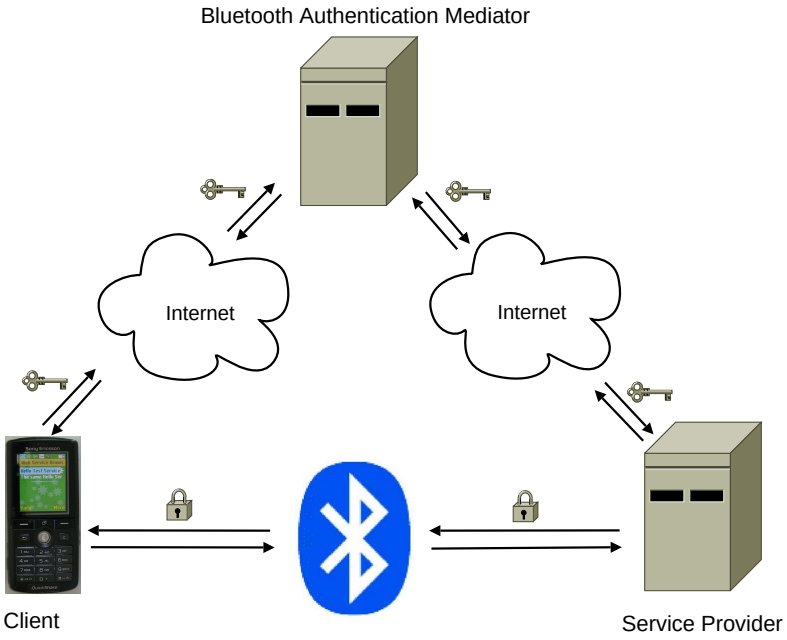
DynaBlu - A Framework for developing Location-Aware Web Service Applications using Bluetooth communication



Group d619a, Room E4-113	Supervisor
Nikolaj Andersen	Lone Leth Thomsen
Morten Vejen Nielsen	
Jørn Rasmussen	

Department of Computer Science
AALBORG UNIVERSITY

TITLE:

DynaBlu - A Framework for developing Location-Aware Web Service Applications using Bluetooth communication

SEMESTER PERIOD:

DAT6,
1st of February - 11th of June 2007

PROJECT GROUP:

d619a (room E4-113)

GROUP MEMBERS:

Nikolaj Andersen, nikko@cs.aau.dk
Morten Vejen Nielsen, mvejen@cs.aau.dk
Jørn Martin Rasmussen, joern25@cs.aau.dk

SUPERVISOR:

Lone Leth Thomsen,
lone@cs.aau.dk

NUMBER OF COPIES: 7

PAGES IN REPORT: 1 - 118

PAGES IN APPENDIX: 127 - 139

TOTAL NUMBER OF PAGES: 139

ABSTRACT:

High speed data communication and increasing processing power of mobile devices makes them well suited for a range of new applications. Combined with the potential location-awareness of mobile devices it is possible to bring context sensitive information to users.
This project deals with the development of DynaBlu, an application framework for creating location-aware web service applications. The framework uses Bluetooth technology for data communication and providing location-awareness. Web services are used to interact with the information services. The communication is authenticated and encrypted to ensure certain security requirements.
The goal of this project is to make the creation of interactive, secure and location-aware applications for mobile devices possible with DynaBlu, and to gain essential insight into the technical foundation of such a framework.

Preface

This report is written at the Department of Computer Science, Aalborg University within the Database and Programming Technology research unit, by project group d619a/E4-113 to document the second of the two masters thesis semesters. The project was written in the spring semester of 2007 to be evaluated on the 25^{th} of June, 2007.
The project was supervised by Lone Leth Thomsen, whom we would like to thank for her participation in supervising the development of the project.
We assume the reader to have a prior knowledge of XML, programming, and distributed systems. When references are written after a paragraph, the reference accounts for the entire paragraph. When references are written after a word the reference accounts for the single word or term preceding it. We will sometimes be using Wikipedia, blogs, etc. as sources to explain concepts. Because of the subjective nature of these sources, we will not be using these as argumentative sources.

Nikolaj Andersen

Morten Vejen Nielsen

Jørn Martin Rasmussen

Contents

Part I

Introduction

Chapter 1

Introduction

Since the invention of "The Internet" by Tim Berners-Lee in 1992, a growing number of web pages serve as an international digital platform for any kind of information. Initially thought as a static information resource, web pages have since begun moving towards being information-providers that can change dynamically. Either by user-interaction like in web-logs or automatically depending on its context. Googles advertising system for instance displays commercials on a page dynamically dependent of the content of the current page. Such context-awareness is often referred to as the new way of the Internet with the expression *Web 2.0*.

WAP and the *Wireless Markup Language* (WML), introduced in the late 1990s, brought the static part of the Internet to mobile phones. And with every new generation of mobile phones, the Internet as we know it from desktop computers gets more and more pervasive in this direction. The Opera Mobile(TM) browser [43] is an example of such a bridge between these two worlds.

Interestingly, the word context-sensitivity attains a whole new dimension in its meaning on mobile devices. The context for a mobile phone can be extended to the physical location of the mobile device. What if Googles advertising system were not only related to the content of a web page, but also related to the actual physical location of the user requesting that web page?

In our belief, bringing dynamicity and location-awareness together opens the mobile device to a whole new realm of possible applications.

Our goal in this project is therefore to bring the need for location-aware services and the possibilities of modern mobile devices together.

A framework that could provide the technical background for such applications would ease the development process. To motivate the need for such a framework, we list a number of possible use cases.

Example 1: Walking by a movie theater, posters at the wall show upcoming and currently shown films at the theater. By using a mobile device, services provided by and within range of the theater could be discovered. One example of such a service could be the possibility to directly download a trailer presenting the movies shown on the posters. As the trailer ends, the service provides possibilities to directly make a reservation or directly pay for a ticket to see the film. Waiting in the ticket-line thus becomes obsolete.

Example 2: A city has put up a number of location-based services in the cultural centers of the city. There are a lot of sights in this district, such as museums and sculptures. Services located at these sights function as a sort of a tour guide. As a tourist approach the sights, she is able to discover the services related to the sights nearby with her mobile device. These services could then provide background information or inform her of alternate sights worth visiting. Background information could for instance be voice messages, text messages, pictures or maps. Tourists thus becomes able to explore a city in a whole new way.

Example 3: At the airport, similar services as in Example 2 could provide the traveler with information depending on the terminal she is currently at. Maps related to the current terminal or flight schedules could be provided.

Example 4: Many buildings today have facilities for physically disabled persons. But ramps for wheelchairs and handicap toilets are expensive to build and the need for these are statistically lower than for a regular toilet. Therefore typically such facilities are limited to only a few locations in a building. Finding these facilities can be difficult without knowing where they are. A location-aware service accessible with a mobile device could ease finding handicap-friendly facilities.

The examples described above show that there are indeed applications for such "mobile systems" that provides information dependent on the physical location of a user.

Initial Problem

Our goal in this project is to provide users of mobile devices the possibility of getting services, that are aware of the users physical location. Therefore, we develop an application framework, that eases the development and deployment of such services.

For such a project, it is crucial to investigate current products on the market.

Both current mobile systems, and what communicative capabilities modern mobile devices have. The following chapter presents the results of our investigation.

Chapter 2

Preliminary Analysis

In this chapter we present an analysis of research relevant to our initial problem, see section 1. We investigate customer demands and real-life experiences from previous and existing mobile systems. This will provide us with insight into what criteria a mobile system should adhere to. We investigate the different technology platforms available to us when focusing on dynamic location-aware service discovery on mobile devices. The purpose of this examination is to identify the network technology we intend to exploit to realize communication in our system. Having identified a network carrier technology we then proceed to discuss the consequences of using this on a mobile device. This chapter will provide us with valuable insight into how our actual system should be designed to satisfy both the user demands and the demands implied from using a mobile platform.

2.1 Existing Systems

2.1.1 SMS Systems

SMS (Short Message Service) has previously been used for making mobile systems. In this section we will be describing some of the experiences and customer attitudes towards mobile SMS systems. This section is based on a mobile marketing whitepaper released by Mobilereact, a marketing firm specializing in mobile solutions, in 2005. [38]

SMS or *Texting* has gained wide popularity amongst mobile phone owners, almost 100% [38] of all teenagers having a mobile phone uses texting facilities on their phones. This popularity is by a large part due to the non-intrusive nature of a text message where the recipient has the possibility of reading

a text message and replying to it when she finds the time to do it instead of having to respond immediately as is the case with an ordinary telephone call.

Experimentation has been done with SMS systems with the purpose of extending the traditional texting functionality into different application areas, the listing, based on [38], below illustrates some of the SMS systems that have previously been implemented.

- SMS-couponing via interactive street displays - JC Decaux (Ireland).
- TV voting, game participation, chat, radio song dedications.
- Mall-based permission marketing - Jurong Point (Singapore).
- Mobile micro-payment systems (Paypal Mobile).
- Instant win contests - Coca Cola (Australia).

However some considerations should be taken when deploying SMS systems into the public. One important obstacle is the public's lack of tolerance to unsolicited information, similar to the general attitude towards SPAM e-mails. Another limitation to consider is the limitations following from using the SMS medium.

A relevant question to state is whether or not the users are interested in mobile SMS systems? An answer to this question is indicated in a study conducted in [1] for Nokia in 2001. This study showed that 88% of mobile users (aged 16-45) would be receptive to receiving mobile SMS advertising messages if the three following conditions are adhered to [38]:

1. **Choice** - Being able to decide whether or not to receive messages.
2. **Control** - The ability to easily opt-out if the user is not interested in the marketing scheme anymore.
3. **Mutual benefit** - Getting something back in return. A reduction in the cost of services, for example, would qualify.

Several case studies have been conducted generally showing increases in sales after the introduction of an SMS system. A rather positive example is the *Coca Cola Summerdays* campaign launched in 2003 lasting for 3 months in Australia. The objective of this campaign was to increase attention based on a coupon-based prize competition. Customers would get a coupon number with their Coca Cola bottle and this number could release a prize. The customers could submit their numbers either by SMS or by going online.

The results from this campaign showed that out of 2.5 million valid entries 97% of customers chose to use the SMS application form, out of these 2.5 million 120000 customers actively chose to receive future announcements from Coca Cola. Case studies such as this shows the feasibility of creating mobile SMS systems that are put to actual use.

When using SMS systems a number of limitations are implied. For one SMS systems typically requires the user to actively send an SMS request to the system in order for it to be activated. Sending this message can be cumbersome for the user as messages for SMS systems typically requires a specific formatting, for instance a message like "COCA-COLA Lottery #", that has to be sent to a specific number to enter a contest. We back this claim by referring to the *Coca Cola Summerdays* campaign [38]. Where 2.5 million entries were valid out of 4 million entries in total.

Besides message formatting issues we also consider the cost of using SMS systems as a restriction. While using simple single-message advertising systems might impose an insignificant cost on the user, consider using SMS systems requiring more advanced user-interaction and thus requiring the user to send multiple messages. This restriction could potentially be a hindrance to the future success of SMS systems.

Another restriction is the implications of using the SMS medium for advertisement messages. Since SMS is a text-based medium sending images, videos, etc. is not possible. One solution to this problem might be to use MMS (Multimedia Messaging Service) to send multimedia content. However using MMS only increases the cost problem as sending an MMS message is typically more expensive than sending a text message.

2.1.2 Bluetooth Systems

This section presents a number of Bluetooth systems and projects related to the area that this report address.

BlipNet

BLIP systems (Bluetooth Local Infotainment Point) is a company located in Vester Hassing near Aalborg, Denmark, that specializes in Bluetooth systems. The Company has built a Bluetooth system called BlipNet. The BlipNet is built by using a combination of special-purpose hardware and a software system. Figure 2.1 illustrates how the BlipNet works. The BlipServer is the main component of the system. This entity controls all other entities and is accessible from the Internet. The BlipServer can be managed from the BlipManager, which is a software program that lets you manage

the BlipServer through a GUI. The BlipServer is connected to a number of BlipNodes through LAN and Internet. A BlipNode is the hardware component of the system. This component allows the server to communicate with Bluetooth devices in range of the BlipNode by providing a bridge between the LAN and the Bluetooth devices. When Bluetooth enabled and discoverable devices come in range of a BlipNode the device is detected and its presence is reported back to the BlipServer. The BlipServer now has the option of pushing objects to the device through the BlipNode. Bluetooth enabled devices can be registered at the BlipServer and granted user accounts, this way it is possible to control which objects gets pushed to which devices. There is an API available to interface with the BlipServer in order to build systems on top of BlipNet.[7]

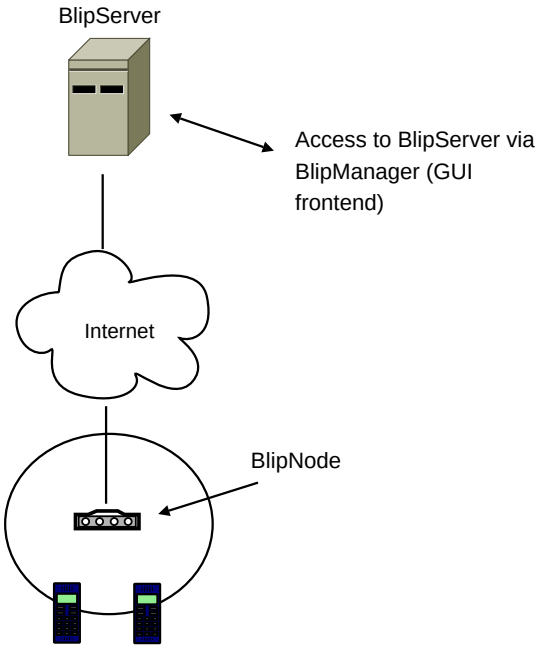


Figure 2.1: Overview of the BlipNet architecture.

The BlipNet can be used as a platform for developing systems that can offer information to users based on their location. It is ideal for creating marketing systems that offer commercial content to users passing by certain areas. It is possible to create a diverse range of Bluetooth systems by using this framework.

From a strict marketing point-of-view the framework has great potential. The BlipNet framework is push-based. It allows you to push information to any device in the range of a BlipNode that has Bluetooth turned on, not requiring any software to be installed in the target mobile device. But from a users point of view this system also has the potential of creating a very

aggressive marketing system. Imagine walking into a mall where BlipNodes have been placed in a number of shops, and the BlipNodes have all been configured to send commercial content to anyone passing by. Now imagine that you turn on Bluetooth on e.g. a mobile phone, this would result in you receiving a number of messages. This scenario resembles SPAM e-mails in an inbox. What makes this possible is the fact that it is the BlipServer that discovers the Bluetooth enabled devices, and it is the BlipServer that pushes objects to the devices. Also a provider might attempt to push malicious content to a Bluetooth device which could potentially expose sensitive information to a service provider.

From a users perspective it would be desirable to be able to discover services, and then choose what information you want from which services.

The BlipNet system has been deployed in a number of cities, including Copenhagen, Denmark, where over 400 BlipNodes have been deployed that are operated and managed by a single BlipServer. [12]

CWhere

In [31] a location system called CWhere is proposed. CWhere allows people at Aalborg University to find each other based on their current location and their interests. The system relies on Bluetooth technology to discover users and it is built on top of BlipNet 2.1.2. The system has a web service as a front end that can be used to manage user accounts and profiles. The front end controls the BlipServer and a database that stores user information.

A number of BlipNodes are positioned around campus, and when a user with a Bluetooth device in discoverable mode enters an area that is in range of one of the BlipNodes, the user will receive information about other users that have matching interests.

BlueBlitz

A German company called BlueBlitz[21] offers a framework for Bluetooth communication that is very similar to BlipNet. The hardware components provided by BlueBlitz solve the same problem as BlipNodes does in BlipNet, however they have a number of different components to choose from. The size of these components vary from small home office to enterprise components. BlueBlitz also provides an Internet solution that can be used to create a mobile communication platform onto an existing web site. For the mobile devices they offer a mobile gateway that implements security features. The architecture of the BlueBlitz systems is very similar to the architecture of BlipNet, and it is also a push-based system.

B-MAD

B-MAD is a location-aware mobile advertising system introduced in [16], that is based on Bluetooth and WAP. The system consists of a Bluetooth Sensor, an Ad Server and a Push Sender. Since the system is permission-based, the Ad Server must maintain a database of users. The database stores information about which users it is okay to send ads to, and which ads have already been sent to which users. When a mobile device comes in range of the Bluetooth Sensor a message is sent from the Bluetooth Sensor to the Ad Server. The message is sent over a WAP connection and contains the MAC address and MSISDN, a unique identification number, of the mobile device and a location identifier. The Ad Server then checks whether there are any undelivered ads associated with the given location waiting to be delivered to the mobile device. If this is the case the ads are delivered to the mobile device through the Push Sender. The Push Sender uses WAP Push SI (Service Indication) to deliver the ad, which means that ads are sent through a Push Proxy Gateway and encoded into a simple SMS message.

The B-MAD system is similar to BlipNet and BlueBlitz, but B-MAD doesn't rely on commercial hardware products to bridge between the mobile devices and the server. Furthermore the authors of [16] highlight that the system is permission-based, which will outrule the possibility of sending unsolicited content.

2.2 Data Connections on Mobile Phones

Modern mobile phones support a wide variety of different communication channels. Some of them are suited for ad-hoc connectivity and some for infrastructure-mode. Infrastructure-mode is characterized by the fact, that a network provider is responsible for routing data from a point A to a point B. The network provider also charges the user for the service. Payment can occur on a per-data-amount or per-connection-time basis.

In this section we present the possibilities a user has to transfer data with her mobile phone. We will not discuss obsolete communication channels for mobile phones, nor go into great detail of each communication protocol. Fourth generation mobile technologies are not part of this discussion either since they are still in preliminary state.

The communication channels can be grouped by several characteristics as figure 2.2 depicts.

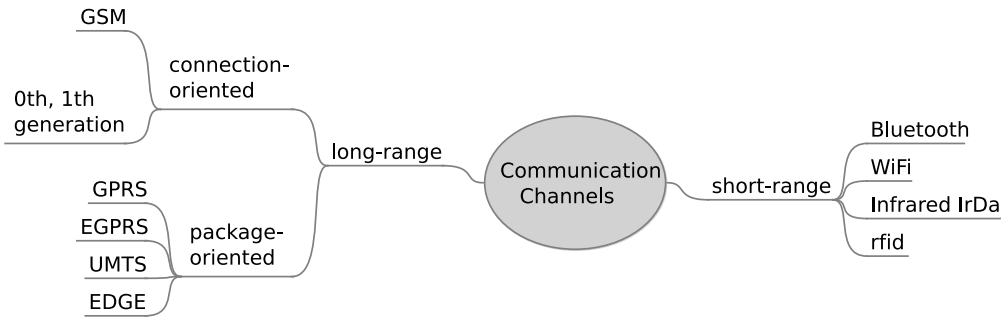


Figure 2.2: Map of the different communication channels available on mobile phones and their characteristics

Long-Range Communication Channels

As the name *mobile phone* suggests, the main purpose of a mobile phone is providing telephony. This service depends greatly on an infrastructure namely the telephone network. Connection to that service is provided by the telephone company which acts as a service provider. Since availability is a main goal for providing telephony, also in the countryside, the communication channel used for this service is a long-range communication channel.

Usually, a mobile phone supports only a subset of those standards. Older phones support the GSM (Global System for Mobile Communications) standard and newer, third and fourth generation mobile phones support UMTS (Universal Mobile Telecommunications System) and UMTS R8.

Connection-Oriented Channels

Mobile telephony emerged from wired telephony. Therefore, making payment dependent the per-connection-time basis was obvious. One of the remaining connection-oriented communication channels is the widely used GSM standard. Working on different frequency bands, GSM is a second generation communication standard that supports both speech and data communication.

The major drawback, that makes data-communication rather unattractive, is that payment is independent of the amount of data that is sent through the connection. Hence an established but idle connection still costs money. Another mayor drawback is the slow data transmission that lies between 6 Kb/s (half-rate) and 13 Kb/s (full-rate).[51]

Package-Oriented Channels

Because of the drawbacks of connection-oriented channels, the GSM standard was extended by several package-oriented protocols. The two most popular ones are GPRS (General Packet Radio Service) and EDGE (Enhanced Data rates for GSM Evolution) also called EGPRS (Enhanced GPRS). The main advantages of those standards are package-orientation and much higher datarates than GSM only. The package-orientation means, that payment depends on the amount of data that has been sent, and not on the duration of the data-connection. The datarates are between 9.6 Kb/s and 80 Kb/s for GPRS and 236.8 Kb/s for EDGE.

As third generation mobile phone technologies no longer build upon GSM technologies, new communication channels such as UMTS and HSDPA (High-Speed Downlink Packet Access) could be designed from scratch. They are much faster than GSM based technologies and allow theoretical datarates of up to 100 Mb/s download and 50 Mb/s upload. Of course, those theoretical maxima are only achieved at locations where superior signal quality can be guaranteed. One major drawback of third generation networks is their signal range which is lower than GSM because of the higher frequency band.[51]

Short-range Communication Channels

Another characteristic of a mobile phone is the fact that people carry their phone with them where ever they go. This opens the possibility of exploiting short-range communication channels to provide location-awareness for mobile phone users. In this scheme the host having the short-range communication device becomes the service provider, instead of the telephone company as is the case with long-range communication channels for mobile phones.

A wide variety of short-range communication channels exists each having their strengths and drawbacks.

Infrared (IrDA)

Formed in 1993, The communication standard defined by the *Infrared Data Association* (IrDA) is the oldest of the mentioned short-range communication technologies. The goals of IrDA was to develop an inexpensive (less than 5$ per device), fast (115 Kb/s) cable-replacement based on infrared signal transmission. Targeted at printer-interfaces and serial communication replacement, IrDA was first introduced to mobile phones in late 1997. By then, the original data-transmission-rate was raised to support 4 Mb/s.

As interoperability was one of the main goals of the IrDA, they also defined application protocols to ensure interoperability. Two of the more noteworthy protocols include *IrCOMM* and *IrOBEX*. The first protocol, *IrCOMM*, provides serial and parallel port emulation over the infrared link. The second protocol, *IrOBEX*, provides exchange of simple data objects, hence the name IrOBEX which stands for Infrared Object Exchange. This protocol can be considered as the IrDA analog of the HTTP protocol since it provides authenticity, reliability and other basic services as HTTP does.

Based on infrared light, the main disadvantage of IrDA is its short range. A clear line-of-sight is necessary, and according to the specification in the range of 0-1m at an angle diverging no more than 15-30 degrees.[64]

Bluetooth

In 1994, the mobile phone company Ericsson invented Bluetooth. Like IrDA, its primary goal was to replace data cables with a wireless connection. In contrast to IrDA, Bluetooth transmits its signals by radio frequency using the unlicensed 2.4 GHz band. Its transmission range is 10m and can be extended to 100m by use of amplifiers. Like IrDA, Bluetooth is designed as a low cost device and the goal was less than 10$ per device. Data transfer rate is at max 780 Kb/s which makes it much slower than IrDA. Bluetooth 2.0, specified in 2004 introduced EDR (Enhanced Data Rate) which makes datarates of up to a theoretical limit of 4 Mb/s possible.

As with the IrDA standard, Bluetooth also supports standard transport protocols. In particular and analog to IrDA, Bluetooth supports RFCOMM and OBEX which serve the same purposes as IrCOMM and IrOBEX. It is therefore clear, that Bluetooth is a direct competitor to IrDA. Since a line-of-sight between communicating parties is unnecessary and the possibility of point-to-multipoint communication makes Bluetooth a strong competitor. [48]

Bluetooth devices in range can be discovered by use of device discovery. Bluetooth discovery takes between 18-25 seconds on average. [58]

WiFi

Wireless Lan, WiFi and IEEE 802.11 all correspond to the same. It is the IEEE standard 802.11 a/b/g which is referred to, that defines a set of wireless LAN/WAN standards. The intentions with WiFi was to create a wireless alternative to Ethernet. WiFi has a range of 100m which is ten times the range of a standard Bluetooth connection. Power consumption is therefore approximately 10 times higher than Bluetooth power consumption. [37]

Using the same baseband as Bluetooth, 2.4GHz, WiFi faces the same interference potentials as Bluetooth does. Microwave stoves and other devices using the unlicensed radio frequency band are known to disturb communication.

Since WiFi is designed for wireless LAN applications, it is well-suited for any kind of Internet-protocol based application such as email, browsing and of course web service invocation. Also the high data-rates at 11 Mb/s and 54 Mb/s respectively for the standards 802.11b and 802.11g allows a high degree of interaction due to short round-trip-time compared to Bluetooth and other short-range communication technologies.[37]

Other Short-Range Technologies

There are a couple of other wireless short-range technologies which could be targeted at mobile devices. Such technologies include RFID, ZigBee, Wibree and WUSB (Wireless USB). These technologies are partially still under development and not widely applied to mobile phones. Furthermore technologies like RFID are not well suited for our application since they are not aimed at supporting transmission of larger data sets.

Bluetooth in Mobile Phones

Since Bluetooth was aimed to replace cables, the primary use-case for Bluetooth on mobile phones are headsets. The ability to ensure the bandwidth required for realtime audio transfer natively by the Bluetooth specification makes headsets practically a killer-application for Bluetooth.

With the implementation of standard transport protocols, especially the implementation of IrDA protocols like IrCOMM (called RFCOMM with Bluetooth) and IrOBEX (OBEX), synchronizing phonebooks and calendars with the mobile phone and a computer is meant to be as simple as plugging in a cable. It gives the user the ability to backup the data stored on the mobile phone.

Using Bluetooth has also some notable basic advantages. First of all Bluetooth is cheap to use. In order to access the Internet from a mobile phone a user needs to have an account with a phone company, and the user will be charged for the Internet traffic. Bluetooth radio communication is totally free of charge for the user. Although connectivity to the Internet and its services using Bluetooth requires a bridging application to be installed, which the phone company otherwise would provide.

Power consumption is also a quality factor for the user of a mobile phone.

Since Bluetooth is a short-range communication channel, its power consumption is much lower than for GPRS or UMTS which are long-range channels.

The fact that Bluetooth is a short-range channel, can be regarded as a feature. Since the typical transmission range of a Bluetooth signal is about 10m, a user can only establish a connection to devices in its neighborhood, hence providing a limited form of location-awareness in the device.

2.3 Summary

The SMS systems section of this chapter described the results of a marketing analysis conducted in 2005. One of the key results from this analysis was the fact that 88% of the questioned mobile users would be interested in using these sorts of systems, if they adhere to the three conditions, *Choice*, *Control*, and *Mutual benefit*. It is our belief that adhering to these principles can help ensure the success of future mobile systems exhibiting a similar behavior as an SMS system.

Also using SMS systems for more advanced user interaction can be cumbersome and costly since the user will have to send a number of specifically formatted SMS messages to the SMS system to achieve interaction. Specifically a case study of the *Coca Cola Summerdays* marketing campaign launched in Australia in 2003 showed a large number of invalid SMS messages sent due to ill-typed and ill-formatted SMS messages. From this we conclude that using mobile systems should be free of charge, to open the possibilities of creating more advanced applications requiring direct user interaction. Also using a mobile system should rely on simple and easy-to-use interfaces with the goal of eliminating or reducing the number of invalid requests sent to a system.

We believe that push-based services is a bad approach for building location-aware mobile systems. When push-based services are used the users have no choice at all whether or not they want to receive a given message, and this is not in accordance with the *Choice*-condition listed in section 2.1.1 that should be met in order for users to be receptive to receive content on their mobile phone. Push-based services take away the initiative for users to engage in interactions. This is a problem because users may be inconvenienced or disturbed by an interaction that has been pushed upon them. For instance, in a push-based system a user could be writing a message on her phone and suddenly be interrupted in the process by a message from a service provider.

Furthermore push-based services empowers the service providers to send anything they want to their users. Even though content may be sent to users

in a permission-based manner, as is the case with the B-MAD system, it is still up to the provider to choose what content to send to the user. This can potentially be dangerous as a malicious provider might attempt to send corrupting data to a client.

The BlipNet and BlueBlitz solutions require that you purchase their hardware together with their software in order to set up Bluetooth systems. What these hardware components do can also be accomplished in software. There are potentially two benefits of going with the software approach. Firstly the providers will not need to invest funds in the hardware components. Secondly the providers will not be constrained in their interaction by the particular firmware that has been installed on the hardware component. The software approach of doing the interaction is more flexible, but the software solution will most likely execute slower than the hardware solution.

Despite of the Instant win contest discussed in section 2.1.1, most of the application areas for mobile systems are dependent on the current location of the customer or user. Therefore location-awareness is an important feature for mobile systems. But location-awareness cannot be provided by long-range communication used in SMS based mobile, since there is no possibility to locate a user solely by long-range signals due do their imprecise nature. GPS (Global Positioning System) is neither feasible, since it is not very common in mobile phones today. It is also not possible to use GPS signals inside buildings.

We suggest a Bluetooth-based mobile system because of their limited signal-range which offers the possibility to provide limited location-awareness. It is limited, because the signal-range depends on the hardware in the sender and receiver and the vicinity both are in. But we are confident, that this limitation only restricts few applications that require location-awareness.

Bluetooth is also favorable as communication channel for mobile systems. While data transferred by SMS or any other channel that requires a carrier (the phone company) cost money, Bluetooth is completely free. Otherwise, customers might pay for receiving commercials. In this case, acceptance would quickly decrease because of the lack of mutual benefit as described in [38]. Of course WiFi is also free to use like Bluetooth. But WiFi has a power consumption that is approximately ten times the power consumption of Bluetooth.

The goal is to be able to build systems that allow users to discover Bluetooth services on their own in a safe way, and let the users choose for themselves which services they want to use. In order to build these systems securely we need an authentication framework that establishes safe communication between mobile devices and Bluetooth services. Establishing safe communication requires the use of a registry to store data about authenticated

services, and a server with access to this registry that can act as a mediator and negotiate between the mobile devices and Bluetooth services.

This summary has provided an accumulation of important observations and arguments, which serve as the basis for a detailed problem statement in the following chapter.

Chapter 3

Problem Statement

This chapter provides a conceptual description of the DynaBlu framework. We introduce the term *authenticated web service invocation over Bluetooth* and explain the concept of the DynaBlu framework. The conceptual overview leads to a set of system requirements and system philosophy. Finally the goals for this project are defined.

3.1 Authenticated Web Service Invocation over Bluetooth

The main task of the DynaBlu framework is to provide the user of a mobile device with a communication gateway to location-aware information services.

To avoid potential malicious service providers we have a built-in security mechanism providing authentication of service providers. This security mechanism requires the use of a third party, which we have called the *Bluetooth Authentication Mediator*. This mediator is responsible for confirming the identity of the communicating parties by exchanging their identifying data.

3.2 System Description of the DynaBlu Framework

Figure 3.1 gives an conceptual overview of the proposed framework.

Our framework consists of three collaborating entities.

- A client. In DynaBlu a service requester is a mobile client with Bluetooth capabilities. This client will detect a signal from a service provider

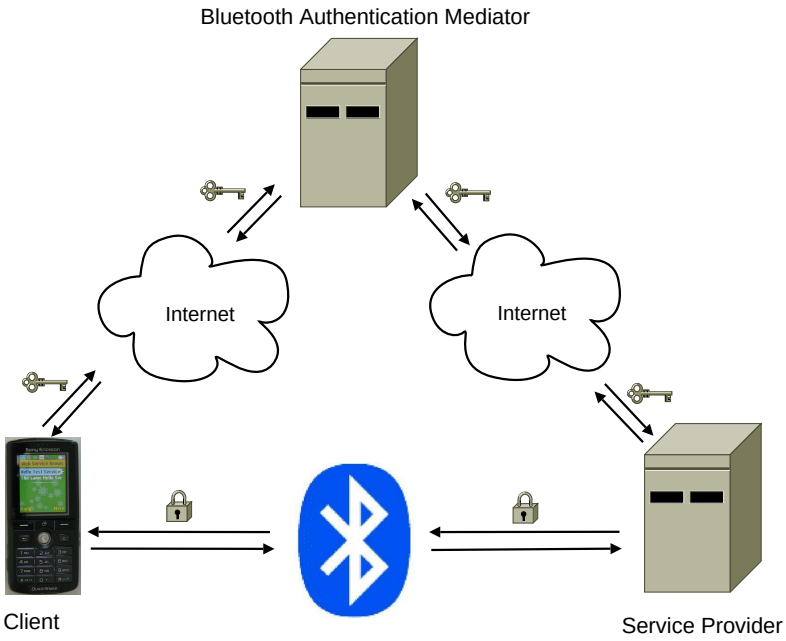


Figure 3.1: Conceptual overview of the framework, allowing authenticated web service invocation from a mobile phone using Bluetooth communication

within its range. Depending on the service being provided, the client can choose to initiate a Bluetooth connection with the provider and interact with its service. We will need to design and implement client software for the mobile client in order for it to be able to interact safely with a service provider. We have chosen to implement the Bluetooth connectivity in a component we call the Bluetooth Communication Bridge, which we describe in detail in chapter 8

- The service provider is responsible for publishing its services and advertising the presence of these using Bluetooth. The service provider should provide capabilities allowing its identity to be confirmed, moreover the service provider should be able to confirm the identity of the client. The Bluetooth connectivity in the provider is also implemented in the bridge component.

- The Bluetooth Authentication Mediator. This entity provides the means for authentication between the client and the provider. It should provide facilities allowing service providers to register their services at this entity. This registration will make it possible for a client and provider to securely establish the identities of each other.

The interaction between the three entities is divided into three phases.

Publication phase The service provider publishes/registers information about its identity and provided services in the Bluetooth Authentication Mediator.

Discovery phase The Client discovers a Bluetooth signal from the provider and subsequently fetches the information needed to communicate.

Authentication phase Both client and provider authenticate each other by communicating with the mediator. The mediator provides both parties with an authentication and encryption key.

Communication phase Based on the authentication results received from the mediator, the client sets up communication with the provider and it's web service applications.

The client can now communicate with the web services deployed in the provider by sending/receiving Soap messages over the Bluetooth Communication Bridge.

3.3 System Requirements

Based on the preanalysis, the use cases described in the introduction and our experience from former projects we define requirements that the DynaBlu framework should adhere to. The requirements are split into two groups. The first group contains the design requirements which address aspects regarding the inner workings of our framework. The second group contains requirements that are aimed at both the end users, which use the client application to consume services in a service provider, and the developers using our framework to implement a location-aware system.

Design Requirements

Extensibility

Our framework should be designed to be extensible. This requirement ensures that we will be able to update and change our framework to meet future demands. Designing our framework to be extensible necessitates a modular and flexible design where individual components can be changed or added without affecting the rest of the system.

To make our framework extensible we must also provide documentation that can aid developers in understanding the existing code. Source code comments and API documentation simlifies integration and extension of existing modules.

Reliability
A goal of our framework is that it should be designed to be reliable. By reliable we mean that error-handling facilities should be provided, ensuring that unexpected actions are handled. In the case of an irreparable error the system should always respond with an appropriate error message.

Efficiency
Because the client application will be running on a resource constrained mobile device, efficiency concerns regarding memory usage and heavy calculations must be made. This improves the performance of the application and increases compatibility with respect to memory consumption. Optimization of the data communication between the client application and the provider is also part of the efficiency requirement. Efficient data communication is important to prevent communication bottlenecks which would have direct impact on the usability requirements.

Usability Requirements

Selectability
The results concerning customer demands for mobile systems learned in the preanalysis chapter (see section 2.1.1 for a detailed description) dictate three conditions that mobile systems should adhere to. Those conditions are *Choice*, *Control*, and *Mutual Benefit*. To satisfy the *Choice* condition a DynaBlu system must never send messages to a mobile user who has not opted to receive a certain message. A user must have requested the message by actively selecting a web service for invocation. This means that the client will never receive unsolicited messages from systems based on our framework. A system can only be activated when the client actively invokes a service chosen using our client application. This method of invoking applications also implicitly satisfies the *Control* condition as the client can simply remove the client application from her mobile phone if she no longer wants to use DynaBlu systems. The *Mutual Benefit* condition is left in the hands of the service provider, choosing themselves which marketing strategy they will be adopting.

Credibility
Mobile clients should always be able to trust the service providers they encounter. Clients should not have to worry about service providers sending malicious content. A solution to this problem is to make sure that the client can always confirm the identity and thus good intentions of the service provider. Also the service provider should be able to establish the identity of the client thus making sure that potential sensitive information is delivered to the correct recipient. Clients and service providers need not necessarily have preceding knowledge of each other, which is the main reason for us to

include a mediating entity in our framework.

Since we rely on the fact that mobile clients should see our Bluetooth Authentication Mediator as a trustworthy entity in the system, we will have to check the content of the services registered in our service registry before they become publicly available. By check we mean we will have to manually test the deployed web service by invoking it.

Security
Users of systems based on our framework should never have to worry about security. Facilities to safely establish the identity of service providers should be available. Also for sensitive applications encryption facilities has to be available.

This requirement makes it possible for us to support a larger number of applications like for instance the movie theatre example from chapter 1 where we discussed the possibility for a user to make ticket payments from her mobile device.

User Experience Our framework has two groups of users, namely client users consuming applications in the service provider and developers making applications available in the service provider. This requirement is aimed at both types of users.

The client should not have to wait for unreasonable amounts of time when using our framework. When for instance a Bluetooth device discovery is made a user will have to wait for 18-25 seconds, discussed in section 2.2, which can be a nuisance. Thus we will be focusing on minimizing the idle-time that a user has to spend when using our client application to interface with the service provider. To aid in this perspective our client application should also be designed to minimize the manual user interaction required by a client to use an application from a service provider. This aspect is backed by the tendencies we found in the preanalysis chapter showing that users often make mistakes when having to manually supply a system with interaction data.

From a developers point-of-view it should be simple to develop applications based on the DynaBlu framework. A developer should not have to spend a lot of time reading documentation to be able to develop applications using our framework. Also setting up a provider should not be costly and require the use of special-purpose hardware, as is the case with the previously mentioned similar systems like Blip Systems and BlueBlitz.

3.4 System Philosophy

We have stated the following system philosophies that our system should be designed towards.

Openness Our framework for creating location-aware Bluetooth applications should be based on the use of open and vendor-neutral technologies. This makes it possible for developers to use our framework regardless of the development platform they are using as long as it supports these open technologies.

Compatibility It was reported that in 2005 708 million Java-enabled mobile devices had been shipped, in other words 7 out of 10 devices being shipped today is Java-enabled [8]. Based on this fact and that this number is expected to grow in the future we will be basing our system on Java technology. By doing this we gain wide support on most modern phones.

3.5 Project Goals

In [18] a proof of concept is proposed for web service invocation over Bluetooth (the details of this proposal are discussed in section 4.3). In this project we use that proof of concept as an inspiration for our own implementation. The main goal is to develop a functional application framework that makes it possible to develop mobile information applications using authenticated service invocation over Bluetooth. The resulting framework is not to be regarded as a finished product, but as a first step towards a production stable product.

Based on this problem statement, the preanalysis and our knowledge and experience we identify the following areas to be investigated prior to an implementation of the proposed framework.

- Web service invocation from a mobile phone using GPRS.
- Web service invocation from a mobile phone using Bluetooth communication.
- Security and authentication mechanisms for use on a lightweight platform as found on mobile phones.
- Dynamic stub generation used for web service invocation.

The overall focus of this project is to gain knowledge, experience and technical insight into the difficulties of developing such a framework. The implementation should therefore be considered as an extended proof of concept,

in which the technical hurdles such a framework comprises are solved or at least discussed thorougly. The understanding and knowledge about central aspects of this framework should yield an expertise that is fundamental for a future successful commercial implementation of this framework.

The proposed system requirements are therefore not only valid for the current project but also appliable to future work. The resulting implementation of this framework is therefore not required to fulfill all of the proposed requirements. But along with the mentioned project philosophies we aim to create a solid foundation for future projects.

Part II

Analysis

Chapter 4

Bluetooth

This chapter presents an analysis of Bluetooth technology. This is necessary because an implementation of the proposed system require an implementation of software that utilizes Bluetooth communication. The analysis will provide us with valuable knowledge of Bluetooth protocols and Bluetooth security technology.

4.1 Bluetooth

As mentioned in section 2.2 Bluetooth was designed by Ericsson in an effort to create short-range wireless connectivity for ad-hoc networking. The project was named after 10th century Danish viking king, Harald Bluetooth. In 1998 four other companies (IBM, Intel, Nokia and Toshiba) joined in on the work on Bluetooth and formed a Special Interest Group (SIG) together with Ericsson. This SIG is responsible for developing and maintaining the Bluetooth specifications. The SIG grew very fast. In 2000 there were about 1500 member companies, and today (2007) there are over 7000, showing that there is still interest in further developing this technology. [61]

It is not surprising that Bluetooth technology has attracted a lot of attention. There are virtually an endless array of applications for Bluetooth technology. For instance as a cable replacement for head-sets and MP3-players or as cordless data transfer between phone and computer, computer and printer or between computer and digital camera.

Bluetooth connectivity is available on a single tiny, inexpensive computer module. For example one module in figure 4.1 is 33*15*1.2 mm and it is equipped with short range transceivers. When these modules are built into

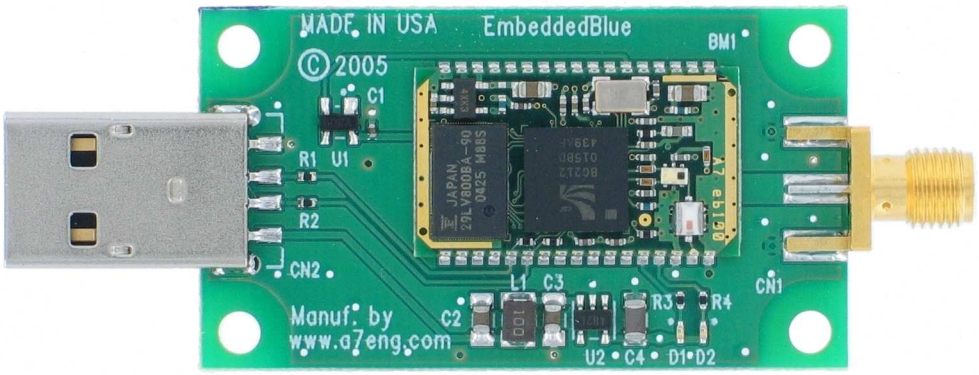


*Figure 4.1: Bluetooth module. The dimensions of the module are roughly 33*15*1.2 mm.*

products such as mobile phones the price of the product is only increased with about 5$, which is only half as much as the design goal mentioned in the introduction, which was 10$. Figure 4.1 shows one of the Bluetooth modules with a possible theoretical data transfer rate of up to 4 Mb/s. [39]

Currently Bluetooth radios are available in three classes dependent on their maximum permitted power usage.

Class 1 100 mW, allowing ranges up to 100 metres.

Class 2 2.5 mW, allowing ranges up to 10 metres.

Class 3 1 mW, allowing ranges up to 1 meter.

Mobile phones are typically class 2 devices allowing communication within a 10m range. Class 2 Bluetooth radios are a well-suited compromise between having a practical application range of the Bluetooth device and a limited power consumption. [60]

Each Bluetooth device is uniquely identified by a 48 bit MAC address, and communication occur over a point-to-point or point-to-multipoint radio link. As mentioned Bluetooth operates on the unlicensed 2.4 GHz part of the radio spectrum, like e.g. microwave ovens, but problems with interference are minimized because the low level Bluetooth communication protocols uses frequency hopping when transmitting data. Bluetooth uses 79 different frequencies around the 2.4 GHz band and 1600 hops are made every second between the 79 frequencies. The sequence of hops that will be made during communication will be given by the master device when a connection is established, which will be explained shortly. Besides coping with interference frequency hopping also allows multiple Bluetooth users in the same room.

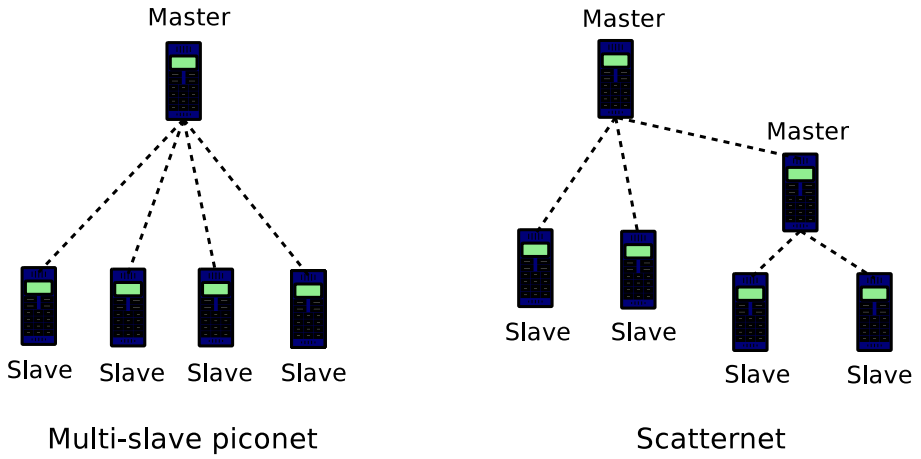


Figure 4.2: Bluetooth networks.

Bluetooth supports both circuit-switched and packet-switched communication. Circuit-switched communication is used in applications where a continuous flow of data is transmitted, and a minimum bandwidth must be reserved. This could for instance be voice communication through headsets. Packet-switched communication is used for data transferal, such as sending a picture from one Bluetooth device to another. Bluetooth supports two different kind of links, ACL and SCO, and both can be supported at the same time. ACL (Asynchronous Connection-Less) links are used for packet-switched data transferal, and SCO (Synchronous Connection-Oriented) links are used for circuit-switched voice communication.

Networks of Bluetooth devices consist of a Master device and up to seven Slave devices. These small networks of Bluetooth devices are called piconets. The device that initially establishes a connection is the Master. This device's clock and frequency hopping sequence is used by all the Slaves in the piconet to synchronize with the Master. Bluetooth piconets can be linked together to form a scatternet. The piconets in a scatternet are not coordinated, which means that they use different frequency hopping sequences. Figure 4.2 shows a multi-slave piconet and a scatternet.

Bluetooth relies on service discovery for locating services on other devices. Service discovery is based on direct interaction between the devices. When Bluetooth devices come in range of each other they are able to search for services via the SDP protocol (Service Discovery Protocol). In traditional networks like the Internet these service lookups are carried out through central directories, like for instance a DNS server. Central directories like DNS servers are not needed in Bluetooth networks. [39]

4.2 Bluetooth Protocol Stack

The Bluetooth protocol stack is built up by a number of layers like the OSI reference model. Figure 4.3 shows the Bluetooth stack. The Bluetooth radio corresponds to the physical layer in the OSI reference model, which is the lowest layer. The Baseband protocol and the Link Manager Protocol (LMP) correspond to the data-link layer in the OSI reference model. The Baseband protocol is responsible for establishing the physical links between Bluetooth devices, and this involves synchronizing the clocks and hopping frequency of the devices in a piconet. LMP is used to control links between Bluetooth devices, which involves negotiation of Baseband packet sizes, authentication and encryption, and controlling the power modes and transmission cycles of the Bluetooth radio.

All upper-layer protocols and applications communicate through ACL links. It is only possible to use SCO links for audio transmission, which runs directly on top of the Baseband protocol. The Host Controller Interface (HCI) is an interface between higher and lower layers of the Bluetooth stack. The HCI is typically the interface between the Bluetooth hardware and the operating system of the host computer. This interface makes it possible to have a Bluetooth module with its own processor implement the lower layers of the protocol stack, and have the host of the module implement the higher layers. The host of a Bluetooth module could for instance be a phone or a desktop computer. The upper-layer protocols are more interesting than the lower-layer protocols from our point of view, because we have to chose one or more of the upper layer protocols to interface with in order to build the proposed software system. Therefore the four main upper-layer Bluetooth protocols are covered in a bit more detail in the following subsections. [39]

L2CAP

Logical Link Control and Adaption Protocol (L2CAP) is a data-link-layer protocol, and it is the protocol that all higher-level protocols interface with. It is possible to develop applications that interface directly with L2CAP. L2CAP provides protocol multiplexing for higher-level protocols. This means that this protocol distinguishes between which higher-level protocol it is communicating with. L2CAP performs segmentation and re-assembly for higher-level protocols that send packets that are larger than what the Baseband supports. The maximum L2CAP packet size is 64 KB. Since the Baseband packets are limited in size (to 341 bytes) large L2CAP packets are typically segmented into multiple Baseband packets. This means that higher-level protocols create a large data overhead by sending large packets. L2CAP uses simple Baseband integrity checks to provide a reliable channel, but reliability

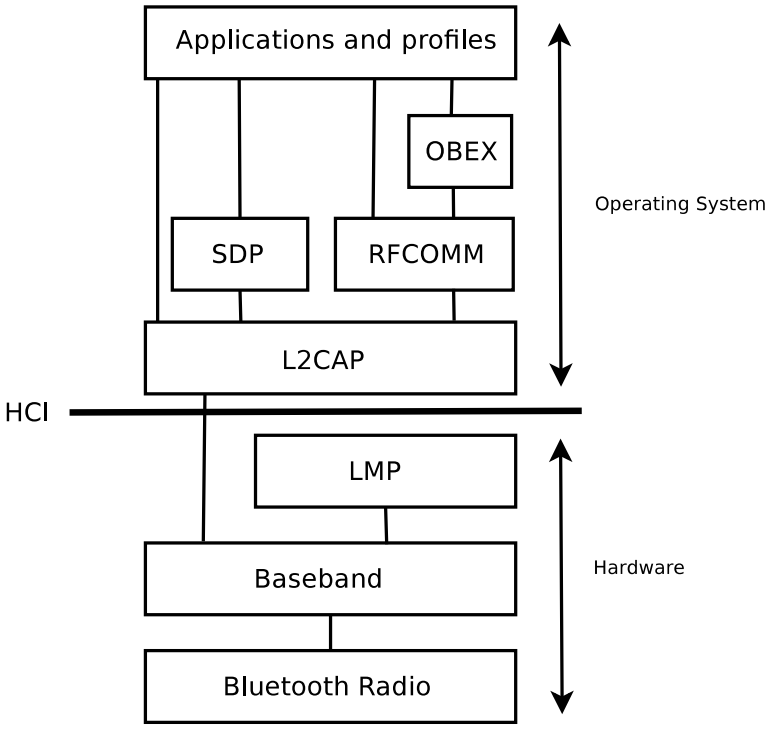


Figure 4.3: The Bluetooth protocol stack.

is not enforced.

L2CAP is a minimalistic protocol. And when using higher-level protocols a large percentage of the data being transmitted is header information, because each layer in the protocol stack uses its own headers. Therefore interfacing directly with L2CAP introduces the lowest possible overhead for applications, and because of this it provides higher bandwidth and lower battery consumption than higher-level protocols. [39]

SDP

The Service Discovery Protocol (SDP) defines how Bluetooth devices publish and discover services. In this context a service is defined as any feature that is usable by another device. SDP runs over a reserved L2CAP channel. A SDP database maintains a local database with information about all the services that are available on the device. This information is stored in Service Records. Each Service Record contains the attributes of a specific service, including the service UUID (Universally Unique Identifier). UUIDs

are global identifiers that are used to classify the type of Bluetooth services being offered. A service client can initiate searches through SDP for services that match a combination of UUIDs and attributes on discoverable SDP databases in range. [39]

RFCOMM

RFCOMM is a cable replacement protocol, and it resides in the transport-layer in the OSI-reference model. It emulates a serial RS-232 port connection over L2CAP, and emulates RS-232 control and data signals. RFCOMM is used as a virtual cable line between Bluetooth devices, and by some higher-level protocols such as OBEX. RFCOMM can for instance be used to support a direct connection between a computer and a printer. [39]

OBEX

OBEX is a session-layer protocol, and one of the adopted protocols in the Bluetooth stack. OBEX was originally developed by the Infrared Data Association (IrDA) with the purpose of supporting easy exchange of data objects. The OBEX protocol was easy to adopt because it is designed to be independent of the underlying transport protocol. In the Bluetooth stack the transport protocol can for instance be RFCOMM or TCP.

OBEX is similar to the HTTP protocol, but much lighter. It is session oriented and like the HTTP protocol also based on the client/server approach. OBEX allows OBEX clients to receive and send objects to OBEX servers, and to change the active directory at the server. The basic operations are connect, disconnect, put, get and setPath. The connect and disconnect operations are used to establish and close a connection to an OBEX server. When a connection has been established data objects can be pushed or pulled from the server via the get and put operations. OBEX has a built in header system that allows clients to specify the kind and name of the data they wish to receive or send.[39]

Other supported protocols

Bluetooth also supports a number of other protocols. Support for PPP, TCP/UDP/IP offers an alternative transport-layer protocol, and interface to the Internet. WAP is supported for sending and reading Internet content and messages. TCS BIN is supported for setting up speech and data calls between Bluetooth devices.[39]

Discussion

The most efficient approach in terms of power consumption and network bandwidth is to interface with the L2CAP protocol. However, using L2CAP implies having to deal with low level issues such as segmenting data into L2CAP packets on one side of the connection, and reassembling the packets on the other side in order to reconstruct the data. Interfacing with RFCOMM relieves us from this programming effort and allows us to work with data streams instead. So there is a tradeoff between efficiency and programming effort. We have chosen to interface with RFCOMM, because in this respect we value programming time higher than the potential increase in efficiency. Furthermore the software will be developed in a modular manner, which means that the module controlling the data flow can be replaced later if a more efficient one is needed.

4.3 Web Service Invocation over Bluetooth

It is not a trivial task to accomplish web service invocation through a Bluetooth channel, because web service invocation is commonly based on common Internet protocols, which means that SOAP node implementations like JAX-RPC (explained in 6.2.1) are based on the HTTP protocol. In order for us to achieve web service invocation over Bluetooth we must invent a scheme that will enable us to send SOAP messages from one Bluetooth device to another, but fortunately existing research have been conducted in this area. Two scientific articles present solutions to this problem [18] and [19].

The first article [18] presents a proof-of-concept of how to use Bluetooth technology for web service invocation. It presents a "lightweight framework" for sending SOAP messages from one Bluetooth device to another. The idea behind the framework is to put in a proxy layer between the web service components and the Bluetooth components on both the client side and the server side. Figure 4.4 shows an illustration of the proposed framework.

When the client Midlet or application issues a Soap request the Soap message is passed to the client Bluetooth proxy. The proxy serializes the Soap message into a byte stream and sends it to the Bluetooth device that runs the Soap server through a serial communication link (RFCOMM). On the server side this byte stream is passed to the server Bluetooth proxy, which then deserializes the byte stream into a Soap message. The Soap message is forwarded to the web service container, where it is invoked, and the response SOAP message is transferred back to the client Midlet in the same way through the proxy layer.

The second article [19] presents a performance evaluation of their framework.

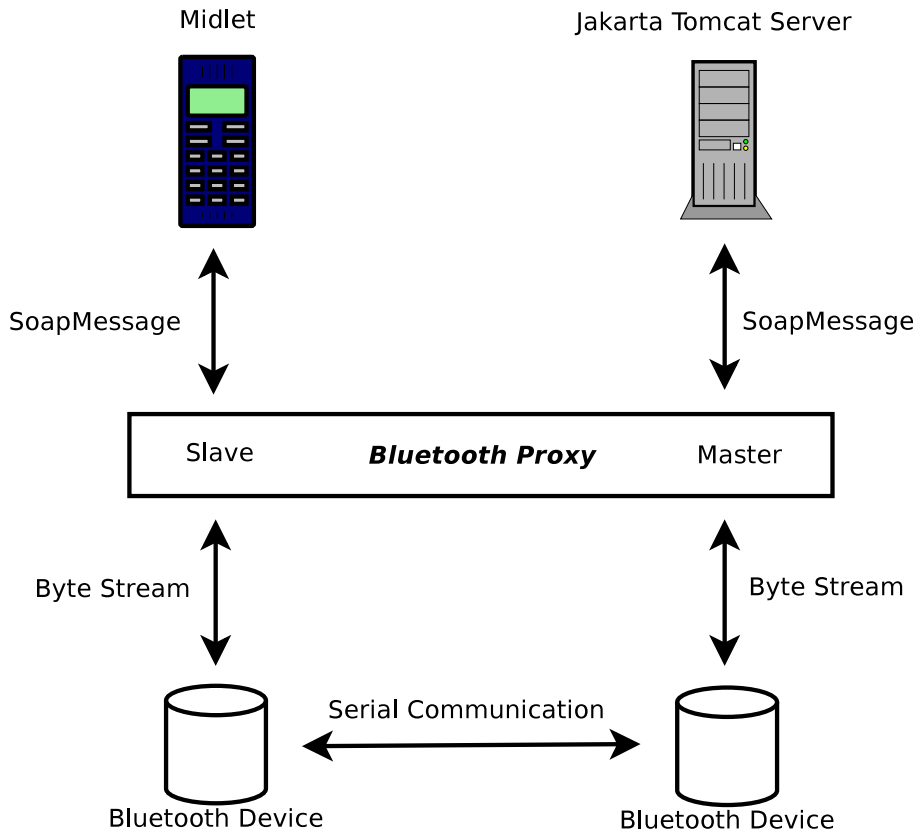


Figure 4.4: Framework architecture, based on figures in [18] and [19].

Here the through-put of the framework, the discovery times and the overhead introduced by the framework is investigated through experimentation. The article concludes that the framework has a high through-put and is applicable in real-world applications.

In our project we need a similar framework for web service invocation over Bluetooth. Therefore we use the basic ideas presented in this section as a starting point for designing the communication infrastructure that is needed.

4.4 Coping with Mobility

Using Bluetooth we have to take the properties of a radio connection that can be distorted and a client in motion into account. While arbitrary disconnections as well as bad signal states occur, the user expects continuous data transmission while she is moving. Solving these communication problems at

a low level without much user-interaction should be taken into account in DynaBlu.

We therefore aim to reduce the implications of reconnections and bad transmissions as much as possible. The first step in finding solutions, is to identify the problems implicated by client mobility and short-range radio communication.

We identify three possible communication problems. Figure 4.5 depicts these problems.

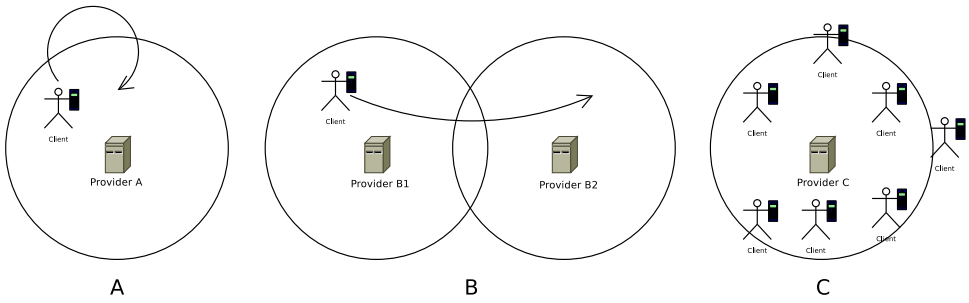


Figure 4.5: The three problems due to the mobility of the user: A *The reconnection problem,* B *The continuity problem, and* C *The discovery problem.*

A The Reconnection Problem

If the user leaves the vicinity of the provider he is currently connected to, the connection is lost and terminated. If the user after a short period of time enters the vicinity of the same provider again, a reconnection mechanism must occur without user interaction. This mechanism should be fast enough, so that the user is not inconvenienced by the time she must wait for a response from the provider. Furthermore such a reconnection must occur without having to restart the authentication procedure. Re-authentication would be too time consuming and entail costs because of the established GPRS-based Internet connection. Especially in the case where the user is moving at the border of the vicinity, frequent reconnection would disturb the seamless functionality of the web services provided by our framework.

B The Continuity Problem

The limited range of Bluetooth communication is a shortcoming that affects use-cases, where the vicinity of one provider is smaller than the area to be covered by a particular service. A natural solution would be to install several

providers that mirror the services of one provider. Figure 4.5*B* depicts such a solution. Provider B1 and B2 provide exactly the same content.

Since B1 and B2 use different Bluetooth devices for communication, the client will not recognize B2 as the same provider, resulting in loss of the former web-service interaction state data with B1 and renegotiating authentication and encryption keys for B2. This takes time and is costly because of the communication with the mediator established over the Internet. To solve this problem, a mirroring service may be adequate.

C The Discovery Problem

The user of the client application should not be concerned with the technical details of Bluetooth communication. Therefore discovery of providers and their services must occur automatically without any user interaction. The client application must dynamically survey the vicinity for available providers all the time to get all possible services.

The discovery of devices takes place in two steps. First all available devices in the vicinity have to be detected. In a second step every device that has been discovered has to be scanned for available services. If there are many Bluetooth devices in the vicinity, the time used for the second step will be multiplied by the number of detected devices because we have to scan every device that may run a provider application. Because of this discovery on Bluetooth devices can be a time consuming process. Device discovery takes approximately 18-25 seconds depending on the mobile device and the environment, and during this time the application that is running the device discovery is forced to halt.[59]

Discussion and Possible Solutions

The discovery problem is the most simple to solve. Device discovery can only take place, if the target device is in *discoverable mode*. If not, the device is invisible for the discoverer and is hence never scanned for services. Assuming push-based mobile systems as described in section 2.1.2 gain widespread use in the future, people could get annoyed with the constant bombardment of OBEX push-objects and would eventually disable the discoverable mode in their mobile devices. Push-based mobile systems do not work if a device is not discoverable. As more people disable discoverable mode, our discovery problem grows smaller. Though we should not rely on this assumption, but find a more elegant solution. A practical solution can be to improve the steps of the discovery process as described above. In the first step devices can be filtered according to the class of the device, for instance scanning of

other mobile phones can be avoided. In the second step an optimization can be to maintain a list of already scanned devices. This way time consuming repetition of the scanning process can be avoided. Delaying the scanning process in the client can also be part of the solution. This would improve the user experience, but might reduce the number of detected services in the vicinity.

Reconnecting to a previously connected provider without having to go through the Authentication phase againg (see page 22) can only occur, if the client-provider association through the encryption keys remains for some time after a disconnection. It also requires both the client and the provider to preserve potential stateful web service interaction to allow the client to continue where it lost the connection. The solution here could be a keyring in the client and the provider, that allows reconnection and transmission with the same keys for some time. With respect to security, the keys should have an expiration timeout. This keyring in combination with the preservation of the web service interaction in the provider, would solve the reconnection problem.

Solving the continuity problem requires the provider to be identifiable despite of using several different Bluetooth addresses.

A way to solve this problem is to let a provider define a number of "mirror devices" that share the same services through some standard web service in each provider. The client could then see which Bluetooth addresses are associated with a provider and reuse existing keys with the specified mirrors. The providers would of course need to exchange keys amongst each other.

We have now discussed a number of communication problems in our framework that would potentially influence the user experience of the system, and discussed possible solutions. This will influence the design and implementation of our framework, since a good user experience is crucial for the acceptance of the DynaBlu framework.

4.5 Summary

To avoid dealing with low level details such as segmentation and re-assembly of L2CAP packets we choose to interface with RFCOMM in the Bluetooth protocol stack. This will allow us to work with a stream connections instead, and we do not consider the overhead introduced by having to send RFCOMM headers as significant.

The communication infrastructure will be based on web service invocation over Bluetooth. To solve this problem we will need proxies on both sides of the Bluetooth connections, which can translate byte streams to and from SOAP messages.

Connection problems must be taken into account in order for us to provide the user with a continuous data transmission, because the communication is based on Bluetooth technology. Device discovery can be optimized by using black and white lists and by using different modes for scanning.

Chapter 5

Security

Security aspects such as confidentiality and authenticity are of vital concern in today's world. Today there is an increasing number of electronic transactions involving sensitive information, such as credit card numbers [50]. Potential malicious parties can misuse this information and make purchases with stolen credit card numbers. Because of the large number of applications that today require the ability to handle and protect sensitive information it is important to provide support for these types of scenarios in our framework.

Addressing security aspects has been a goal from the beginning of this project which was also stated in the our system requirements. In this section we listed the system requirements *Credibility* and *Security* stating that a user should always be safe when using our system. This implies that it should not be possible to communicate with a party in our system without having authenticated the identity of the party. We also stated that we want to provide support for encryption schemes enabling the possibility of developing information-sensitive applications with our framework.

In this chapter we discuss security aspects when communicating between the three parties in our system, namely the Client, Provider and Mediator, see figure 3.1 for an overview of the entities' roles in our framework. The communication types between these entities can be classified into the following two types of communication.

- Client-Mediator and Provider-Mediator communication over the Internet. We describe security aspects of these connection types in section 5.1.
- Client-Provider communication over Bluetooth. Security aspects of this type is described in section 5.2.

5.1 Internet

Internet communication will be used by our clients and providers to communicate with the Bluetooth Authentication Mediator. The purpose of the mediator is, as previously mentioned, to authenticate the identity of the communicating parties. Yet we have not discussed issues concerning authentication of the mediator's identity. A malicious user might impersonate this entity and give false authentication information. We thus need to provide mechanisms for securely establishing the identity of the mediator, and the mediator must be able to verify that it is communicating with the correct client and provider. Moreover we discuss methods of exchanging encryption keys securely. In this section we outline how these mechanisms work and how they can be used in our framework.

5.1.1 Authentication

Most Internet systems dealing with sensitive information today use the HTTPS protocol, which is a an encrypted version of the HTTP protocol. HTTPS is today understood by most browsers and also most modern mobile devices can communicate using HTTPS. Support for HTTPS was released in the Mobile Information Device Profile (MIDP) in 2001 [30], discussed in section 6.1.2.

HTTPS describes HTTP communication using encrypted Secure Sockets Layer (SSL) or using its successor Transport Layer Security (TLS). SSL/TLS initializes HTTPS communication by performing a handshake session where information is exchanged between the two communicating parties that enable them to encrypt and decrypt messages.

In this handshake the identities of the communicating parties are established, and protocol versions and encryption algorithms are agreed upon. This creates the possibility of a client/server to have support for a number of security schemes. As long as both parties have support for a common security scheme they can communicate securely with each other. [23]

An identity certificate is required by the web server in order for it to support HTTPS communication. A certificate contains a public and private key which are used to authenticate the identity of the communicating parties and used for the exchange of encryption keys. A public key is used to encrypt a message. This message can only be decrypted using a private key related to the public key. Note that in this section we only outline the workings of public/private key-based algorithms. For a more detailed description on how these work we refer to [35]. We outline the steps required to encrypt messages in figure 5.1. In this example Alice sends an encrypted message to Bob who decrypts it after receiving it.

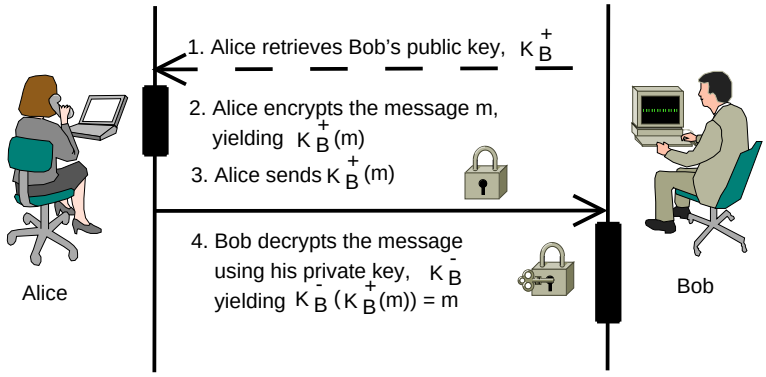


Figure 5.1: Alice and Bob communicating using public/private key-based encryption

This idea of using different keys to encrypt and decrypt messages is also referred to as asymmetric encryption. The most commonly used asymmetric encryption algorithm is RSA, which is explained further in [36]. If another user where to intercept the message $K_B^+(m)$ being sent in figure 5.1 it could not be decrypted as they would not have access to Bob's private key K_B^-. [35]

Signing is used to ensure that a message was indeed received from the correct sender. Signing a document requires the following steps. Illustrated in figure 5.2. [35]. Here Bob signs a message m using his private key K_B^-. Now Alice can verify that Bob indeed sent the message by applying Bob's public key K_B^+ to the received message.

However as asymmetric encryption relies on computationally heavy algorithms, faster symmetric encryption schemes are often employed in combination with the asymmetric encryption schemes. In this scheme a shared secret key to be used in the symmetric encryption is negotiated using asymmetric encryption. This procedure is handled by the SSL/TLS security protocol in the handshaking procedure [35]. [62][23]

To use HTTPS security we need to create an identity certificate in the Bluetooth Authentication Mediator. As it is though anybody can create a false identity certificate claiming that they are the mediator and thus act as a false mediator entity in our system. Thus we need to verify the validity of the identity certificate. This is done using a Certificate Authority (CA), for instance VeriSign [56] or Thawte [54]. A CA is responsible for providing trustworthy certificates that binds the public key of a party to an identity. This identity is represented in a certificate which is digitally signed by the CA. Another advantage of having our mediator certificate signed by a CA is that our mediator will then be able to issue certificates to providers using our framework and thus making the mediator a CA itself. [35] And due

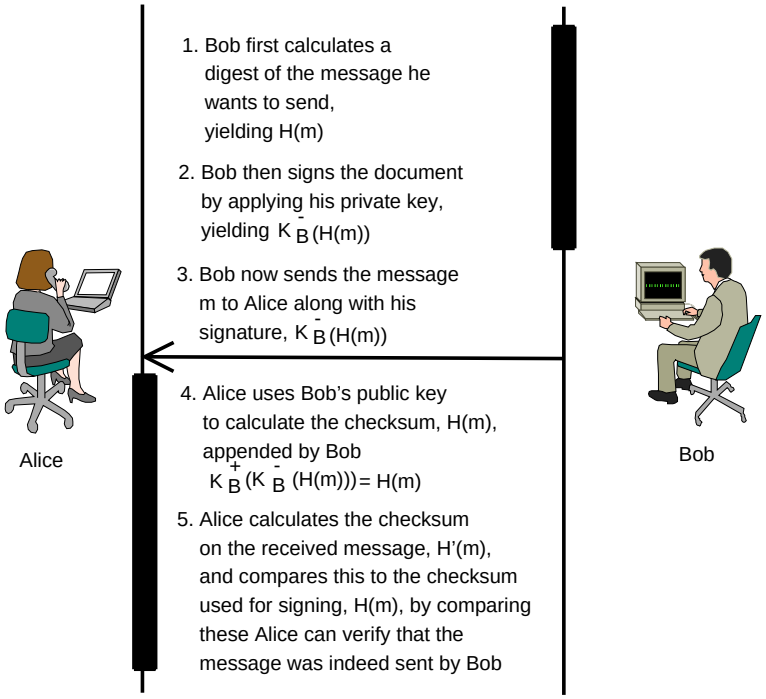


Figure 5.2: Bob digitally signs a document using his private key and sends the document to Alice who verifies the that Bob indeed sent the message

to our security constraints a provider must have a certificate signed by the Bluetooth Authentication Mediator in order for it to use our system.

However one problem still remains. The client application could be changed manually by malicious parties to skip the authentication process and thus allow the client to access unsafe information from non-authenticated providers. We solve this issue by digitally signing the client application with the certificate issued to our mediator. This will ensure that our mobile client application was indeed published by us.

5.1.2 Encryption

Data being transmitted using SSL/TLS, after the symmetric key has been exchanged in the handshaking procedure, is typically encrypted using the RC4 (Rivest Cipher 4) algorithm. This algorithm is not recommended for systems requiring high levels of security such as military systems. Though it is regarded safe to use for most Internet payment systems. RC4 is a *stream cipher* encryption algorithm, which means that it operates on the plaintext data one digit at a time. As opposed to a *block cipher* where the plaintext data is encrypted block-wise [23]. RC4 has been used in a

number of encryption protocols and standards including WEP and WPA for wireless LAN security, and also SSL and TLS for HTTPS. RC4 is referred to as a simple and fast encryption algorithm. Another issue with the RC4 algorithm is that it becomes easier, for a malicious third-party, to obtain the shared key being used in the encryption the longer the communication runs. This is because it will enable the third party to gather more data to be used for reverse-engineering the shared key. Thus for longer communication sessions, where large amounts of data are being transmitted, RC4 is not recommended. Yet we do not consider this to be a problem considering the application domain of our framework.

RC4 has never been officially released by RSA labs as it is trademarked. However an unofficial reverse-engineered implementation commonly referred to as ARC4 (Alleged RC4) was released in 1994. To avoid potential trademark issues ARC4 is actually the algorithm used in most security protocols today. The RSA labs license states that unofficial implementations of RC4 may be used freely as long as they are not referred to as RC4 implementations, thus we will henceforth in this report refer to RC4 as ARC4. We refer to [33] for a detailed description of the ARC4 algorithm. [63]

5.2 Bluetooth

Bluetooth was originally designed as a simple cable replacement. But transferring data by a radio link instead of a cable implies a threat to the confidentiality of a connection. The frequency hopping scheme used in Bluetooth provides no real protection against eavesdropping. The hopping scheme is derived from the master's clock and given to clients as soon as they try to connect to the master. A simple device discovery (by the inquiry procedure) will provide an attacker with the needed information for an attack [26].

Therefore, one of the design goals for Bluetooth was that confidentiality of a data link can be ensured using authentication and encryption. Thereby, security concerns were addressed from the beginning by the Bluetooth SIG. The Bluetooth core provides link-level authentication and encryption. This means, that only the link between Bluetooth cores is protected. The data flow between the application and the Bluetooth core through the mobile phones operating system is not secured. The operating system of the mobile phone must thus be fully trusted.

Authentication is based on user input and a challenge-response scheme. The encryption uses a symmetric key, which is derived from a link key generated by the authentication procedure.

5.2.1 Authentication

As mentioned before, authentication of Bluetooth devices is based on a mutual challenge-response scheme. The participants of a connection encode a random value with a shared link-key, which the counterpart sends back in decoded form. Thereby, the authentication of one device is granted. This challenge-response procedure must be initiated from both communicating parties to ensure mutual authentication. The link-key that both devices use for the challenge, is generated in the so-called *pairing*-procedure.

Pairing The pairing procedure consists roughly of the following phases, which are briefly outlined below. For a detailed description of the pairing procedure we refer to [26]

1. **Generating Keys.**
   On mobile phones, generating initialization keys involves user input. A common passkey entered at both mobile phones is needed. This passkey is used to generate the keys needed to provide authentication of messages, more specifically a *Link Key* is generated which is used to encrypt and decrypt messages between the communicating parties. Also a *Combination Key* is generated which we explain further in the next bullet.

2. **Link Key Exchange.**
   The combination keys are then exchanged. With these combination keys, each device now uses the combination keys to calculate a common link key. The advantage of using the combination keys to calculate a common link key makes the key exchange secure because the actual link key is never exchanged.

3. **Authentication.**
   To prove that the pairing procedure was successful, each device generates a random plaintext value m_{rand} and sends it to the other device. The other device encrypts it with the link-key $K_{AB}(m_{rand})$ and sends it to the device that sent the random value. The first device can easily check if they have the same link-key by calculating the same encryption on m_{rand}. This is called a challenge-response scheme, and is carried out on both sides. Thereby, mutual authentication is established, and the pairing was successful.

If the pairing process fails, it has to be retried. Otherwise, the link key is stored in the memory of each mobile phone, along with the Bluetooth address of the paired device. If a connection has to be established at a later time, a link key already exists, which makes the user-interaction unnecessary.[26, 11]

5.2.2 Encryption

If encryption is desired, an encryption key K_C has to be generated. This encryption key is generated from the link-key which is never used as an encryption key directly.

All algorithms used in the Bluetooth core are based on symmetric keys. The shared secret is thus the passkey entered by the user. For the pairing and authentication process Bluetooth uses a 128 bit block cipher named *SAFER+*. Using this particular algorithm was simply a decision made by the Bluetooth designers based on performance and licensing concerns. SAFER+ is freely available and has been one of the contenders for the AES algorithm.

For the encryption the Bluetooth core uses a stream cipher called E_0. This algorithm is based on work in the mid 1980's. Neither the SAFER+ algorithm nor the E_0 algorithm have known weaknesses as of the date of this writing[26].

We will not perform any cryptoanalysis on these two algorithms, but refer to Gehrman et al. in [26] instead. They have done exhaustive cryptoanalysis and conclude on the most effective attack found for impersonating that "in a practical system where encryption is activated, it is not at all easy to make something useful of this attack beyond the point of just disrupting the communication". And after a exhaustive cryptoanalysis of E_0 they simply conclude that "currently there is no attack known that breaks the complete encryption procedure with reasonable effort".[26]

The pairing process seems to be the greatest weakness of the Bluetooth core. If a short pass-key is entered, an attacker may have little trouble guessing the correct key K_C by listening in on the pairing process. The Bluetooth specification therefore recommends long pass-keys for sensitive applications.[26]

5.3 Discussion

Using long pass-keys, Bluetooth seems to be rather immune against security threats like eavesdropping and impersonation. This makes Bluetooth an eligible candidate for many security-sensitive applications. The designers of the Bluetooth core have been very careful in designing its security aspects. Though Bluetooth on the mobile phone is not secure by definition. Implementation flaws in the application layer and operating system, have in the past lead to several security breaches that could easily be exploited [26].

In our case, none of the yet known security breaches would affect us, since they all aim at services provided by the operating system and not by any

J2ME applications. From this perspective, and with regards to power consumption, authentication and encryption provided by the Bluetooth core would seem to be the best way to provide proven security within our application.

The only problem is, that the shared secret used in the pairing process cannot automatically be set by the J2ME application. It always has to be requested through the operating system in the mobile phone, which either already has a link-key stored, or has to request a passkey from the user to create a link-key. This is not feasible regarding the examples in chapter 1 which motivate our project. All of those examples motivate an automated detection and invocation of services available at some physical location of a user. Using pairing, the user would be forced to enter a passkey on the mobile phone for every service that requires authentication. Besides the required user-interaction, the passkey must be different for every user. Otherwise the whole pairing process would be worthless since an eavesdropper, recording the pairing process, could easily generate her own link-keys. Generating passkeys for every user would simply be impractical in many application domains of our project.

Therefore we have to implement authentication and encryption on the application level. Like the Bluetooth designers, we are not going to develop our own algorithm. In the perspective of lightweightness and confidentially, an existing algorithm based on symmetric keys is preferable. Based on our discussion of the ARC4 algorithm, we have chosen to use this algorithm to provide data confidentiality on the Bluetooth channel. This choice was based on the fact that ARC4 is a lightweight algorithm that provides a security level that meets the requirements of most Internet based credit card systems, which is more than adequate for most mobile applications in this domain.

We will be using the ARC4 implementation provided in the open Bouncy Castle security API [41]. Bouncy Castle is a Java lightweight cryptography API providing implementations of most commonly used encryption algorithms.

We expect that most of the applications developed with our framework will not be requiring data encryption security. However authentication is by our system requirements always required in our framework. We thus need to use secure and lightweight authentication mechanisms for use with the mobile client. For this reason we have based our Internet communication on using the HTTPS protocol.

Design and implementation aspects of the security mechanisms are discussed further in chapter 11.2.

Chapter 6

Development Platform

In this chapter we investigate the development platforms that we will need to create DynaBlu. We start by investigating the platform available to us on most client devices, namely the Java 2 Micro Edition (J2ME). Next we continue to describe specifications and third-party software frameworks of interest to us. The purpose of these descriptions is to identify and discuss the use of relevant specifications needed to implement our framework.

6.1 J2ME

Java 2 Micro Edition (J2ME) is a set of APIs targeted at small devices having limited hardware capabilities. J2ME is a limited cut-down version of the larger J2SE and J2EE Java versions.

Since a wide variety of devices exist with differing purposes and capabilities it has become impossible to create a single lightweight software product to suit all purposes. Therefore J2ME has been designed in a modular fashion. Instead of being designed as a single bulk specification J2ME was designed as a collection of individual specifications that can be combined to suit the needs of a specific device's hardware platform.

J2ME relies on *configurations* and *profiles*. Configurations are used to classify device types by their minimum hardware capabilities. Configurations serve as a common denominator for a specific device classification. Profiles are used to extend the configurations with specific functionality not present in the device's configuration, for instance Bluetooth capabilities is implemented in a specific profile. [55]

To provide an overview of the J2ME platform we refer to figure 6.1. The

configuration part in this figure is illustrated with the CDC/CLDC block explained further in section 6.1.1. The profile part in this figure is the MIDP block explained further in section 6.1.2.

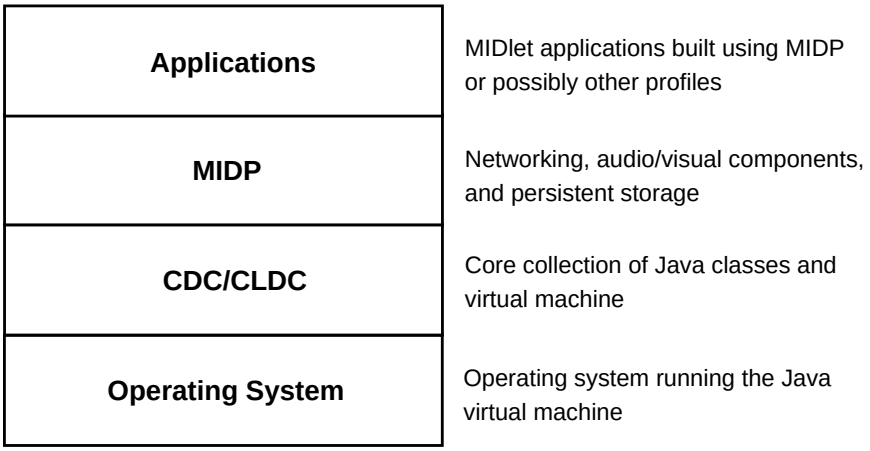


Figure 6.1: Overview of the J2ME architecture.

6.1.1 Configurations

As already mentioned J2ME is a development platform targeted at hardware constrained devices. Hardware constrained devices in this respect covers a variety of devices having different computational power and memory. The specifications available on a device thus should also be dependent on the hardware limitations of the device. For instance a simple mobile phone would probably be more constrained than a top-of-the-line PDA or smart phone. [32]

To address this problem two configurations have been specified to distinguish between high-end and low-end devices. A configuration here consists of a core collection of Java classes and a Java Virtual Machine. [55].

Connected Limited Device Configuration (CLDC) As specified in the CLDC 1.1 Java Specification Request, JSR 139 [5]. CLDC defines a standard set of classes available for devices having the following hardware capabilities.

- At least 160 KB of total memory available for the Java platform and CPU speed of at least 8 MHz. The CPU being a 16/32-bit processor.
- Limited power, often battery operated.

- Connectivity to some type of network, although with possibly limited (9.6 Kb/s or less) bandwidth.
- High-volume manufacturing (usually millions of units).
- User interfaces with varying degrees of sophistication down to and including none.

Connected Device Configuration (CDC) Classifies high-end devices. This configuration addresses devices having more memory, increased CPU capabilities and more network bandwidth than CLDC devices. This configuration is supplied in devices being too limited to use J2SE and too powerful to use only the class-functionality supplied with the CLDC. CDC devices are typically PDAs or smart phones.

The CDC 1.1 specification, JSR 218 [9], defines like CLDC a standard set of classes available for devices with the following hardware specifications.

- An increase over a CLDC device in the ROM size available ranging from 128-256 KB.
- 512 KB minimum RAM available.
- Robust connectivity to some type of network.
- Same requirements to user interfaces as in the CLDC specification.

Mobile phones today are typically shipped with the CLDC configuration. Yet as modern phones are becoming increasingly sophisticated we should expect one of two future scenarios. Either more CDC configured phones will be shipped or we should expect more sophisticated CLDC configurations to be released.

As the client in our system will be using a mobile phone, we will focus on the CLDC 1.1 configuration in the remainder of this project.

The virtual machine conforming to the CLDC 1.1 specification known as the KVM (K Virtual Machine) has a number of lacking features compared to the J2SE virtual machine. Of noteworthy restrictions in the KVM is the lack of dynamic class loading. We discuss this restriction further in section 6.5. For more information on the KVM we refer the reader to [5].

6.1.2 Profiles

The CLDC 1.1 configuration is limited to only supporting a minimal set of the hardware in a device. To reach the full potential of the hardware in a device the concept of profiles was introduced. A profile complements

a configuration by introducing an additional set of classes. The additional classes serves the purpose of supporting specific hardware in devices. [55]

The Mobile Information Device Profile (MIDP 2.0) defined in JSR 118 [10] provides a base platform extending the CLDC with more specific functionality. The MIDP 2.0 profile gives developers the capability of creating more advanced audio/visual applications having advanced network capabilities.

MIDP 2.0 provides 7 packages aimed at the following:

1. Enhanced network support
2. Better user interface capabilities
3. Game development tools
4. Media tools for audio playback
5. The MIDlet framework, a MIDlet being a Java program compliant to the J2ME virtual machine
6. Providing persistent storage for MIDlets
7. Security capabilities

For our system we will be requiring a number of the features provided by the MIDP profile. We will be taking advantage of the enhanced network support and security capabilities provided by MIDP to connect to and invoke services deployed at the Bluetooth Authentication Mediator securely. Also the enhanced UI capabilities and media tools opens the opportunity for creating more advanced Bluetooth services with respect to user presentation.

Today MIDP profiles have become widely spread in modern mobile phones also MIDP have become integrated in most integrated development environments aimed at mobile devices. This makes programs conforming to MIDP portable to other devices as long as they support the MIDP profile, for this reason MIDP 2.0 is also backwards compatible with older MIDP profiles. [32] This makes MIDP profiles attractive for use in our project as one of our stated philosophies is *Compatibility*, see section 3.4.

6.2 Web Services

The reader is required to have preceding knowledge of web services to read this section. The report in [17] provides an introduction of the web services framework.

To realize network connectivity in our framework we were required to use a distributed platform that makes it possible for us to make RPC calls to external resources using either Bluetooth or GPRS. Reflecting on our Compatibility and Openness philosophies a platform that is both standardized, widely used and familiar to developers should be considered. Firstly the platform should not be locked to a specific vendor platform but should be accessible from several development platforms. Secondly the platform should use open formats, which makes potential integration with external systems easier.

Based on these philosophies and the growing interest in the web services framework [17] we have chosen to focus on this platform to implement our network connectivity.

Using web services though implies having to send Soap formatted messages, which involves data overhead when sending and receiving RPC messages. To cope with this on a mobile device we need to examine lightweight web service frameworks. We present two web service frameworks aimed at mobile devices in this chapter.

6.2.1 JSR 172: J2ME Web Services Specification

The JSR 172 Web Services Specification [6] released in 2004 provides the necessary class functionality to create web service clients on mobile devices. This specification provides a WSDL and Soap parser API, based on a subset of the well-known SAX2 (Simple API for XML) parser. This parser can be used to create so-called web service stubs. A stub is a local Java representation of a remote web service. Having this stub relieves the developer of having to manually deal with the tasks usually involved with invoking a web service, e.g. creating and sending a Soap request conforming to the web service's remote WSDL description. These tasks are handled by the JSR 172 runtime components.

Although JSR 172 provides easy-to-use facilities for creating web service clients it is aimed at an abstraction level that is too high for us when dealing with Bluetooth web services. By this we mean that when using JSR 172 we cannot manually create or manipulate the actual Soap messages, these are instead generated by the JAX-RPC runtime. In order for us to make web service invocation on Bluetooth available we need to create our own Soap node in the client software. This implies building functionality for generating, parsing, sending, and receiving Soap messages.

6.2.2 kSOAP

To address this issue we have investigated third-party Soap frameworks. Based on multiple citations in related articles we found kSOAP (Kilobyte SOAP) [34]. kSOAP is a lightweight web services framework for mobile devices, allowing developers to create web service clients. kSOAP was developed before SUN Microsystems decided to release the JSR 172 reference specification and thus kSOAP was not developed to conform to the this specification. Instead kSOAP was developed independently by a smaller development team, currently 4 persons are involved.

The kSOAP framework is based on the kXML framework which is a lightweight XML framework providing an XML parser on a mobile device. We have chosen the newest kSOAP 2.0 version since this provides us with needed functionality for serialization of Soap messages. After having examined the API for kSOAP and conducting practical experiments, by creating Soap messages and using these to invoke a web service, it was quickly revealed to us that kSOAP provides functionality for manually creating and parsing Soap messages, which is exactly what we need for our Bluetooth services because it enables us to not only invoke web services using HTTP but also Bluetooth.

Using this framework implies having to include the kSOAP and kXML packages in our client application. Though considering that these packages requires approximately 650 KB this is a reasonable trade-off instead of having to spend our time developing a complete Soap node (And XML parser) ourselves.

Another advantage of using kSOAP is that we will not be requiring the clients mobile device to have support for the JSR 172 specification, which has not yet gained wide spread due to its recent release in 2004 [6]. This will also further enhance the compatibility of DynaBlu.

6.3 JSR 82: Java APIs for Bluetooth communication

In 2002 JSR 82 was released by the Bluetooth SIG. This specification defines the Java APIs for developing Bluetooth enabled applications targeted at J2ME. This API is designed to run on the CLDC as an extension package for a J2ME profile, like for instance the MIDP.

JSR 82 allows programmers to interface with the higher-layer Bluetooth protocols, and to develop applications that discover and publish Bluetooth services that can be accessed from other Bluetooth enabled devices.

JSR 82 defines a number of classes that makes it possible to search for nearby Bluetooth enabled devices, and to search for services on the devices that have been discovered. When a service have been discovered the framework returns all necessary information about the service, such as connection strings and service attributes.

Establishing connections to services that have been discovered is transparent through the Generic Connection Framework (GCF). The GCF is defined in CLDC 1.0 as a set of classes that facilitate access to resources (the profile MIDP 1.0 extends the GCF framework). Because of the way the framework is designed resources are accessed through the same method regardless of what type of resource it is. A connection string can simply be passed to the *Connector.open()* method and this will open a connection to the service. JSR 82 contains a number of classes for publishing Bluetooth services at different levels in the Bluetooth protocol stack.

Since the communication between the Client and the Provider in the proposed system is based on Bluetooth, we need to develop software for the client that interfaces with JSR 82. The client must use functionality that enables service discovery and communication over RFCOMM. [11]

6.4 Bluetooth Connectivity with J2SE

In contrast to J2ME, there is no standard API for Bluetooth communication in the J2SE core library. There is of course the JSR 82 API for J2ME, but no standard implementation for any PC platform. The reason for this is that the implementation of a Bluetooth stack is dependent on the operating system and often also dependent on the Bluetooth hardware. This is the reason for SUN's decision not to implement Bluetooth in J2SE but leave such implementations to third party developers.

We need Bluetooth connectivity in the Java application of the provider (see figure 3.1 on page 22). Several third party libraries exists that we took into consideration. They differ mainly in the number of supported operating systems, communication protocols and their license of use. Table 6.1 lists these libraries.

As we concluded in section 4.2, we use the RFCOMM protocol to communicate between the Bluetooth devices. Support for this protocol is therefore the main criteria for the choice of a suitable Bluetooth library. The Bluecove, Impronto and Avetana libraries all support RFCOMM.

For a potential deployment of our project, support for various operating systems is preferable, this is also in accordance with our compatibility system philosophy. According to table 6.1, the Avetana library supports the largest

Name	Operating Systems	Protocols	License
Bluecove [52]	Windows XP SP2	SDP, RFCOMM	LGPL
JBlueZ [53]	Linux(BlueZ stack)	HCI interaction only	GPL
Impronto [46]	Linux	SDP, L2CAP, RFCOMM, OBEX	Commercial
Harald [25]	Linux(BlueZ stack)	HCI interaction only	GPL
Avetana [20]	Linux(BlueZ stack)	SDP, L2CAP, RFCOMM, OBEX	GPL
Avetana [20]	Windows, MacOS X, Windows Mobile	SDP, L2CAP, RFCOMM, OBEX	Commercial

Table 6.1: Third party Bluetooth libraries for J2SE and their features and limitations

number of operating systems. Furthermore the Avetana library implements the JSR 82 API partially. This is beneficial for the implementation of the Bluetooth communication in the client and the provider, since the implementations are syntactically similar to each other.

The Linux implementation of the Avetana library is based on JBlueZ and uses the BlueZ Bluetooth stack and driver to access Bluetooth devices. Since Avetana extends JBlueZ and JBlueZ is released under the terms of the GPL license, Avetana is forced to release their product under the GPL license as well. This is comfortable for us, because the source code is thereby made available for free download.

For a possible deployment scenario, we might decide to choose another library because the GPL license possibly forces us to release the source code as well. This could eventually destroy our business strategy. But for now, we use the Avetana library which seems suitable for the first steps.

6.5 Dynamic class loading

The DynaBlu framework is based on being able to invoke web services that have been discovered dynamically. In order to do this stub classes must be generated at runtime that match the WSDL files describing the interface of the web services. The standard way to do this is to download a class file from a server and load the class file with a custom class loader. This concept of loading new classes at runtime is called dynamic class loading. When the class is loaded it can be instantiated as a stub class and web service operations can be invoked with calls through this stub. This dynamic generation of stub classes presents a problem for us, because of limitations

in the KVM. In the following sections we discuss this problem further and investigate possible solutions to overcome this problem.

Dynamic class loading in the KVM

The Java Virtual Machine (JVM) supports dynamic class loading by allowing custom class loaders. A custom class loader is created by subclassing java.util.ClassLoader and can essentially be implemented to interpret any Java class represented in a bytestream.

The KVM disallows the use of custom class loaders. There are two reasons for this:

1. Security. The KVM and the CLDC disallows dynamic class loading to prevent execution of malicious applications without the knowledge of the user. This decision was made early in the CLDC 1.0[4] specification in 2000. Although the hardware platform of mobile devices has changed since then, CLDC 1.1[5] from 2003 still disallows dynamic class loading. Restrictions of dynamic class loading is an implication of the so-called sandbox model imposed on CLDC applications.

   The restrictions of the sandbox model are explained as follows in the CLDC 1.1 specification: *"Java applications cannot escape from this sandbox or access any libraries or resources that are not part of the predefined functionality. The sandbox ensures that a malicious or possibly erroneous application cannot gain access to system resources."*[5]

   In addition to the sandbox model the CLDC 1.1 has further restrictions on the use of dynamic class loading: *"by default, a Java application can load application classes only from its own Java Archive (JAR) file. This restriction ensures that Java applications on a device cannot interfere with each other or steal data from each other."*[5]. This restriction prohibits us from sending a Java Archive (jar) file containing stub classes, to a CLDC device and then executing this jar to invoke web services through its stubs.

2. Hardware restrictions. The low amount of memory available in mobile devices and the slow CPU's would make dynamic class loading impractical. The reason for this is the cost of executing the CLDC preverifier on a mobile device. The CLDC preverifier is the component responsible for converting compiled Java class files into a file that will work in a CLDC compatible VM. The memory costs of applying the preverifer is discussed further in section 5.2 of the CLDC 1.1 specification [5]

Possible Solutions

From our perspective there are three ways to solve the problem with dynamic class loading in the KVM.

Extending CLDC
One idea could be to extend the CLDC to include a dynamic class loader for the KVM. In the article [44] they discuss and compare different possible solutions for the introduction of a dynamic remote class loader in the CLDC. However, despite of their efforts they do not provide any reference implementation. Nor have we seen any signs of an implementation of a dynamic class loader in future CLDC specifications. A problem with a potential solution like this is that it would require the implementation of a custom KVM in the mobile device, and as a result of this dynamic class loading would only work on phones using this non-standard KVM.

Hacks and Workarounds
Another idea is to attempt to facilitate dynamic class loading artificially by using hacks and workarounds to manipulate the system of a mobile device.

In [3] a method to provide dynamic class loading in the KVM is proposed. This method builds on the fact that the Palm implementation of the KVM stores class information in a local database, that is accessible by client applications at runtime. By modifying this database at runtime, introduction of new classes into the system is made possible. The problem with this solution is that it only works with the Palm implementation of the KVM.

In [2] a workaround that allows accessing native methods from a Midlet is explained. This workaround makes it possible to implement a kind of "dynamic class loading". This solution is based on interacting with native methods through socket calls to localhost. However, platform-independence is only given by the methodology of using socket connections to localhost. The workaround is implemented in C++ and bound to Windows Mobile, Linux, and Symbian OS distributions on mobile devices.

Runtime Proxy Information
A third solution is to avoid dynamic class loading altogether. Instead of representing a web service application in a stub class and using the stub to interact with the application it is possible to parse the information needed to communicate with a web service directly from the service's WSDL interface. This parsed information can be used to create and send SOAP messages at runtime using the kSOAP framework. Thus instead of having to dynamically load a stub class to invoke web service operations we can instead use the information stored in the WSDL interface to instantiate the communication classes needed to communicate with the web service. We call this method *runtime proxy information*.

The runtime proxy information required when using this workaround includes the name of the web service, its operation names and their associated parameter signatures, and finally the XML namespace that the web service is deployed in.

Obtaining the runtime proxy information from a WSDL interface though obviously requires the client to know the location of a web service's WSDL interface.

Conclusion

The methods we investigated concerning extending the CLDC and using hacks and workarounds proved to be inadequate for our use because they suffer from being either hardware dependent and/or bound to a specific mobile operating system. These constraints contradict our philosophy of Compatibility, see 3.4.

The only viable solution we have found to solve our problem is to use the workaround involving runtime proxy information and parse this from the WSDL interface describing a web service application. Using this information we can instantiate the communication classes needed to interact with the web service by using kSOAP. Though as previously mentioned we will need to solve the problem concerning the location of a web service applications WSDL interface, in other words the client must be able to obtain a WSDL file describing a web service at runtime. We discuss our solution to this problem in chapters 9 and 10.

Part III

Design

Chapter 7

System Design

In this chapter the general design and interaction pattern between the main components of the system will be discussed. The design of each of the components in figure 3.1 from the problem statement is presented, which includes important design decisions and interesting implementation details. The Bluetooth communication link in the bottom of figure 3.1 from the problem statement will be implemented in a separate software module called the Bluetooth Communication Bridge. This bridge is responsible for maintaining secure and reliable connections between Bluetooth clients and providers, and to enable transmission of Soap messages through the bridge. To improve readability the Bluetooth Authentication Mediator in figure 3.1 is referred to merely as the mediator in the rest of the report.

Before elaborating on the design of the individual components the remaining part of the introduction gives an overview of the overall system interaction design. Figure 7.1 presents a sequence diagram of the system design.

In the left side of the figure there is a legend indicating which phase the interaction is part of, which are the communication phases that were presented in the problem statement (note that the publishing phase is not part of the interaction design, this phase is part of the installation of a new provider).

Discovery phase In this phase the client and provider communicate through the Bluetooth channel. The client initiates service discovery and a Bluetooth connection to the provider. Note that the provider does not immediately set up a bridge for this client in case negotiations with the mediator fail (doing this would make the system vulnerable to denial-of-service attacks).

Authentication phase The client and provider communicate with the mediator over the Internet (GPRS in the case of the client). The client and provider attempt to authenticate each other and negotiate connection de-

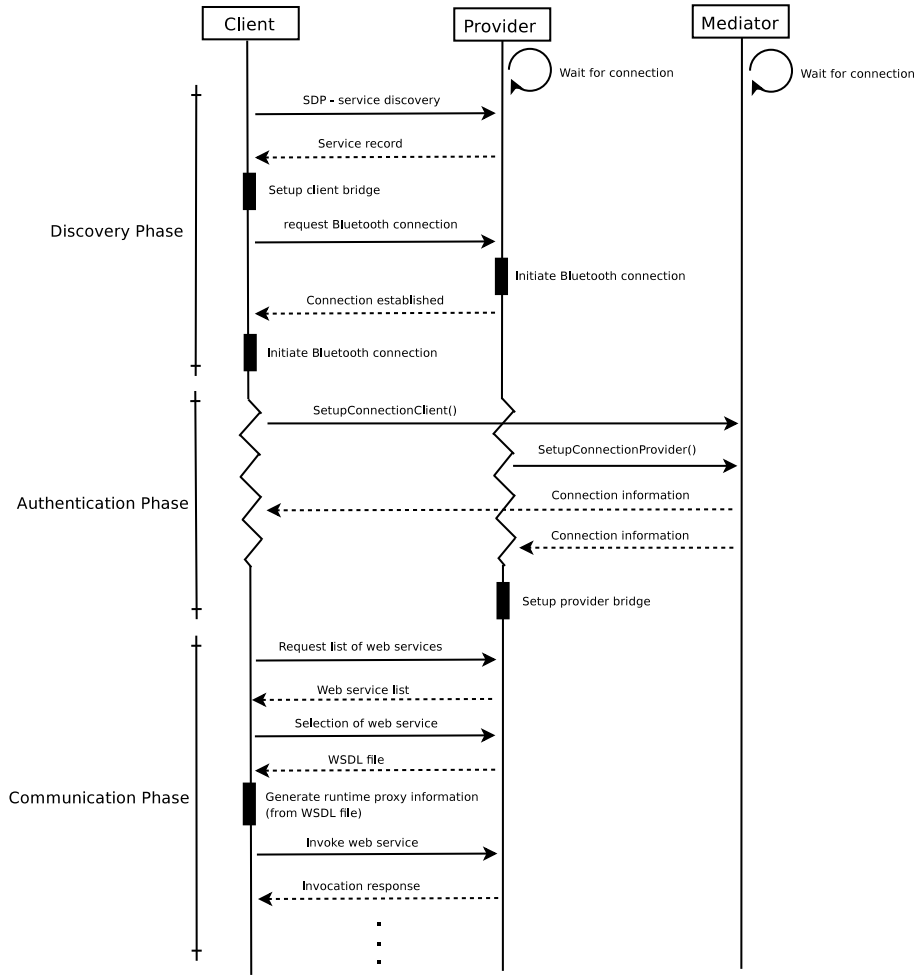


Figure 7.1: Overview of system interaction design. The zigzag part of the line implies that the interactions do not happen in any particular order.

tails. If this negotiation is successful the provider sets up a provider bridge for the client.

Communication phase Further communication between this client and provider now proceeds through the bridge by exchanging SOAP messages. First the client requests a list of available web services, and upon selecting one of them a WSDL file is returned. The client uses this WSDL file to generate runtime proxy information needed to invoke the selected web service.

Chapter 8

Bluetooth Communication Bridge

As we described in chapter 3, the central aspect of our framework is to invoke web services using an application on a mobile phone. Technically, the web service is not invoked on the mobilephone, but on the provider. To realize this we build a communication bridge over a Bluetooth channel that makes this possible, hence the name *Bluetooth Communication Bridge*.

In this chapter we will describe this bridge and its design on both the client side and the provider side.

8.1 Bridge Design

When web service requests and responses pass through a Bluetooth connection there are certain requirements to the data transport. Reliability, integrity, authenticity and security are provided by the HTTP and the HTTPS protocols when web services are invoked over the Internet. The Bluetooth communication bridge must at least provide the same level of reliability and security. Therefore these quality-of-service aspects have to be implemented over the Bluetooth channel. These aspects are enforced in the communication protocol of the bridge.

An overview of the Bluetooth communication bridge is given in figure 8.1. Inspired by the OSI reference model and [42], we chose to apply a layered architecture for the implementation of the communication protocol. The QoS requirements can be separated into groups, which are ideal candidates for individual layers. This architecture eases maintenance of the sourcecode since every layer is decoupled from each other. Also because of the similarity between the provider side and the client side (see figure 8.1) parts of the

Bluetooth communication bridge have to be implemented only once. Furthermore the layered architecture makes it easy to add new functionality by adding new layers, and to exchange existing functionality by replacing layers. This adheres to our system requirement of extensibility. We illustrate this shortly.

Each layer in our bridge is responsible for its own isolated functionality. The following listing briefly characterizes each layer and their respective responsibilities.

Bluetooth Layer Is responsible for sending and receiving the individual bytes of a complete data stream over a Bluetooth connection.

Integrity Layer According to [22], Bluetooth enforces reliability for the individual bytes being sent and not for the total amount of data being sent in a message. Therefore we have to ensure the integrity of a complete message ourselves.

The Integrity layer is responsible for checking the integrity of an entire message. This is achieved by embedding the byte array data being sent with a checksum. This checksum is used on the receiving side to check the integrity of a message. The details of this layer are explained in section 8.2.1.

Security Layer Is responsible for the authentication and encryption of a message. It is based on symmetric encryption and thus needs a common key between the client and the provider which is negotiated with the mediator in the *Authentication phase* (see page 22). Further description of this layer is in section 8.2.2. This layer was implemented to satisfy our *Security* and *Credibility* philosophies.

Routing Layer After a provider has received a SOAP message. This needs to route the SOAP message to the web service application it was intended for. The routing layer is responsible for transferring the required routing information to the provider. Because of the triviality of this layer we will not be describing it further. The implementation of this layer can be seen in `d619a.common.bridge.routing.RoutingLayer` [24] .

Serialization Layer Is responsible for converting Soap objects to a byte array and vice versa. This serialization and deserialization process is done by converting the SOAP messages being sent to and from byte arrays. We have derived most of the implementation in this layer from the kSOAP framework and we will thus not be describing this layer further. The implementation of this layer can be seen in `d619a.common.soap.serialization.SerializationLayer` [24] .

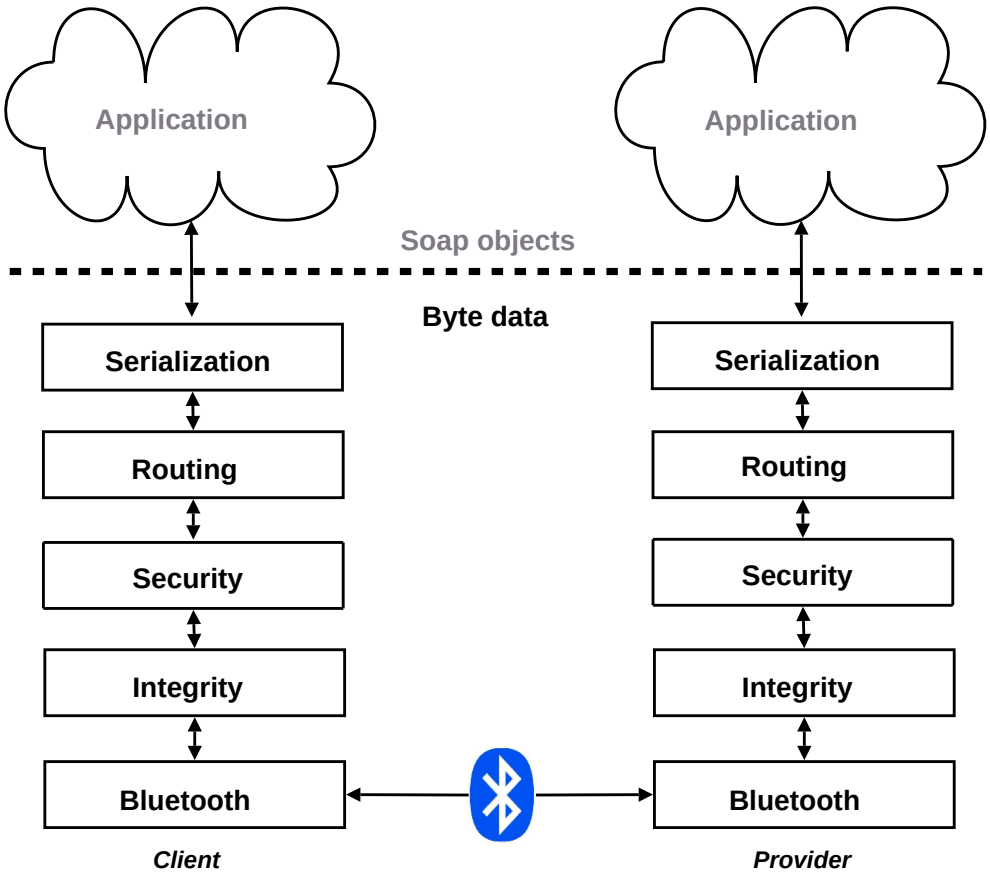


Figure 8.1: Overview of the layered design of our Bluetooth communication bridge. The dashed line parts the bridge from the rest of the provider and the client application (greyed out).

At the implementation level, we accomplish the decoupling of the layers by using interfaces. Each layer implements the same interfaces for sending and receiving data. The layers can then be put together by passing references to the individual objects. Our interfaces have been implemented in the classes `Transport` and `TransportCallback`, located in the Common Modules component in `d619a.common.bridge.interfaces` [24]. The Transport interface describes the functionality for sending data down the stack while the TransportCallback interface describes the functionality that is used for sending notifications to the upper layers by using callbacks.

To keep the abstraction between the application and the Bluetooth communication bridge, we implement on both the client and provider side a *Bridge Manager* which is responsible of instantiating and putting the layers together. It is also responsible for providing the individual layers with additional information such as the encryption key. Setting up the bridge layers

is the same on both sides. Listing 8.1 shows how the bridge is set up by connecting all the layers. In line 3 the integrity layer is created with the Bluetooth layer as a parameter. This has the effect that the integrity layer uses the Bluetooth layer to send data. Line 7 sets the callback receiver of the Bluetooth layer, which has the effect that data received in the Bluetooth layer will be passed to the integrity layer. This way layers are initialized and connected to each other.

```
1  private void setupBridge() {
2      bluetoothLayer = new BluetoothTransport();
3      integrityLayer = new IntegrityLayer(bluetoothLayer);
4      serializationLayer = new SerializationLayer(
           integrityLayer);
5
6      //Set up callback receiver references
7      bluetoothLayer.setCallbackReceiver(integrityLayer);
8      integrityLayer.setCallbackReceiver(serializationLayer);
9      serializationLayer.setCallbackReceiver(this.
           dataReceiver);
10   }
```

Listing 8.1: Initialization of bridge layers.

As we stated earlier, the authentication and encryption key for the Bluetooth communication is negotiated with the mediator after the Bluetooth connection has been established. Since the security layer depends on this key, the layer is inserted in the protocol after the negotiation with the mediator. Listing 8.2 shows this insertion. In lines 9 through 10 the security layer is connected to the rest of the bridge. This example demonstrates the flexibility of the bridge design.

```
1  public void addAuthEncToConnection(BigInteger authencKey,
       boolean onlyAuthentication){
2      //instantiate authenc layer
3      if(onlyAuthentication)
4        securityLayer = new SecurityLayer(EncryptionMode.OFF,
             authencKey);
5      else
6        securityLayer = new SecurityLayer(EncryptionMode.ONN,
             authencKey);
7
8      //connect authenc layer to the rest of the protocol-
           stack
9      integrityLayer.setCallbackReceiver(securityLayer);
10     serializationLayer.setLowerLayer(securityLayer);
11   }
```

Listing 8.2: Adding the security layer to the bridge.

Because of the problems implied by the mobility of the user (see section 4.4), the bridge manager also has to take care of connecting, disconnecting and reconnecting Bluetooth communication channels. Furthermore each side of the bridge has individual requirements to the handling of connections. The

provider bridge, where many clients try to connect at the same time, must be capable of handling many connections at the same time. The client bridge must only handle one connection at a time, but it has to scan regularly for new providers and monitor the signal of registered providers. Since these requirements for the connection management are different in the provider and in the client, we explain the design of both sides as part of the Bluetooth Communication Bridge in the next two sections.

8.1.1 Provider Bridge

As we have discussed in section 4.4, the moving nature of our client must be taken into account in the design of our framework. In the provider especially with regards to the handling of multiple client-connections at the same time.

Designing a provider that is capable of dealing with only one possible client-connection is trivial. Figure 8.2 depicts the design of such a bridge. Because the provider knows which client connects to the provider as well as the authentication and encryption key, the provider just has to wait for the client to connect. After a successful connection establishment, the client can invoke a web service in the provider. Since there is only one client, connecting and disconnecting to and from the provider frequently, requires the provider to wait for the client to reconnect only. The web service responses from the provider are always sent through the one and only existing bridge and communication channel to the client. And regardless if the web-service interaction is state full or not, since there is only one client, the web service-connector in the provider persists.

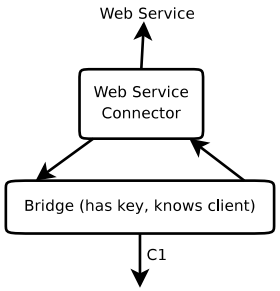


Figure 8.2: Schematic drawing of a trivial provider. This provider is only capable of connecting to one known client (noted C1 as connection to client 1), using the same web service and a known authentication and encryption key.

In reality, clients are not known beforehand and the assumption of only one client connecting at the same time is not realistic. Therefore we must design the provider to be robust for handling many concurrent connections as well as continuous connection loss and reconnection attempts.

To ensure maximum flexibility in number and identity of the clients, the basic idea is to let the provider to be a hub for a theoretically infinite number of providers described in the trivial case above and depicted in figure 8.2. Using this template of *virtualization*, the implementation gains scalability and implementation transparency.

Figure 8.3 shows the principle of this virtualization. A process inside the provider establishes the service entry in the Bluetooth registry and waits, until a client connection is detected. As soon as a client tries to establish a connection using Bluetooth communication with the provider, a new communication stack (named "Bridge" in figure 8.3) and a web-service connector instance is created.

These two elements together form a provider-instance for one given connection. This technique is very scalable, since the provider can theoretically handle an infinite number of connected clients. Each connection is also completely separated from the other clients making the handling of several different web service requests trivial.

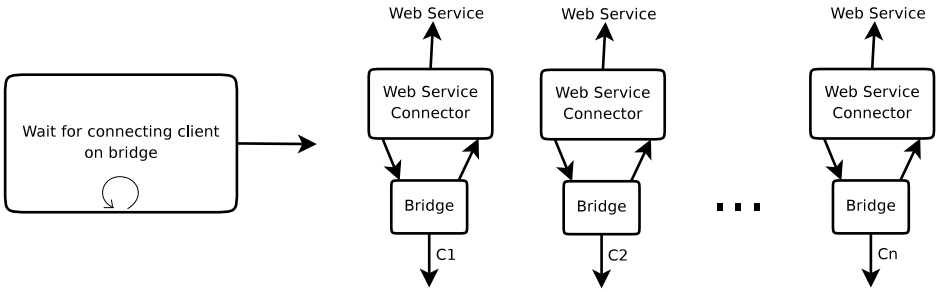


Figure 8.3: Schematic drawing of the virtualization of the provider. Each client (1,2,3..n) that connects (connection C1, C2 .. Cn) to the provider over the Bluetooth bridge, gets a new instance of the provider.

Implementation

As figure 8.3 shows, the provider is virtualized for each connecting client. This virtualization is the main responsibility of the class `d619a.provider.bridge.bluetooth.BridgeManager` [24] . Since the BridgeManager constantly listens for incoming client connections, it is implemented as a thread. Listing 8.3 shows the `run()` method which is the entry point for the thread. As soon as a client connection is established, the method returns with the

Bluetooth address of the connected client. This address is then along with the `bluetoothLayer` object passed to a new instance of the BridgeConnector class which is a inner class of the BridgeManager. The BridgeConnector object is responsible for establishing the Bluetooth communication bridge as well as connecting it with a service connector. According to the provider virtualization principle, the combination of bridge and web service connector has to be decoupled from the BridgeManager, Hence, the BridgeConnector class is implemented as a `Thread` . That causes `bridge.start()` in line 14 of listing 8.3 to return immediately. The `while(doRun)` loop starts over again and waits for another client connection which completes the virtualization as shown in figure 8.3.

```
1  public void run(){
2
3    //* run the bridge manager as long
4    //* as there is no shutdown event
5    while(doRun){
6      BluetoothTransport bluetoothLayer;
7      try {
8        bluetoothLayer = new BluetoothTransport
9                                (uuid, serviceName);
10
11       //wait for a client to connect
12       BTAddress clientAddr = bluetoothLayer.
13                                waitForClientConnection();
14
15       //a client has connected! ->run the bridge building
16       BridgeConnector bridgeCon = new BridgeConnector
17                                      (bluetoothLayer,
                                           clientAddr);
18
19       bridgeCon.start(); //build the bridge by starting it
20     } catch (TransportIOException ex) {
21       ex.printStackTrace();
22       //... errorhandling
23     }
24   }
25 }
```

Listing 8.3: The BridgeManager waits until a client attempts to connect to the provider. As soon as that happens, the BridgeManager starts a new thread, that becomes responsible of handling a new connection.

As the BridgeManager starts the Bridge thread in line 14 of listing 8.3, the `run()` method of the BridgeConnector sets up the layers of the bridge almost exactly as the client application does. The only difference is that the BluetoothLayer object is provided by the BridgeManager and already connected to a client.
After the bridge is set up the BridgeConnector connects it to a web service connector provided by an object implementing the interface `d619a.provider.service.interfaces.ServiceManager` [24] .Listing 8.4 shows

the details of connecting a bridge and a web service connector together. In line 18 of listing 8.4 the web service connector starts to do the processing. This method waits for incoming web service requests, processes them and sends an answer back to the client. `doProcessing()` is blocking an returns only if either the client disconnects from the provider, or the web service connector detects that the communication has finished. When it returns, the Bluetooth Layer is disconnected and the thread terminates. It might look as being unnecessary complicated with a ServiceManager, but this is a long sighted design construct that is motivated by two reasons. If the Bluetooth connection has to be reconnected, the ServiceManager is able to connect the reconnecting client with the same service connector that waits to send its response. Furthermore is it possible to use the ServiceManager for device-dependent service access and reliable connections to state full services.

```
1  //* get a service from the service-manager
2  //* that must be used together with
3  //* the connected client and set them together.
4  //* After that, let the Service handle the processing.
5  String clientId = bluetoothLayer.getClientAddress().
       toString();
6  theService = myServiceManager.getServiceFor(clientId);
7  theService.setBridge(serializationLayer);
8  serializationLayer.setCallbackReceiver(this);
9
10 try {
11   //when everything is in place, connect the Bluetooth
         layer!
12   //(connect just starts the listener thread.)
13   bluetoothLayer.connect();
14 } catch (IOException ex) {
15   ex.printStackTrace();
16 }
17
18   //and let the service do the processing.
19   theService.doProcessing();
20
21   bluetoothLayer.disconnect();
```

Listing 8.4: The Bridge object calls the ServiceManager to get a service for the connecting client. The service is then connected to the bridge and the service can do the processing. As soon as the service finishes, the BridgeManager disconnects the client.

The implementation of the BridgeManager is multi threaded to allow multiple clients to be connected to the provider at the same time. However, the Avetana Bluetooth library used to communicate with the Bluetooth device does not provide simultaneous connections to more than one client. This is a flaw in the Avetana library that is not mentioned in its sparse documentation. Therefore this problem has been observed only after the implemen-

tation of the provider. By making a call to `LocalDevice.getLocalDevice().getProperty("bluetooth.connected.devices.max");` the flaw is revealed.
There are possible solutions to this problem. Since the Avetana library is open source, it can be changed and extended by anyone interested. There is already one extension called JBluetooth, available at [45]. Changing the library for Bluetooth communication would also be a possibility. At the current state of the development no solution is implemented since the project goals only define a functional implementation and no solution that could directly be used in a production environment.

8.1.2 Client Bridge

Although the bridge in the client resembles the provider bridge, there are big differences. While the provider bridge is designed towards the capability of handling many connections at the same time, the client bridge adheres to completely different design criteria.

Since the client is frequently moving, we have to take into account a changing environment in terms of connectivity to providers. We discussed central points of mobility and communication in section 4.4. But we have paid little attention to the properties of discovering devices and scanning for services. Due to the design of the Bluetooth communication, device inquiry and service scanning takes time in the magnitude of tenths of seconds.

This is an unacceptable property for the client application if it implies that the user has to wait while the application scans for available providers. Therefore we designed the client bridge so that it meets the requirement of dynamically discovering providers along with the need for smoothless user interaction. A module called *SearchUnit* `d619a.client.bridge.bluetooth.searchunit` [24] is responsible for all scanning and discovery.

We have implemented the following features that allows us to minimize the time spent for device discovery and scanning for available services in these devices.

- Filtering on the *Classes of Devices* attribute as specified in the Bluetooth specification document [22] makes scanning for services on devices that are not interesting unnecessary.

- To minimize the number of rediscoveries of devices and their services, we maintain lists to keep track of the devices and services already discovered. A whitelist holds the devices we know are providers while a blacklist has all discovered devices listed that do not provide any

services of use. As soon as a known device is being rediscovered, rescanning of its services is prevented by use of the lists.

- By monitoring devices in the whitelist for the availability of their services we can provide the user with a preliminary list of services in the vicinity without the need for a complete device and service discovery every time the user requests a list.

These features are only optimizations to reduce the time consumption. But the user still has to wait for a complete list of available providers. The most elegant solution would be to permit concurrent communication with a providers web service and discovery of new providers and services. But since Bluetooth devices in mobile phones are restricted in their capabilities, they do not necessarily support such a dual-mode (`javax.bluetooth.LocalDevice.getProperty()` reveals these capabilities). Therefore we split the discovery of new providers in two modes. The *idle-mode* which actively scans the vicinity for new devices and the *interaction-mode* which only passively checks already discovered devices for their availability. The names *idle-mode* and *interaction-mode* refer to the interaction between the client application and a web service. While connected to a web service, the SearchUnit is in *interaction-mode*. While not connected, the SearchUnit is in *idle-mode*. The two modes and their tasks are depicted in figure 8.4 and in figure 8.5.

Idle-Mode

While the client is not connected to a provider, the SearchUnit, which handles the discovery of devices and services is set in *idle-mode*. The left side of figure **??** depicts how the idle-mode works. The first step, where we check the elements in the whitelist for availability is the same as in interaction-mode and will be explained separately.

In idle-mode, the main task of the SearchUnit is to discover all devices in the vicinity and find the devices which provide valid services for our framework. Initially, all discovered devices that are not in the blacklist nor in the whitelist are marked as "undecided". Each of those devices is then scanned for services. If a device does not provide any usable services, it is stored in the blacklist and never scanned again until the list is reset. If a device provides usable services, it is stored in the whitelist and a providerlist, which holds all available providers. As soon as all devices are discovered and scanned, the process starts over again.

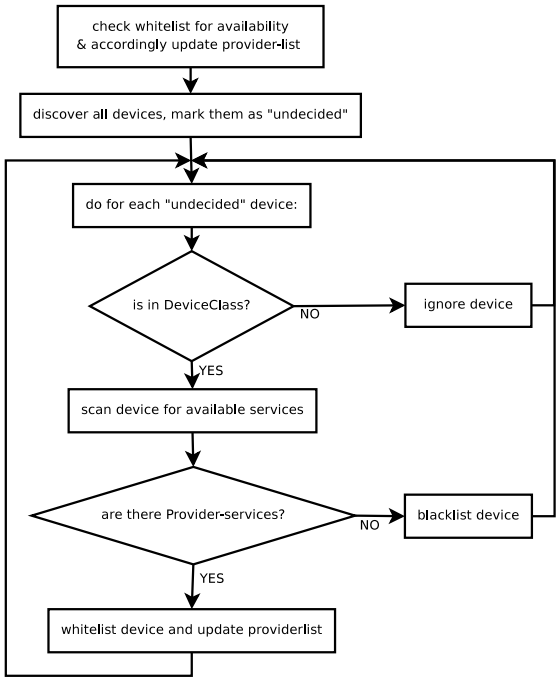


Figure 8.4: Flow chart of the idle-mode *in the SearchUnit of the client bridge.*

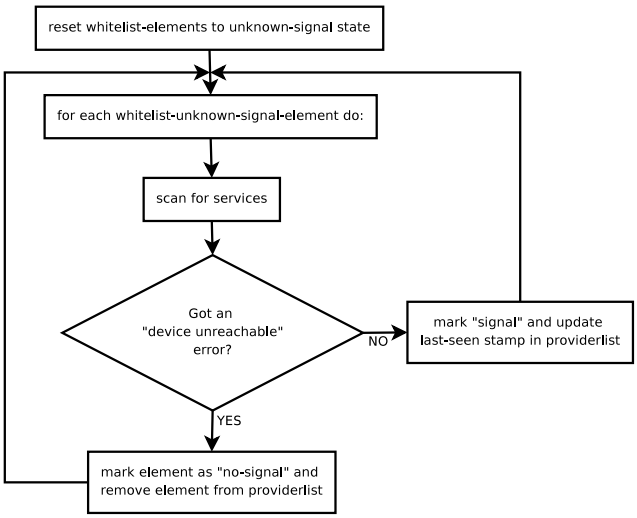


Figure 8.5: Flow chart of the interaction-mode *in the SearchUnit of the client bridge.*

Interaction-Mode

As soon as a connection is established to a provider, the interaction-mode as the right side of figure **??** depicts is started in the SearchUnit. In this

mode, the SearchUnit does not discover new devices but restricts itself on monitoring already known whitelisted devices. This is done by repeatedly scanning the known devices for its services. If a device is unreachable, the scanning fails and the signal state of the device gets the mark "no signal". This causes the providerlist to be altered by removing the device which is no longer available. It will not be removed from the whitelist, since it might be available soon again, while the client is moving.

Implementation

Service discovery on Bluetooth devices is a complex task. There are a number of details that must in place before the Bluetooth API can successfully be used to discover services. The following has to be specified when doing a service search with a call to *searchServices(...)* in the local *DiscoveryAgent*(*javax.bluetooth.DiscoveryAgent*):

UUIDs Every service is started with an array of UUIDs, which indicates the type(s) of services that you are looking for. We must generate a random UUID that nobody else is using, and use this to start provider services. This UUID must be used by clients in order to discover services that can be used in this framework.

Attributes Every Bluetooth service has a service record in a database or service registry in the local Bluetooth device. When a search is initialized it must be specified which attributes that are to be retrieved from the service record during the search. This works by passing an array of hexadecimal numbers, which each correspond to a particular attribute. These numbers are specified by the Bluetooth SIG in [27].

Remote Device Each service search is initiated on a remote device that have been discovered by a device search. A device search can be started by calling the method *startInquiry(...)* in the local *DiscoveryAgent*. The search unit we have implemented uses separate modules for doing device searches and service searches.

Discovery Listener When initiating searches for devices or services a discovery listener must be specified. This is a reference to the class that will handle call-backs when a service has been found or the search is finished or aborted. The class that is referenced must implement the interface *DiscoveryListener* located in *javax.bluetooth*.

When a service has been found the discovery listener will be notified with a callback, which takes the service record of the discovered service as a

parameter. This way attributes can be extracted from the service record, and compared with further search criteria, like for instance if the name of the service is correct. The search unit module can be found in appendix V in listing 2.

8.2 Bridge Layers

8.2.1 Integrity Layer

Since the RFCOMM protocol which we use for Bluetooth communication emulates a serial cable connection, there is one major shortcoming. The protocol guarantees the order of the transferred bytes and the integrity of individual bytes. But since the nature of a serial connection is a stream connection, there is no beginning or end of a message. Hence integrity of a single Soap message as we send it can not be guaranteed by the RFCOMM protocol. We thus developed the *integrity layer* in our bridge. This layer ensures that an entire message has been received and that the message is valid. We do this by adding a checksum to the message when sending a message, and check and remove the checksum upon receiving the message.

When sending a message, we calculate a 128 bit checksum of the data using the MD5 hashing algorithm. The MD5 algorithm is a fast and widely used hash function that generates a 128 bit message digest using trap-door or one-way mathematical functions. This means that it is impossible to reverse engineer the original message from a digest (excluding brute-force methods) [23]. This checksum is inserted into the first 16 bytes of the message.

Figure 8.6 shows a message that has been embedded with integrity information. As can be seen from the figure the message has also been embedded with length information in the 4 bytes following the checksum. The length information contains the length of the n bytes of data being sent. This information is needed when receiving messages in the integrity layer.

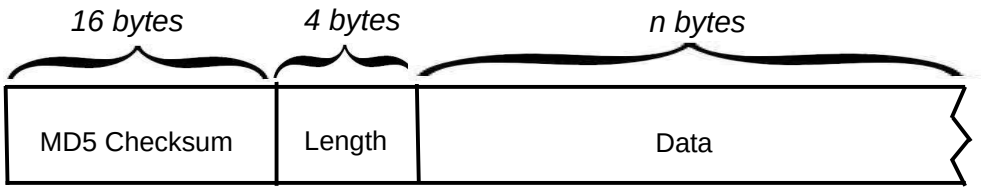


Figure 8.6: Message embedded with integrity information

When receiving a message the integrity layer must first perform the integrity check when the entire message has been received. This is handled by waiting

until the 4 bytes representing the length information has been received. Now having read this information we read in data until the number of bytes dictated by the length information has been received.

The length information can be of an arbitrary length dependent on the number of data bytes n, though because we have allocated 4 bytes for length information we limit the size of messages to of maximum 2^{32} bytes (Approx. the amount of data on a DVD) which should be more than adequate for any mobile device today.

When the message data has been received we calculate a checksum on the data and compare this to the checksum saved in the first 16 bytes in the message. If the two checksums match we will have verified the integrity of the message. Now we can remove the integrity information from the message and safely signal the layer above the to continue processing the message.

8.2.2 Security Layer

The security layer implements symmetric key encryption functionality, which is used for both encryption and signing of messages. The layer operates on messages and each of the messages are represented in a byte array.

There are two modes of operation: *Encryption* and *signing*. The mode of operation depends on whether encryption is turned on or off by the surrounding environment. If encryption is turned off, the mode of operation is signing. The mode is set up when the security layer is initialized along with the encryption key, which is a 128 bit randomly generated, large integer, which is obtained by the environment from the mediator. The security layer is implemented in **Security.java** `d619a.common.bridge.security` [24] .

Encryption

In this mode all traffic will be encrypted. Figure 8.7 shows how a message is encrypted and decrypted. First an MD5 digest is generated from the data message and appended. The digest is needed later to ensure that decryption was successful. Next the entire data message is encrypted (including the digest). When the message is received on the other side of the bridge the entire message is decrypted. After decryption the MD5 digest that was appended before encryption is stripped from the message. A new MD5 digest is generated on the remaining data, and the new digest is compared to the appended digest. If the two digests match we can be sure that the message was decrypted successfully with a correct encryption key. If they do not match the key was incorrect or the data was altered, and an exception is raised.

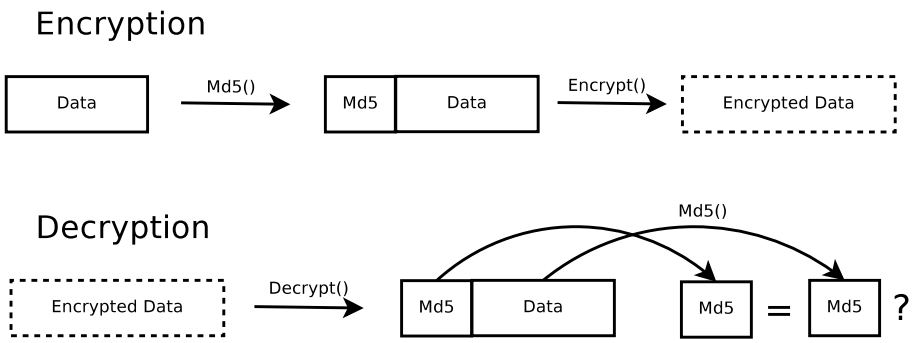


Figure 8.7: Encryption and decryption of a message. Encrypted data is marked with dashed box lines.

Because the message can only be read if it is decrypted with the correct key both confidentiality and authentication is provided, satisfying our Security and Credibility system requirements. If the message was altered after encryption, the decryption would fail, therefore integrity is also provided.

Signing

In this mode of operation all messages will be signed with an encrypted digest of the message. Figure 8.8 shows how a message is signed and authenticated. The difference between signing and encryption is that only the attached MD5 digest is encrypted. To authenticate a message the attached MD5 digest is stripped from the message and decrypted, and a new digest is generated on the remaining data. A message is successfully authenticated if the decrypted MD5 digest matches the new digest. If they do not match the key was incorrect or the data was altered, and an exception is raised.

The message can only be authenticated with the correct key and if the data is altered the digests will not match. Therefore authentication and integrity is provided, satisfying our Credibility requirement.

In this process it is important that the encryption key cannot be reverse engineered from looking at the data that was encrypted (since an eavesdropper could calculate the MD5 sum and compare it with the encrypted MD5 sum). By using large 128 bit encryption keys this risk is minimized.

Encryption engine

The encryption engine that is used in the security layer is Bouncy Castle's ARC4 implementation (`org.bouncycastle.crypto.engines.RC4Engine`

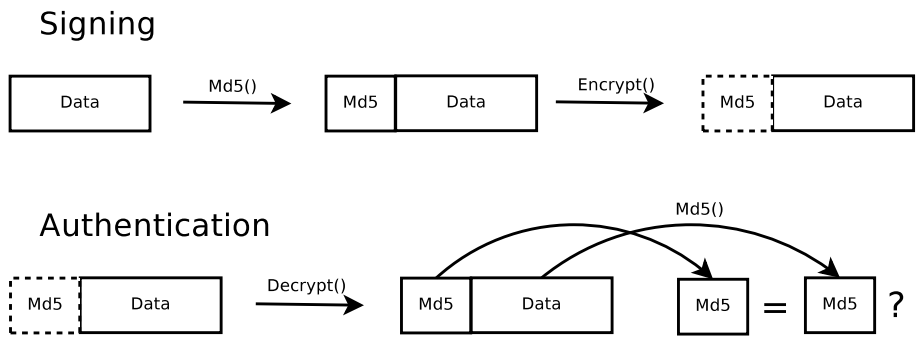


Figure 8.8: Signing and authentication of a message. Encrypted data is marked with dashed box lines.

[24] [41]). This is a lightweight, open source implementation of the widely used, stream cipher ARC4 algorithm.

Listing 8.5 shows how the encryption engine is initialized in the security layer. The second parameter to the constructor is the 128 bit encryption key. This key is written to a byte array in line 4, and passed to the encryption engine as an initialization parameter in line 5. The encryptionMode parameter denotes whether the security layer should operate in encryption mode or signing mode.

```
1 public SecurityLayer(int encryptionMode, BigInteger
      encryptionKey) {
2         this.encryptionMode = encryptionMode;
3         this.encryptionEngine = new RC4Engine();
4         byte[] key = BigIntegers.asUnsignedByteArray(
              encryptionKey);
5         this.encryptionEngine.init(true, new KeyParameter(
              key));
6     }
```

Listing 8.5: Initialization of encryption engine.

After initialization of the engine data can be encrypted by calling the method *processBytes()*. Listing 8.6 shows how this works. Lines 1 through 3 creates a new checksum on the data and adds it to a new byte array together with the data. Line 4 creates a buffer for the encrypted data. In line 5 processBytes() is called with the following parameters: (input data, offset into input data, length of input data, output buffer, offset into output buffer). The data in the input buffer is encrypted with the ARC4 algorithm and written to the output buffer.

```
1     MD5 md5 = new MD5(data);
2     byte[] checksum = md5.doFinal();
3     byte[] dataToBeEncrypted = addChecksum(checksum, data);
4     modifiedData = new byte[dataToBeEncrypted.length];
```

```
5       encryptionEngine.processBytes(dataToBeEncrypted, 0,
            dataToBeEncrypted.length, modifiedData, 0);
```

Listing 8.6: Encryption of data using processBytes().

Chapter 9

Client

In this chapter we present the design and implementation of our client application running on the mobile device of the user. We first describe the overall design of the the client GUI forms and the relationship between these forms. Next we proceed to disuss issues concerned with the use of the web service applications in a service provider. Finally we discuss implementation details and issues experienced while developing the client application.

9.1 Design

The purpose of the client application is to allow a user to interact with a service provider. We have implemented a number of user interfaces which allow a user to do the following.

ServiceBrowserList Discover a nearby service provider and choose a Bluetooth service.

WebServiceBrowserList Choose a web service application exposed via the Bluetooth service.

OperationBrowserForm Choose an operation to invoke in the selected web service application.

Primitive/DynamicResultsForm Show the results of the operation invocation in an appropriate results window.

In order for us to separate the user interaction forms into seperate components used in the client application Midlet, a design based on what we call

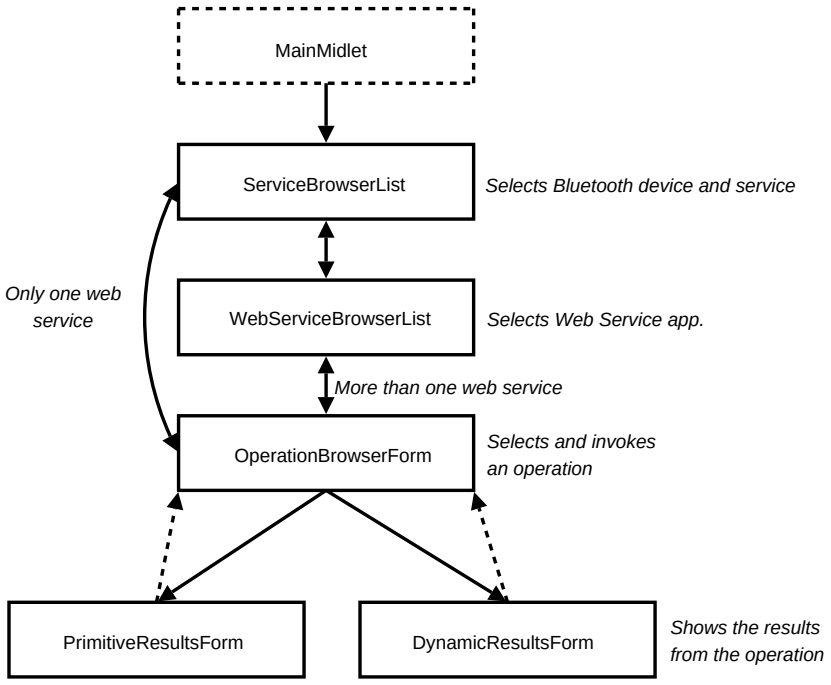


Figure 9.1: Navigation-diagram showing the dialog flow in our user interaction. Note that the dashed box represents a non-visible entity.

parent-child interfaces have been created. For instance if a service browser form opens a web service browser form, the service browser would become the parent to the web service browser, which would become the child of the service browser. This relationship is required because we want a user to always be able to return to a previous form by pressing a "Back" button. The relationship between our forms is shown in figure 9.1. In this example the parent-child relationships is represented top-down, meaning that the upper forms are parents to the lower forms. This way of buiding the client GUI is superior to the standard design proposed by most GUI modeling tools for Midlet applications, where an entire applications UI flow is modeled in a single Midlet class, which becomes difficult to understand and maintain.

In this figure we see that if a user selects a Bluetooth service from a provider and this service only offers a single web service application, we skip the web service browser and directly show the user the operation browser. This decision was made because we want to minimize the required interaction from a user in order for her to consume a provider application.

In order for us to communicate with a service provider and find out what applications it has to offer, we communicate with a predefined provider web service, which we discuss in detail in chapter 10. The interaction steps taking place between the client and this provider web service is shown in figure 9.2

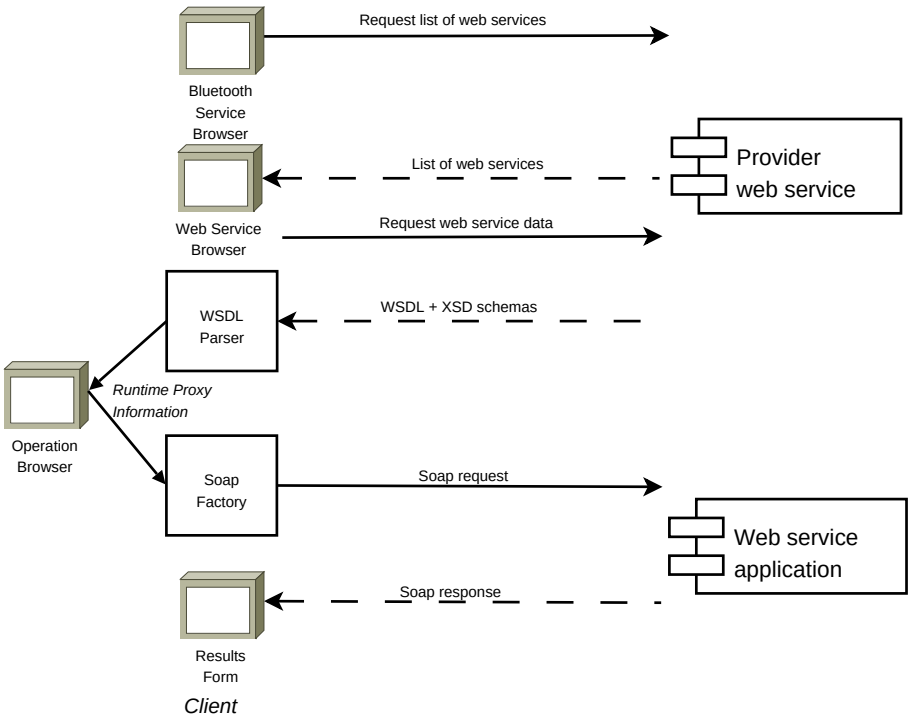


Figure 9.2: Example interaction between the client and the provider. The figure shows the involvelment of the WSDLParser and SoapFactory modules.

Here we see that the client first requests a list of the available web service applications placed in the service provider. This list is shown in the web service browser. Now the user selects a web service from this list. Based on this selection the framework then requests the WSDL file and associated XSD schemas of this web service. Next this information is passed to the WSDLParser component (implemented in `d619a.client.wsdl` [24]), which is a custom WSDL parser we have developed to parse out information to use *runtime proxy information*, mentioned in section 6.5. This runtime proyx information is stored in a WSDLContainer object, implemented in `d619a.common.wsdl.datatypes` [24] . We use this information to show a list of operations to the user in our operation browser. If any of these operations require input from the user this is also represented in the operation browser. Now from the data supplied by the user and the runtime proxy information a Soap request message is generated. We generate this by using the SoapFactory module, implemented in `d619a.common.soap` [24] . This module is used to generate kSOAP Soap messages. Now having generated an appropriate Soap message we send this to the chosen web service in the provider, which in turn responds with a Soap response that we show in our client GUI. We have included an illustration of the actual client application GUI flow in

table 9.1.

We discussed the restriction of not having dynamic class loading in the CLDC, in section 6.5. Our conclusion from this analysis was to use runtime proxy information. This conclusion was adequate for our dynamic communication with web services. However another consequence of not having dynamic class loading is that the web service applications deployed in a service provider will not be able to return arbitrary data types. Web service operations may only return data in a format that can be deserialized by standard kSOAP routines, in other words the web service operations may as a starting point only return primitive datatypes. However as we also want to be able to represent complex data such as images we have incorporated a mechanism that allows an application to also return a number of predefined datatypes, that we supply. To implement this mechanism we were required to be able to return serialized complex data types by use of primitives. We use a naming convention that allows a client to distinguish between methods returning specific complex types and primitive types. Our mechanism is based on embedding type information into the name of a web service operation.

Now when we invoke an operation we will be able to handle a complex response by examining the name of the operation we are invoking, and using the ClassLoader we can load an appropriate class needed to handle the return data. For example an operation, called *showImage* returning an image in our framework would for instance have the following complete name *d619a_client_gui_datatypes_Image___showImage*. Now from this name we can parse out the path of the Custom data type to handle the return data, load it, and use it to show the image. For test this design have included this Image class in our framework, see `d619a.client.gui.datatypes` [24] . This class describes how to generate a form that can show an image response. Note that to send binary data in web services we use a Base64 encoder to send this in a String representation. We use the kobjects Base64 encoder to accomplish this. As a convention we assume that an operation returns a primitive type if no type information has been embedded into its name.

9.2 Implementation

One of the restrictions of using J2ME is that you are only allowed to change the currently displayed form from a Midlet. Meaning that we cannot change to another form from inside one of our form classes. One solution to this problem could be to implement each form class in a seperate Midlet. Then instead of changing a form we would be required to simply start a new Mi-

(a)

(b)

(c)

(d)

Table 9.1: screenshots of the running client application

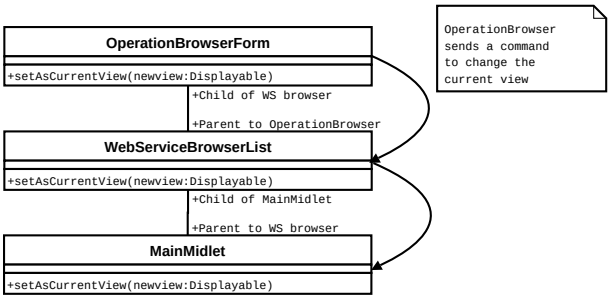


Figure 9.3: Example of method call propagation to reach the top-parent or MainMidlet, calling setAsCurrentView to change the currently displayed form.

dlet representing the form. However using J2ME you are not allowed to start Midlets from inside a Midlet, thus this solution is not possible. We have used our parent-child relationship to solve this problem. The method *setAsCurrentView* has been implemented to change the currently displayed form. When making a call to this method it uses the parent-child relationship to propagate the method call through all the defined parents until the top-parent or the MainMidlet class is reached and the currently displayed form is changed. Figure 9.3 illustrates the propagation of a call to this method. Also listings 9.1 and 9.2 shows the implementation details handling the propagation of calls to *setAsCurrentView* and the code in the MainMidlet class where the actual form change occurs.

```
1 //Propagates view change to parent
2 public void setAsCurrentView(Displayable newview) {
3   parent.setAsCurrentView(newview);
4 }
```

Listing 9.1: A form class propagating a call to setAsCurrentView to its parent, implemented in the form classes in the Client module `d619a.client.gui` *[24]*

```
1 //Changes the actual view
2 public void setAsCurrentView(Displayable newview) {
3   getDisplay().setCurrent(newview);
4 }
```

Listing 9.2: The MainMidlet class receiving a propagated call to setAsCurrentView, implemented in the Client module in `d619a.client.gui.MainMidlet` *[24]*

To implement our custom classes we have implemented a common interface that must be implemented by all custom data types. We call this *CustomDatatype* placed in `d619a.client.gui.interfaces` [24] . This use of this interface makes it possible for us to use our datatypes in a generic manner by calling methods on the datatype solely through this interface. This means that it becomes possible to create a GUI form to show the interaction results of a custom data type in a generic manner, this form is implemented in `d619a.client.gui.DynamicResultsForm` [24] .

Chapter 10

Provider

In this chapter we discuss the design and implementation of the software developed in the provider and aspects of the client-provider communication. The provider software has been implemented as a web service called the provider web service, not to be confused with the web service applications deployed in the provider. The provider web service is used to inform the client of the web services that are available in the provider and how these can be used.

The provider web service also provides information used to generate the user interfaces in the client application by mapping WSDL/XML names to user-friendly textual descriptions that can be shown in the client application's GUI.

10.1 Design

When a client wants to invoke an operation in a web service application deployed at the provider, the client first needs to download information about how to use that service. To retrieve this information the client first needs to contact the provider web service which is responsible for delivering WSDL information about the web service applications deployed in the provider.

The provider needs to save information about the mapping of service descriptions to their WSDL locations, and the mapping of operation names to their textual descriptions. We have implemented this functionality in the following two configuration files, which by default are placed at the root of the application server in the provider:

- *service_ mapping.conf* Maps service descriptions to a WSDL URL location. Note that the service description is the data that will be shown in the web service browser GUI (explained in chapter 9). We show an example of a service mapping in table 10.1.

Service Description	WSDL URL
Hello World Service	http://localhost:8080/HelloWS /HelloWS?wsdl

Table 10.1: Example of a mapping. A Hello World service is mapped to the location of its WSDL file

- *operation_ mapping.conf* Maps an operation name which refers to the name defined in the WSDL file associated with the service, to an operation description, which is a string describing the operation in user-friendly terms. An example of an operation mapping is shown in table 10.2.

Service Description	Operation name	Operation description
Hello World Service	sayHello	Make service say Hello to the world

Table 10.2: Example of a service mapping mapping the operation names in the Hello World service WSDL to user-friendly descriptions

There are other alternatives for storing this meta-data about the web services and their operations, however as we dicovered in [17] a solution to this problem is a huge topic in itself, and not the focus of this project. Section 13.2 presents a discussion of other alternatives for representing this meta-data.

10.1.1 Client-Provider Communication

Figure 10.1 illustrates the communication taking place between the client and the provider during a communication session. When the client has chosen to browse a Bluetooth service at a provider a list of web services is first downloaded to the client user interface. This constitutes the first of the interaction steps in which the client fetches a list of the service descriptions defined in the service mapping. In this case this list would contain the service descriptions, contained in the service mapping, for WS App 1 and 2.

Now a service description is chosen by user input in the client application and this service description is used as a parameter to locate and download the WSDL file and associated XSD schemas of the selected web service application. To spare the overhead of having to send an extra SOAP message we also piggyback the operation mapping data to this message.

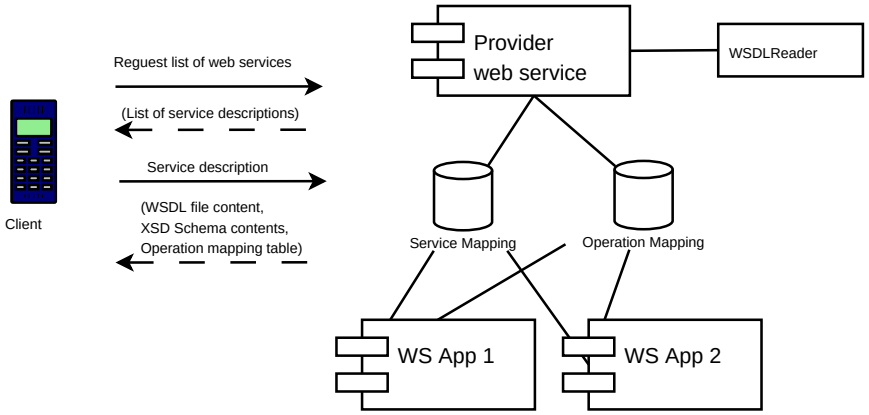


Figure 10.1: The communication taking place between the client and the provider. In this case the provider has two deployed web service applications.

When the provider web service receives a request for a WSDL file it first locates the location URL of the WSDL file in the service mapping. It then downloads the WSDL file using the WSDLReader component, implemented in the Provider web service in `wsdl.WSDLReader.java` [24] . Now the WSDLReader analyzes the WSDL file to check for any associated XSD schemas and downloads these as well if any exists. The schema data is appended to the WSDL data as a single string, which will later be parsed in the client application, as explained in chapter 9.

Next the relevant operation mapping data associated with the requested WSDL file is retrieved from the operation mapping file. This is added to the response message and sent back to the client as an XML complex datatype representing the WSDL and operation mapping response data.

We plan to extend our client application in the fall semester of 2007 with an embedded browser, which makes it possible for us to use HTML code to format our response messages. This opens the possibility of making more advanced operation mappings in our provider, where an operation name could be mapped to HTML elements such as images, buttons, etc. instead of merely mapping these to textual descriptions.

10.2 Implementation

Implementing web service operations to return objects or in XML language, complex data types, requires us to make sure that the data is sent and received properly by the client. In this section we describe the implementation details regarding the sending and receiving of complex data types using kSOAP.

The two operations *getServices* and *getServiceData* implemented in the provider web service in `provider.ProviderInformationWS.java` [24] each returns a complex data type. The alternative to this would be to only use primitive datatypes. This would require us to send several SOAP messages between the client and provider, which could yield significant data overhead.

We have thus chosen to use complex data types. Consequently we need to make sure that these are properly serialized and sent to the client, on the provider side. And on the client side we need to make sure that these responses are properly deserialized into their respective Java class representations. kSOAP [34] provides the KvmSerializable interface, included in Common Modules `org.ksoap2.serialization.KvmSerializable.java` [24] . kSOAP makes it possible for us to serialize custom Java objects into SOAP representations by implementing this interface.

To implement this we are required to first model our response messages into two Java classes named *ServicesResponse.java* and *ServiceDataResponse.java*, see `provider.responses` [24] for implementation details of these. These classes have to implement the KvmSerializable interface and specify the properties of the classes, properties here being the member values in the classes. Furthermore the types of these properties must be specified. kSOAP supports by default a number of types that can be serialized, which consist of a number of primitives (such as Strings and integers) and the Vector class specified in Java. If a response message consists of properties that do not map to these types kSOAP has an Object type that can be used. However using this type requires the developer to manually write more advanced deserialization routines on the receiving end.

In our ServiceDataResponse operation mapping information is sent as a Hashtable, where the key (operation name) maps to a value (operation description). This is an example of a case where more advanced deserialization is needed on the receiving end. The code snippet in listing 10.1 shows the deserialization routine that rebuilds the Hashtable after it has been received by the client.

```
1 //Create an empty Hashtable that will contain the
      deserialized data
2 Hashtable operationMapping = new Hashtable();
3 //Deserialize operation mapping Hashtable
4 /* Get complex Object property named operationMapping,
      this contains a list of key-value pairs */
5 SoapObject operationMappingSoapObject = (SoapObject)
      response.getProperty("operationMapping");
6 //Get they key-value pair values from the operationMapping
       SoapObject
7 for (int i=0; i<operationMappingSoapObject.
      getPropertyCount(); i++) {
8         SoapObject keyValueSoapObject = (SoapObject)
```

```
                operationMappingSoapObject.getProperty(i);
9           String key = keyValueSoapObject.getProperty(0).
                toString();
10          String value = keyValueSoapObject.getProperty(1).
                toString();
11          //Add values to operationMapping Hashtable
12          operationMapping.put(key, value);
13      }
```

Listing 10.1: Deserialization of a Hashtable, implemented in the Client module in `d619a.stub.ProviderInformationStub.java` [24]

The kSOAP framework saves complex objects in SoapObjects and known types in SoapPrimitives. Deserializing a complex object is a matter of recursively decomposing each SoapObject into its SoapPrimitives and fetching the values from these. Figure 10.2 shows a tree representation of the Hashtable deserialized in listing 10.1

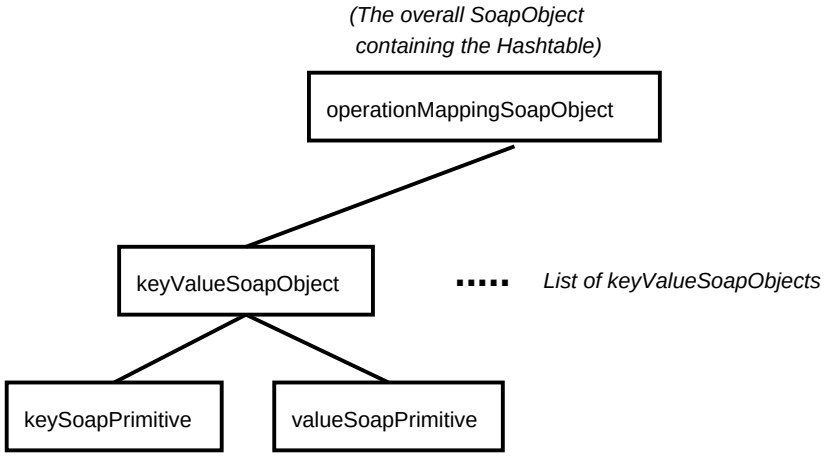


Figure 10.2: kSOAP representation of a Hashtable shows the relationship between SoapObjects and SoapPrimitives

Chapter 11

Mediator

This chapter presents the design and implementation of the mediator, which is labeled as the Bluetooth Authentication Mediator in figure 3.1 in the problem statement. The mediator is responsible for negotiating connections between clients and providers, and in this process to facilitate authentication, encryption keys and compatibility. This enitity was implemented to satisfy the *Credibility* and *Security* requirements.

Figure 11.1 illustrates how the negotiation takes place. To begin with the client device initiates a Bluetooth connection to a service provider **(A)**. After this both the client and provider sends a request to the mediator with a set of flags signalling requirements for the communication **(B)**, like for instance that encryption should be turned on. When both the client and provider has received a response from the mediator **(C)**, the parties can continue communicating through the Bluetooth channel **(D)**, unless the mediator was unable to facilitate a connection for instance if the client software is out of date or if the provider was not authenticated.

11.1 Design

The mediator is designed as a web service. This web service has access to a database that contains records of all service providers. It is impossible to connect to service providers that are not in this database using our system. For each service provider the database has a record with the following attributes: Service name, Bluetooth address, key timeout, WSDL version, admin password. The key timeout attribute determines how long time a key is valid when communicating through the Bluetooth channel. When the timeout expires a new key must be negotiated through the mediator. The

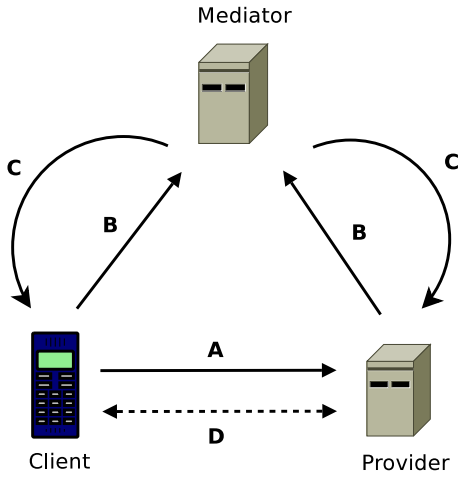


Figure 11.1: Connection negotiation.

WSDL version attribute is used to inform clients to update their stubs.

In order to be able to establish connections between clients and providers the mediator needs a mutually exclusive shared resource that is accessible from concurrent calls to the mediator web service. When a web service is invoked the hosting servlet container of the web service starts a new process. The method call to the web service will be executed in this new process. This means that invocations of the same web service cannot directly share Java objects. To solve this problem we use a DBMS to manage the shared resource. Synchronization is handled by letting clients write a record in a table that holds current connection attempts, and then by letting providers look for this record and set a flag in the record upon discovery to inform the client that there is a match. In this process an encryption key is generated and written to the same record.

In order for the mediator application to scale we need a reliable DBMS that can handle multiple connections. We use the free MySQL [40] Community Edition, which it is one of the most widely used free DBMSs.

Figure 11.2 shows the design of the mediator component and what kind of information that will be passed to and from the web service. The client and provider will pass the same type of information to the mediator, and upon receiving two matching requests the mediator generates an encryption key, and sends it back to the client and provider.

In addition to the encryption key there are a number of other output parameters that will be returned from the mediator. Both the input message and output message has an encryption flag. This flag is used to specify the encryption needs of the client and provider. It may not always be necessary

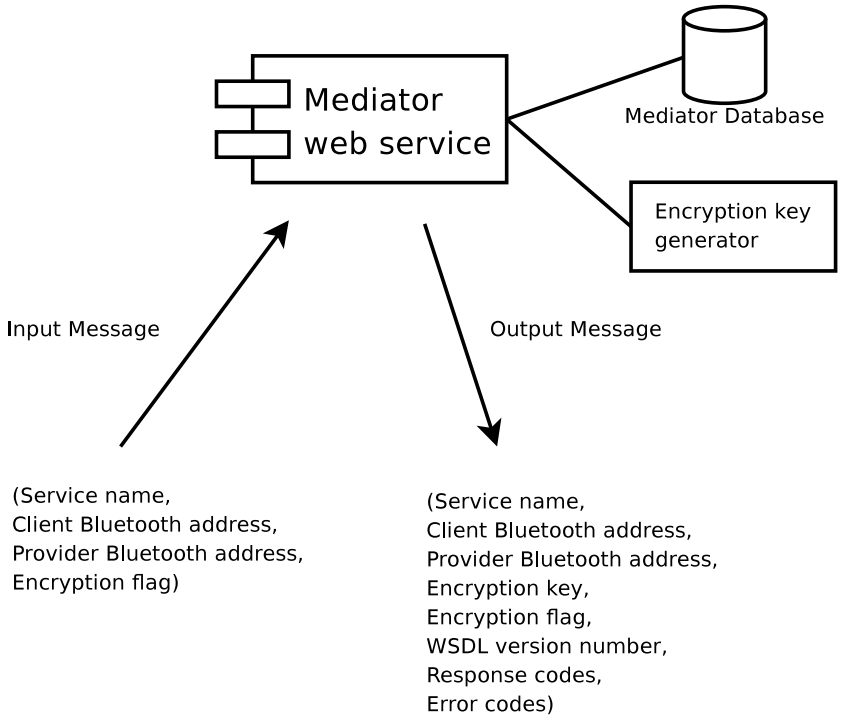


Figure 11.2: Mediator component.

to encrypt the traffic between the client and provider if the content that is being sent is not sensitive, like for instance if it is a museum service providing information about sights. Also it may sometimes be imperative to encrypt all traffic if for instance the service provider is a bank. Based on the encryption flags encryption will be set to on or off. If encryption is set to off the traffic between the client and provider will still be authenticated by using the encryption key to sign messages (see chapter 5). Table 11.1 shows the mediator encryption policy. The policy is used by the mediator to determine whether a connection should be encrypted or not. The encryption flag in the output message from the mediator informs the clients and providers of whether encryption is on or off, or signals an error if the encryption needs were incompatible (in this case the client must change the encryption mode in order to communicate with the provider).

Encryption flag	**Optional**	**Mandatory**	**None**
Optional	off	on	off
Mandatory	on	on	error
None	off	error	off

Table 11.1: Mediator encryption policy.

The WSDL version number output parameter is used to notify clients of the

version of the stub that is needed to communicate properly with the service provider. This version number is used to notify clients to update their stubs, and thus their client application. The response codes are used to signal if the connection attempt was successful, and are defined in the interface **Status.java** `d619a.mediator` [24] . The error codes contain descriptions of potential errors.

Encryption key generator

The encryption generator is used by the mediator to retrieve keys for the negotiated connections. It is implemented in **Encryption.java** `d619a.mediator.encryption` [24] . The keys that are generated are 128 bit random integers.

11.2 Implementation

Connection requests in the mediator must be synchronized properly in the mediator to avoid deadlock and starvation situations. Connections between clients and providers are established by using a table in the database as a shared resource. If client and provider requests were to go through the same web service operation, the database would have to be locked while the operation call either creates or matches a record in the database. Otherwise deadlock situation are possible. Since the DBMS has to be able to serve many[1] requests at the same time it is not desirable to lock the database. Therefore client requests are made by calling the web service operation *SetupClientConnection()*, and provider requests are made by calling the web method *SetupProviderConnection()*. Figure 11.3 shows a flow chart of how the synchronization between the two web service operation calls is handled. The implementation of the mediator web service can be seen in appendix V in listing 1. The source file is located in `d619a.mediator` [24] .

[1] This number depends on how many service providers that are registered in the mediator, and how many users they have on average.

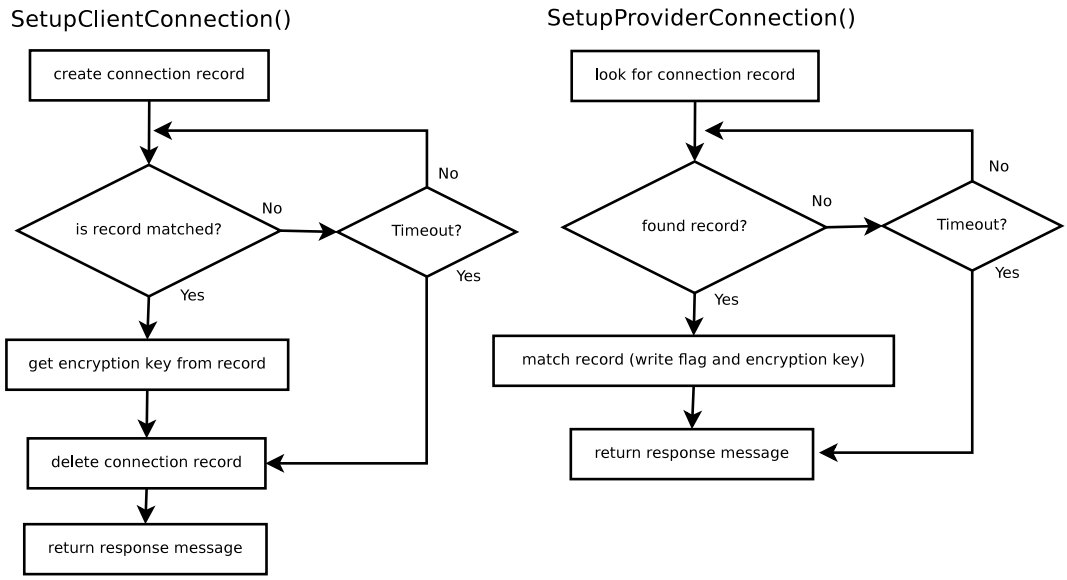


Figure 11.3: Flow chart of connection requests.

Part IV

Conclusion

Chapter 12

Conclusion

After having designed and implemented our framework we reflect on the product and evaluate it with respect to the system requirements and philosophies stated in chapter 3.

We start by evaluating our framework's adherence to the design requirements, namely the extensibility, reliability and efficiency requirements. We first evaluate the system as a whole and then discuss the individual components of the system. Secondly we evaluate whether our system adheres to the usability requirements, namely the credibility, security, selectability and user experience requirements. Finally we discuss our system philosophies and their influence in the design and implementation of our system.

12.1 Evaluation

Design Requirements

Our system is generally well documented in that every class and public method in the framework has been documented using SUN's Javadoc tool [14], see [13] for our documentation. The decision to use Javadoc was made from the beginning of the project. Since work on our framework will be continued on future semesters we made this decision in order for us to be able to quickly understand and change implementation details in future versions of our system. Our documentation style adheres to the documentation requirement in our extensibility system requirement.

Our framework has been developed as a distributed system aimed at both mobile devices and stationary platforms. This necessitates the use of two

Java editions, namely the J2ME and J2SE platforms. Based on the fact that J2ME was designed to be a subset of J2SE, the platforms are similar in many respects. This makes it possible to write code that is compatible for both platforms. We have exploited this capability in creating our Common Modules library where platform independent code has been placed. The advantage of having this library is that maintaining code becomes easier since we are only required to change it in one place. For instance we have exploited this library in our bridge component where some of the layers are platform independent and thus placed in only one place. This method of having code in one place, where possible, adheres to our extensibility system requirement in making our code maintainable.

As discussed in chapter 8 our Bluetooth Communication Bridge has been designed in a modular fashion by using a layered architecture. Using a layered architecture gives an extensible design where individual layers can be updated/inserted without affecting the other layers in the architecture. [42] The use of common interfaces in our layers makes it possible for us to communicate with the layers in a generic fashion, which we exploit when using our callback mechanisms.

To satisfy the reliability requirement in our bridge component we have implemented several mechanisms to handle potential errors in the bridge communication. Our use of custom exceptions makes it possible for us to further distinguish between errors and handle these. These mechanisms aid in making our bridge more reliable in the sense that unexpected behavior in our bridge in minimized by either handling errors where possible or informing the user that an error has occurred by propagating error messages to the client application. Since errors in our system are likely to occur during communication sessions we have prioritized reliability in the bridge component, this also helps in making the client and provider reliable in that they are both reliant on the use of the bridge.

Reflecting on our bridge component with respect to efficiency we have identified an issue that is a result of our use of the JSR 82 specification, used for Bluetooth communication. In our current implementation it is only possible to receive a single byte at a time in the Bluetooth layer. This impedes on performance because every time a byte is received in the Bluetooth layer it notifies the integrity layer, which then runs a number of checks. Therefore data transmission rate of the Bluetooth communication bridge is not yet good enough for a production stable solution. In future versions of our bridge we will need to improve on the problems faced from using JSR 82, for instance the use of buffers in the Bluetooth layer could help in reducing the number of callbacks being made to the integrity layer. Another solution that could improve on our Bluetooth throughput would be to use the L2CAP protocol instead of RFCOMM, this solution would minimize the amount of

header information sent with the byte data. A third solution would be to incorporate a compression layer into our bridge which would help in limiting the number of bytes being sent though the bridge. We discuss this scenario in more detail in chapter 13.1.

The use of callbacks in our bridge makes an efficient design where the layers in our bridge respond to callbacks instead of having to continuously poll for data updates between each layer.

The client application has been designed with extensibility as an important requirement. This decision was made because the future work presented in chapter 13.1 implies that a number of changes in the client application will be made in the future. Because of this we designed our Midlet user interfaces in separate classes to ease updates and insertions of forms into our application. This design is superior to the standard design proposed by most GUI modeling tools for Midlet applications where an entire applications UI flow is modeled in a single Midlet class. Another aspect dealing with extensibility is our use of class loading to represent custom data types. This feature creates a loose coupling between our client application and the custom data types supported by our framework making it easy to introduce new custom data types.

Our provider component has been designed with efficiency in mind. In this respect efficiency refers to communication-level efficiency. We optimized our Provider web service to send as few SOAP messages as possible. This optimization was implemented by piggybacking data to SOAP messages. This communication-level efficiency also affects the time spent processing byte data in the bridge. Because by reducing the number of SOAP messages to be sent over the bridge we also reduce the SOAP overhead introduced in each message.

The mediator component has been designed with reliability in mind. Our mediator uses a DBMS which allows it to scale in the number of connections it can handle. This is an important point since our mediator can potentially become a bottleneck in the future use of our system if it gains success. The ability to scale in our belief partly meets the reliability system requirement. However an issue we have not addressed in this project is the fact that our mediator is in effect a single-point-of-failure in our system. We have not addressed this issue yet but we will be required to investigate methods of dealing with this problem if our framework gains success in the future. One solution could be replicate our mediator component in the future yet referring to the goal of our project, in section 3.5 we have not focused on this perspective.

Usability Requirements

Since the implementation of the system uses certificates and secure connections to authenticate each other through a mediating entity, the requirement of credibility is partly satisfied. However being able to confirm the identities of the providers using our framework does not imply that the content of the providers is secure. Thus we need to develop a method for how the content published by providers will be checked. Content verification could be achieved in a number of ways including manually checking the validity of a providers content and incorporating mechanisms for automatically checking for malicious content in providers. We have chosen not to focus on this aspect as this in itself is a large research area.

Encryption functionality has been implemented in the security layer in the Bluetooth Communication Bridge, and encryption keys are negotiated through the mediator using secure connections. Our current implementation uses the ARC4 encryption algorithm, which is not recommended for systems requiring high-grade/military level security. Yet reflecting on the usage examples, we presented in chapter 1, our choice of encryption algorithm should be sufficient to satisfy the security requirement. This is also based on the fact that the ARC4 encryption algorithm is sufficiently secure to be used in Internet payment systems, see section 5.1 for more information.

When a connection between a client and service provider has been set up, the user interacts with the client application through a service browser and an operation browser as explained in chapter 9. This enables users to manually select the services they want to interact with, and which operations to invoke in them. Therefore the requirement of selectability is fulfilled implicitly. This requirement in turn satisfies the *Choice* and *Control* requirements stated by the marketing analysis we presented in section 2.1.1. In a future version of our framework we expect to improve the capabilities of creating user interaction by implementing an HTML browser into our application. This idea is discussed further in section 13.3.

One aspect of the user experience requirement concerns the users of our client application that interfaces with service providers. Searching for service providers takes on average 18-25 seconds. These delays have been minimized because of our optimizations in the implementation of both the search unit, see 8.1.2, and the Bluetooth communication. Our use of whitelists and blacklists in the search unit minimizes the time spent performing service discoveries.

Another aspect regarding the user experience requirement is aimed at the second group of users for our framework, namely the developers that set up a service provider. In this respect we have found two advantages of using

our software compared to using the push-based systems we mentioned in section 2.1.2. First our framework is cheap because it is implemented in software and thus we do not require the developer to purchase any hardware products besides a Bluetooth device. Secondly our software is simple to install and use. The developer is only required to have an understanding of web services and how to use these on an application server. The only requirement stated by our framework is that the developer has to manually map her web service applications in our mapping files, and that the developer installs the provider web service.

System Philosophies

Our two system philosophies of openness and compatibility is in our opinion intrinsically related. If using an open standard it does not imply that this standard is actually widely adopted. This is the reason why we have also included the compatibility philosophy. We chose to use the web services framework based on the fact that the specifications describing the web services framework has been developed both openly and in a vendor-neutral platform based on XML.

Using SOAP however contradicts our system requirement of efficiency so choosing this format was a trade-off decision preferring openness and compatibility over efficiency. Yet the potential use of XML accelerators, discussed in chapter 13.1 might help in reducing the overhead of using SOAP.

Our choice of using Java to implement our system was made to satisfy the compatibility philosophy. Unfortunately no programming language exists that applies to 100% of the mobile devices shipped today. Thus in lack of such a language we chose the one that is most compatible on todays mobile devices. As mentioned in section 3.4 Java is currently supported on 70% of the mobile devices being shipped today.

12.2 Conclusion

In section 3.5 we defined some areas of investigation for our project. In the analysis part of this report we covered those areas thoroughly. The knowledge foundation gathered from this analysis has provided us with the necessary insight and knowledge needed to implement an extended proof of concept of the framework proposed in the problem statement.

The design of the individual components in DynaBlu reflect the insights gained from the analysis, and many implementation issues have been resolved during the project by revising and debating design details.

The design requirements are important to ensure that future work on DynaBlu is successful. Furthermore as the requirements are meant to be long-term requirements we apply them in order to be able to identify future improvements that is needed for DynaBlu to enter a production-stable state. Our bridge is an example of a component that fulfills the extensibility and reliability requirements. Future work could improve the efficiency of the bridge in order to raise it to a production-stable level.

The usability requirements of selectability, credibility and security are satisfied in the current implementation of DynaBlu, which is a good starting point for creating a system superior to existing similar systems. For instance satisfying the credibility and security requirements opens the door for new application domains, such as banking services.

Chapter 13

Perspectives

It is an open question whether DynaBlu will be successful. In this context success implies that the framework is used as a platform for implementing Bluetooth systems at a number of different locations. The success of DynaBlu depends on the attention and popularity that this project gains in the local environment, the user acceptance of DynaBlu and the developer acceptance of DynaBlu.

Popularity
If this project receives a lot of attention at Aalborg University and in the local media the chances for success will increase significantly. Media attention would for example lighten the burden of having to spend money and time on marketing efforts.

User Acceptance
In order for a user to be able to interact with a Bluetooth system that is based on DynaBlu she will need to obtain the client software that will run on her phone, which could for instance be downloaded from a web page. This is a drawback with regards to spreading the framework rapidly.

For the users to accept DynaBlu the performance of the Bluetooth communication and discovery must be fast enough to relieve the users of waiting a lot of time to receive data or searching for services. The Bluetooth communication is not yet fast enough for a production stable solution because of implementation details with using the JSR 82, but as mentioned in the conclusion this problem can be solved by using additional buffering and interfacing with L2CAP instead of RFCOMM. Another way of improving the performance of the Bluetooth communication would be to implement a compression layer into a future version of our bridge. Such a layer could contain XML compression features as available in several commercially available XML accelerators

such as Intel®'s XML accelerator [29] and IBM's WebSphere DataPower XML Accelerator XA35 [28]. The performance of the Bluetooth discovery is on the other hand acceptable in the current state because of the optimization implemented in the client bridge.

The acceptance of DynaBlu also depends on how well the client application is accepted by users. The client GUI is not very appealing in the current state, but is simple and functional. And since it is designed to be extensible, improvements can be made as the needs arise in the future.

Another factor that will be important for the success of the framework is how much time it will take for it to be developed into a production stable solution. This depends on how much is needed in order for the system to be production stable, and to answer this question we would have to test the system with actual users.

Developer Acceptance
For developers to accept DynaBlu it must be simple to develop and publish services. In the current state web services are published that offer data that is either text or pictures. There are many tools available that enable developers to build and deploy web services, and this is an advantage. Furthermore since services are published as web services it is possible to integrate them in other applications. In the future we would like to be able to handle HTML data in the provider services as well. This way dynamic content could be viewed in a browser on the mobile device, and developers would be able to work with a language that is straight forward for most developers. Section 13.3 discusses this option in detail.

Important Qualities
There are a number of qualities of DynaBlu that are particularly important factors that will have an influence on its success. The selectability from the users point-of-view is a clear advantage compared to existing systems. Furthermore the security implemented in our framework opens the door for new types of applications that involve sending sensitive information, such as instance payment systems and banking services. The existing Bluetooth systems that we have examined in the preanalysis are all based on hardware components that have been developed by the various companies. Our framework is based entirely on software and is therefore a cheap way of creating a location-aware system.

Technological Perspectives

As the mobile hardware platform is under continuous development, and the resources of mobile devices will increase, it is possible that dynamic class loading will be incorporated in the mobile platform at some point. Dynamic

class loading would create new opportunities for us. For instance the client software could be downloaded directly from a service provider as a class file, and loaded directly into the memory of the mobile phone, which would make it easy to spread the client software. Furthermore dynamic class loading would make it possible to download and run dynamic stubs at runtime, which would eliminate the need for parsing of WSDL files at run-time in the client.

Another aspect of the continuing development of the mobile hardware platform is increased battery capacity, which is needed to support the integration of faster CPUs. This increased battery capacity also has the benefit of making the integration of faster short-range communication technologies feasible. For instance WiFi is faster than Bluetooth and covers a larger area, but also comsumes significantly more battery. Advances in short-range communication technology on mobile devices would be a welcome an addition that could help ensure the future success of DynaBlu.

13.1 Future Work

In this section we present a few problems that are particularly interesting to investigate further, and discuss possible solutions to these problems.

13.2 Service and Operation Mapping

As described in chapter 10 we have implemented descriptions of web services and operations in text files located on the same application server as the provider web service.

It is necessary to store service mappings in the provider web service. This way users are limited to access the services for which the provider web service has a mapping for, and the mappings can link to web services on any application server.

The problem with storing the operation mappings is that the rest of the web services are not necessarily located in the same place as the provider web service. Therefore the descriptions can be located on a different computer than the web services that they describe. Furthermore the descriptions are not stored in a standard format. This way the web services are not only separated from the descriptions, but access to the descriptions is not achieved through a standard gateway, like for instance a registry or the provider web service. Therefore the current solution would imply possible maintenance problems.

Web service operations that are published by service providers need to be described in order for clients to identify which operations they want to use. The web services are syntactically described by the use of WSDL documents, but a semantic description is also needed to explain the meaning of an operation invocation. In this case a semantic description would contain consists of information like service capabilities, preconditions and relations to other services. It would be desirable for us to be able to express what a particular service has to offer clients.

In [17] we looked into the relatively new research area *semantic web* (first proposed by the creator of the Web, Tim Berners-Lee, in 2001). This area deals with adding semantic information to web pages and services, with the purpose of automating task that will otherwise involve user interaction. Languages like OWL-S and SAWSDL can be used to add meta-data to web service descriptions, and using one of these languages is a possible solution for our problem with describing web service capabilities.

OWL-S is an XML language that can be used in conjunction with WSDL documents to add semantic information to services and operations. The OWL-S documents must be published in a registry, because the user must be able to retrieve the semantic documents, and to avoid maintenance problems the documents should be kept in one place. The OWL-S documents can for instance be stored in an UDDI repository together with the WSDL files they reference. In our case the obvious place to put this registry would be the mediator. But this way clients would have to download XML documents from the mediator over a GPRS connection, which is not free like Bluetooth communication. As we discovered in [17] there are other disadvantages with using an UDDI registry, for instance complexity, overhead and maintenance problems.

SAWSDL (Semantic Annotations for WSDL)[15] is a new initiative by W3C. SAWSDL is currently in a draft version and will build on WSDL 2.0, which is also under development. The vision of SAWSDL is to use the new extensibility features of WSDL 2.0 to add semantic information directly into WSDL documents. This is an interesting initiative from our perspective, because it would enable us to store the operation descriptions in the WSDL file. The advantage of using SAWSDL from our perspective is that all the information needed to understand and communicate with a service is contained within a single document, which can be downloaded from the service provider. This would eliminate the maintenance problems introduced by storing descriptions in separate text files, or by having a registry with web service meta data.

The best solution would be to use SAWSDL documents in our framework in the future, but depending on how much time it takes for the language to be released we may need to consider other alternatives.

13.3 HTML interaction

In this section we describe aspects regarding the adoption of HTML interaction into our framework. This basic research will be used as a starting point for our future work regarding the integration of this feature.

We discuss existing research and products in order for us to be able to identify possible solutions and describe some of the potential issues involved with using these. This section should not be regarded as a detailed design document describing HTML interaction modules.

We have chosen to focus on HTML as a scripting language to represent the graphical elements and structure of the GUI data in our client application. There are a number of reasons to include HTML which we list below.

- HTML is a widely used language for representing user interfaces. HTML is today a grounding used to show data in many applications and is thus becoming a standard not only for representing web pages but also in standard applications.
- HTML is today known by most developers, which makes it use straightforward for most developers to create interaction forms using our framework.
- A number of freely available HTML parsers or browsers are available for mobile devices.
- Web services developed using our framework would potentially be able to access external Internet HTML resources directly and send the content of these resources to our client application. Thus creating a gateway to access the Internet through our Bluetooth bridge.

Existing Systems

In order to display HTML documents in the client the client application will be required to have an embedded HTML browser available. Embedded means a browser that runs from within our software. Several available mobile J2ME browsers exist, for instance the Opera Mobile(TM) browser [43] and the Protheus J2ME Browser [49]. Though depending on how we will be implementing the HTML interaction we could be faced with some potential requirements to our browser. For instance if we need to modify the browser we will need access to the source-code of the browser thus the browser should be an open-source product released under a license usable by us. One of these modifications we are faced with is that we will need to redirect all HTTP traffic from the browser to our Bluetooth bridge.

In [47] the design of a Bluetooth Wireless Internet Gateway (BWIG) is described. The purpose of the BWIG is to provide seamless Internet access to mobile devices utilizing Bluetooth as a transport medium. The design of the BWIG architecture is depicted in figure 13.1. Here we see the interaction between the Bluetooth clients and BWIG. Two modules have been developed that are responsible for providing seamless Internet access namely the BWIG client and the BWIG server.

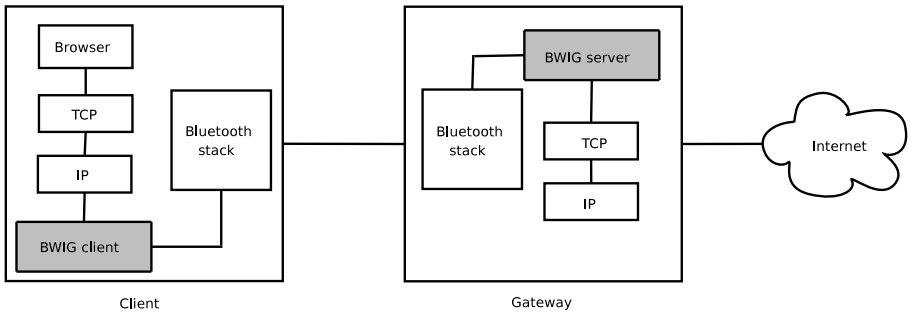


Figure 13.1: The Bluetooth Wireless Internet Gateway (BWIG) architecture

BWIG client The BWIG client is responsible for intercepting TCP/IP packets sent from the client at the application level and redirecting these over the local Bluetooth stack to the Bluetooth gateway.

BWIG server At the server side the BWIG server component receives the client TCP/IP request from the gateway's Bluetooth stack and forwards this to the web server intended to receive the data. The data received from the web server is sent back from the BWIG server to the BWIG client in the same fashion.

Integrating HTML Interaction

In this section we describe possible solution scenarios to adopt when integrating HTML interaction into our framework. By discussion we have identified two overall integration scenarios.

The first solution is to develop our own interpreter implementing a subset of HTML. This component should essentially interpret HTML code into GUI elements from the UI API available in the MIDP profile. In this scenario HTML would be used as a markup language describing native MIDP GUI elements and would not require us to include an embedded browser in our client application. Unfortunately this solution will require us to limit the use of HTML elements to those supported by our implementation. Furthermore this solution inhibits direct access to Internet resources through a Bluetooth

gateway component as we cannot ensure that the markup code describing the Internet resource will conform to our subset implementation.

The second solution is to use an already available browser and modify the source code of this to suit our needs. The browser should be modified to be able to show web pages from the data format we pass to it. This leads to our provider component. A design here could be inspired by the BWIG architecture. We have depicted a possible solution in figure 13.2 where we have developed an Internet Gateway web service in the provider. The idea is that web service applications deployed in the provider should be able to use the Internet Gateway web service to access either external HTML pages from the Internet or locally deployed HTML pages.

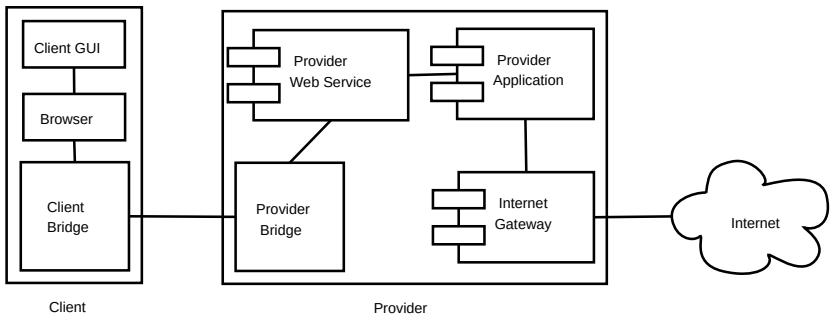


Figure 13.2: Possible design accommodating the integration of HTML interaction

This Internet Gateway web service should support the downloading of an HTML page along with all the resources referenced in the page for instance images and stylesheets contained in separate CSS files. This data should be bundled into a data representation that we can send to the client and show in the embedded browser.

Another issue is that we will need to be able to emulate HTTP transport. This implies that we should be able to handle HTML form data, forms are used in HTML to realize interaction by sending variables as either POST or GET data [57]. When the user of the client application is presented with a form in the embedded browser and sends data from the form we need to package this into a HTTP request data representation that can be interpreted by the provider and passed onto the Internet Gateway web service. The gateway would then forward the form data to the web server hosting the HTML page using HTTP.

Our current framework implementation forces the client to open a GPRS connection to the mediator when establishing a connection with a provider. Mobile phone providers often charge subscribers for using this kind of communication. Integrating an Internet Gateway web service opens the possibility of routing the client communication with the mediator through the provider and thus making the Internet Gateway contact the mediator on behalf of the client. This would make it possible for clients to invoke web service applications implemented using our framework free of charge. However implementing this would require us to develop features for being able to send SSL encrypted data over a Bluetooth connection.

13.4 Final Remarks

These suggestions for future work provide us with a starting point for a continued project into our next semester. The extensibility of DynaBlu makes us flexible in that we can quickly adapt to new user requirements. This gives us a good basis for doing further user and market analysis and to respond dynamically to changes produced by these analysis.

The long-term goals of our continued work on the DynaBlu framework is to make it enter a production-stable state where we can market it and examine the possibilities for creating a commercial product.

Bibliography

[1] HPI Research Group. http://www.hpiresearch.com/.

[2] *Accessing Native Methods from a Midlet -> A Powerful Workaround.* Arvind Gupta. http://www.microjava.com/articles/techtalk/dynamic?PageNo=1.

[3] *Dynamic Classloading in the KVM.* MicroDevNet, 2000. http://www.microjava.com/articles/techtalk/dynamic?PageNo=1.

[4] *JSR 30: Connected Limited Device Configuration 1.0.* Java Community Process, 2000. http://jcp.org/en/jsr/detail?id=30.

[5] *JSR 139: Connected Limited Device Configuration 1.1.* Java Community Process, 2003. http://jcp.org/en/jsr/detail?id=139.

[6] *JSR 172: J2METM Web Services Specification.* Java Community Process, 2004. http://jcp.org/en/jsr/detail?id=172.

[7] *BlipNet Technical White Paper.* BLIP Systems A/S, January 2005.

[8] *JavaTM Technology in Mobility At-A-Glance.* SUN Microsystems, 2005. http://www.sun.com/aboutsun/media/presskits/javaone2005/mobility_aag_final0605_v2.pdf.

[9] *JSR 218: Connected Device Configuration (CDC) 1.1.* Java Community Process, 2005. http://jcp.org/en/jsr/detail?id=218.

[10] *JSR 118: Mobile Information Device Profile 2.0.* Java Community Process, 2006. http://jcp.org/en/jsr/detail?id=118.

[11] *JSR 82: JavaTM APIs for Bluetooth.* Java Community Process, 2006. http://jcp.org/en/jsr/detail?id=82.

[12] *BlipNet.* BLIP Systems, 2007. http://www.blipsystems.com/.

[13] *Javadoc documentation for the DynaBlu framework.* Nikolaj Andersen, Morten Vejen Nielsen, Jørn Martin Rasmussen, 2007. http://www.cs.aau.dk/~mvejen/dat6/javadoc.

[14] *Javadoc Tool Home Page.* SUN Microsystems, 2007. http://java.sun.com/j2se/javadoc/.

[15] *Semantic Annotations for Web Services Description Language Working Group.* W3C, 2007. http://www.w3.org/2002/ws/sawsdl/.

[16] L. Aalto, N. Göthlin, J. Korhonen, and T. Ojala. Bluetooth and wap push based location-aware mobile advertising system. *ACM*, june 2004.

[17] N. Andersen, T. L. Kjeldsen, C. P. Larsen, M. V. Nielsen, and J. M. Rasmussen. A technical view on soa and related acronyms. Technical report, Department of Computer Science, AAU, 2006.

[18] V. Auletta, C. Blundo, E. D. Cristofaro, and G. Raimato. A lightweight framework for web service invocation over bluetooth. *IEEE*, 2006.

[19] V. Auletta, C. Blundo, E. D. Cristofaro, and G. Raimato. Performance evaluation of web service invocation over bluetooth. *ACM*, 2006.

[20] avetana GmbH. *avetanaBluetooth JSR82 Implementation.* 2007. http://www.avetana-gmbh.de/avetana-gmbh/produkte/jsr82.eng.xml.

[21] BlueBlitz. *BlueBlitz.* 2007. http://www.blueblitz.com/.

[22] bluetooth.com. *Specification of the Bluetooth System.* http://www.bluetooth.com/Bluetooth/Learn/Technology/Specifications/.

[23] Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[24] d619a. *Sourcecode to our framework.* See enclosed CD-ROM.

[25] J. Eker. *Harald - A Java Bluetooth Stack.* 2007. http://www.control.lth.se/~johane/harald/.

[26] C. Gehrmann, J. Persson, and B. Smeets. *Bluetooth Security.* Artech House, Inc., Norwood, MA, USA, 2004.

[27] B. S. I. Group. *Bluetooth Assigned Numbers.* 2007. http://www.bluetooth.org/assigned-numbers/.

[28] IBM. *WebSphere DataPower XML Accelerator XA35*. http://www-306.ibm.com/software/integration/datapower/xa35/features/.

[29] Intel®. *Intel® XML Accelerator*. http://www.intel.com/support/network/xml/accelerator/.

[30] ITworld.com. *SSL and Mobile Devices*. 2001. http://www.itworld.com/nl/java_sec/04202001/.

[31] A. S. Jensen. *CWhere, Bluetooth in a Mobile Positioning Context*. 2005.

[32] M. D. Jode. *Programming Java 2 Micro Edition on Symbian OS*. John Wiley & Sons, Ltd., 2004.

[33] K.Kaukonen and R.Thayer. *A Stream Cipher Encryption Algorithm Arcfour*. 1999. http://www.mozilla.org/projects/security/pki/nss/draft-kaukonen-cipher-arcfour-03.txt.

[34] E. kSOAP project. *kSOAP 2*. 2006. http://ksoap2.sourceforge.net/.

[35] J. F. Kurose and K. W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet Package, 3rd International Edition*. Addison-Wesley Longman Publishing Co., Inc., 2005.

[36] P. Landrock and K. Nissen. *Kryptologi*. Forlaget ABACUS, Vejle, Denmark, 1992.

[37] B. A. Miller. *The Phony Conflict: IEEE 802.11 and Bluetooth Wireless Technology*. november 2001. http://www.informit.com/articles/article.asp?p=24240&seqNum=1&rl=1.

[38] Mobilereact. *Practical Mobile Marketing White Paper*. 2005. http://www.mobilereact.co.th/downloads/Eng_WP.pdf.

[39] N. J. Muller. *Bluetooth Demystified*. McGraw Hill Professional, 2000.

[40] MySQL. *MySQL Community*. http://www.mysql.com/.

[41] T. L. of the Bouncy Castle. *bouncycastle.org*. http://www.bouncycastle.org/.

[42] L. M. A. M.-M. P. A. N. og Jan Stage. *Objekt Orienteret Analyse & Design*. Marko, 2001.

[43] Opera. *Opera Mobile™*. http://www.opera.com/products/mobile/.

[44] L. L. Petrea and D. Grigoras. Dynamic class provisioning on mobile devices. *IEEE*, 2006.

[45] M. Pracucci. *JBluetooth*. http://download.pracucci.com/java/jbluetooth/readme.html.

[46] rococo software. *Impronto Bluetooth Library*. 2007. http://www.rococosoft.com/java.html.

[47] N. Rouhana and E. Horlait. Bwig: Bluetooth web internet gateway. In *ISCC '02: Proceedings of the Seventh International Symposium on Computers and Communications (ISCC'02)*, page 679, Washington, DC, USA, 2002. IEEE Computer Society.

[48] K. Sairam, N. Gunasekaran, and S. Redd. Bluetooth in wireless communication. *Communications Magazine, IEEE*, 40(06):90–96, june 2002.

[49] SourceForge.net. *Protheus J2ME Browser*. http://sourceforge.net/projects/protheus/.

[50] D. Statistik. *Befolkningens køb via internettet efter hyppighed, type og tid*. 2007,. http://www.dst.dk.

[51] S. Stemberger. *Is Bluetooth Wi-Fi?* april 2002. http://www-128.ibm.com/developerworks/wireless/library/wi-net.html.

[52] B. D. Team. *Bluecove*. 2007. http://code.google.com/p/bluecove/.

[53] J. D. Team. *JBlueZ*. 2007. http://jbluez.sourceforge.net/.

[54] Thawte. *SSL digital certificates with extended validation from thawte the global SSL certificate authority*. http://www.thawte.com.

[55] K. Topley. *J2ME in a nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[56] VeriSign. *VeriSign - Security (SSL Certificate), Communications, and Information Services*. http://verisign.com.

[57] W3C. *Forms in HTML documents*. http://www.w3.org/TR/html4/interact/forms.html#h-17.3.

[58] A. I. Wang, M. S. Norum, and C.-H. W. Lund. Issues related to development of wireless peer-to-peer games in j2me. In *AICT-ICIW '06: Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services*, page 115, Washington, DC, USA, 2006. IEEE Computer Society.

[59] A. I. Wang, M. S. Norum, and C.-H. W. Lund. Issues related to development of wireless peer-to-peer games in j2me. *IEEE*, 2006.

[60] Wikipedia.org. *Bluetooth*. 2007. http://en.wikipedia.org/wiki/Bluetooth.

[61] Wikipedia.org. *Bluetooth Special Interest Group*. 2007. http://en.wikipedia.org/wiki/Bluetooth_sig.

[62] Wikipedia.org. *HTTPS*. 2007. http://en.wikipedia.org/wiki/HTTPS.

[63] Wikipedia.org. *RC4*. 2007. http://en.wikipedia.org/wiki/RC4.

[64] S. Williams. Irda: past, present and future. *Personal Communications, IEEE*, 7(01):11–19, february 2000.

Part V

Appendix

Appendix

Project Code Samples

.1 Mediator Web Service

The following listing contains the implementation of the mediator web service, which is responsible for negotiating connections between clients and providers. In chapter 11 there is a flow chart (figure 11.3) that simplifies how the web service synchronizes access to the database.

```
1
2  /*
3   * Mediator.java
4   *
5   * Created on April 4, 2007, 2:21 PM
6   *
7   */
8
9  package mediator;
10
11 import d619a.common.bridge.security.EncryptionMode;
12 import d619a.mediator.database.DBMediator;
13 import d619a.mediator.database.DatabaseErrorException;
14 import java.util.Random;
15 import javax.jws.WebService;
16 import javax.jws.WebMethod;
17 import d619a.mediator.encryption.Encryption;
18
19 /**
20  * This class implementes the mediator web service.
21  * @author Nikolaj Andersen
22  */
23 @WebService()
24 public class Mediator {
25
26     /**
27      * Clients can call this method to attempt to set up Bluetooth
            connections
28      * with service providers.
29      * @param providerName Name of the service
30      * @param BTAddressClient Bluetooth address of the client
31      * @param BTAddressProvider Bluetooth address of the service
            provider
32      * @return a 'ConnectInfo.java' object, which contains the
            information
33      * needed for further communication between client and provider.
34      * Including response codes and error codes.
35      */
36     @WebMethod
37     // this method should only be called by the client
38     public ConnectInfo setupConnectionClient(String providerName, String
           BTAddressClient, String BTAddressProvider, int encryptionMode) {
39         System.out.println("running client method...");
40         DBMediator db = new DBMediator();
41         int TimeOut = Integer.parseInt(System.getProperty("timeout"));
```

```
42
43          // check if it is a valid provider
44          if (!db.validProvider(providerName, BTAddressProvider)) {
45              ConnectInfo info = new ConnectInfo();
46              info.setBTProviderAddress(BTAddressProvider);
47              info.setBTClientAddress(BTAddressClient);
48              info.setSucces(Status.PROVIDERNOTFOUND);
49              info.setErrorCode("No such provider in Database");
50              return info;
51          }
52
53          // create connection record in Connections table, and wait
                for the provider to match it (with timeout)
54          try {
55              db.createConnectionRecord(BTAddressClient, BTAddressProvider,
                    encryptionMode);
56              System.out.println("created record for client: "+ providerName
                    + " " + BTAddressClient + " " + BTAddressProvider);
57          } catch (DatabaseErrorException ex) {
58              ConnectInfo info = new ConnectInfo();
59              info.setBTProviderAddress(BTAddressProvider);
60              info.setBTClientAddress(BTAddressClient);
61              info.setSucces(Status.CONNECTIONNOTAVAILABLE);
62              info.setErrorCode("Internal database problem");
63              return info;
64          }
65          ConnectInfo info = new ConnectInfo();
66          info.setSucces(Status.CONNECTIONNOTAVAILABLE);
67          info.setErrorCode("No response from other party");
68          info.setBTClientAddress(BTAddressClient);
69          info.setBTProviderAddress(BTAddressProvider);
70          int counter = 0;
71          while (counter < TimeOut) {
72              if (db.isMatch(BTAddressClient, BTAddressProvider)) {
73                  System.out.println("match found for client: "+ providerName
                        + " " + BTAddressClient + " " + BTAddressProvider);
74                  String key = "";
75                  String wsdlVersion = "";
76                  int encMode = 0;
77                  int keyTimeout = 0;
78                  try {
79                      key = db.getEncKey(BTAddressClient, BTAddressProvider);
80                      wsdlVersion = db.getWsdlVersion(providerName,
                            BTAddressProvider);
81                      keyTimeout = db.getKeyTimeout(providerName,
                            BTAddressProvider);
82                      encMode = db.getEncryptionMode(BTAddressClient,
                            BTAddressProvider);
83                  } catch (DatabaseErrorException ex) {
84                      info.setSucces(Status.CONNECTIONNOTAVAILABLE);
85                      info.setErrorCode("Internal database error");
86                      break;
87                  }
88                  if (encMode == EncryptionMode.INCOMPATIBLE) {
89                      info.setSucces(Status.INCOMPATIBLEENCRYPTIONMODES);
90                      info.setEncryptionMode(EncryptionMode.INCOMPATIBLE);
91                      info.setErrorCode("The encryption modes were
                            incompatible.");
92                      break;
93                  }
94                  info.setEncKey(key);
95                  info.setEncryptionMode(encMode);
96                  info.setSucces(Status.CONNECTIONESTABLISHED);
97                  info.setWsdlVersion(wsdlVersion);
98                  info.setKeyTimeout(keyTimeout);
99                  info.setErrorCode("No errors");
100                 break;
101             } else {
102                 // try to sleep to wait for the provider to match the
                        connection record
103                 try {
104                     System.out.println("sleeping: "+ providerName + " " +
                            BTAddressClient + " " + BTAddressProvider);
105                     Random rand = new Random(System.currentTimeMillis());
106                     // random number between 2000 and 4000
107                     long sleepInterval = ((Math.abs(rand.nextLong())) %
                            2000) + 2000;
108                     Thread.sleep(sleepInterval);
109                 } catch (InterruptedException ex) {
110                     continue;
111                 }
112             }
113             counter++;
114         }
```

```
115
116             // delete connection record and return connection information
117             try {
118                 db.removeConnectionRecord(BTAddressClient, BTAddressProvider);
119             } catch (DatabaseErrorException ex) {
120                 // ignore, record will be cleaned up later
121             }
122             return info;
123         }
124
125         /**
126          * Service providers can call this method to attempt to set up
127          * Bluetooth connections with clients.
128          * @return a 'ConnectInfo.java' object, which contains the
                information
129          * needed for further communication between client and provider.
130          * Including response codes and error codes.
131          * @param providerName Name of the service
132          * @param BTAddressClient Bluetooth address of the client
133          * @param BTAddressProvider Bluetooth address of the service
                provider.
134          */
135         @WebMethod
136         // this method should only be called by the provider
137         public ConnectInfo setupConnectionProvider(String providerName, String
                BTAddressClient, String BTAddressProvider, int encryptionMode) {
138             System.out.println("running provider method...");
139             DBMediator db = new DBMediator();
140             int TimeOut = Integer.parseInt(System.getProperty("timeout"));
141
142             // check if it is a valid provider
143             if (!db.validProvider(providerName, BTAddressProvider)) {
144                 ConnectInfo info = new ConnectInfo();
145                 info.setBTProviderAddress(BTAddressProvider);
146                 info.setBTClientAddress(BTAddressClient);
147                 info.setSucces(Status.PROVIDERNOTFOUND);
148                 info.setErrorCode("No such provider in Database");
149                 return info;
150             }
151
152             // look/wait for a connection record in Connections table,
                    and match it when found (with timeout)
153             ConnectInfo info = new ConnectInfo();
154             info.setSucces(Status.CONNECTIONNOTAVAILABLE);
155             info.setErrorCode("No response from other party");
156             info.setBTClientAddress(BTAddressClient);
157             info.setBTProviderAddress(BTAddressProvider);
158             int counter = 0;
159             while (counter < TimeOut) {
160                 if (db.isRecord(BTAddressClient, BTAddressProvider)) {
161                     System.out.println("record found for provider: "+
                            providerName + " " + BTAddressClient + " " +
                            BTAddressProvider);
162                     // get the encryption mode, and apply encryption
                            policy
163                     int encMode = EncryptionMode.INCOMPATIBLE;
164                     try {
165                     encMode = db.getEncryptionMode(BTAddressClient,
                            BTAddressProvider);
166                     } catch (DatabaseErrorException ex) {
167                         info.setSucces(Status.CONNECTIONNOTAVAILABLE);
168                         info.setErrorCode("Internal database error");
169                         break;
170                     }
171                     encMode = this.getEncryptionMode(encMode, encryptionMode);
172                     // if the encryption modes are incompatible return an
                            error
173                     if (encMode == EncryptionMode.INCOMPATIBLE) {
174                         info.setSucces(Status.INCOMPATIBLEENCRYPTIONMODES);
175                         info.setEncryptionMode(EncryptionMode.INCOMPATIBLE);
176                         info.setErrorCode("The encryption modes were
                                incompatible.");
177                         try {
178                             db.writeMatch(BTAddressClient, BTAddressProvider, "
                                    ", encMode);
179                         } catch (DatabaseErrorException ex) {
180
181                         }
182                         break;
183                     }
184                     // generate encryption key, and write match to db
185                     Encryption enc = new Encryption();
186                     String key = enc.getEncryptionKey();
187                     String wsdlVersion = "";
```

```
188                 int keyTimeout = 0;
189                 try {
190                     db.writeMatch(BTAddressClient, BTAddressProvider, key,
                            encMode);
191                     wsdlVersion = db.getWsdlVersion(providerName,
                            BTAddressProvider);
192                     keyTimeout = db.getKeyTimeout(providerName,
                            BTAddressProvider);
193                     System.out.println("match written for provider: "+
                            providerName + " " + BTAddressClient + " " +
                            BTAddressProvider);
194                 } catch (DatabaseErrorException ex) {
195                     info.setSucces(Status.CONNECTIONNOTAVAILABLE);
196                     info.setEncKey("");
197                     info.setErrorCode("Internal database problem");
198                     break;
199                 }
200                 info.setEncKey(key);
201                 info.setEncryptionMode(encMode);
202                 info.setSucces(Status.CONNECTIONNOTAVAILABLE);
203                 info.setWsdlVersion(wsdlVersion);
204                 info.setKeyTimeout(keyTimeout);
205                 info.setErrorCode("No errors");
206                 break;
207             } else {
208                 // try sleeping to wait for the client to write a
                        connection record
209                 try {
210                     System.out.println("sleeping: "+ providerName + " " +
                            BTAddressClient + " " + BTAddressProvider);
211                     Random rand = new Random(System.currentTimeMillis());
212                     // random number between 2000 and 4000
213                     long sleepInterval = ((Math.abs(rand.nextLong())) %
                            2000) + 2000;
214                     Thread.sleep(sleepInterval);
215                 } catch (InterruptedException ex) {
216                     continue;
217                 }
218             }
219             counter++;
220         }
221         // return connection information
222         return info;
223     }
224
225     /**
226      * Service providers can call this web method to update the WSDL
               version
227      * of wsdl files for a particular service.
228      * @param providerName Name of the service.
229      * @param providerBTAddress Bluetooth address of the service
               provider.
230      * @param version new version of the wsdl file.
231      * @param adminPassword the administrative password required to
               perform updates for
232      * this dervice provider.
233      * @return returns an integer which correspondes to values
               defined in
234      * interface 'Status.java'
235      */
236     @WebMethod
237     public int updateWSDL(String providerName, String providerBTAddress,
            String version, String adminPassword) {
238         // try to update the wsdl version
239         System.out.println("running update method...");
240         int retur = Status.ERROR;
241         DBMediator db = new DBMediator();
242
243         if (db.checkPassword(providerName, providerBTAddress, adminPassword
                )) {
244             try {
245                 db.updateWSDL(providerName, providerBTAddress, version);
246                 retur = Status.WSDLUPDATESUCCESSFUL;
247             } catch (DatabaseErrorException ex) {
248                 retur = Status.ERROR;
249             }
250
251         } else {
252             retur = Status.WRONGPASSWORDORPROVIDERNAME;
253         }
254
255         return retur;
256     }
257
```

```
258     /* This method is used internally. It implements the encryption
            policy logic
259      * @return returns an integer that signal the encryption mode,
            defined in
260      * the EncryptionMode.java interface.
261      */
262     private int getEncryptionMode(int clientFlag, int providerFlag) {
263         if (clientFlag == EncryptionMode.ONN && providerFlag ==
                EncryptionMode.ONN )
264             return EncryptionMode.ONN;
265         else if (clientFlag == EncryptionMode.OFF && providerFlag ==
                EncryptionMode.OFF )
266             return EncryptionMode.OFF;
267         else if (clientFlag == EncryptionMode.ONN && providerFlag ==
                EncryptionMode.OFF )
268             return EncryptionMode.INCOMPATIBLE;
269         else if (clientFlag == EncryptionMode.OFF && providerFlag ==
                EncryptionMode.ONN )
270             return EncryptionMode.INCOMPATIBLE;
271         else if (clientFlag == EncryptionMode.OPTIONAL && providerFlag ==
                EncryptionMode.OPTIONAL )
272             return EncryptionMode.OFF;
273         else if (clientFlag == EncryptionMode.OPTIONAL && providerFlag ==
                EncryptionMode.OFF )
274             return EncryptionMode.OFF;
275         else if (clientFlag == EncryptionMode.OFF && providerFlag ==
                EncryptionMode.OPTIONAL )
276             return EncryptionMode.OFF;
277         else if (clientFlag == EncryptionMode.OPTIONAL && providerFlag ==
                EncryptionMode.ONN )
278             return EncryptionMode.ONN;
279         else if (clientFlag == EncryptionMode.ONN && providerFlag ==
                EncryptionMode.OPTIONAL )
280             return EncryptionMode.ONN;
281
282         return EncryptionMode.INCOMPATIBLE;
283     }
284 }
```

Listing 1: Mediator.java: Implementation of the mediator web service.

.2 Search Unit

The following listing shows our implementation of a Bluetooth module that is used for searching for Bluetooth services on nearby Bluetooth devices. The search unit is optimized for limiting search time by using different modes of operation and white- and blacklists for devices (see section 8.1.2).

```
1
2  /*
3   * SearchUnit.java
4   *
5   * Created on March 19, 2007, 2:14 PM
6   *
7   */
8
9  package d619a.client.bridge.bluetooth.searchunit;
10
11 import d619a.client.bridge.bluetooth.exceptions.BTDisabledException;
12 import d619a.client.bridge.bluetooth.test.DebugLogger;
13 import java.io.IOException;
14 import java.util.Date;
15 import javax.bluetooth.BluetoothStateException;
16 import javax.bluetooth.DataElement;
17 import javax.bluetooth.DeviceClass;
18 import javax.bluetooth.DiscoveryAgent;
19 import javax.bluetooth.DiscoveryListener;
20 import javax.bluetooth.LocalDevice;
21 import javax.bluetooth.RemoteDevice;
22 import javax.bluetooth.ServiceRecord;
23 import javax.bluetooth.UUID;
24
25 /**
26  * This class implements functionality for discovering bluetooth
        devices
```

```
27   * and scan for the availability of services with a specified UUID.
28   * Once started, the object of this class does the discovery all by
         itself
29   * and maintains a ProviderList which contains all the devices found
30   * in the vicinity that provide one or several service-instances we
         where
31   * looking for.
32   * Notification for changes in that list are realized through
33   * the SearchUnitCallback interface.
34   * @author nikko, jmr
35   */
36  public class SearchUnit extends Thread implements DiscoveryListener {
37    private static int SU_MODE_IDLE = 1;
38    private static int SU_MODE_INTERACTION = 0;
39    private static int SU_MODE_TERMINATED = -1;
40
41    private static int ATTR_SERVICENAME = 0x0100;
42    private static int ATTR_SERVICERECORDHANDLE = 0x0000;
43    private static int ATTR_SERVICECLASSIDLIST = 0x0001;
44    private static int ATTR_PROTOCOLDESCRIPTORLIST = 0x0004;
45
46    private UUID[] uuid;
47    private SearchUnitCallback parent;
48    private DiscoveryAgent discoveryAgent;
49    private LocalDevice localDevice;
50
51    private ProviderList myProviderList;
52    private DeviceList myDeviceList = new DeviceList();
53
54    private int currentMode = SU_MODE_IDLE;
55
56
57  //cache for the ServiceSearch
58    private ServiceRecord[] serviceRecordCache;
59    private int serviceDiscoTransID;
60    private RemoteDevice serviceDiscoCurrentDevice;
61    private boolean bIsScanning;
62
63    //properties
64    private boolean prop_monitorReachability = true;
65    private int[]   prop_filterDeviceClasses = null;
66    private boolean prop_filterServiceDuplicatesOnSameDevice = false;
67
68
69    /**
70     * Creates a new instance of SearchUnit
71     * After creation, start the Thread with the start() method.
72     * Switch between idle-mode and interaction-mode with
73     * setInteractionMode() and setIdleMode().
74     * @param uuid the thread will search for services having this uuid
75     * @param parent the receiver of callbacks when services are found.
76     * receiver must implement 'SearchUnitCallback' interface.
77     */
78    public SearchUnit(UUID uuid, SearchUnitCallback parent) {
79      this.uuid = new UUID[] {uuid};
80      this.parent = parent;
81      this.myProviderList = new ProviderList(parent);
82    }
83
84    /**
85     * terminate the SearchUnit-Thread.
86     * This will stop all processes. Reinvocation is done with start()
87     */
88    public void terminateSearch(){
89      currentMode = SU_MODE_TERMINATED;
90      this.discoveryAgent.cancelInquiry(this);
91      this.discoveryAgent.cancelServiceSearch(this.serviceDiscoTransID);
92    }
93
94    /**
95     * Starts the Thread
96     */
97    public void run(){
98
99      while(currentMode != SU_MODE_TERMINATED){
100       if(currentMode == SU_MODE_IDLE){
101         try {
102           //check WHITE elements, if they still are in the vicinity
103           surveyKnownDevices();
104           //now, discover new devices in the vicinity
105           this.startInquiry();
106         } catch (BTDisabledException ex) {
107           //we can't throw an exception, so we just terminate this
                search.
```

```
108             this.terminateSearch();
109           }
110
111           //wait for the discovery to complete
112           waitWhileScanning();
113
114           try {
115             //scan all UNDECIDED devices from the DeviceList for
                    services
116             //UNDECIDED are all those devices, that have just been
                    discovered.
117             //they are neither black nor white yet.
118             surveyUndecidedDevices();
119           } catch (BTDisabledException ex) {
120             //we can't throw an exception, so we just terminate this
                    search.
121             this.terminateSearch();
122           }
123
124         } else if(currentMode == SU_MODE_INTERACTION){
125           try {
126
127             //if the mobile properties allows it, monitor WHITE devices
128             //-> check WHITE elements, if they still are in the
                    vicinity
129             //update ProviderList (remove those without signal)
130             surveyKnownDevices();
131           } catch (BTDisabledException ex) {
132             ex.printStackTrace();
133           }
134
135
136         }
137       }
138
139     }//END run
140
141     /**
142      * get Instance to LocalDevice.
143      * If no Device is available, it's certainly because bluetooth
144      * is not enabled on the phone. We throw an exception.
145      *
146      * @throws d619a.client.bridge.bluetooth.exceptions.
                BTDisabledException
147      */
148     private void getLocalDevice() throws BTDisabledException {
149       getLocalDevice(false);
150     }
151
152     /**
153      * get Instance to LocalDevice.
154      * If no Device is available, it's certainly because bluetooth
155      * is not enabled on the phone. We throw an exception.
156      * @param forcenew forces the method to retrieve a new instance of
                the local device
157      * @throws d619a.client.bridge.bluetooth.exceptions.
                BTDisabledException
158      */
159     private void getLocalDevice(boolean forcenew) throws
             BTDisabledException {
160       try {
161         if(forcenew || localDevice == null)
162           localDevice = LocalDevice.getLocalDevice();
163         if(forcenew || discoveryAgent == null)
164           discoveryAgent = localDevice.getDiscoveryAgent();
165       } catch (BluetoothStateException ex) {
166         //the only reason that makes it impossible to get the
                localDevice
167         //and get a DiscoveryAgent, is that the Bluetooth Device is
                disabled
168         //in the mobile phone. So we throw an exception!!
169         throw (BTDisabledException) ex;
170       }
171     }
172
173
174
175     /**
176      * Start to inquiry the vicinity for bluetooth devices.
177      *
178      * @throws d619a.client.bridge.bluetooth.exceptions.
                BTDisabledException
179      */
180     private void startInquiry() throws BTDisabledException{
```

```
181
182       getLocalDevice();
183
184       try {
185         discoveryAgent.startInquiry(DiscoveryAgent.GIAC, this);
186         setIsScanning(true);
187       } catch (BluetoothStateException ex) {
188         //I assume, here we fail on the same cause as in getLocalDevice
189         throw (BTDisabledException) ex;
190       }
191     }
192
193     /**
194      * Stop the Inquiry. (aka cancel)
195      */
196     private void stopInquiry(){
197       if(discoveryAgent != null)
198         discoveryAgent.cancelInquiry(this);
199
200       setIsScanning(false);
201     }
202
203     /**
204      * stop an ongoing servicesearch
205      */
206     private void stopServiceSearch(){
207       if(discoveryAgent != null)
208         discoveryAgent.cancelServiceSearch(this.serviceDiscoTransID);
209     }
210
211     /**
212      * set the SearchUnit to IdleMode.
213      * In IdleMode, the SearchUnit does an exhaustive search for new
214      * Devices and their services. This mode is on most mobilephones
215      * only available, if no connection is established.
216      */
217     public void setIdleMode(){
218       stopServiceSearch();
219       stopInquiry();
220       currentMode = SU_MODE_IDLE;
221     }
222
223     /**
224      * set the SearchUnit to InteractionMode.
225      * In interaction-mode, the SearchUnit only checks frequently if
226      * already discovered devices still are in the vicinity.
227      * And this only, if the mobilephone is capable of doing this while
228      * a connection is established.
229      */
230     public void setInteractionMode(){
231       stopServiceSearch();
232       stopInquiry();
233       currentMode = SU_MODE_INTERACTION;
234     }
235
236
237
238     /**
239      * Start the discovery for Services on a specified Device.
240      * @param btDevice The device to be discovered on
241      * @throws d619a.client.bridge.bluetooth.exceptions.
           BTDisabledException
242      * Bluetooth disabled
243      */
244     private void startServiceDisco(RemoteDevice btDevice) throws
          BTDisabledException{
245
246       getLocalDevice();
247
248       //reset the cache
249       serviceRecordCache = null;
250       serviceDiscoCurrentDevice = btDevice;
251       try {
252
253         //start the Search, which is non-blocking
254         int[] attrSet = { ATTR_SERVICERECORDHANDLE, ATTR_SERVICECLASSIDLIST,
               ATTR_PROTOCOLDESCRIPTORLIST, ATTR_SERVICENAME };
255         serviceDiscoTransID = discoveryAgent.searchServices(attrSet,uuid,
              serviceDiscoCurrentDevice,this);
256         setIsScanning(true);
257       } catch (BluetoothStateException ex) {
258         //if it doesn't work, it just doesn't work! :-S
259         DebugLogger.getInstance().addEntry(ex.getMessage());
260         this.discoveryAgent.cancelServiceSearch(this.serviceDiscoTransID);
```

```
261
262       } catch (Exception e) {
263         DebugLogger.getInstance().addEntry("disco NULLPOINTER.\n");
264         //a nullpointer exception indicates, that searchServices can
                not establish a connection to
265         //the remote device. remove it from the list then.
266         //update DeviceList and ProviderList and set current Device to
                "OFFLINE"
267         myDeviceList.changeSignalState(this.serviceDiscoCurrentDevice,
                DeviceList.OFFLINE);
268         myProviderList.removeProvider(serviceDiscoCurrentDevice.
                getBluetoothAddress());
269         this.discoveryAgent.cancelServiceSearch(this.serviceDiscoTransID);
270       }
271     }
272
273
274
275
276     /**
277      * Callback receiver from the DiscoveryAgent
278      * @param btDevice The discovered Device
279      * @param cod Class of Device. ->see Bluetooth Specification
280      */
281     public void deviceDiscovered(RemoteDevice btDevice, DeviceClass cod) {
282       //if we have enabled the DeviceClass filtering, only
283       //add current device to the DeviceList, if it belongs to one of
              the
284       //accepted Classes.
285       //int majorClass = cod.getMajorDeviceClass();
286       //TODO:add DeviceClass filtering capability
287       //update the DeviceList with the found device
288       myDeviceList.upsert(btDevice, DeviceList.DEV_IS_UNDECIDED, DeviceList.
              ONLINE);
289
290     }
291
292     /**
293      * Callback from DiscoveryAgent.
294      * Services have been discovered by the Discovery initiated with
295      * a certain Transaction ID
296      * @param transID The Transaction ID that identifies the discovery
297      * @param serviceRecord the serviceRecord of the remote device that
              matches the search.
298      */
299     public void servicesDiscovered(int transID, ServiceRecord[]
          serviceRecord) {
300
301       //the serviceRecord contains ONLY those services, having the UUID
302       //we have scanned for.
303       //we cache those records now, and do any updating ONLY when the
              search completes.
304       //So we update the ProviderList and the DeviceList
305       //and return to the serviceDiscovery.
306       if(this.serviceDiscoTransID == transID)
307         serviceRecordCache = serviceRecord;
308     }
309
310     /**
311      * Callback from DiscoveryAgent.
312      * Search for services has been completed.
313      * @param transID The Transaction ID which identifies the Search.
314      * @param respCode the Response code from the DiscoveryAgent which
              tells,
315      * how the discovery went.
316      */
317     public void serviceSearchCompleted(int transID, int respCode) {
318       if(transID == this.serviceDiscoTransID){
319
320         if(respCode == DiscoveryListener.SERVICE_SEARCH_DEVICE_NOT_REACHABLE)
                {
321           //update DeviceList and ProviderList and set current Device
                  to "OFFLINE"
322           myDeviceList.changeSignalState(this.serviceDiscoCurrentDevice,
                  DeviceList.OFFLINE);
323           myProviderList.removeProvider(serviceDiscoCurrentDevice.
                  getBluetoothAddress());
324           DebugLogger.getInstance().addEntry("NOT_REACHABLE:"+
                  serviceDiscoCurrentDevice.getBluetoothAddress() +"\n");
325
326         } else if(respCode == DiscoveryListener.SERVICE_SEARCH_NO_RECORDS){
327           //mark the current Device BLACK in the DeviceList
328           myDeviceList.changeState(this.serviceDiscoCurrentDevice,
                  DeviceList.DEV_IS_BLACK);
```

```
329
330         } else if(respCode == DiscoveryListener.SERVICE_SEARCH_COMPLETED){
331           //mark the current Device WHITE in the DeviceList
332           myDeviceList.changeStates(this.serviceDiscoCurrentDevice,
                  DeviceList.DEV_IS_WHITE, DeviceList.ONLINE);
333
334           //update the ProviderList with the device AND the services
335           //each service in a provider gets its own entry.
336           for(int i = 0; i<this.serviceRecordCache.length;i++){
337             String url = serviceRecordCache[i].getConnectionURL(ServiceRecord
                    .NOAUTHENTICATE_NOENCRYPT, false);
338             Date lastSeen = new Date();
339             String btAddr = serviceRecordCache[i].getHostDevice().
                    getBluetoothAddress();
340             String sname = (String)serviceRecordCache[i].getAttributeValue(
                    ATTR_SERVICENAME).getValue();
341               String devname = "unnamed";
342               try {
343                 devname = serviceRecordCache[i].getHostDevice().
                        getFriendlyName(false);
344               } catch (IOException ex) {
345                 ex.printStackTrace();
346               }
347             Provider newP = new Provider(btAddr, devname, sname, url,
                    lastSeen);
348             //update/insert the new/updated provider entry
349             myProviderList.upsert(newP);
350           }
351
352         } else {
353       //either the serviceSearch was terminated, or there was an error.
354         }
355       }
356       setIsScanning(false);
357     }
358
359     /**
360      * Callback from DiscoveryAgent.
361      * The inquiry of the vicinity for Bluetooth Devices is completed.
362      * @param discType status of the inquiry.
363      */
364     public void inquiryCompleted(int discType) {
365
366       if (discType == DiscoveryListener.INQUIRY_COMPLETED) {
367       } else if (discType == DiscoveryListener.INQUIRY_TERMINATED) {
368       } else if (discType == DiscoveryListener.INQUIRY_ERROR) {
369       }
370
371       setIsScanning(false);
372     }
373
374     /**
375      * Check all the devices marked as WHITE in the Devicelist,
376      * if they still are in the vicinity.
377      * If necessary, update the ProviderList.
378      * @throws d619a.client.bridge.bluetooth.exceptions.
             BTDisabledException Bluetooth disabled
379      */
380     private void surveyKnownDevices() throws BTDisabledException {
381       //check WHITE elements, if they still are in the vicinity
382       //update ProviderList (remove those without signal)
383       myDeviceList.setWhitesToUnknownSignalState();
384
385       //do a service scan on each of the white devices.
386       //if a device does not respond, we change its signal state and
387       //delete the device from the providerlist.
388       RemoteDevice dev = myDeviceList.getNextWhiteUnknownSignal();
389       while(dev != null){
390         DebugLogger.getInstance().addEntry("disco "+dev.getBluetoothAddress()
                +"\n");
391
392         this.startServiceDisco(dev);
393
394         //wait for this discovery to end
395         waitWhileScanning();
396
397         //if the SearchUnit mode has terminated, return immediately
398         //a change to idle-mode changes nothing, because this
                surveillance
399         //is a optimized pre-step for the idle-mode
400         if(currentMode == SU_MODE_TERMINATED)
401           return;
402
403         //get the next device.
```

```
404         dev = myDeviceList.getNextWhiteUnknownSignal();
405       }
406     }
407
408     /**
409      * Do a Service discovery on each Device in the devicelist,
410      * that is neither a BLACK nor a WHITE element.
411      * @throws d619a.client.bridge.bluetooth.exceptions.
             BTDisabledException Bluetooth disabled
412      */
413     private void surveyUndecidedDevices() throws BTDisabledException{
414       //do a service scan on each of the white devices.
415       //if a device does not respond, we change its signal state and
416       //delete the device from the providerlist.
417       RemoteDevice dev = myDeviceList.getNextUndecided();
418       while(dev != null){
419         this.startServiceDisco(dev);
420
421         //wait for this discovery to end
422         waitWhileScanning();
423
424         //if the SearchUnit mode has changed to interaction, return
               immediately
425         if(currentMode == SU_MODE_INTERACTION || currentMode ==
               SU_MODE_TERMINATED)
426           return;
427
428         //get the next device.
429         dev = myDeviceList.getNextUndecided();
430       }
431     }
432
433
434     /**
435      * Wait non-blocking for the inquiry or the service scan to
             complete.
436      */
437     private synchronized void waitWhileScanning() {
438       while(this.currentMode != SU_MODE_TERMINATED && this.isScanning()) {
439         try {
440           wait();
441         } catch (InterruptedException e) {}
442       }
443       isScanning();
444     }
445
446     /**
447      * Is a thread actively scanning or inquirying?
448      * @return true if yes
449      */
450     private boolean isScanning(){
451       return bIsScanning;
452     }
453
454     /**
455      * Set true if a thread currently is scanning or inquirying
456      * @param yesorno yes for true
457      */
458     private synchronized void setIsScanning(boolean yesorno){
459       this.bIsScanning = yesorno;
460       notifyAll();
461     }
462   }
```

Listing 2: SearchUnit.java: Implementation of the client search unit.

Appendix

Source Code

This CD-ROM contains the source code of DynaBlu. The Javadoc documentation can also be found online at [13].

The software has been tested using the following development tools. We have included prebuilt versions of the software using these tools on the CD-ROM.

- J2SE 1.6 (For the provider software).
- MIDP 2.0, CLDC 1.1 and JSR 82 (For the client software).
- Netbeans 5.5 IDE.
- Apache Tomcat 5.5.17 (As the application server).
- MySQL Community Server 5.0 (For the mediator component).
- The Avetana Bluetooth J2SE libraries have been tested on an Ubuntu 7.0.4 Linux installation having the BlueZ libraries installed.