. In case 1, the two users belong to different LANs, and the message is therefore routed as u1-p1-p2-u2. Case 2 is much simpler, since both the clients are under the coverage of one local proxy. This local proxy knows the receiver's information and is able to route the message directly. Either of the two cases places the same requirements on our SIMPLE client. Considering the current test environment, Case

2 is more feasible and possible. The presence server is like a central repository. It manages the presence information and is responsible for notifying any status change.

Although the local network is represented by a solid bar, this does not mean in any way that the network must be a wired network. It could perfectly well be a wireless network.

The bar representing the network can be any kind of network that supports TCP or UDP.

In particular, the SIMPLE client is designed for small wireless devices and implemented in J2ME, thus the SIMPLE Client could be any device that contains a Java processor.

1) Case 1: User1-Proxy1-Proxy2-User2

2) Case 2: User1-Proxy-User2

Figure 4-4 SIMPLE Client Deployment Diagram

In the following figure, Figure 4-5, there are three swim lanes: the end user, the SIMPLE Client application, and the Reader. The client is split into two main processes: one for reading incoming information from the presence server or other user; another for sending out request messages. These two processes can run on their own threads concurrently. Since the incoming messages can arrive at any time, the Reader thread should always be active.

Figure 4-5 Main Activity Diagram of SIMPLE Client

The following class diagram, Figure 4-6, depicts the overall structure of the classes for the SIMPLE client. It shows the main classes and packages and their relationship.

There are five main functional classes/interfaces and two supplement packages:
 SIMPLEClient, Settings, Dialog, DialogListener, MsgHandler, SIPMessage, SIPHeader.

Figure 4-6 Overall Class Diagram of SIMPLE Client

Based on the high-level class diagram, the packages can be decomposed into the concrete classes shown in the following Table 4-6.

Class Name	Description
Address	Interface that defines the methods within a SIP Address Object.
AddressImpl	Implementation of Address interface.
Allow_Event	Represents the Allow_Event field of the SIP header. This class extends the SIPHeaderImpl class. It specifies the type of event allowed.
CallID	Represents the CallID field of the SIP header. This class extends SIPHeaderImpl class.
ComplexElement	Represents the complex element defined in XML Schema.
Constant	Static class containing all the value for constant variable.
Contact	Represents the Contact field of the SIP header. This class extends the SIPHeaderImpl class.
ContactRecord	Represents the entry for the contact list in the storage.
ContentLength	Represents the Content-Length field of the SIP header. This class extends the SIPHeaderImpl class. If the message body appears, the value should be greater than 0.
ContentType	Represents the Content-Type field of the SIP header. This class extends the SIPHeaderImpl class. It specifies the type of the media.
CSeq	Represents the CSeq field of the SIP header. This class extends the SIPHeaderImpl class.
Dialog	The interface that defines methods for the SimpleClient to send out requests and receive responses.
DialogImpl	The detailed implementation of the Dialog interface.
DialogListener	Listener that wraps up the incoming message notification. It listens to the incoming messages and handles them accordingly.
DocHandler	The adapted XML parser to handle XML document.
Element	The simple element in XML Schema.
Event	Represents the Event field of the SIP header. This class extends the SIPHeaderImpl class.
Expire	Represents the Expire field of the SIP header. This class extends the SIPHeaderImpl class.
From	Represents the From field of the SIP header. This class extends the SIPHeaderImpl class.
MaxForwards	Represents the Max-Forwards field of the SIP header. This class extends the SIPHeaderImpl class.
MethodNames	Static class containing the name of the method defined in SIP/SIMPLE.
msgHandler	Class that processes each incoming SIP message to take action and generate response information
PresenceHandler	Class that handles the presence information only.
SchemaHandler	Class that handles the XML Schema only.

Settings	Stores the information about the SIMPLE client and its related information in a permanent storage.
SIPHeader	Interface that includes the methods a SIPHeader Object contains.
SIPHeaderImpl	Implements the SIPHeader interface. This is the base class for the SIPHeader object (i.e. From, To etc.).
SIPMessage	Interface that includes the methods of the general parts of all SIP message objects, including SIP request and SIP response.
SIPMessageImpl	Implementation of the SIPMessage interface. This is the base class for each SIPMessage type(i.e. Register, Request, Response etc.)
SIPMessageSend Receiver	The adapter class for sending and receiving SIP messages.
SimpleClient	This class manages the user/GUI interaction and dispatches the incoming SIP messages (Message and Notify)
SimpleParser	Class that goes through each incoming SIP message to validate the integration of the message body.
SIPRequest	Implementation of a SIP Request object, which extends the SIPMessageImpl object. MESSAGE, NOTIFY and SUBSCRIBE all belong to this class.
SIPResponse	Implementation of a SIP Response object, which extends the SIPMessageImpl object.
StringTokenizer	Class that tokenizes a string based on the token defined.
To	Represents the To field of SIP header. This class extends the SIPHeaderImpl class.
UDDIInfo	Contains the service description information.
URI	Indicates the SIP address in URI format.
Via	Represents the Via field of the SIP header. This class extends the SIPHeaderImpl class.

Table 4-6 SIMPLE client Class Description

Figure 4-6 gives the state chart of the SIMPLE client. It shows the overview states and their changes triggered by the basic operations. For example, the “start” and “quit” actions trigger the state change from “Initial” to “Idle”. The instant messaging, the subscription and service invocation are represented in the message sending and processing states. A brief explanation of each state is given in Table 4-7.

State	Description
Initial	Basically this state is the start point and the end point of the client. After the client has finished registering with the server, it will move to the next state.
Idle	In this state, the client will simply wait for either an incoming message or the human user's command to send out the constructed message.
Processing incoming message	If the waiting state is triggered by an incoming message, the client will handle the message as described in Table 4-5. After the task is finished, the client will iterate to the waiting state.
Sending outgoing message	If the user wants to send a message, the client will be in this state. The operations were stated in scenario 3 in the previous section. After the task is finished, the client will iterate to the waiting state.

Table 4-7 Description of the SIMPLE Client States

Figure 4-7 SIMPLE Client State Chart

4.5.5 SIMPLE Client API

Based on the previous analysis of the basic classes, we need to specify the operations for those classes that are referred to as “methods” in Java. First, these APIs have to cover the minimum required operations defined in the standards. Second, we would like to offer a set of APIs that is comfortable for both the service provider and the end user. It should be as general as possible, so that the end user and the service provider can use these APIs without an extra layer of translation. Third, it should be convenient to expand for all other similar APIs and future applications built on top of it. Therefore, we need to review the relevant definitions first.

4.5.5.1 Reference API Overview

There are a few Java Specification Requests (JSR) proposed by the Java Community Process (JCP) [63] working groups that are somehow related to our SIMPLE client API definition. Some of them are fully defined and already have corresponding sample implementations, while others are still under construction or have not even been started yet. A brief overview of these JSRs, as provided in the following Table 4-8, will help us better understand our requirements and come up with a more practical design.

JSR#	Title	Description	Current Status
32	JAIN SIP API	Standard interface for SIP clients and SIP Servers to exchange information in the deployment of service-driven networked applications and services. Designed for Java Advanced Intelligent Network (JAIN) and targeted on IP telephony.	Final Release (August, 2001)
123	JAIN Presence and Availability Management	Java API to distribute and manage presence information securely across heterogeneous networks and applications. It is a user-preference- and enterprise-policy-based	Public Review (April, 2003)

	(PAM) API	presence information management system. The target platform is J2SE and J2EE.	
125	JAIN SIP Lite	A light version of Java API that wraps around the SIP protocol, allowing the application developer to write the high-level application on top of SIP without an extensive knowledge of the SIP protocol. The target platform is J2SE, J2EE.	Public Review (July, 2002)
164	JAIN SIMPLE Presence	A JAIN API that leverages the SIP extension for the Presence service to manage the presence information between a SIMPLE client and a presence server. The target platform is J2SE and J2ME.	Public Review (August, 2003)
165	JAIN SIMPLE Instant Messaging	A JAIN API that leverages the SIP extension for the Instant Messaging service to exchange short messages between SIMPLE clients. The target platform is J2SE and J2ME.	Public Review (August, 2003)
180	SIP API for J2ME	SIP API special for J2ME client. It is a multipurpose API not only for SIP but also for Instant Messaging, Presence and Gaming services. The target platform is J2ME.	Public Review (May, 2003)
186	JAIN Presence	A generic and protocol-neutral API for both IP telephony and legacy network to manage the presence information between Presence clients and servers. The target platform is J2SE and J2ME.	Unavailable for community review.
187	JAIN Instant Messaging	A generic and protocol neutral API for both IP telephony and legacy network to exchange short messages between IM clients. The target platform is J2SE and J2ME.	Unavailable for community review.

Table 4-8 Relevant JSRs overview

As described above, every JSR has its own focus. An analysis of each JSR and its relation to our SIMPLE client API design will be provided below.

JAIN SIP API is a full set of APIs that covers both low-level signaling protocol (SIP) for the session control, and high-level application development. It is relatively complicated and large-scaled. It is designed for the whole, integrated network and adopts the

JavaBean/EJB Event Model in J2EE. It includes more than 50 Java classes/interfaces, transaction control and server side operations. Given the limited resources available to our SIMPLE client, this API is too big to fit any mobile device and J2ME environment. In addition, since our SIMPLE client is just a typical end-user application, choosing a framework designed for service development across the whole telecommunication network would be overkill.

PAM is a high-level, abstract, protocol neutral API for presence information management [64]. Instead of trying to replace the existing protocols for communication and networked services, it attempts to provide a generic mechanism to manage the presence and availability information over multiple services and networks. Its coverage spans various communication systems, not just instant messaging. It only focuses on issues on the server side and does not specify the protocol for the Presence service. In our case, we need the API for a client, and the protocol used will be SIMPLE. Therefore, PAM is not appropriate and should be eliminated.

JAIN SIP Lite has a small footprint while still maintaining the core features of the underlying SIP protocol. It is defined for high-level applications for a SIP user agent. The developer does not need to have an in-depth knowledge of the SIP protocol. Aside from that, it provides session management and other support for the handling of SIP multimedia dialogs, which go beyond the scope of Instant Messaging and the Presence service. The platform targeted is J2EE/J2SE and does not take into account J2ME CLDC terminal limitations [65]. It seems that part of this API satisfies the situation for our SIMPLE client and the advantage is obvious. However, fully implementing this API for our SIMPLE client is still not the best choice.

The JAIN SIMPLE Presence and IM API defines an application level API for the Client and Presence server on top of SIP, according to the IETF standard. It offers a limited extension to the JAIN SIP and JAIN SIP Lite to support instant messaging and presence, as well as the session management. It provides a standard method of cooperating with the SIP/SIMPLE stack. The architecture is minimal without sacrificing functionality. It is dedicated to J2ME, in order to meet the stringent requirements of small devices. Compared to the others, it is simple and easy for common developers to implement it. JAIN SIMPLE JSR defines eight interfaces. They are Address, Message, IMClient, IMListener, PresenceAgent, PresenceUserAgent, SimpleListener and Watcher. As shown by the names, each interface specifies the behaviors associated with the corresponding entity. Table 4-9 summarizes the main functions of those entities.

Name	Description	Comment
Address	It represents the SIP address whose format is described in [33].	It gives the generic format of SIP address.
Message	It represents the SIP/SIMPLE message, including SIP request and SIP response.	All the header fields are expressed by the text string.
IMClient	It is capable of sending and receiving instant messages.	It is only defined for IM.
IMListener	It processes the incoming instant messages.	It is only defined for IM.
PresenceAgent	This agent is responsible for handling SUBSCRIBE requests and generating notifications if status change happens.	It is supposed to function as the presence server. It is not the focus of the work of this thesis.
PresenceUserAgent	It is responsible for managing the presence information for a user agent.	It is specially defined for Presence service.
SimpleListener	It processes the incoming SIP /SIMPLE messages.	It is generic for all SIMPLE messages.
Watcher	It is responsible for generating SUBSCRIBE requests and handling the notification.	It is specially defined for Presence service.

Table 4-9 JAIN SIMPLE API (JSR 164/165)

As we can see from the above table, the IM and presence service are treated separately. The main difference comes from the different SIP requests. It is possible to integrate those two parts together and maximize the code reuse. Besides that, another drawback is that the header field is represented as pure text string. It is hard to add and process new headers. So we should consider offering a new entity dedicated to SIP/SIMPLE headers. Considering the size, the environment, and the functionality, this one is the best reference for our SIMPLE client, although the session management component may not be necessary in our situation.

SIP API for J2ME is a lightweight, generic SIP API, specially designed for J2ME. Any other extension could build on top of it and thus make use of it. In JSR 180, there are more than ten interfaces or classes. The structure is quite different from JSR 164/165.

Table 4-10 gives an overview of the main content.

Name	Description	Comment
SipAddress	It represents a generic SIP address.	It is defined in all related JSRs.
SipClientConnection	It represents SIP client transaction. It generates SIP request and response.	It is for SIP client.
SipClientConnectionListener	It is for incoming SIP response.	It is associated with SipClientConnection.
SipConnection	This interface is implemented by SipClientConnection and SipServerConnection.	It is used for both client and server to create SIP request and response.
SipConnectionNotifier	This interface defines a SIP server connection notifier.	It is for SIP server only.
SipDialog	It represents the SIP Dialog. It can be retrieved through SipConnection.	It connects the two end users of the SIP call.
SipHeader	It defines the operations on SIP headers.	The SIP header is represented as name-value pair.
SipRefreshHelper	This class is used to refresh subscriptions or registrations.	It is used for SIP client periodically.
SipRefreshListener	Listener interface for	It is associated with

	RefreshHelper events	SipRefreshHelper.
SipServerConnection	It represents SIP server transaction. It is created by the SipConnectionNotifier when a new request is received.	It is for the SIP server only.
SipServerConnectionListener	Listener interface for incoming SIP requests.	It is for the SIP server only.

Table 4-10 SIP API for J2ME

From above table, we can see that JSR 180 targets SIP only. It involves both the client and server. Its functionalities are performed through transactions and SIP Dialogs. Its focus is different from SIMPLE, especially for the SIMPLE client in our case. For example, the SipRefreshHelper is not necessary for our SIMPLE client and message exchange can be simplified rather than using SipConnection. SIP header is defined here, but SIP message is not described. The concept of Dialog is not important for pager-mode instant messaging, but we would like to keep it in our design for later expansion of session-mode instant messaging. Therefore, we are seeking a hybrid which is simpler and more object oriented targeting only on SIMPLE clients. The JAIN SIMPLE API is a better fit for our requirement in term of the size and the supporting functionalities. Therefore, we adopt the main part from it. For the detailed class description of our SIMPLE client, please refer to Table 4-6.

After comparing the currently available JSRs, we can see the advantage of the newly defined JAIN SIMPLE Presence and IM API. Our SIMPLE client API should adopt its main part and put the focus only on the client side. Furthermore, considering the unprecedented growth of hardware processing capability and the nature of the Java language, our design will be more detailed, more object-oriented and thus easier to extend by taking the advantage of polymorphism. Without scarifying the simplicity, we

should also consider other reference JSRs to complement our design for the service access to the end user. As for the actual implementation, more helper classes will be provided for clarification and the user's convenience.

4.5.5.2 SIMPLE Client API in J2ME

According to above analysis, we defined the J2ME API for our SIMPLE Client. To retain the concept of object-orientation across our project, all the necessary entities in SIMPLE are represented by classes. In terms of the size and functionality, some of them are relatively complicated. Inheritance is applied to the concrete entities that contain the abstract and general format.

The APIs for the underlying classes such as Address and Message will be similar to those in [65] and [66]. The only difference is that we tried to present the helper entities, such as URL and SIP header, as classes, while they are treated as pure text strings in [65] and [66]. Note that for the message construction and processing we borrowed the mechanism from JSR 32 [67] which, in our minds, is more classical and better for the message processing.

<i>API</i>	<i>Description</i>	<i>Comment</i>
SimpleClient()	Create a SimpleClient object. The SimpleClient object holds the GUI. It starts a server socket on 5060 port to receive incoming SIP messages (NOTIFY, MESSAGE, etc.). The response is received through the DataInputStream associated with the DataOutputStream that sends out the SIP message. Each outgoing message (REGISTER, MESSAGE etc.) is handled by Dialog object. The incoming message is handled by the	

	msgHandler	
startApp() : void	Starts running the SimpleClient.	For J2ME. Only.
pauseApp() : void	Stops the running SimpleClient instance. The application stops receiving user input while the instance is not destroyed.	Same as above.
destroyApp (unconditional:Boolean) : void	Destroy the Midlet object created in the SimpleClient object.	Same as above.
CommandAction (c: Command, d: Displayable) : void	Handles the user input command and takes action.	For GUI interaction in J2ME.
postMessage(message:String): void	Posts a String in the Midlet Display screen.	Display on screen.
Dialog(called:String, from:Address, to:Address, body:String)	Constructs a Dialog object which takes responsibility for sending out the SIP request and receiving the response.	Similar to JSR 32. (1)
getOutgoingMessage() : SIPMessage setOutgoingMessage msg:SIPMessage) : void	Getter and setter of outgoing message associated with this DialogImpl object.	Similar to JSR 32.
getLocalAddress():Address getRemoteAddress():Address	Gets the address from the "From" and "To" headers.	Similar to JSR164 /165. (2)
setDialogListener(d:DialogListener): void	Adds the listener associated with the outgoing message.	Similar to JSR 32. (3)
createRequest(method:String,content _type:String, payload:byte[]):SIPRequest	API for constructing the outgoing SIP message.	(4)
createResponse(in_request:SIPMessage ,statuscode:String, reason:String, payload:byte[]):SIPMessage	Constructs a Response object according to the income SIPMessage.	(5)
sendMessage(msg:SIPMessage):String	This method is used to send both requests and responses based on the instance type of msg. If it is a SIPRequest, it returns the response message as a String. If it is a response, it returns null.	(6)
notifyClientResponse (client:DatagramClient):void	It receives the response returned and posts the information according to the status code.	Similar to JSR 180. (7)
getServiceList():String()	This method parses an XML message and retrieves the list of	Defined for service

	available web services	access.
getServiceInfo():String	This method retrieves the Web service schema file from an incoming message	Same as above.
getService():String	This method gets the web service data from a message	Same as above.
displayWebService(String):void	This method constructs the networked service user interface based on the schema file	Same as above.

Table 4-11 SIMPE Client API

Regarding the SIMPLE client API described in Table 4-11, the following detailed comments are given to explain the rational behind our proposal in Table 4-12. The ID no. corresponds to the number indicated in the last column in Table 4-11.

ID	Detailed Explanation
1	In JSR 180, Dialog is associated with SipClientConnection which deals with SIP messages. Since we discard the use of SipConnection, the Dialog will be associated with the SIP messages directly. As mentioned earlier, this design gives the flexibility for further expansion to the session-model instant messaging.
2	These methods are quite useful to identify the sender and the receiver of a SIP message. The information will be fetched from the headers of the concrete messages. In JSR164/165, these methods are defined as getTo() and getFrom(). We just replace them with more self-explanatory signatures.
3	The DialogListener is the only listener associated with Dialog. It works in the same way as SimpleListener in JSR 164/165 and SipClientConnectionListener in JSR 180.
4	In JSR 164/165, a SimpleFactory is used to create SIP request. Instead of defining a separate factory class, we just place this function right inside our SIMPLE client for simplicity. But it is feasible to expand it and replace it with a factory class.
5	In JSR 164/165, the SIP response is created without content. According to the standard, the SIP response may carry a body. To make this method more general, we add a parameter to specify the content. If the SIP response body is empty, the content will be set to null.
6	In JSR164/165, different SIP request and response are sent out through different method. For example, sendSubscription() is defined for SUBSCRIBE SIP request only. Our message delivery method is generalized to all SIP messages including SIP request and response. Because all the SIP messages are constructed with proper headers and the sender and receiver's information can be retrieved through getLocalAddress() and getRemoteAddress(), the whole SIP message is passed as the input parameter.

7	In JSR164/165, different methods are defined for instant messaging and presence service. For generality purpose, we would prefer a generic format. In JSR 180, notifyResponse() is associated with SipClientConnection. Since we do not define such transaction based entity, we have this notification mechanism work with our SIMPLE client directly.
---	--

Table 4-12 Comments on SIMPLE Client API Design

4.6 SIMPLE Server

There are many SIP servers in the market. In our case, we need a SIP server that also supports SIMPLE. It means the server should support the MESSAGE, SUBSCRIBE and NOTIFY methods, and the events associated with the Presence service. There are some candidates: for example, Antepo's OPN SIMPLE server [29], Indigo's Communications server [68] and nLoci's Atrium server [69]. Those applications all claim to comply with the standards. The differences only come with the choice of background technology, such as the J2EE environment, database and user interface. Therefore, our client should be compatible with all of them.

In our case study, we use the Atrium SIP/SIMPLE server. It is a J2EE SIP server for Windows and Linux. It complies with SIP and SIMPLE standards and is implemented in Java. It is built on top of the JBoss system [70] and the MySQL database [71].

4.7 Message Parsing and User Interface

Two types of message parsing are required, for different purposes. One is for the SIP/SIMPLE messages that are used to carry all the necessary information across the network. Another one is for the XML-formatted document, which contains the service invocation information.

With regard to the former, the message processing will simply follow the standard, such as the header verification and the header value parsing, etc. After checking all the mandatory SIP/SIMPLE fields, the content will be left for further processing and action. For example, after receiving a valid NOTIFY message, the end user will update its contact list accordingly. When a MESSAGE SIMPLE message is received, the end user should check the body of the instant message. If it is a normal text message, it will be prompted on the screen. If it involves service invocation, the message needs further parsing by an XML parser. This XML parser will be able to understand the XML document and strip off the XML Schema and XML document by examining the corresponding XML tags.

For the message content in XML, a proper XML parser is required to fetch the key information for the service invocation. In the case study, we just need the basic filtering functionality of the XML parser, preferably in J2ME so that we can adapt it to our program directly.

There are two main techniques for XML parsing. One is DOM (Document Object Model); the other is SAX (Simple API for XML). They adopt very different approaches for dealing with the XML document. DOM creates a hierarchical tree of nodes based on the XML document; the information can be obtained by manipulating those tree nodes. With SAX, the parser catches the desired information and notifies the user application in the form of a parsing event. Each of them has pros and cons with regard to speed and resource consumption, and they are widely used in practice.

In our case study, we used a small Java parser called the QD (Quick-and-Dirty) parser [72]. It is an event-based XML parser, similar to SAX but with limited functionality and

a small footprint, which fits J2ME. It satisfies our minimum requirements for XML document parsing, such as recognizing the tags and attributes, etc.

Regarding the user interface, we kept it as simple as possible, since J2ME provides very limited GUI components. Only some soft buttons and a command list need to be shown for the end user to perform the actions. The content of the instant message or the service invocation result will be displayed within a text field. Here the possibility of developing a fancy GUI is left up to the individual.

4.8 Summary

The SIMPLE client API defined here provides a basis that can be improved in the future. Extra features will be supported by extending this set of APIs. Although it is targeted to J2ME in this thesis, the structure and the mechanism can be migrated onto other platforms. So far, it is the first SIMPLE client API in J2ME.

Chapter 5 Case Study

This chapter gives a sample scenario to illustrate how our proposed framework can be used in the real world. To avoid reinventing the wheel, we will take advantage of the existing products and APIs and only provide our own implementations when necessary.

5.1 UDDI / UDDIe Server

In our proposed framework, a very important service proxy represents the UDDI/UDDIe service. Therefore, in this example, we need a UDDI/UDDIe server that manages the information for a number of services.

UDDI is well-acknowledged by the industry and research groups. Many companies, including IBM, Sun, and Microsoft, have given their support to UDDI. For example, Microsoft's Enterprise UDDI server is based on the .Net framework and associated with Microsoft Data Engine by default. It provides the development APIs in its own language, such as Visual Basic. IBM's UDDI server is deployed under its WebSphere environment and is supported by the DB2 database.

Currently UDDIe is more at the research stage. Although UDDIe is enriched with more service information and inquiries, we will consider the use of UDDIe's features as future work, because in this research study only the basic idea of our framework needs to be demonstrated.

For simplicity, we chose the GLUE UDDI server [73]. The GLUE UDDI server was developed by The MindElectric for publishing and searching Web services. It is implemented in Java and integrated with a relational database with a small footprint.

Compared to other UDDI servers, GLUE is simple, lightweight and free. It offers the corresponding Java API and supports SOAP1.2, WSDL1.1, and UDDIv2.

5.2 Networked Service – Google

To illustrate the idea of our framework, we need a concrete example of a networked service that will be discovered and accessed by our SIMPLE IM client through the mechanisms described in Chapter 3.

The Google Web service was chosen as an example of a networked service here, mainly because Google is very popular and has free API for developers. Google provides a set of Java APIs that enable software developers to implement their own applications to execute the Google web query. Google Web APIs are implemented as a Web service [74], which is a type of networked service. The corresponding Google IM proxy will use those APIs to query the Web on behalf of the end user.

As a key part of our framework, a UDDI IM proxy associated with the GLUE UDDI server will be implemented as well. The UDDI IM proxy is the entry point for the IM client to find the information about the desired services. It is an IM peer, so it should comply with SIMPLE in our case. Since the service IM proxy can reside on any device, the implementation does not have to be in J2ME. In this example, both the UDDI IM proxy and Google IM proxy are implemented in J2SE.

5.3 An Example of Networked Service Discovery and Access

Figure 5-1 depicts how the IM entities cooperate with each other to access the Google service, based on our proposed framework. The details will be shown later.

Figure 5-1 Interactions for Google service discovery and access

0) Start UDDI Server

The GLUE UDDI server is started and waits for service registration, as in Figure 5-2.

```

D:\Program Files\Xinox Software\JCreator Pro\GE2001.exe
[STARTUP] GLUE Standard 4.1.2 (c) 2001-2003 The Mind Electric
[STARTUP] soap/http server started on http://192.168.0.2:8004/glue/inquiry. /uddi
inquiry/
[STARTUP] soap/http server started on http://192.168.0.2:8005/glue/publication.
/uddipublication/
[WARNING] GLUE standard UDDI server only holds up to 5 services
[WARNING] for unlimited storage, upgrade to GLUE professional
  
```

Figure 5-2 Start GLUE UDDI server

1) Publish Google Web Service

The information about the Google Web Service is published in the UDDI server, including the <Business> entry and the corresponding <tModel>. The details are shown

in Figure 5-3 for <tModel> and Figure 5-4 for <businessEntity>. The following code segments illustrate how to create a UDDI client and register the related information with the GLUE UDDI server. The detailed procedure is compliant with the UDDI standard.

```
// create UDDI client
UDDIClient uddi = new UDDIClient( inquiryURL, publicationURL, user, password );

// create TModel for Google Web Service
TModel tModel = new TModel( "Google Web Service" );

// add a Description to the TModel
tModel.addDescription( new Description( "Google Search Engine" ) );

// publish the TModel
TModel savedTModel = uddi.saveTModel( tModel );
```

Figure 5-3 <tModel> of Google service registration

```
// create a Binding for the specific implementation of the Google Web Service.
Binding binding = new Binding();
binding.addTModelInstance( new TModelInstance( tModelKey ) );
AccessPoint accessPoint = new AccessPoint( "google@mydomain.com", "other" );
binding.setAccessPoint( accessPoint );

// create a service with the single SOAP binding
Service service = new Service( "google" );
service.addBinding( binding );

// save the service
```

Service savedService = uddi.saveService(service);

```
Dr:\Program Files\Xinox Software\JCreator Pro\GE2001.exe
[STARTUP] GLUE Standard 4.1.2 (c) 2001-2003 The Mind Electric

saved
<Business>
  <businessEntity businessKey='22A6999B-E25C-059B-9690-BAF17D9684E5' operator='U
ddi.me'>
    <discoveryURLs>
      <discoveryURL useType='businessEntity'>http://192.168.0.2:8004/glue/inquir
y/?businessKey=22A6999B-E25C-059B-9690-BAF17D9684E5</discoveryURL>
    </discoveryURLs>
    <name xml:lang='en'>Google</name>
    <name xml:lang='fr'>name two in french</name>
    <description>Google Search Engine</description>
    <contacts>
      <contact useType='CEO'>
        <description xml:lang='en'>Google Web Service</description>
        <personName>Google</personName>
        <phone useType='toll-free number'>1-800-222-0000</phone>
        <email useType='not secure'>service@google.com</email>
        <address useType='headquarters' sortCode='72245'>
          <addressLine>Google Web Site</addressLine>
          <addressLine>World Wide Web</addressLine>
        </address>
      </contact>
    </contacts>
    <categoryBag>
      <keyedReference keyName='ntis-gov:naics.1997' keyValue='4532' tModelKey='U
uid:C0B9FE13-179F-413D-8A5B-5004DB8E5B62' />
    </categoryBag>
  </businessEntity>
</Business>
Press any key to continue...
```

Figure 5-4 Google service UDDI registration

2, 3) Register Google Proxy and UDDI IM proxy

The Google Proxy registers itself with the Atrium SIP/SIMPLE server by sending a SIP REGISTER method. The UDDI IM proxy does the same. The detailed message is shown in Figure 5-5 for UDDI IM proxy registration.

A screenshot of a Windows command window titled "D:\Program Files\Xinox Software\JCreator Pro\GE2001.exe". The window displays a SIP REGISTER message in text format. The message content is as follows:

```
The send message is:-:-  
REGISTER sip:mydomain.com SIP/2.0  
Via: SIP/2.0/UDP 192.168.0.2:1234  
From: <sip:uddi@mydomain.com>;tag=aed40abc-86a1-4d1b-a6bc-a404847b91fe  
To: <sip:uddi@mydomain.com>  
Call-ID: 1aef64b3-c612-4d36-b065-2b686976d32e@192.168.0.2  
CSeq: 1 REGISTER  
Contact: <sip:192.168.0.2:1234>;methods="INVITE, MESSAGE, INFO, SUBSCRIBE, OPTIO  
NS, BYE, CANCEL, NOTIFY, ACK"  
User-Agent: Windows RTC/1.0  
Expires: 1200  
Event: registration  
Allow-Events: presence  
Content-Length: 0
```

Figure 5-5 Register UDDI proxy with SIMPLE Atrium Server

4) Register end user (SIMPLE Client)

When the SIMPLE client is started, it is configured to register with the Atrium server automatically by sending a REGISTER message. The format is the same as shown above.

5) Subscribe to UDDI IM proxy and search for services

The UDDI IM proxy can be treated as a normal IM peer. It can be added as a friend into the contact list by sending a SIMPLE SUBSCRIBE message to the Atrium server, so that the end user will be able to monitor the availability of the UDDI service. The SIP address will represent the IM peer's identity, which is *uddi@mydomain.com* for UDDI IM proxy.

The following Figure 5-6 shows step 4) and 5) respectively.

Figure 5-6 Start SIMPLE Client (left) & Subscribe to UDDI proxy (right)

6) Get available services

The end user, a SIMPLE client in this implementation, then sends SIMPLE MESSAGE messages to the UDDI proxy and asks for the available service list stored in the UDDI server. As shown, the Google web service is returned within the `<WebServiceList>`. A screen snapshot is shown in Figure 5-7.

A screenshot of a command window titled "D:\Program Files\Xinox Software\JCreator Pro\GE2001.exe". The window displays the following text:

```
The send message is===  
MESSAGE sip:uddi@mydomain.com SIP/2.0  
Via: SIP/2.0/UDP 192.168.0.2:6789;branch=z9hG4bKxktxnedjbadqxjmxzzgw  
Via: SIP/2.0/UDP 192.168.0.2:5060  
CSeq: 9 MESSAGE  
Call-ID: 1077504703307@192.168.0.2  
To: <sip:sion@mydomain.com>;tag=sion-1077504703287  
From: <sip:uddi@mydomain.com>  
User-Agent: Windows RTC/1.0  
Record-Route: <sip:192.168.0.2>;lr=true  
Contact: <sip:192.168.0.2:1234>  
Content-Type: text/plain  
Content-Length: 102  
  
_WebService  
<WebServiceList>  
Google Web Service||google@mydomain.com  
</WebServiceList>  
_WebService  
after the send
```

Figure 5-7 Get Service List

7) Select Google proxy and ask for service

The Google service is implemented as a selectable item in the left of Figure 5-9 to facilitate the user's screen interaction. By simply selecting the icon, the end user will send an IM message to request the service access template, as described in Chapter 3.

8) Get Google service access template

The corresponding service access template will be returned to the end user. The XML Schema for service invocation will be encapsulated in the IM message. Figure 5-8 shows the SIMPLE MESSAGE containing the service invocation XML Schema.

A screenshot of a Windows command window titled "D:\Program Files\Xinox Software\JCreator Pro\GE2001.exe". The window displays the details of a SIP message being sent. The text is as follows:

```
The send message is====  
MESSAGE sip:google@mydomain.com SIP/2.0  
Via: SIP/2.0/UDP 192.168.0.2:6789;branch=z9hG4bKzqtrngioxzgjlfkzumzi  
Via: SIP/2.0/UDP 192.168.0.2:5060  
CSeq: 11 MESSAGE  
Call-ID: 1077507051133@192.168.0.2  
To: "sion" <sip:sion@mydomain.com>;tag=sion-1077507051113  
From: "google" <sip:google@mydomain.com>  
User-Agent: Windows RTC/1.0  
Record-Route: <sip:192.168.0.2>;lr=true  
Contact: <sip:192.168.0.2:2345>  
Content-Type: text/plain  
Content-Length: 157  
  
<?xml version="1.0" ?><xs:element name="Service Access"><xs:element name="Search"  
" type="xs:String"/><xs:element name="Return" type="xs:String"/></xs:element>  
after the send
```

Figure 5-8 Google Service Access Template

9) Generate GUI for Google service

The QD parser on the end user side will first parse the XML Schema and turn it into a form. In this example, only the search string and the number of returned results will be required. Figure 5-9 shows step 7) and the GUI generated for the end user to perform the Google search.

Figure 5-9 Display Google Service (left) & Generate GUI for Service Access

10) Access Google service

The end user's input will be wrapped within an XML file and sent out to the Google proxy through a SIMPLE MESSAGE message. The detailed SIMPLE MESSAGE is shown on the right hand-most side of the window in Figure 5-10.

Figure 5-10 Message Details for Google Service Access

11) Contact the backend Google server

Upon receiving the above SIMPLE messages, the Google IM proxy will parse the XML file and forward the required information (search string and number of returned result here) to the background Google server to invoke the service. The following code segments show how to perform a Google search using Google Search APIs.

```

//Get Number of search result
int maxResults =integer.parseInt(keyword.substring(keyword.indexOf("||")+2));
Vector value=new Vector();

//Perform Google Search
IGoogleSearchPort google = GoogleSearchServiceHelper.bind();
GoogleSearchResult result = google.doGoogleSearch( key,
keyword.substring(0,keyword.indexOf("||")), start,
maxResults, false, null, true, nul,null,null );
ResultElement[] results = result.resultElements;
//Read Search Results
for( int i = 0; i < results.length; i++ )

```

```

{
    System.out.println( "Title: " + results[i].title );
    System.out.println( "Summary: " + results[i].summary );
    String item=new String(results[i].URL + "\r\n");
    value.addElement(item);
}

```

12) Return the search result

The search result will first be sent back to the Google proxy, using its specific protocol.

Then the Google proxy will construct an IM message containing the search result within an XML document. Here, the search result is simply embedded within a <data> element.

Figure 5-11 shows the content of the SIMPLE message.

Figure 5-11 Returned result from Google proxy

13) Display the result for the end user

The QD parser will parse the IM message and filter out the results. For demonstration purposes, the message details are displayed completely on the screen, as shown in Figure 5-12.

Figure 5-12 Result received by the SIMPLE Client

5.4 Related Statistics in Case Study

This section will show some time related data collected during our case study. A short analysis will follow. However, the performance issue is out of the scope of this thesis. Please note the service registration and other configuration issues are considered as the prior condition for our case study. Therefore, only the steps involving service access are evaluated in the following table. The data may vary due to different network conditions and the individual computer's processing power.

Function Description	Time	Comment
Get available service list and post it on the display screen of the J2ME simulator. (Step 5&6)	4.25 seconds	The SIIMPLE client communicates with the UDDI service proxy and gets the available service list registered in UDDI Glue server. The communication is carried through SIP MESSAGE requests. It includes the time for inquiring the UDDI server. The bottleneck lies in the internal search operation of GLUE Server. The search operation involves classloader technology which is time consuming. It also takes time to search the database and parse XML file.
Get Google service access template and generate a form on the display screen of the J2ME simulator. (Step 7, 8 & 9)	<1 second	The SIMPLE client requests the service access template from Google service proxy and translates the XML Schema file into a simple form. It is fast since the SIMPLE client just needs to parse a very simple XML Schema file and generates several text fields accordingly.
Access Google service and display the search result. (Step 10, 11 & 12)	4.98 seconds	The SIMPLE client sends the desired parameters to Google service Proxy and gets the search result. It includes the time for communication between SIMPLE client and Google service proxy and the time for executing the Google search. By calling the Google search API, the Google proxy actually has to go through the internet to connect the Google search engine and query its database. It depends on the speed of the internet connection and the structure of the database. Therefore, it consumes relatively long time.

Table 5-1 Statistics for Case study

The SIMPLE server runs on Pentium IV 1.9GHz PC with 256M memory. The hardware platform for the rest applications is Pentium III 733MHz PC which has 384M memory and 1M bps internet connection. The two PCs are connected on local area network. The data in Table 5-1 are calculated by taking the average of 5 experimental runs with the variances of 0.0156, 0.003 and 0.022 respectively. With better hardware and faster

internet connection, the performance is expected to be improved. Ideally, the SIMPLE client should be tested on handheld devices under wireless environment. Due to the limitation of current laboratory, we just want to provide some related information here.

5.5 Summary

The above example illustrates the main idea and mechanism of our proposed framework. Although there are still some potential features that are worth exploring, we can see the feasibility and simplicity of our solution. It provides a prototype for more sophisticated scenarios and most importantly, it promotes a new concept that enriches our traditional communication methods.

Chapter 6 Summary and Future Work

6.1 Summary

As we have seen in this thesis, there are many service discovery and access protocols that address the same problem via different solutions and foci. Some of them offer serializable service objects, as in JINI, and others rely on IP broadcast. Coupled with the Presence service, Instant Messaging enables the user to get the most up-to-date status of the desired services. Combined with the UDDI web service categorization mechanism, our proposed service discovery can be improved not only through the traditional pulling method, but also through the push method. The user does not have to keep asking if the specific service is available or not. Instead, once the user subscribes to the specific service, which is represented as a service proxy in our framework, the user will be notified of any status change in the service. This saves bandwidth usage and reduces the network traffic.

Furthermore, we offer a service representation schema. With a minor extension, it allows the normal user to access more complicated services through a simple and friendly interface. Therefore, the current millions of IM users are almost ready to receive the real advantage of networked services.

Our solution is built on existing industry infrastructure. It is easy to adopt and extend. For the service provider, all the work will be done simply by implementing its own service IM proxies and attaching them to the IM domain. To enjoy the networked services, the end user only needs to load a small program that is able to interact with the service IM

proxies. It is efficient and economical. SIMPLE serves as a good candidate to show our ideas in this thesis.

By introducing an IM-based service provisioning and access framework, the thesis proposes answers to the questions in Chapter 1.4. Instant messaging and presence service can serve as both the communication and the service discovery protocol to support networked service provisioning and access. The service IM proxy, which appears as a normal IM peer, facilitates the IM end user to access the service by providing a service access representation scheme. The Web service categorization mechanism can be generalized to any networked service and represented as a special service in our framework. By giving the consideration to the current existing infrastructure and the industry support, SIMPLE was chosen for our sample implementation.

6.2 Future Work

Although this thesis proposes a generic service-provisioning framework, there are still some issues left for future research. The work of this thesis could be enhanced in the following ways:

1. The service discovery through UDDI could be refined. How can the desired service be located more precisely and specifically? One of the possible solutions is to use UDDIe. Since UDDIe support searching by attribute, it will offer the end user more search options, such as by geographic location or by service property. A more powerful search function can be implemented.
2. As mentioned earlier, the use of context information will be another challenge for wireless devices, due to their mobility. Some JAIN solutions may provide

supplementary methods of allocating nearby services with regard to some specific situations.

3. Instead of using XML Schema, as proposed in this thesis, an alternative might be to use XML Stylesheet language to convey the service invocation information. In this case, the end user will be informed not only of the XML document for service invocation details, but also on how to parse the attached XML file.
4. Support for SOAP can be implemented on the service IM proxy side, so that the backend service will interact with the service IM proxy as a general service call.
5. Last but not least, the security issue is open for consideration. In our case, since SIMPLE is an extension to SIP, we may inherit the security features from SIP directly.

References

- [1] Instant Messaging, NAR's Web Intelligence Summary, July 2001
<http://www.realtor.org/webintell.nsf/0/9d9b6f0d9a364b7886256aa900510d4d?OpenDocument>
- [2] Wireless Village: The Mobile IMPS Initiative, white paper
http://www.openmobilealliance.org/wvdocs/wv_white_paper.pdf
- [3] Wireless Village: The Mobile IMPS Initiative, System Architecture Model
Version 1.1", 2001
http://www.openmobilealliance.org/wirelessvillage/docs/WV_Architecture_v1.1.pdf
- [4] Fredrik Espinoza. Towards Individual Service Provisioning. In W. Lewis
Johnsson, Elisabeth André, and John Domingue, editors, In 2003 International
Conference on Intelligent User Interfaces (IUI 2003), pages 239--241. ACM Press,
January 2003.
- [5] Cheng, S., Liu, J, Kao, J., Chen C., A New Framework for Mobile Web Services.
In 2002 Symposium on Applications and the Internet (SAINT) Workshops, pages
218-222. January 2002.
- [6] Maamar, Z., Mansoor, W., Mahmoud, Q.
Software Agents to Support Mobile Services. A poster presentation (two pages
published in the proceedings) in the First International Joint Conference on
Autonomous Agents and Multi-Agent Systems, July 2002.
- [7] Googlematic
<http://www.softonic.com/ie/20187>

- [8] Woods, B., 2002. Search the Web via IM.
<http://www.internetnews.com/bus-news/print.php/1012411>
- [9] Curtis, B & Hsu, J. 2003. An e-mail user interface to Web servers.
<http://www-106.ibm.com/developerworks/webservices/library/wi-email/>
- [10] ActiveBuddy
<http://www.activebuddy.com>
- [11] YIMGoogle
<http://www.tropo.com/techno/java/yimgoogle/>
- [12] Forno, F. SOAP Over Jabber
<http://www.jabber.org/jeps/jep-0072.html#sect-id2595921>
- [13] W3C SOAP 1.2 Specification
<http://www.w3.org/TR/soap>
- [14] JOXA Project
<http://joxa.sourceforge.net/>
- [15] Webmethods's GLUE
<http://www.webmethods.com/Solutions/Glue/>
- [16] KSOAP Project
<http://ksoap.enhydra.org>
- [17] Krause, S., Magedanz, T., Mobile Service Agents enabling "Intelligence on Demand" in telecommunications, In proceeding of IEEE Globecom'96, November 1996
- [18] Fredrik Espinoza and Lucas Hinz. Generic Peer-to-Peer Support for a Personal Service Platform. In Sumi Helal, Yuji Oie, Carl Chang, and Jun Murai, editors, In

2003 Symposium on Applications and the Internet (Saint 2003), pages 218--225, Los Alamitos, California, January 2003. IEEE Computer Society.

- [19] JXTA Web site
[http:// www.jxta.org](http://www.jxta.org)
- [20] Activebuddy Product Web site
<http://www.activebuddy.com/products/index.shtml>
- [21] M. Day, J. Rosenberg, H. Sugano, A Model for Presence and Instant Messaging, RFC 2778, February, 2000.
<http://www.ietf.org/rfc/rfc2778.txt?number=2778>
- [22] M. Day, J. Rosenberg, H. Sugano, Instant Messaging / Presence Protocol Requirements, RFC 2779, February 2000.
<http://www.ietf.org/rfc/rfc2778.txt?number=2778>
- [23] The Wireless Village initiative: White Paper.
http://www.openmobilealliance.org/WirelessVillage/docs/WV_White_Paper.pdf
- [24] Wireless Village Web site
<http://www.wireless-village.org>
- [25] Wireless Village 1.0 Specifications
<http://www.openmobilealliance.org/WirelessVillage/index.html>
- [26] Jabber Web site
<http://www.jabber.org>
- [27] Jurvis, J., 2001. Jabber: Smooth Talking for Short Messages.
<http://www.camworld.com/archives/000667.html>
- [28] <http://www.streampath.com>

- [29] Antepo Company Home page
<http://www.antepo.com>
- [30] Peter Saint-Andre, 2002. XMPP and Jabber.
<http://www.saint-andre.com/jabber/xmpp+jabber.html>
- [31] P. Saint-Andre, Extensible Messaging and Presence Protocol (XMPP): Core, Internet-Draft. January, 2004.
<http://www.ietf.org/internet-drafts/draft-ietf-xmpp-core-22.txt>
- [32] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence, Internet-Draft. January, 2004.
<http://www.ietf.org/internet-drafts/draft-ietf-xmpp-im-21.txt>
- [33] J. Rosenberg, H. Schulzrinne, G. Camarillo.
SIP: Session Initiation Protocol, RFC 3261. June 2002.
<http://www.ietf.org/rfc/rfc3261.txt?number=3261>
- [34] A. B. Roach. SIP-Specific Event Notification, RFC 3265. June 2002.
<http://www.ietf.org/rfc/rfc3265.txt?number=3265>
- [35] B. Campbell, J. Rosenberg, H. Schulzrinne. Session Initiation Protocol (SIP) Extensions for Instant Messaging, RFC 3428. December 2002.
<http://www.ietf.org/rfc/rfc3428.txt?number=3428>
- [36] J. Rosenberg, D. Willis, R. Sparks. Session Initiation Protocol (SIP) Extensions for Presence, Internet Draft. November 2002.
<http://www.ietf.org/proceedings/02mar/I-D/draft-ietf-simple-presence-05.txt>

- [37] Ubiquity Web site
<http://www.ubiquity.net>
- [38] MessageMachines Web site
<http://www.messagemachines.com/>
- [39] SUN's Java product home page
<http://java.sun.com>
- [40] Monson-Haefel, R. & Chappell, D., 2000. Java Message Service.
O'Reilly, Beijing. 2000 (ISBN: 0596000685)
- [41] C. Perkins, E. Gutman, J. Veizades, M. Day.
Service Location Protocol, RFC 2608. June 1999.
<ftp://ftp.isi.edu/in-notes/rfc2608.txt>
- [42] M. Barbeau. Mobile, Distributed, and Pervasive Computing. Chapter 27 -
Handbook of Wireless Networks and Mobile Computing.
John Wiley and Sons, Inc., February 2002 (ISBN: 0-471-41902-4)
- [43] C. Perkins, E. Gutman, J. Kempf.
Service Templates and service: Schemes, RFC 2609. June 1999.
<ftp://ftp.isi.edu/in-notes/rfc2609.txt>
- [44] Discovery and Join, Jini Technology Core Platform Specification.
<http://www.sun.com/software/jini/specs/jini1.1html/discovery-spec.html>
- [45] UDDI Technical White Paper. September 2000.
http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf
- [46] About UDDIe.
<http://www.wesc.ac.uk/projects/uddie/uddie/about/>

- [47] DAML Web site
<http://www.daml.org/>
- [48] Bechhofer, S., Broekstra, J., et al., 2000.
An informal description of Standard OIL and Instance OIL.
<http://www.ontoknowledge.org/oil/downl/oil-whitepaper.pdf>
- [49] DAML-S: Semantic Markup for Web Services.
<http://www.daml.org/services/daml-s/0.7/daml-s.html>
- [50] Christensen, E., Curbera, F., Meredith, G. & Weerawarana, S., 2001.
Web Services Description Language (WSDL) 1.1.
<http://www.w3.org/TR/2001/NOTE-wsdl-20010315>
- [51] UpnP Web site
<http://www.upnp.org>
- [52] Salutation Web site
<http://www.salutation.org/>
- [53] C. Perkins, E. Gutman.
DHCP Options for Service Location Protocol, RFC 2610, June 1999.
<ftp://ftp.isi.edu/in-notes/rfc2610.txt>
- [54] Helal, S., 2002. Standards for Service Discovery and Delivery.
http://www.ee.oulu.fi/~skidi/teaching/mobile_and_ubiquitous_multimedia_2002/standards_for_service_discovery_and_delivery.pdf
- [55] XML Schema.
<http://www.w3.org/XML/Schema>

- [56] Wireless Java.
http://www.onjava.com/topics/java/Wireless_Java
- [57] SUN's J2ME Web site
<http://java.sun.com/j2me/>
- [58] CLDC Specification V.1.0, JSR 30, 2000.
<http://jcp.org/en/jsr/detail?id=30>
- [59] White Paper: CLDC Hotspot Implementation White Paper.
http://java.sun.com/products/cldc/wp/CLDC_HI_WhitePaper.pdf
- [60] Mobile Information Device Profile 2.0 (Final Release), JSR 118, November 2002.
<http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html>
- [61] White Paper: MIDP APIs for Wireless Applications, Sun Micro System.
<http://java.sun.com/products/midp/midp-wirelessapps-wp.pdf>
- [62] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and T. Berners-Lee, Hypertext Transfer Protocol - HTTP/1.1, RFC 2616, June 1999.
<ftp://ftp.isi.edu/in-notes/rfc2616.txt>
- [63] Java Community Process Web site
<http://www.jcp.org>
- [64] JAIN Presence and Availability Management (PAM) API, JSR 123, April 2003.
<http://jcp.org/aboutJava/communityprocess/review/jsr123/index.html>
- [65] JAIN SIMPLE Presence, JSR 164, August, 2003.
<http://jcp.org/aboutJava/communityprocess/review/jsr164/index.html>

- [66] JAIN SIMPLE Instant Messaging, JSR 165, August, 2003.
<http://jcp.org/aboutJava/communityprocess/review/jsr165/index.html>
- [67] JAIN SIP API Specification, JSR 32, August, 2003.
<http://jcp.org/aboutJava/communityprocess/review/jsr164/index.html>
- [68] Indigo SIP Server and SDK release note.
<http://www.indigosw.com/html/prodsipsdk.html>
- [69] Atrium SIP/SIMPLE Server User's Guide.
<http://www.nloci.com>
- [70] JBoss Server Overview.
<http://www.jboss.org/overview>
- [71] MySql Reference Manual.
<http://www.mysql.com/doc/en/index.html>
- [72] Brandt, S.R., May 2002. Create a quick-and-dirty XML parser, *Java World*.
<http://www.javaworld.com/javaworld/javatips/jw-javatip128.html>
- [73] Web Method GLUE UDDI Server User's Guide.
http://www.webmethods.com/solutions/wM_Glue/
- [74] Introduction to Google Web APIs.
http://www.google.com/apis/api_faq.html