ETH zürich

ETH Library

JXTA over Bluetooth

Master Thesis

Author(s):
Käppeli, Daniel

Publication date:
2003

Permanent link:
https://doi.org/10.3929/ethz-a-004595748

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Information and
Communication Systems
Research Group

JXTA over Bluetooth

Daniel Käppeli

Diploma Thesis

May 5, 2003 – September 4, 2003

Supervising Professor: Prof. Gustavo Alonso
Supervising Assistant: Andreas Frei

Contents

Preface 1

1 Introduction 3
- 1.1 Motivation 3
- 1.2 Messaging System 3
- 1.3 Demo Application 4
- 1.4 Definitions 4

2 Bluetooth 6
- 2.1 Basic Concepts 7
  - 2.1.1 Stack Overview 9
- 2.2 Device and Service Discovery 10
- 2.3 Security 11
- 2.4 A Brief Wireless Technology Comparison 11
- 2.5 Java Bluetooth Stack 12
  - 2.5.1 Stack Evaluation 13
  - 2.5.2 Rococo's Impronto Developer Kit For Linux 13

3 Messaging System 16
- 3.1 Overview 16
  - 3.1.1 Alternatives 16
  - 3.1.2 Proprietary Development vs JXTA 16
  - 3.1.3 Proprietary Development vs JXME 17
  - 3.1.4 Controllability 17
- 3.2 Architecture 17
  - 3.2.1 Connection Establishment 18
  - 3.2.2 Connection Management 18
  - 3.2.3 Data Processing 19
- 3.3 Messages 20
  - 3.3.1 Message Format 21
- 3.4 Addressing Devices 23

4 Benchmarking 25
- 4.1 Motivation 25
- 4.2 Stack Architectures 25
- 4.3 Test Environment 27
- 4.4 Results 28

5 Sample Application BAD2 **32**
5.1 What is BAD2 . . . 32
5.2 Appointment Protocol . . . 33

6 Future Work **36**
6.1 Scatternet . . . 36
6.1.1 Routing . . . 36
6.1.2 Device Discovery . . . 37
6.2 Security . . . 37
6.3 Reliability . . . 37
6.4 Multicast Groups . . . 38
6.5 Interconnection with JXTA/JXME . . . 38

A Objectives **39**

B Installing and Configuring BlueZ **41**
B.1 News . . . 41
B.2 Configuration on Linux/RedHat . . . 41
B.3 Installation . . . 41
B.3.1 iPAQ H3970 . . . 41
B.3.2 Installing BlueZ on RedHat 7.3 . . . 42
B.3.3 Installing BlueZ on RedHat 9 . . . 44
B.4 Start hcid and sdpd at Boot Time . . . 46
B.5 Environment Variables . . . 46
B.6 Set the user-friendly name . . . 47
B.6.1 Adapting the HCID configuration file . . . 47
B.6.2 Using the hciconfig tool . . . 47

C TCP Over Bluetooth **48**
C.1 TCP over Bluetooth . . . 48
C.2 Network Access . . . 48
C.3 Resources . . . 50

D Installing and Configuring Impronto v1.1 **51**
D.1 Impronto v1.1 for Linux . . . 51
D.2 Impronto v1.1 for ARM . . . 52
D.3 Java Bluetooth Programming Hints . . . 56
D.4 Environment Variables . . . 60

E Developing Java Bluetooth Applications **61**
E.1 Using Multiple Bluetooth Devices . . . 61
E.1.1 Developing with Eclipse . . . 61

F Enabling Bluetooth Security **63**

G Benchmarks **64**
G.1 Configuration . . . 64
G.2 Data Ascertainment . . . 64

H Compiling CLDC Personal Basis Profile **70**
H.1 Installing J2ME Personal Basis Profile 70
H.2 Compiling J2ME Personal Basis Profile 71

I Mail about Proxyless JXME **72**

Bibliograhpy **74**

List of Figures **76**

List of Tables **77**

Listings **78**

Preface

Abstract

This thesis is concerned with the development and implementation of an ad-hoc messaging system which uses Bluetooth channels to communicate. The implemented messaging system is completely written in Java and allows each device with a JSR-82 compliant Bluetooth stack to be a part of it. The messaging system is designed to support ad-hoc environments where participating devices join and leave very often. There is no network infrastructure required.

Additionally an application was developed to demonstrate the messaging system. These application is a small appointments book where groups can automatically find dates for meetings.

Chapter Overview

Chapter one includes a general motivation behind this thesis and a gives brief overview of the topic. Chapter two gives a general introduction to the Bluetooth Wireless Technology and discusses the Bluetooth stacks used in this thesis. Since there are several methods to setup Bluetooth enabled networks, some benchmarking efforts were made along with their performance analysis. These results are summarized in chapter three. Chapter four introduces the implemented messaging system and its principles and concepts. The developed demo application (BAD2) is introduced and explained in chapter five. BAD2 is an implementation of a rudimentary diary where several participants can make new appointments almost without any human interaction. Chapter six gives an overview of the future tasks.

Appendix A shows the objectives of this thesis as stated by the supervisor of this thesis. Appendix B is a BlueZ installation guide that shows the necessary steps to install and configure the BlueZ Bluetooth stack. Appendix C explains how to set up a TCP/IP network on top of the BlueZ stack. Appendix D shows how to install Rococo's Impronto Developer Kit for Linux. Appendix E gives some hints about programming Java Bluetooth applications using Eclipse. Appendix F shows how to enable the Bluetooth security features using the implemented messaging system. Appendix G gives some additional information about the benchmarking. Appendix H contains instructions on building and installing J2ME's Personal Basis Profile. Appendix I contains an E-mail written by Kuldip Singh Pabla that offers an opportunity to integrate the implemented

messaging system in the JXTA project.

Acknowledgment

First of all I am highly grateful to Doris who for the last few weeks has managed our household without any help from my part. Doris has given the much needed support during my studies, especially during the last four months. I am also grateful to my parents, sister, and brother for their silent support all these years.

I am grateful to all people who supported me in writing this thesis. I am especially thankful go to all people who proof read this thesis and gave helpful ideas. I am highly obliged to Andreas Frei who is the supervisor of this project and who gave always the much needed support. I am also grateful to Prof. Gustavo Alonso and his entire team.

Last but not least I thank Leo for patiently and promptly answering all my questions.

Chapter 1

Introduction

1.1 Motivation

Nowadays devices are getting smaller and their interconnections are getting more complex. Computers are becoming wearable and follow the humans to all possible places. One medium to interconnect such small and mobile devices is Bluetooth. Bluetooth is suitable for Personal Area Network (PAN) communication.

Another trend is the development of event-oriented-messaging systems. The vision of this thesis is to bring together these two technologies, namely Bluetooth and messaging systems. The goal is to create a messaging system which allows such small and mobile devices to talk to each other in an ad-hoc environment. A messaging system, which allows also limited devices to be a part of such a communication system, has been designed. The implemented messaging system does not require any additional infrastructure such as a Network Access Point (NAP).

Bluetooth competes against Infrared (IrDA) and Wireless Local Area Network (WLAN) in terms of PAN networking. Since IrDA connections are limited to two devices with a direct line of sight, IrDA is unqualified to build such a messaging system. Bluetooth Wireless Technology is nowadays in widespread use and many different devices contain built-in Bluetooth units such as Handhelds, Mobile and Smart Phones, laptop computers, etc. Usually such devices do not have a WLAN interface and to integrate them in a network the Bluetooth technology is a key factor.

The team leaded by Prof. Gustavo Alonso (ETH Zurich) has expert knowledge in dealing with LAN and WLAN based messaging and event systems (please refer to [4]). So it is obvious to launch a project analyzing Bluetooth as an alternative technology.

1.2 Messaging System

The goal of this diploma thesis is to develop a multi-hop messaging system where the participating devices are connected with a Bluetooth channel. The objectives of this thesis is given in Appendix A.

Before we decided to implement a new messaging system other alternatives were analyzed and technical know-how have been acquired. Such alternatives are existing peer-to-peer messaging systems such as JXTA and JXME.

There are also some different methods to establish a Bluetooth based connection. To compare the performance of these methods some benchmarking tests were executed, the results are summarized in Chapter 4. During a two-month period of technical research, all these different alternatives were analyzed. At the end of the day a decision was made to implement our own dedicated messaging system based on a Java Bluetooth stack. The reasons for this are illustrated in Chapter 3.

The messaging system provides message exchange within a Bluetooth piconet. The messages, which are generated and transmitted, are compatible to the messages defined in the JXME project. So it is possible to interconnect these two systems in the future.

1.3 Demo Application

Based on the above described messaging system a demo application was developed. This demo application is called BAD2 (Bluetooth Ad-hoc Distributed Diary). BAD2 is a rudimentary appointment calendar and the core feature of it is its mechanism to insert appointments. Within a group of people it is often troublesome to make a new appointment where everybody has a free slot. BAD2 finds such a slot and fixes up an appointment. This date and the corresponding task are automatically inserted in the diaries of all participants. The details of BAD2 are given in Chapter 5.

1.4 Definitions

This section contains some definitions most of them are explained also later on in the corresponding chapters.

The nodes participating in the implemented messaging system have three names corresponding to the layer of abstraction being looked at. The lowest level comprises of *masters* and *slaves*. These roles are located on the Bluetooth's connection level. A more abstract sight of a Bluetooth network depicts some devices offering services (*servers*) and some others consuming the services (*clients*). The highest level *peers* and *rendez-vous peers* are depicted. These devices offers and consumes a special service that is related to the implemented messaging system.

Master: Simple Bluetooth networks have a star topology. The Bluetooth device in the center of such a star topology is called a master. The master is the "manager" of such a simple network. Details are described in Section 2.1.

Slave: A slave is a Bluetooth device that is connected to a master. It is located at the periphery of a Bluetooth network. Please refer to Section 2.1 to get more information.

Server: A server is a Bluetooth device offering a service to other Bluetooth devices.

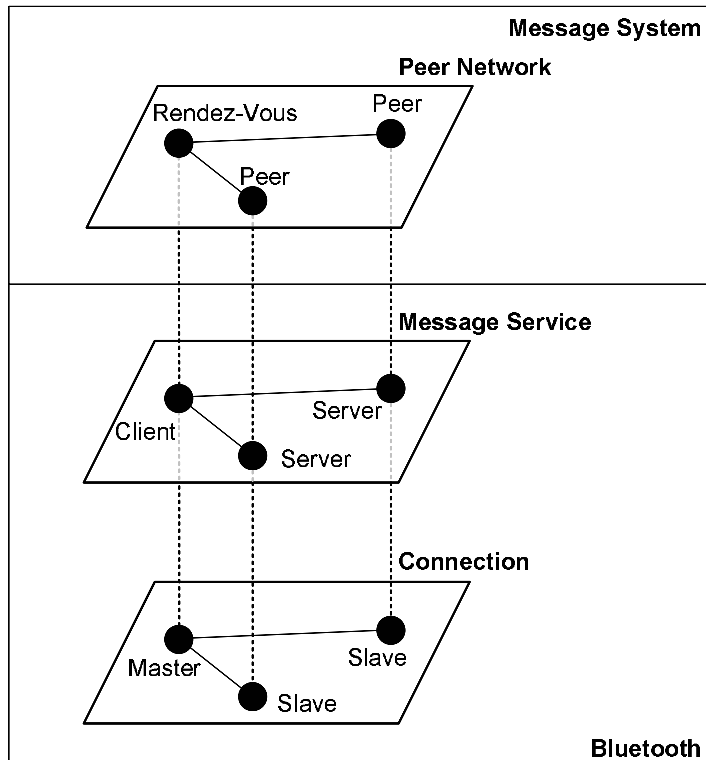
Client: A client is a Bluetooth device consuming another device's service.

Peer: A peer is a host that is a part of the peer-to-peer system. In our implementation of a messaging system each peer has to be connected to a rendez-vous peer.

Rendez-Vous Peer: A rendez-vous peer is interconnecting different peers to a peer-to-peer system. Basically each device can be a rendez-vous peer.

The definitions above and their corresponding layers are illustrated in Figure 1.1. In principle the defined layers are independent of each other but some configurations offer huge advantages. The configuration showed in Figure 1.1 provides such an advantage and also describes the implemented messaging system (see Section 3.2.1).

Figure 1.1: Illustration of the Terminology at Different layers



Chapter 2

Bluetooth

The history of Bluetooth dates back to 1994 when Ericsson launched an initiative to study a low-power, low-cost radio interface between mobile phones and their accessories. In 1998 they came out with the first Bluetooth chip and in the same year the Bluetooth Special Interest Group (SIG) was founded by Ericsson, IBM, Intel, Nokia and Toshiba. The SIG published the Bluetooth specification version 1.0 in 1999 [3], specification 1.1 in 2001 [2], and specification 1.2's release is announced for autumn 2003 [5].

The goal of Bluetooth developers was to replace cable as a communication medium. The wireless Bluetooth technology is much more flexible than the up to now used cable and infrared communication. It has been developed to allow ad-hoc short range communication between different kinds of devices. Bluetooth uses a very low transmission power of about 2.5 mW, which allows operations over distances up to 10 m. The Bluetooth specification allows the transmission power to increase up to 100 mW, which enables the device to operate over distances up to 100 m [16]. Table 2.1 shows the respective maximum output power versus the distance range.

Table 2.1: Bluetooth Radio Power Classes [11]

Power Class	**Max Output Power**	**Range**
Class 1	100 mW	100 meters+
Class 2	2.5 mW	10 meters
Class 3	1 mW	1 meter

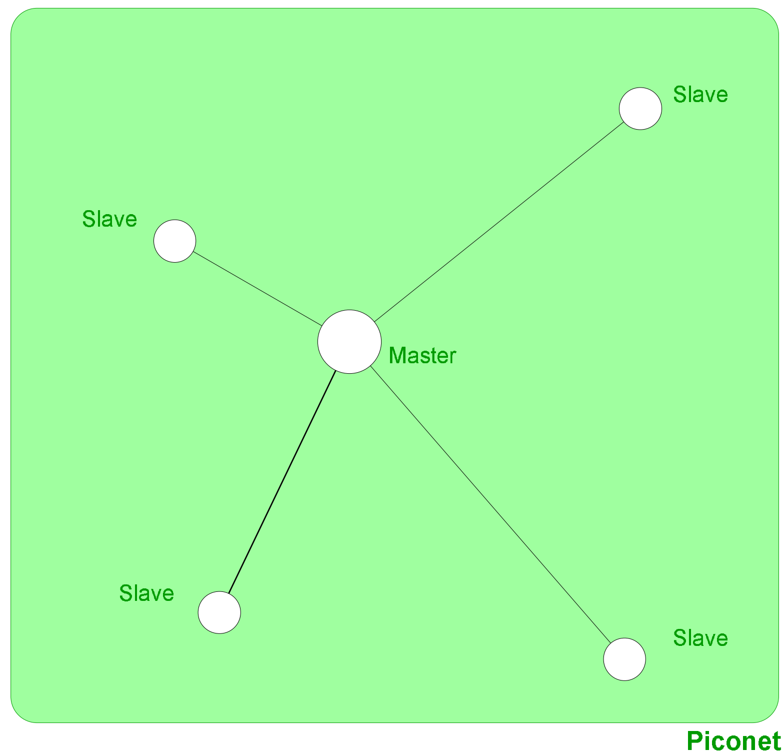
Another aim of Bluetooth is to keep the price as low as possible. The vision that the developers have is to produce a Bluetooth chip which costs around $5 USD only. Currently such a chip costs around $20 USD [1] and Ericsson plans to produce chips that will cost around $10 USD.

2.1 Basic Concepts

Each Bluetooth device has its own unique 48 bit IEEE MAC Bluetooth device address (BD_ADDR), which identifies it unambiguously to other devices. Usually the BD_ADDR is divided into six parts separated by a colon where each part consists of two hexadecimal digits.

To avoid collisions between different senders (interference) and for security reasons, a frequency hopping mechanism is implemented. Hence the frequency of transmission changes 1600 times per second. The base band is divided into 79 different channels that are used in a pseudo random sequence by the frequency hopping algorithm. Each pair of two connected devices needs a "synchronous" clock and a common hopping sequence. The device giving the frequency hopping sequence and its clock is called the *master* and the other device beeing connected to is called *slave*. The master assigns time slices to its slaves which they can use to send data. By default the device initializing the connection establishment process is the master device. Bluetooth stacks such as BlueZ (see Section 2.5.2) also offer the possibility to configure devices to be always master. A master can have up to seven connected slaves, such a network is called a *piconet* (see Figure 2.1).

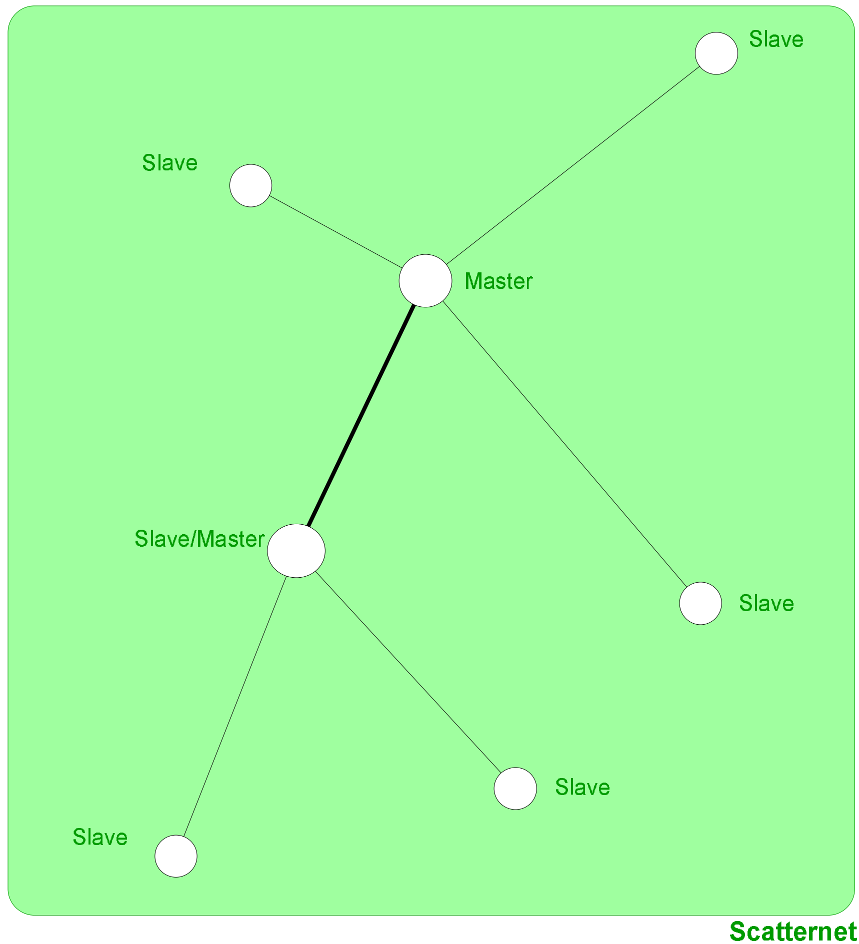
Figure 2.1: Piconet Overview



It is possible to combine multiple piconets to a so called *scatternet*. To build a scatternet one device has to be a member of two piconets. Such an intercon-

necting device may be a master of one piconet and slave of another or it can be a slave in both nets. However it cannot be master of both nets, since in this configuration both nets would have the same clock, same hopping sequence, and the same master, which is by definition a single piconet. Figure 2.2 illustrates a scatternet having an interconnecting master/slave device.

Figure 2.2: Scatternet Overview



Setting up scatternets is a new topic even though the principle is already known since years. Before setting up scatternets two preconditions must be satisfied.

- The used hardware and firmware must be scatternet enabled.
- The Bluetooth stack has to provide the necessary support.

It is very difficult to determine which devices and stacks are scatternet enabled. For this thesis the BlueZ Bluetooth stack has been utilized since it supports scatternets[1]. BlueZ is an open source project and can be installed on Linux

[1]as read on the posted messages in the corresponding news groups

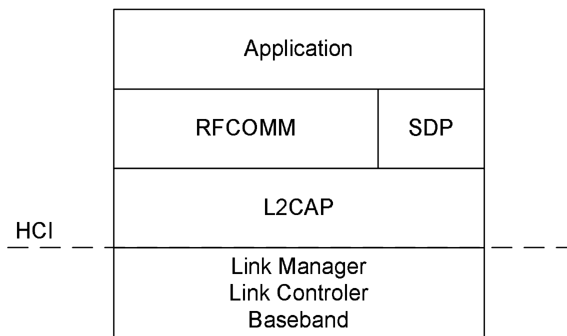
driven hosts.

The next topic is to verify the Bluetooth device because the hardware and firmware also need to be scatternet enabled. A lot of commercial products, such as PDAs, laptop computers and Bluetooth dongles, have chips manufactured by Cambridge Silicon Radio (CSR). In using CSR based devices the critical factor is the firmware installed on the device. A firmware of version number 16.4 and more supports the building of scatternets. The firmware version 16.4 was released in November 2002 [8] and we found the first devices in the market in July 2003. We didn't find any information about the chip sets of other manufacturers such as Ericsson, Silicon Wave, and Broadcom but it was reported[2] that some Ericsson modules (Ericsson ROK 101 007) also support scatternets. The conclusion reached is that building scatternets is certainly feasible but being a "brand new" technology it has not been analyzed in this thesis.

2.1.1 Stack Overview

This section will give a brief overview of the Bluetooth protocol stack. This overview is not all-embracing and only details the topics that are relevant in reference to this work. The complete stack is described in [2]. The Bluetooth specification specifies the functionality, instructions, and the corresponding instruction set that a Bluetooth chip must offer. The specification also contains software layers and protocols such as L2CAP[3], RFCOMM[4], Service Discovery (SDP), and Device Discovery. Figure 2.3 illustrates the simplified Bluetooth stack and its layers. The layers shown below the Host Controller Interface (HCI), Baseband and Bluetooth Radio, are implemented in the hardware, whereas the layers above this interface are implemented in the software.

Figure 2.3: Bluetooth Protocol Stack Overview



The layers below the HCI (Radio Baseband, Link Controller, and Link Manager) format the over-air transmission, handle error detection and re-transmission,

[2]Matthias Ringwald, Distributed Systems Group, ETH Zurich, July 2003
[3]Logical Link Control and Adaptation Protocol
[4]This name is composed of Radio Frequency (RF) and COM according the well-known COM port

as well as manage the links between different devices [11]. To learn more about these layers please refer to [2].

The first software layer is the L2CAP layer. L2CAP has to care about managing the different connections and channels as well as generating data packets out of data streams and vice versa.

The RFCOMM and the SDP base on the L2CAP layer. RFCOMM is a serial connection over the air.

> "RFCOMM emulates full 9-pin RS232 serial communication over an L2CAP channel. It is based on the TS 07.10 standard for software emulation of the RS232 hardware interface." [11]

Service Discovery Protocol (SDP) is a service that allows to find services offered by the remote device. SDP does not offer any opportunity of data transmission to any applications.

> "The SDP is not designed to interface to an existing higher layer protocol, but instead addresses a specific requirement of Bluetooth operation: finding out what services are available on a connected device. The SDP layer acts like a service database. The local application is responsible for registering available services on the database and keeping records up to date. Remote devices may query the database to find out what services are available and how to connect to them." [11]

2.2 Device and Service Discovery

To be able to create connections to other devices, one needs to know which devices are within the range of one's Bluetooth device. To discover those devices, a so-called *inquiry* process is executed. Since the inquiry is a non-deterministic process it is possible that devices within the scanning range may not be detected. Also all devices, having a firmware less than version number 16.4, that are already connected and playing the role of a slave, are not discovered by the inquiry. After the inquiry the devices in the neighborhood are known but there is no knowledge about the services they offer. To find out what services are offered by these devices, a *service discovery* process is executed. Each local application deploys its services by registering them with the local *SDDB* (Service Discovery Database). A remote device searching for services will establish a L2CAP connection with the SDDB and will ask the attributes of the offered services. At the end of the service discovery the resulting information is used to create a connection. Services are basically identified by a UUID (Universally Unique Identifier).

> "A UUID is an Universally Unique Identifier that is guaranteed to be unique across all space and all time. UUIDs can be independently created in a distributed fashion. No central registry of assigned UUIDs is required. A UUID is a 128-bit value." [2]

2.3 Security

Bluetooth has three integrated security mechanisms namely *authorization*, *authentication*, and *encryption*. The applied security can be divided into three levels: *none*, *authentication*, and *encryption* where authentication requires authorization, and encryption requires an authentication.

After the inquiry process has been terminated a device may start establishing a connection. We have to differ the connection establishment between two different cases. The first case describes two devices which have already been connected and the second case devices that have not yet been connected. Basically the authenication is based on the link key (K_{link}). The link key is a shared secret that is only known by the two devices having a connection. Every device needs some memory to store the link keys. In general each connection between any two devices has its own and randomly calculated link key. The first time the two devices are connected a so-called *initialization key* (K_{init}) is generated based on a PIN (Personal Identification Number). This process is called *pairing*. After the pairing, the authentication process is started and this process is the same for all connection establishments. The authentication process is based on a challenge/response mechanism. To verify the link key or the initial key in case the connection is established the the first time, the verifier sends a random number to the claimant. Both devices apply the link key to this random number and compare the results. If both have the same result then the claimant is verified. After the initial key has been verified a link key is generated and semi-permanently stored on these two devices.

The link between two devices may be encrypted. By default there is no encryption applied. To enable encyption any device can request it at any point of time during the connection. Encryption has to be requested each time a new connection is established. Authentication is a precondition for encryption. For each session a new key is generated based on the link key and a random number.

Having very small devices without any memory resources the security concept is vulnerable. Such devices have a hard coded PIN code which don't resist brute force attacks. Another vulnerability is the very limited memory they have. Often such a device is able to store only one single link key. Devices without the necessary memory to store the different link keys, can use their own private unit key (K_{unit}) instead of a randomly generated link key. Each device ever been connected to this small device (A) has the same link key that corresponds to A's unit key. At the end of the day each device ever been connected to A is able to eavesdrop A's network traffic.

2.4 A Brief Wireless Technology Comparison

Often Bluetooth is compared to other wireless technologies such as Wireless Local Area Network (WLAN) and Infrared Data Association (IrDA). Table 2.2 gives a short overview of the different technologies, their advantages and disadvantages.

Table 2.2: Brief Comparison of IrDA, Bluetooth, and WLAN

Topic	IR[20]	Bluetooth	WLAN[16]
Range	1 m	10 m	300 m
Bandwidth	4 Mbps	1.1 Mbps	11 Mbps
Power Consumption	low	low	high
Device's Size	small	small	large
Circumference of Transm.	30° cone	omni-directional	omni-direct.
Device Discovery	—	yes	—
Service Discovery	—	yes	—
Security	—	yes	yes
Price	$1 USD	$20 USD[20]	n/a

2.5 Java Bluetooth Stack

To develop Java based Bluetooth applications, a Java enabled Bluetooth stack is necessary. Since the Bluetooth specification does not care about the interfaces provided by the stacks there is no standardized API neither in Java nor in any other programming language. This jeopardizes the "Write Once Run Everywhere" principle of Java which is indeed one of the most important ones. In the years 2001 and 2002 the Java Community Process[5] (JCP) designed the JSR-82[6] (Java Standard Request) that defined such a Java Bluetooth API that is also called Java API for Bluetooth Wireless Technology[7] (JABWT). However, there exist also a couple of Java Bluetooth stacks that do not fulfill JSR-82 such as Harald[8], JBlueZ[9], etc. Stacks not fulfilling the JSR-82 have not been analyzed in this thesis since the author is convinced that the number of devices fulfilling JSR-82 will rapidly increase within the next few years. Because the Java Bluetooth API is an optional package of J2ME (Java 2 Micro Edition) there will be many more JSR-82 enabled JVMs in devices as PDAs and Smart Phones in the future. At present that is Aug-2003, there are no such devices in the market but a few cell phone manufacturers such as Nokia, Motorola, and Ericsson have announced mobile phones that implement JSR-82. Those phones will be in the market in the third quarter of 2003.

Analyzing the stacks that are in the market at the moment (Aug-2003) there are two types of Java/Bluetooth stacks. Most of the available Java stacks are based on a native Bluetooth stack and simply wrap the native stack's functionality within a Java interface, e.g. Rococo's Impronto, JBlueZ, BlueJava, etc. On the other hand there are pure Java stacks that are completely implemented in Java, e.g. Harald, Atinav, Esmertec's Jbed, etc. Many of these products are described as "platform independent" and "totally implemented in Java"

[5] http://jcp.org
[6] http://jcp.org/en/jsr/detail?id=082
[7] http://www.jabwt.com
[8] http://www.control.lth.se/∼johane/harald
[9] http://jbluez.sourceforge.net

but they also bring along their own dedicated JVM. These stacks target OEMs of those products that are Bluetooth enabled and need to be equipped with a JVM. These stacks cannot be applied to Sun's VMs.

2.5.1 Stack Evaluation

The most important criteria in selecting a stack, is its compatibility with the JSR-82 standard. Because this makes the developed code independent of the underlaying system and it can be widespreadly used, "Write Once Run Everywhere". Other important criteria are price and availability. The availability is another important critera, many vendors sell their products only to OEMs (Original Equipment Manufacturer) such as Atinav[10] and Esmertec[11]. Some other stacks, which come along with their own dedicated hardware such as Zuccotto[12], can cost around $3000 USD. There are also a few open source projects to develop an implementation of JSR-82 on SourceForge[13], but none of them equaled an usable status. Either they are still at the planning stage or the projects have never released any software. At the end of the day we have decided to use Rococo's[14] Impronto Developer Kit for Linux which fulfills the JSR-82 standard and is free for private use and universities. Otherwise it costs around 2500 EUR. Table 2.3 gives a brief overview of the analyzed stacks.

2.5.2 Rococo's Impronto Developer Kit For Linux

Rococo's Impronto Developer Kit for Linux is a Java wrapper that sets up on top of the BlueZ Bluetooth stack. To see how to install Impronto Developer Kit on a Linux box please refer to the Appendix D.

> "Impronto for Linux can be utilized as an optional package with many profiles as it is based on the smallest subset of Java configurations, CLDC (Connected Limited Device Configuration). CLDC is a subset of the CDC (Connected Device Configuration)" [17]

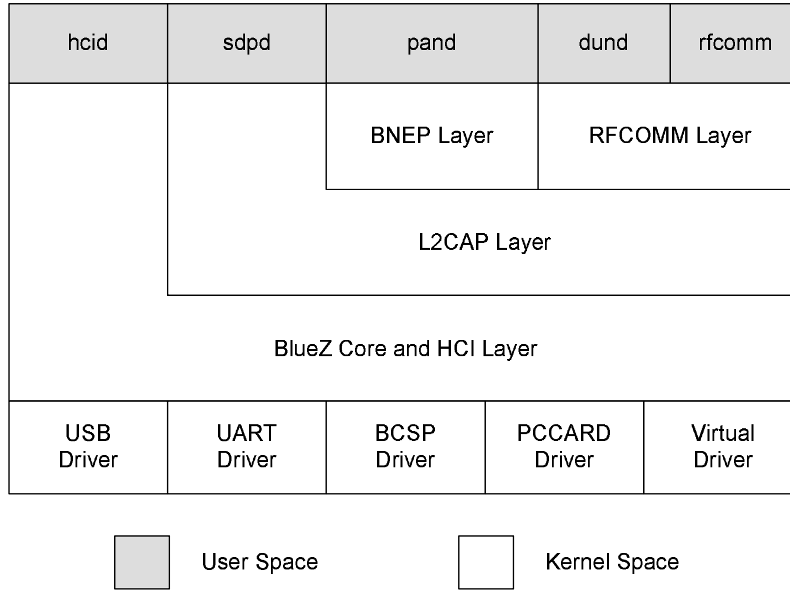
To execute a Java Bluetooth application the delivered JAR file (*idev_bluez.jar*) and the directory containing the license (`LinuxLicense.txt`) have to be included in the classpath. After installing Impronto the JAR file can be found in the following directory: `/usr/share/java`. The license is delivered separately by E-mail. To execute a Java/Bluetooth application the classpath must be defined as described above and the library `/usr/lib/libimpronto.so` has to be included in the library path of the host system (see Appendix D for more details).

[10] http://www.atinav.com
[11] http://www.esmertec.com
[12] http://www.zuccotto.com
[13] http://www.sourceforge.net
[14] http://www.rococosoft.com

BlueZ

BlueZ is the official Linux Bluetooth stack that is written in C/C++. Figure 2.4 gives an overview of the parts of the BlueZ Bluetooth stack relevant to this thesis. This stack supports a huge number of devices that are in today's market.

Figure 2.4: Fragment of the BlueZ Bluetooth Protocol Stack Overview [9]



To install BlueZ, the kernel of the target system must be patched and recompiled (please refer to Appendix B to get the detailed installation guide). BlueZ is compatible with most commercial Bluetooth modules. A list of compatible devices can be downloaded from Marcel Holtmann's web page [6].

> "In general all devices with a CSR, Broadcom or AVM chip will work fine. Others will work too, as long as they are compatible to the H:2 specification[15]." [7]

[15] see [2], Part H:2: HCI USB Transport Layer

Table 2.3: Brief Comparison of Different Stacks

Product	**JSR-82**	**J2SE**	**J2ME**		**Pure Java**	**ARM v5**	**OEM only**	**Price**
			CLDC	**CDC**				
Rococo Impronto v1.1	•	•	•	•	no	•	no	free
Rococo Impronto v1.0	•	•	—	•	no	—	no	free
Atinav	•	•	•	•	yes	n/a	yes	n/a
Zuccotto	•	•	•	•	n/a	n/a	no	$2995 USD
Esmertec's jbed	•	•	•	•	yes	n/a	yes	n/a
Harald	—	•	—	—	no	—	no	free

Chapter 3

Messaging System

3.1 Overview

The messaging system we implemented is able to operate in an ad-hoc environment without any network infrastructure. Such a system is qualified to build the transport layer for event systems as described in [4]. This section gives an overview of the different alternatives available as well as of our implementation of such a message system.

3.1.1 Alternatives

Before we decided to implement our own dedicated messaging system other alternative solutions were analyzed. These alternatives include JXTA and JXME, where JXME is a small footprint implementation of JXTA. The following sections point out the reasons to implement a new messaging system.

JXTA and JXME are both based on HTTP which requires TCP/IP. Theoretically it is possible to setup TCP/IP over a Bluetooth channel (see Section 4.2) but there are some general problems in it which we could not resolve, namely routing and bridging [21]. Each connection established using BNEP creates a virtual network interface. These different interfaces need to be connected to each other and to other interfaces such as wired or wireless LAN. Bridging is typically used to interconnect subnets of the same network and routing is used to interconnect different networks. To use routing to interlink these network interfaces one needs a huge configuration effort since each interface and the corresponding the routing tables have to be configured. However the usage of bridging is simpler in terms of configuration but caused serious crashes on the used RedHat 9 systems. The installation and configuration is documented in Appendix C.

3.1.2 Proprietary Development vs JXTA

JXTA is a full featured peer-to-peer software based on TCP/IP and uses XML to transmit data. To learn more about JXTA see [14]. JXTA has high memory and CPU requirements and is therefore inapplicable to small devices such as

Personal Digital Assistants (PDA) and Smart Phones. JXTA is not the way to go to integrate these small devices in a messaging system.

3.1.3 Proprietary Development vs JXME

JXME is a small footprint implementation of JXTA. It is designed to operate on very limited devices such as PDAs and Smart Phones with a Java 2 Micro Edition (J2ME) platform. The main handicap of JXME is its need to have a JXTA proxy, which makes an ad-hoc usage of JXME impossible since this requires a JXTA proxy server and TCP/IP network infrastructure. Two devices that are places next to each other do not communicate directly. To allow such a direct communication both devices need a HTTP server. This is too heavy weight for such a small device. These HTTP servers are required since such a message system communicates asynchronously.

3.1.4 Controllability

Comparing TCP/IP over Bluetooth to L2CAP or RFCOMM, TCP/IP has another disadvantage, the controllability. Using L2CAP or RFCOMM it possible to use the service and device discovery mechanisms offered by Bluetooth which are lost using TCP/IP. The JSR-82 interface includes mechanisms to take influence to the process of connection establishment. One can think about implementing filter that blocks the connection to given devices based on the device's name or address, or based on the offered services.

Hence a decision was made to implement a new system that allows message exchange within a system of small and ad-hoc inter-networking devices. The following chapter will give an overview of the implemented system which is inspired by the JXME system.

3.2 Architecture

The architecture of the system is designed to support message distribution within a piconet. During the phase of research and design, there was no a scatternet enabled device available, so it was not possible to analyze it. A piconet is composed of maximum eight active Bluetooth devices, where there is one master and up to seven slaves (see Section 2.1).

A node participating in the messaging system is called a peer. There are two different kinds of peers: the rendez-vous and regular peers. Each peer must have a direct connection to a rendez-vous peer. Basically there are two types of rendez-vous peers: Threaded and non-threaded rendez-vous peers. A threaded rendez-vous peers, which search automatically for regular peers and establish connections to them. Whereas having a non-threaded peer the application using the messaging system has to initiate the process of connection establishment by calling the peernetwork's *connect()* method.

The architecture of the messaging system can be divided into three parts: connection establishment, connection management, and data processing. The

parts connection management and data processing are the same for all peer types, the connection establishment depends on the type of peer chosen.

3.2.1 Connection Establishment

In a piconet obviously the master node should also be a rendez-vous peer. The simplest way to give the master role to a device is by letting it initiate the connection establishment. In other words this device has to pass through the device and service discovery procedures, followed by establishing the connection. An alternative way to reach this arrangement is to change the HCI daemon's configuration file (see Appendix B), which requires super user privileges. The HCI daemon must be restarted after having modified the configuration.

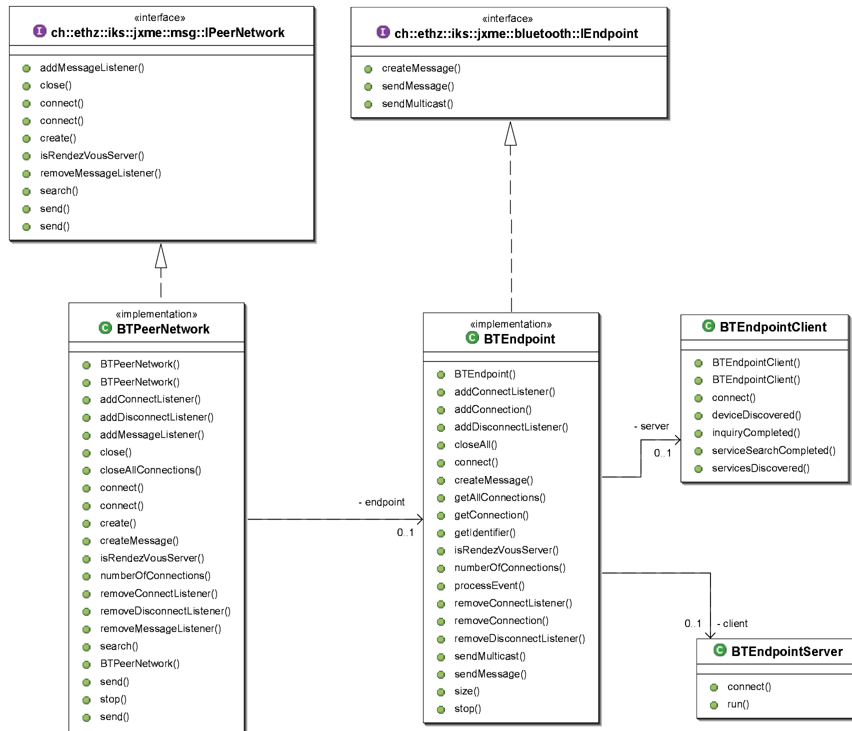
To avoid frequent reconfiguration the master of the piconet should initiate all connections. The master should also be a rendez-vous peer interconnecting the adjacent peers. The master has to take care about inquiry and service discovery while the regular peers simply need to deploy a well known service. In other words the rendez-vous peer invites interested peers. In terms of Bluetooth: Each regular peer is a server and the rendez-vous peer is a client since it consumes the services offered by the peers. The Bluetooth Java API identifies services by an UUID (see [10]). The UUID value of our messaging system service is UUID generated by the following seed: `8800`. In addition to the service the underlaying transport protocol is important. We use RFCOMM connections (see Chapter 2). Based on these RFCOMM connections input and output streams are created to provide data transmission. The establishment of connections is done by the following classes *BTEndpointClient* and *BTEndpointServer* of the package *ch.ethz.iks.jxme.bluetooth.impl* (see Figure 3.1). Connections are not related to the class that created them (*BTEndpointClient* or *BTEndpointServer*). After the establishment process a connection is stored in a connection pool managing all of them. Two nodes are connected by exactly one connection. Starting a rendez-vous peer there are two possibilities to create connections. Automatically by starting a dedicated thread or manually by calling the peer's (*BTPeerNetwork*) *connect()* method. The automated process expects a timeout value specifying the time to wait between two inquiry processes and must not be started by calling the *connect()* method. Applying the non-automated procedure the call to the method *connect()* is blocking until the inquiry and service discovery processes are completed. This can take around half a minute or even longer.

When a connection is established an associated reading thread will be created and started. Its task is to listen on this given connection for incoming data.

3.2.2 Connection Management

All connections and the corresponding reading threads are managed by the connection pool using a wrapper class (*BTReadThreadConnectionHandle*). The connection pool offers access to the connections by an identifier (see Section 3.4). Broken connections will be removed automatically from the connection pool.

Figure 3.1: UML Diagram illustrating a PeerNetwork



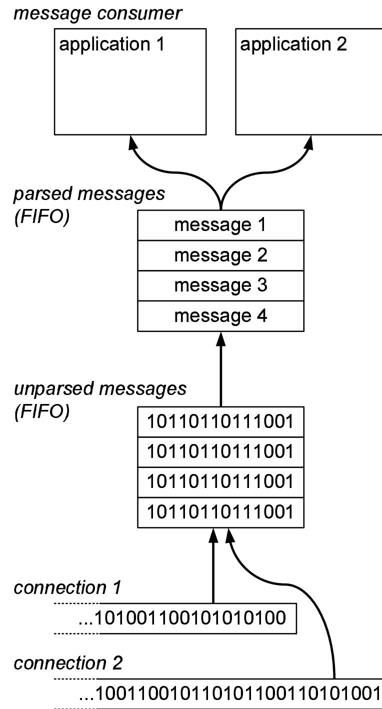
Every time a connection is added or removed to the connection pool, a corresponding event is generated (*ConnectionEvent*). A client application can register its implementation of the corresponding listener's interface (*IConnectListener* resp. *IDisconnectListener* defined in the package *ch.ethz.iks.jxme.bluetooth*) to be notified.

3.2.3 Data Processing

Incoming data is processed by the reading thread extracting messages out of the data stream. That raw data is stored within a queue. A parser thread processes these data to create a message objects (*IMessage*). These objects are then stored in another message queue. The thread operating on this second queue has to notify all registered listeners (*IMessageListeners*) about the new message. Any customer application may implement it's own message listener and register it to the peer network. The processing of incoming messages is illustrated in Figure 3.2.

Outgoing data is transmitted immediately. The *send* method blocks until the message is transmitted. The outgoing data flow is illustrated in Figure 3.3.

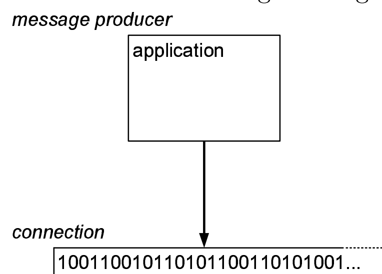
Figure 3.2: Data Processing of Incoming Data



3.3 Messages

A message is composed of any number of elements. A message is a container for such elements. Each element has a name, some data, a namespace, and a MIME type (Multi-Purpose Internet Mail Extensions). The messages delivered to a client application are defined according to the interface *IMessage* of the package *ch.ethz.iks.jxme.msg* and the corresponding elements *IElement* defined in the same package (see Figure 3.4).

Figure 3.3: Data Processing of Outgoing Data



Listing 3.1: Sample Message

```
1  jxmg 0 02 11 temperature 07 address 03
2  jxel 3  0 06 sender 12 009988776655
3  jxel 3  0 08 receiver 12 112233445566
4  jxel 2  1 12 TestMimetype 08 sensor10 2 4C
```

3.3.1 Message Format

The format of the messages is based on the message format defined in the JXME project [15]. The specified format is a binary format because parsing XML messages is very CPU power and memory intensive. Listing 3.1 illustrates how such a message can look like. Real messages do not contain any line numbers, any spaces, or any line breaks. These have been added to make the message code more readable. A message consists of a header (see Listing 3.1, line 1) and an arbitrary number of elements (see Listing 3.1, lines 2 – 4). The header includes information about the message version, namespaces, and the number of elements contained in this message. To see detailed information about the message's header see Table 3.2. The body of a message contains any number of elements storing the message's data. Each element is assigned to a namespace and a MIME type, has a name, and a payload section. The namespaces used in the elements are defined in the header of the message. Inside an element namespaces are referenced by a number representing the position of its declaration in the message's header. 0 and 1 are special namespaces (see Table 3.1) whereas 0 refers to empty namespace (no namespace defined) and 1

Figure 3.4: UML Diagram of a Message and its Elements

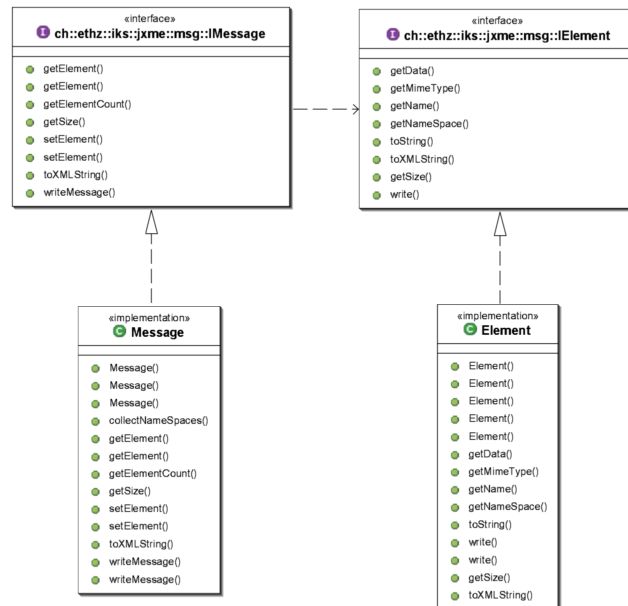


Table 3.1: Predefined Namespaces

Number	Namespace
0	empty namespace
1	”jxta”
2	first user defined namespace
⋮	⋮
n	last user defined namespace

refers to the “jxta” namespace. These predefined namespaces are well known and therefore not defined in the header section of a message. User defined namespaces start with number 2 and will increase following the position they are defined in the header of the message. Each element can also have a MIME type. The default MIME type (“**application/octet-stream**”) is indicated by the number 0. User defined MIME types are indicated by setting the MIME type flag to 1. If the MIME type identifier is set to 1 the declaration of the MIME type must follow to this flag. A MIME type is defined by its name's length encoded as a short integer (*short*, 2 bytes) and its name as an array of characters (*char[ ]*). The details of the element's format having the default MIME type is described in Table 3.3 and user defined MIME type in Table 3.4. Compare also line 2 and 3 to line 4 of Listing 3.1.

Table 3.2: Description of the Message Header of Listing 3.1, line 1 [18]

Header Part	Length	Java Type	Description
jxmg	4 bytes	char[ ]	message header identifier
0	1 byte	byte	message version
02	2 bytes	short	number of namespaces
11	2 bytes	short	length of namespace 2 in bytes
temperature	—	char[ ]	namespace 2
07	2 bytes	short	length of namespace 3 in bytes
address	—	char [ ]	namespace 3

The described message format complies with JXME's message format. We have defined an additional header during this project (see Figure 3.5). This header contains information about sender, receiver, the length, and the type of the message (unicast or mulitcast). The advantage of such an additional header is the fact that a message must not be parsed to be forwarded only. In the future it will be possible to add more data units to the header such as check sums, etc.

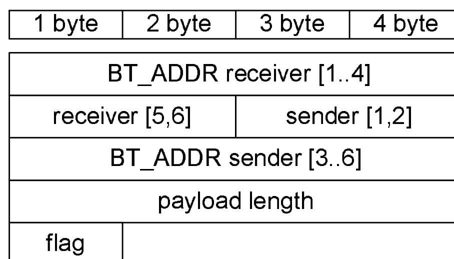
Table 3.3: Description of a Message Element having the Default MIME Type (see Listing 3.1, line 2)

Element Part	Length	Java Type	Description
jxel	4 bytes	char[ ]	element header identifier
3	1 byte	byte	identifier of the namespace
0	1 byte	byte	identifier of the mime type
06	2 bytes	short	length of element's name
sender	—	char[ ]	name of this element
12	4 bytes	int	length of element's data block
009988776655	—	char[ ]	element's payload

Table 3.4: Description of a message element having a user defined mime type (see Listing 3.1, line 4)

Element Part	Length	Java Type	Description
jxel	4 bytes	char[ ]	element header identifier
3	1 byte	byte	identifier of the namespace
1	1 bytes	byte	identifier of the mime type
12	2 bytes	short	length of mime type's name
testMimeType	—	short	name of the mime type
08	2 bytes	short	length of element's name
sensor10	—	char[ ]	name of this element
02	4 bytes	int	length of element's data block
4C	—	char [ ]	element's payload

Figure 3.5: Message Header



3.4 Addressing Devices

Each Bluetooth device has an unique MAC address but this address is neither handsome nor handy. There is another more user-friendly possibility to identify Bluetooth devices, the so called *user-friendly name*. The user-friendly name has a length of 248 bytes and is UTF-8 encoded [2]. See Appendix B how to set the user-friendly name using the BlueZ Bluetooth stack. It is the system ad-

Listing 3.2: Code Snippet of Class *BTConnectionHandle*

```
1 public String getIdentifier() {
2   try {
3     return _remoteDevice.getFriendlyName(true);
4   } catch (IOException e) {
5     LOG.fatal("..." , e);
6   }
7   return _remoteDevice.getBluetoothAddress();
8 }
```

ministrator's task to specify the device's unique name. It has to be taken into consideration that a system may have more than one connected Bluetooth devices whereas each of them needs to have its own unique name. By default each BlueZ driven device's name is "BlueZ(x)", where x is the number specifying the device's HCI. This number is given by the BlueZ software and is according to the order BlueZ is initializing the Bluetooth devices. The range of x is limited between zero and the number of Bluetooth devices minus one.

Using the JSR-82 API the device addresses are specified by a hexadecimal string (*java.lang.String*) of 12 characters (0..9A..F) without any separating colons or spaces. To get the Java compatible Bluetooth address out of a real device's address one simply removes the separating colons.

The messaging system is able to identify connections either by Bluetooth device address or user-friendly name but not both of them simultaneously. The default policy is using the user-friendly name and if this is not available the Bluetooth device address is used instead. This identifier can be changed by editing the source code of the class *BTConnectionHandle* in the package *ch.ethz.iks.jxme.bluetooth.impl* or by programming a new implementation of the interface *IConnetionHandle* in the package *ch.ethz.iks.jxme.bluetooth.impl*. To change the policy from user-friendly name to the Bluetooth address comment line 2–6 of listing 3.2.

The connections are stored in a connection pool which is implemented by a hash table (*java.util.Hashtable*). The key of the key-value pair is the identifier (here the user-friendly name) and the value is the connection itself. If there are two devices – both of them having the same identifier – the existing connection is overwritten by the newer one.

The all-zero address and a `null` value representing the address are reserved broadcast addresses. In other words to send a broadcast, call the send method (*send(String id, IMessage data)*) of class *BTPeerNetwork* in package *ch.ethz.iks.jxme.bluetooth.impl* setting the parameter *Sting id* to `null` or to "000000000000". The all-zero address is also defined as broadcast address in the Bluetooth Specification (see [2], page 51). This doesn't cause any problems since the Java API does not offer any methods to send mulitcasts out of Java. Sending messages to the broadcast address the messaging system worries about the distribution of the message and not the Bluetooth module as one might expect.

Chapter 4

Benchmarking

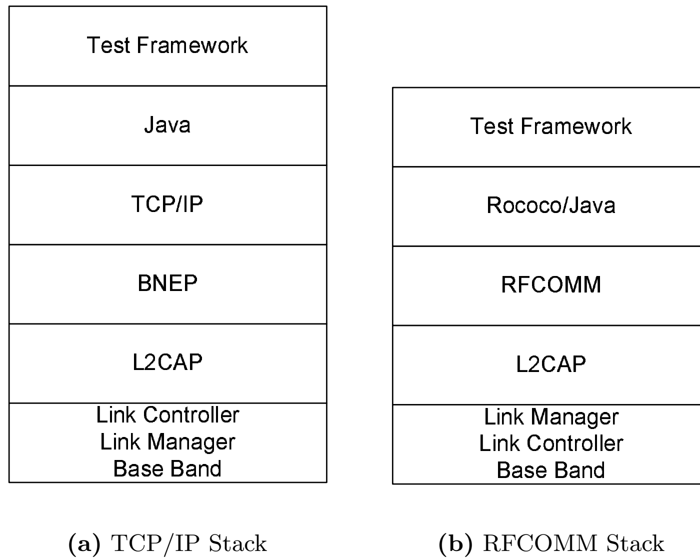
4.1 Motivation

During technical research we have found different architectures that allow Java applications to communicate over a Bluetooth channel namely TCP/IP over Bluetooth and Bluetooth RFCOMM connections (see also Section 3.1.1). We feared that TCP/IP slows down the performance of such a connection. To measure the loss of performance a benchmark environment was implemented. This benchmark environment consists of two interconnected devices, where one of them is a simple echo server that immediately returns any incoming data. The second device sends a predefined amount of data to the echo server and measures the time needed until it receives back the sent data. If the amount of data is very small, these tests behave like a ping measuring the round trip time. However by sending large amounts of data the bandwidth can be measured.

4.2 Stack Architectures

Since data transmission is the L2CAP's task the protocol stacks are identical up to this layer. The higher levels indeed are totally different. Looking at the TCP/IP stack (see Figure 4.1(a)) on top of L2CAP is the BNEP layer (Bluetooth Network Encapsulation Protocol) that implements the PAN profile (Persona Area Network). BNEP encapsulates the data packages of other network protocols such as IPv4, IPv6, and IPX and redirect these packages to the underlaying L2CAP layer [19]. On a Linux driven system each BNEP connection is represented by a virtual network interface. Such an interface can be configured like any other real existing network device, using tools such as `ifconfig`. A detailed description of setting up a BNEP based TCP/IP network is shown in Appendix C. On top of BNEP resides the TCP/IP layer which functionality is used by the test framework. Incidentally there is also an alternative way to set up TCP/IP over Bluetooth, that has not been analyzed in this thesis. It is possible to set up PPP (Point-To-Point Protocol) over RFCOMM and TCP/IP on top of PPP using the Bluetooth's Dial Up Network Profile (DUN).

On the other hand, Rococo's Java API allows accessing the RFCOMM layer which is on top of the L2CAP layer. The corresponding protocol stack



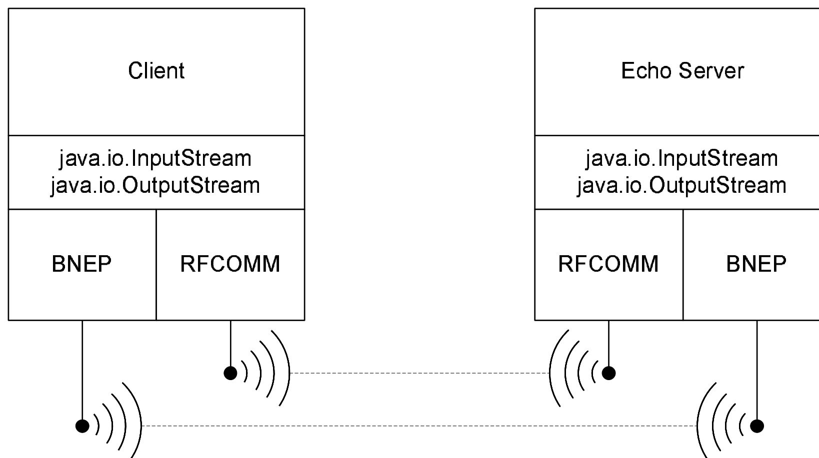
(a) TCP/IP Stack **(b)** RFCOMM Stack

Figure 4.1: Overview of RFCOMM and TCP/IP Stack

is illustrated in Figure 4.1(b). The L2CAP and the RFCOMM layers are part of the BlueZ Bluetooth stack that is described in detail in Section 2.5.2. On top of the BlueZ stack is Rococo's Impronto located which offers the stack's functionality to Java driven applications.

Basically the test framework's architecture is very simple. The interface to the core testing system is an input stream (*java.io.InputStream*) and an output stream (*java.io.OutputStream*). This architecture is illustrated in Figure 4.2. This design guaranties that the overhead produced by the test framework is always the same. The test framework consists of a server (*benchmark.Server-*

Figure 4.2: Architecture Overview of Benchmark Environment



Routine) and a client component (*benchmark.ClientRoutine*). The constructor of these classes requires an input and output stream, as well as some test data (*byte[ ]*) to be transmitted. To implement a test case a class has to be implemented which creates an input and an output stream and passes those to the test routine. The server routine simply reads all the incoming data from the input stream and writes them immediately to the given output stream. The client routine is a bit more complicated because it has to measure the round trip time. Before the first byte is transmitted the current system time (t_{start}) is captured and after the reception of the last byte the system time is captured again (t_{stop}). The resulting time t is the difference between the start and the end time (see Equation 4.1).

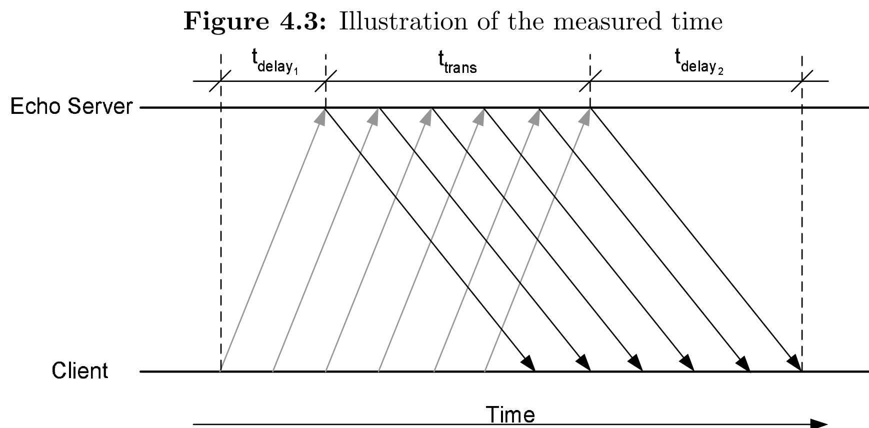
$$t = t_{stop} - t_{start} \tag{4.1}$$

This time t is composed of the delay from the client to the echo server (t_{delay_1}) and vice versa (t_{delay_2}), and the time needed to transmit the data t_{trans} (see Equation 4.2). Figure 4.3 illustrates Equation 4.2.

$$t = t_{trans} + t_{delay_1} + t_{delay_2} \tag{4.2}$$

On the client side the number of iterations is passed as another argument to the method *run()*, which invokes the benchmark routine. This method returns an array of long integers containing the duration of each transmission (t, see Equation 4.1) in milliseconds.

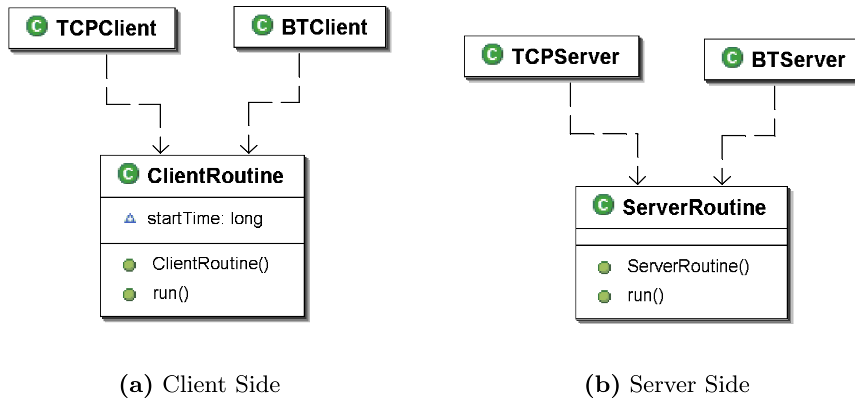
Figure 4.3: Illustration of the measured time



4.3 Test Environment

The distance between the two devices was maintained to one meter in all test cases. Transmission through walls or over distances of 10 meters and more may cause a loss in performance. The configurations of the used machines are given by Table G.1 in Appendix G. To avoid errors during the measurement, all other applications were closed and no cron job was executing on the test systems.

To be able to compare the performance of TCP/IP and RFCOMM, we implemented two different clients and servers. To test the RFCOMM connection the classes *BTClient* and *BTServer* of the package *benchmark* have been implemented. These classes crate an RFCOMM connection to each other. The establishment of the TCP/IP connection is done by the classes *TCPClient* and *TCPServer*. The concrete implementations of the client and server classes has to pass input and output streams to the *ServerRoutine* and the *ClientRoutine*. The dependencies are illustrated in Figure 4.4. It is obvious to see that the tests themselfs are always executed by the classes *ServerRoutine* and *ClientRoutine*.



(a) Client Side **(b)** Server Side

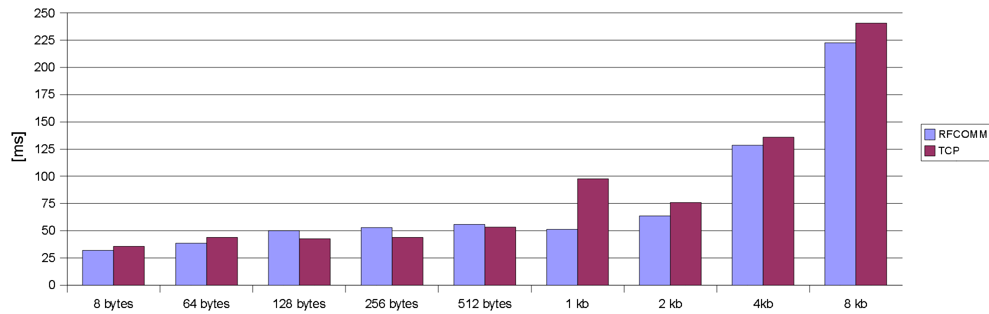
Figure 4.4: Dependency Graph of the Benchmark Environment

4.4 Results

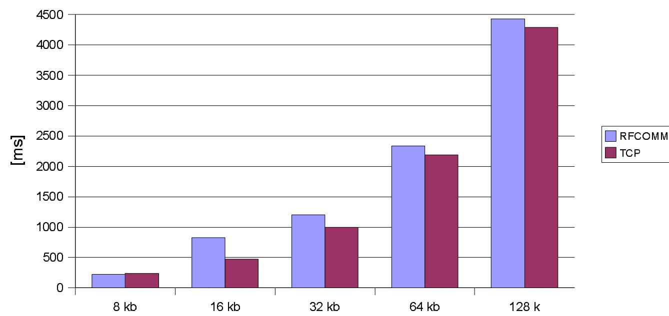
All measurements were repeated 20 times and the amount of data transmitted varied between the rang of 8 bytes and 128 kB. The recorded data is listed in Appendix G.

TCP/IP is found to be not significantly slower than RFCOMM and having a look at the standard deviation of these two data series TCP/IP seems to have lesser fluctuations as well (see Figure 4.6). Figure 4.5 illustrates the measured periods of time the transmission took.

The time needed to send and receive the data increases slowly up to a data volume of 512 bytes, and is then approximately doubled each time the amount of data is doubled. This is because the default package size is 768 bytes [17]. Up to 512 bytes one single package only needs to be transmitted. The result is approved by the virtual round trip time per byte. In both series this time is smaller than 0.05 ms when the amount of data is larger than 512 bytes (see Figure 4.7). The resulting jitter of ± 0.01 ms can be explained by the missing efficiency during the transmission, when the packets are not completely filled or due to the precision of measurement. The lower bound of the round trip time per byte is 0.03 ms using both technologies.

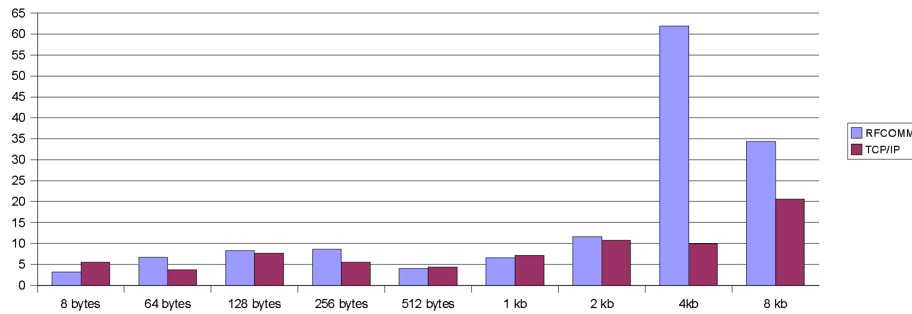


(a) Average transmission time (8 bytes – 8 kB)

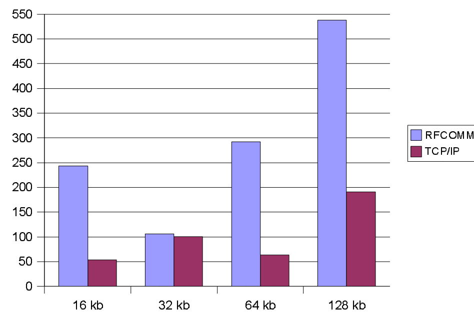


(b) Average transmission time (8 kB – 128 kB)

Figure 4.5: Measured time [ms] by given amount of data

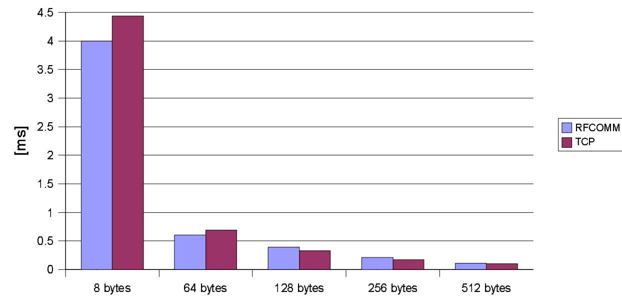


(a) Standard Deviation (8 bytes – 8 kB)

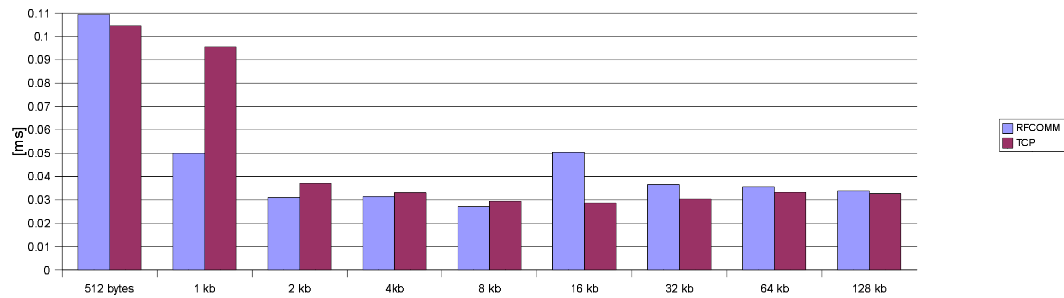


(b) Standard Deviation (8 kB – 128 kB)

Figure 4.6: Comparison of the Standard Deviation



(a) Average Round Trip Time (8 bytes – 512 bytes)



(b) Average Round Trip Time (512 bytes – 128 kB)

Figure 4.7: Comparison of the average round trip time per byte

Chapter 5

Sample Application BAD2

5.1 What is BAD2

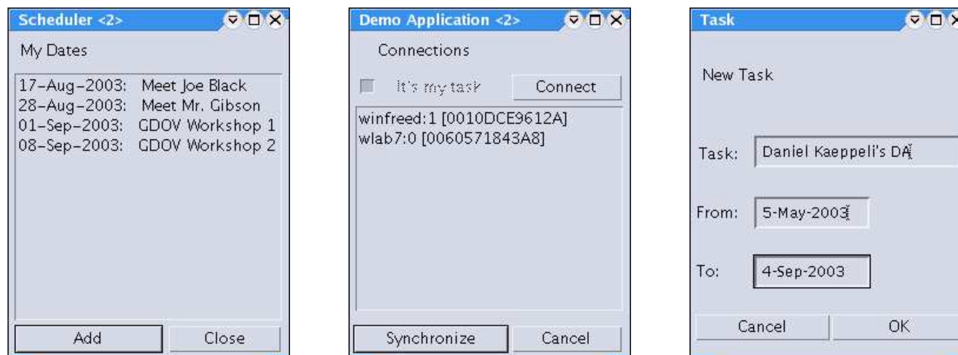
BAD2 (Bluetooth Ad-hoc Distributed Diary) is a very simple computer based diary. It allows a group of up to eight people to make an appointment nearly without any human interaction. Before making such a new appointment BAD2 will check all the different appointment books of the participants and finds a date where nobody is occupied yet and — if there exists such a date — adds this appointment into the diaries of all participants. A diary is managing tasks whereas each task is composed of a title and date when the task takes place. The granularity of the diary is currently at level of days so a person can only participate one task per day. Each task has a moderator whereas a moderator is the speaker or the person moderating the meeting.

The user interface (UI) is designed to be displayed on small devices such as iPAQs. The dimension of the windows is 240x300 pixels that matches perfectly to the iPAQ's display (240x320 pixels). A height of 300 pixels fits to the iPAQ's screen in such a way that the menu bar is still visible. The UI is implemented in Java's Abstract Window Toolkit (AWT) to make it quick.

BAD2 is built on top of the implemented message system (see Chapter 3). To make a new appointment all participants – but the moderator – must start a regular peer. This can be done by starting BAD2, press the "Add" button and after wards the "Connect" button. The moderator needs to start a rendez-vous peer that is interconnecting all participants. To start a rendez-vous peer the moderator also needs to start BAD2 and to press the "Add" button. The difference is that the moderator has to select "My Task" before clicking the "Connect" button. At this point the connect procedure is initiated. The moderator's device will search all participants and establish a connection to each of them. If the connections have been established, the moderator defines a new task. After defining the new task the moderator presses the "Synchronize" button to start the synchronization process. This is provided by sending a multicast message to all participants that includes information about the task such as the name and the range of time it should take place. Invoked by this initial message each of the different participants will transmit the dates within the specified range that they are occupied. In the current implementation it is

the rendez-vous peer's task to determine the first date that everybody is unoccupied. Basically each device is able to determine such a date by itself but in this application it is done by the rendez-vous peer. If the moderator's device found such a date it sends out the task's date to the connected participants that insert it in their local diaries.

BAD2 consists of three screens: the schedule, the connection, and the task screen. The schedule screen (see Figure 5.1(a)) displays all appointments in the diary. Clicking the schedule screen's "Add" button the connection screen (see Figure 5.1(b)) appears. This screen shows the connections the host device actually maintains. This screen will be empty until at least one connection is established. If the user is not a moderator he has simply to click the "Connect" button and to wait. If the user is a moderator he has to select "It's my task" and then also to click the "Connect" button. If the device and service discovery procedure is terminated the "Synchronize" button becomes enabled. Clicking the "Synchronize" button, the task screen (see Figure 5.1(c)) appears. This screen collects the information necessary about the new task that is to be created, such as its name and the range in time it will take place. The dates that are entered to the application must conform the following format: `dd-MMM-yyyy`. After clicking "OK" the synchronization process will be executed. After a few seconds the new task is going to be inserted in the schedules of all participants.



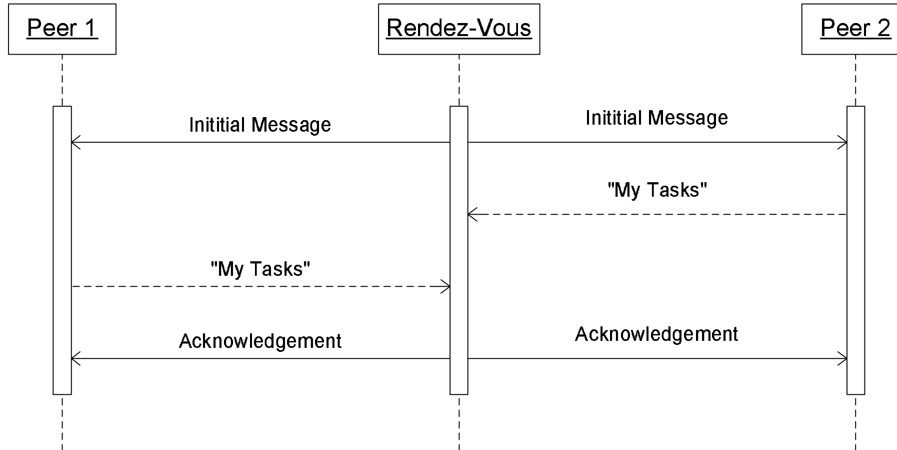
(a) Schedule Screen **(b)** Connection Screen **(c)** Task Screen

Figure 5.1: These Screen Shots Show the Different Screens of the BAD2's User Interface.

5.2 Appointment Protocol

The core feature of BAD2 is its ability to find dates for appointments. This section explains the algorithm implemented. The principle of this mechanism is very simple. Each participating device propagates the dates it is occupied. In this matter each node can build a list of all appointments of all the other participants. In our implementation only the moderator's device will build such

Figure 5.2: Message Sequence Diagram



a list. This device searches the first possible date where nobody has another appointment and will propagate this date to all participants.

The synchronization process is initialized after the moderator clicked the "OK" button on the task screen (see 5.1(c)). This screen appears only on the moderator's device. The message sequence diagram is illustrated in Figure 5.2. The very first message is sent by the moderator's rendez-vous peer to all connected peers (see Table 5.1). This message contains information like the task's name, its range in time to take place, and additionally also the number of participating devices. Invoked by the receipt of this message each peer returns a notification (see Table 5.2). This notification is composed of the following elements: an element notification without any data, and it's tasks. The name field of such a task's element contains "task$x$" and its payload data that contains the date of the task in milliseconds since 1-Jan-1970 (see Java API, Section *java.util.Date*). All transmitted dates are encoded this way. After receiving a notification of each participating peer the moderator's peer determines the date the new task will take place. After doing this it sends out an acknowledgment (see Table 5.3) to the peers informing them about the determined date. All peers — inclusive the one of the moderator — adds this task to their local schedule.

Table 5.1: Example of an Initial Message

Element	**Data**
numberOfPeers	3
task	example task
fromDate	15-Aug-2003 [in ms]
toDate	30-Aug-2003 [in ms]

Table 5.2: Example of a Notification Message

Element	**Data**
notification	—
task1	17-Aug-2003 [in ms]
task2	21-Aug-2003 [in ms]
⋮	⋮
task$n$	30-Aug-2003 [in ms]

Table 5.3: Example of an Acknowledgment

Element	**Data**
acknowledgment	—
task	18-Aug-2003 [in ms]

Chapter 6

Future Work

Since the time to write a diploma thesis is limited to four months, there was no time to have a look at all possible tasks and skills. This chapter gives an overview of the tasks that should be considered in future work.

6.1 Scatternet

At the moment the messaging system supports piconets. That means the messaging system is limited to eight participating devices. During the next few months and years the number of scatternet enabled devices will increase rapidly. So there will be a need to adapt the messaging system to scatternet's topology. The following tasks have to be checked to enable scatternet support to the messaging system.

6.1.1 Routing

At the moment the routing mechanism is very rudimentary and it is based on the piconets star topology. In a piconet there are up to seven active slaves, each of them connected to the same master. In other words each slave has exactly one connection interconnecting it to its master. That means each device having more than one connection is by definition a master. This is exactly what the routing algorithm is based on. Each device having more than one connection must forward messages to all interconnected devices. This strategy fits perfectly to the underlaying piconet topology.

A scatternet network is a general, cyclic graph. Applying the described routing mechanism to scatternets is dangerous and is going to cause a never-ending message storm since in a scatternet there are at least two interconnected devices having more than one connection. These to devices would send the message to each other (Ping-Pong). It is proposed to implement a "smarter" routing algorithm such as flooding (see [21], p 353–354) or any other routing algorithm (see [21] pp 355 or [22]). Keep in mind that this message system is a very dynamic network and the participating devices might often join and leave the network.

6.1.2 Device Discovery

At the moment the messaging system is able to propagate broadcast messages over the given piconet and send unicast messages to adjacent nodes. A device has zero-knowledge about non-adjacent nodes. To support sending unicast messages all over a scatternet the sending device must have knowledge about nodes connected somewhere to this network. Also intermediate nodes that route messages from the source to the destination must have knowledge about the entire network. Therefore it is necessary to add a device discovery mechanism discovering devices that have joined and leaved the network. An important point is the detection of disappearing devices. A leaving node may destroy or divide such an ad-hoc network. If the master of a piconet is turned off the complete network will break down. A scatternet may be divided in two autonomous and non-connected parts if a interconnecting master/slave or slave/slave node leaves the network.

6.2 Security

Bluetooth offers three mechanisms of security: authorization, authentication, and encryption. Authorization takes place during the first time two devices establish a connection, the so called pairing (see Section 2.3). The authentication process is started each time a connection is established but the very first time. Encryption can be used to secure the communication between two adjoining and connected nodes. At the moment there are neither authentication nor encryption enabled (to enable encryption and authentication see Appendix F). It might be necessary to have endpoint encryption rather than link encryption. Currently there are no mechanisms supporting endpoint encryption, so the top level application has to care about. Another issue is gaining access to the messaging system. At the moment everybody can gain access to the messaging system, there are no user names, passwords, or other authentication mechanisms but the Bluetooth security options.

6.3 Reliability

In the current state the transmission of messages is not reliable. It is assigned to higher level applications to make the communication reliable. The RFCOMM connection used does itself not guarantee any reliability.

> “Data frames do not require any response in the RFCOMM protocol, and are thus unacknowledged.
>
> Therefore, RFCOMM must require L2CAP to provide channels with maximum reliability, to ensure that all frames are delivered in order, and without duplicates. Should an L2CAP channel fail to provide this, RFCOMM expects a link loss notification, which should be handled by RFCOMM . . . .” [2], page 422

6.4 Multicast Groups

The current system processes every incoming message and delivers it to the application level. To make the system more efficient it is advisable to implement filters. These filters should forward a message if needed but only process the message if the current user was registered to that group the message was addressed to. Such a filtering mechanism should allow the top level application to define rules that are applied to the incoming messages by the messaging system.

6.5 Interconnection with JXTA/JXME

In the current state the implemented system is not able to cooperate with JXTA peer-to-peer networking software. The message format defined in this implementation is fully compatible to the message format defined in JXME. To compare the message formats see Section 3.3.1, [15], [18], and analyze also the following classes of the JXME source code: *Message.java* and *Element.java* in the package *net.jxta.j2me*.

There are a few different ways to achieve compatibility with JXTA. The first one is to implement a proxy having a TCP/IP enabled and a Bluetooth network interface. This proxy picks up the incoming messages from the Bluetooth interface and pushes them to the TCP/IP interface and vice versa. Since JXME uses a polling mechanism and the Bluetooth messaging system uses a push mechanism the proxy needs to poll the JXTA rendez-vous server for getting new messages. After receiving such a message the proxy has to push it to the Bluetooth device. Such a proxy can also be a device having limited resources e.g. a J2ME CLDC VM.

The second possibility is to implement a dedicated JXTA proxy. This is a ordinary JXTA node having a Bluetooth endpoint implementation and serving as proxy for mobile devices. This solution has not been analyzed in detail in this thesis.

The last possibility is to join Kuldip Singh Pabla's project "Proxyless JXME" (see Appendix I). The main problem using JXME is that each participating device needs a HTTP connection to a full featured JXTA node. Based on this fact it is impossible to build ad-hoc networking groups without having any network infrastructure. Two neighboring JXME nodes can only communicate using network and JXTA infrastructure whereas the location context is lost. The project of Kuldip tries to eliminate this drawback. The project presented here and the corresponding UDP implementation of Andreas Frei can be used as a prototype for such a system.

Appendix A

Objectives

Multihop Event-System basierend auf JXTA-Bluetooth Layer

Student:	Daniel Käppeli
Beginn:	5. Mai 2003
Ende:	4. September 2003
Assistent:	Andreas Frei

Motivation

In der heutigen Zeit werden die Geräte immer kleiner und die Vernetzung untereinander hat stark zugenommen. Es werden Szenarien möglich die noch bis vor kurzem undenkbar waren. Mit der allgegenwärtigen Vernetzung treten aber auch Probleme der verteilten Systeme in den Vorschein. Um diese Systeme zu entkoppeln gibt es verschiedene Ansätze wie Objekt-Orientierte und Komponenten Middleware (z.B. Corba, RMI), Message-Orientierte Middleware oder Ereignis-Orientierte Systeme. Bluetooth spielt dabei vermehrt eine wichtige Rolle im Personal Area Network (PAN). Also in der Umgebung in der sich der Mensch aufhält. Im weiteren scheinen Ereignis-Orientierte Systeme eine gute Wahl für verteilte Systeme zu sein, wo keine dauerhaften Verbindungen vorhanden sind.

Aufgabe

Die Aufgabe dieser Diplomarbeit besteht darin ein multihop Event-System für Mobile Geräte basierend auf Bluetooth aufzubauen. Die mobile Infrastruktur besteht dabei aus Ressourcen armen Geräten wie iPAQs bis zu Ressourcen reicheren Geräten wie Laptops oder Desktop Maschinen.

Das multihop Event-System soll auf JXTA, einem Peer to Peer Framework aufbauen.

JXTA um Bluetooth erweitern

In unserer Ad-hoc Infrastruktur, bestehend aus iPAQs, Laptops und Desktop Geräten, verfügen alle Geräte über einen Bluetooth Adapter. JXTA definiert verschiedene Protokolle unter anderem das Endpoint Protokoll, welches um Bluetooth erweitert werden soll.

Multihop Event-System

Herkömmliche Event-Systeme basieren auf verteilten Systemen welche für statische und meist zuverlässige Netzwerke konzipiert wurden. In ad-hoc Netzwerken wo mobile Geräte jeder Zeit den Standort wechseln können sind neue Konzepte nötig um Ereignisse, Events, den interessierten Konsumenten zur Verfügung zu stellen. Die Produzenten der Ereignisse,

die Publisher, publizieren ihren Event um den Konsumenten, den Subscriber, zu informieren. Publisher können Sensoren sein die gerade genügend Ressourcen zur Verfügung haben um den Event an ein nächstes Gerät zu senden. Es ist deshalb eine Infrastruktur auf den Sensoren oder den Kleinst-Geräten nötig, welche einer JXME Architektur ähnelt um diese Events zu verschicken.

Events werden meist nicht direkt vom Publisher zum Subscriber gelangen. Verschiedene Devices sind in der Weiterleitung und Verteilung dieser Events involviert. Es soll nun ein Event-System entwickelt werden, welches auf Kleinst- Geräten funktioniert und auf Ressourcen reicheren Geräten erweitert werden kann.

Benchmarking des Bluetooth, Multihop Event-System

Das entwickelte Event-System soll während der Entwicklungszeit und insbesondere am Schluss laufend getestet werden um die Möglichkeiten und Grenzen auszuloten.

Evaluation

Die Schlussnote der Diplomarbeit stützt sich auf den entwickelten Prototypen, die dazugehörigen Benchmarks und entsprechender Dokumentation.

Dabei setzt sich die Note nach folgendem Schema zusammen:

50% für die Qualität der Dokumentation, sowohl Code als auch Bericht,

25% für Vollständigkeit der Unit Tests und Benchmarking,

25% für die Qualität und Vollständigkeit der gestellten Anforderungen.

Appendix B

Installing and Configuring BlueZ

BlueZ is the official Bluetooth stack for Linux which is included in the official kernel's source code. To install BlueZ on a computer it is necessary to recompile the kernel. The following sections give the support necessary to install BlueZ on a Linux computer and on a Linux driven iPAQ.

B.1 News

Users of BlueZ and developers of Java Bluetooth applications discuss their problems in the following news groups. If you have a problem check the mail archives or post a new message.

- BlueZ Users: http://bluez.sourceforge.net/mlist.html
- JABWT Mailing list: http://groups.yahoo.com/group/JABWT/

B.2 Configuration on Linux/RedHat

Use the following tools to start and configure BlueZ. The Usage of this tools is also described in the following sections.

- `hciconfig hci0 up` (or `hcid`, which reads `/etc/bluetooth/hci.conf`)
- `hcitool dev` (to check if device is online)
- `hcitool inq` (to find other devices
- `hcitool scan` (to find other devices and their services)

B.3 Installation

B.3.1 iPAQ H3970

To install BlueZ on a Linux driven iPAQ execute the following instructions.

- install the `bootgpe2-v0.7-rc1-h3900.jffs2`
- `wget wget http://familiar.handhelds.org/releases/v0.7/base/armv4l/-bluetooth-uart-modules_2.4.19-rmk6-pxa1-hh11_ipaqpxa.ipk`
- `ipkg install task-bluez` (as root)

Links

http://handhelds.org/pipermail/ipaq/2003-January/016527.html

B.3.2 Acer Bluetooth on IBM A31/ RedHat 7.3

Remove old bluetooth from kernel

To install BlueZ on a kernel older than 2.4.20 you have to rebuild the kernel without any Bluetooth features which your kernel offers.

- `cd /usr/src/linux`
- `make xconfig`
- delete from Bluetooth all modules
- delete from USB Support the module **USB Bluetooth Support (Experimental)**
- `make dep`
- `make clean`
- `make bzImage`
- `cd /boot`
- `cp /usr/src/linux/arch/i386/boot/bzImage vmlinux-2.4.18-10_1`
- `rm vmlinuz`
- `ln -s /boot/vmlinux-2.4.18-10_1 vmlinuz`
- `rm /lib/modules/"your_kernel_version"/kernel/drivers/usb/bluetooth.o`
- `rmmod bluetooth`
- reboot

BlueZ Kernel

The new Bluetooth kernel module can be downloaded from the BlueZ homepage [13].

- `tar -xzvf bluez-kernel-2.3.tar.gz`
- `cd bluez-kernel-2.3`
- `./configure`
- `make`
- `make install` (as root)
- `vi create_dev` (Listing B.1 shows the source code of the script to be created and executed)
- `chmod a+x create_dev`
- `./create_dev` (as root)

Listing B.1: Shell script to create the devices (`create_dev`)

```
1  #!/bin/sh
2  #
3  # Create Bluetooth devices in /dev
4  #
5  #
6
7  VHCI_MAJOR=10
8  VHCI_MINOR=250
9
10 RFCOMM_MAJOR=216
11
12 #
13 # Create device for VHCI
14 #
15 if [ ! -c /dev/vhci ]; then
16    mknod /dev/vhci c ${VHCI_MAJOR} ${VHCI_MINOR}
17    chmod 664 /dev/vhci
18 fi
19
20 #
21 # Create devices for RFCOMM
22 #
23 for i in `seq 0 255`
24 do
25    if [ ! -c /dev/rfcomm$i ]; then
26       mknod -m 666 /dev/rfcomm$i c ${RFCOMM_MAJOR} $i
27    fi
28 done
```

BlueZ Libs

The BlueZ libs have to be downloaded from the BlueZ project web page [13].

- `tar -xzvf bluez-libs-2.2.tar.gz`
- `cd bluez-libs-2.2`
- `./configure`
- `make`
- `make install` (as root)

BlueZ Utils

The BlueZ utils have to be downloaded from the BlueZ project web page [13].

- `tar -xzvf bluez-utils-2.1.tar.gz`
- `cd bluez-utils-2.1`
- `./configure`
- `make`
- `make install` (as root)

Modules

Add the following lines to the module's configuration file (`/etc/modules.conf`). To edit this file super user privileges are required.

```
alias net-pf-31    bluez
alias bt-proto-0   l2cap
alias bt-proto-3   rfcomm
```

Links

- http://bluez.sourceforge.net
- http://www.holtmann.org/linux/bluetooth/
- http://www-106.ibm.com/developerworks/wireless/library/wi-handblue/
- http://www.harbaum.org/till/palm/bluetooth/index.html, how to setup modules on the laptop

B.3.3 Installing BlueZ on RedHat 9

These instructions have been successfully tested on the following systems:

	IBM T30	IBM R32	AMD Athlon 1400
Acer BT-500	•	•	•
MSI Dongle	•	•	•
IBM Daughter Card	•	—	—

If you have downloaded your kernel from www.kernel.org you have to patch your kernel before you can start installing BlueZ. If you have a RedHat distribution (version 9 or newer) ignore the patch instruction. To patch your kernel you have to download the corresponding patch file[1]. To patch the kernel type the following commands:

- `cd /usr/src/`*"your kernel"*
- `patch -p1` <*"path to the patch file"* (as root)

Using the original RedHat kernel start here!

- `cd /usr/src/linux-2.4.20-8`
- `make xconfig`
- select from *Bluetooth Support* all modules (y)
- select from *Bluetooth* → *Bluetooth Device Drivers* the entry *HCI USB Driver*
- `make clean`
- `make dep`
- `make bzImage`
- `cd /boot`
- `cp /usr/src/linux-2.4.20-8/arch/i386/boot/bzImage \-`
  `vmlinux-2.4.20-8_1`
- `rm vmlinuz`
- `ln -s /boot/vmlinux-2.4.20-8_1 vmlinuz`
- reboot
- `vi create_dev` (Listing B.1 shows the source code of the script to be created and executed)
- `chmod a+x create_dev`
- `./create_dev`

BlueZ Libs

The BlueZ libs have to be downloaded from the BlueZ project web page [13].

- `tar -xzvf bluez-libs-2.2.tar.gz`
- `cd bluez-libs-2.2`
- `./configure`
- `make`
- `make install` (as root)

[1] http://www.holtmann.org/linux/kernel/

BlueZ Utils

The BlueZ utils have to be downloaded from the BlueZ project web page [13].

- `tar -xzvf bluez-utils-2.1.tar.gz`
- `cd bluez-utils-2.1`
- `./configure`
- `make`
- `make install` (as root)

Modules

Add the following lines to the module's configuration file (`/etc/modules.conf`). To be able to edit this file super user privileges are required.

```
alias net-pf-31    bluez
alias bt-proto-0   l2cap
alias bt-proto-3   rfcomm
```

B.4 Start hcid and sdpd at Boot Time

It is possible to start and stop the HCI daemon and the SDP daemon when the system is started resp. is shut down. The corresponding start and stop scripts are stored in the script folder (`/etc/rc.d/init.d`) during the installation of BlueZ. To start and stop these daemons a link has to be inserted pointing to this script in the corresponding run level's folder.

- `cd /etc/rc.d/rc5.d`
- `ln -s ../init.d/bluetooth S99bluetooth` (as root)
- `ln -s ../init.d/bluetooth K01bluetooth` (as root)

B.5 Environment Variables

Since most of the Bluetooth utilities are stored in `/sbin` or `/usr/sbin`, it is useful to make sure that these directories are included in the `PATH` environment variable of your system. If these directories are not already in in the system path you might execute following instructions to add them permanently to the system's path.

- `cd /etc/profile.d`
- create as root a new text file called `path.sh` containing the following line:

  ```
  export PATH=$PATH:/sbin:/usr/sbin
  ```

- `chmod 755 path.sh`
- after restarting the XServer (Log out and press ALT + CTRL + BACKSPACE) these directories are included in the `PATH` environment variable.

B.6 Set the user-friendly name

The messaging system identifies devices by its user-friendly name. By convention a user-friendly name is composed of the host name and the HCI device number, separated by a colon, e.g. **wlab5:1**.
To set a devices user-friendly name apply one of the described methods. To see your device's name type the following command in a shell: `hciconfig -a hciX name`, whereas `X` is your device's number.

B.6.1 Adapting the HCID configuration file

The device's name can be configured in the HCI daemon's configuration file.

- open the file **/etc/bluetooth/hcid.conf** in an editor (as root)
- search the following line: `name "BlueZ (%d)";`
- comment this line by adding a "#" in front of this line
- add the following line: **name "<your host's name>:%d"**;
- before these changes become operative the HCI daemon has to be restarted
  - `killall hcid` (as root)
  - `/sbin/hcid` (as root)
- If there is one Bluetooth device connected to your computer its name is **wlabX:0**. If you have five devices your devices friendly their names are **wlabX:0**, ..., and **wlabX:4**

B.6.2 Using the hciconfig tool

Using the `hciconfig` command the user-friendly name may be set or modified at run time without restarting the HCI daemon. Since you are directly interacting with hardware super user privileges are required.

- `hciconfig -a hciX name "your device's name"` (as root)
- The parameter `-a hciX` is optional. If it is not specified the command is applied to the default device which is `hci0`.

Appendix C

TCP Over Bluetooth

The following sections give an overview of how to set up a TCP/IP network using Bluetooth network infrastructure. All instructions of this section require an installed BlueZ Bluetooth stack. Make sure that BlueZ is installed on your system otherwise check Appendix B how to install BlueZ on your computer. A kernel version 2.4.20 or newer is required. All trials failed to install the components necessary using older kernels.

C.1 TCP over Bluetooth

Installation

Make sure that the following BlueZ modules are installed on your computer:

- BlueZ Utils
- BlueZ Libs
- BlueZ SDP
- BlueZ PAN

This howto shows you how to configure an Bluetooth Network Access Point (NAP) and a Personal Area Network User (PANU) accessing the NAP. At the end the author was able to send some pings into the ETH LAN. This howto creates a private LAN (192.168.0.x) and allows the members of this private LAN to use the NAPs Internet connection.

C.2 Network Access

Start the Network Access Point (NAP)

1. Start the HCI Daemon `/sbin/hcid` (as root)
2. Check the HCI Daemon by typing `hcitool dev`. The expected result is the Bluetooth address of your Bluetooth device.
3. Start the Service Discovery Daemon: `/usr/sbin/sdpd`
4. Now you can start the PAN Daemon using the following command:
   `pand --listen --role NAP`
5. **Now see Section C.2 how to start the client!** Continue here after wards.

6. Check the kernel log file for the following line:
   ... New connection from 00:11:22:33:44:55 bnep0
   This indicates the successful establishment of a connection.

7. Configure the `bnep0` interface:
   `/sbin/ifconfig bnep0 192.168.0.1`

8. Now we have to adapt the routing behavior of the NAP:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
iptables -P FORWARD DROP
iptables -A FORWARD -i bnep0 -o wlan0 -j ACCEPT
iptables -A FORWARD -i wlan0 -o bnep0 -m state --state \-
                             ESTABLISHED,RELATED -j ACCEPT
iptables -t nat -A  POSTROUTING -o wlan0 -j MASQUERADE
```

9. Now you should be able to ping the connected host
   `ping 192.168.0.2`

Prepare the PAN User (PANU)

Create the interface's configuration file '`ifcfg-bnep0`' in the following folder:
`/etc/sysconfig/network-scripts/`

```
DEVICE=bnep0
BOOTPROTO=static
IPADDR=192.168.0.x
NETMASK=255.255.255.0
ONBOOT=no
```

where $1 < x < 254$

Start the PAN User (PANU)

1. Start the HCI Daemon
   `/sbin/hcid`

2. To check the HCI Daemon type
   `hcitool dev`
   The expected result is the Bluetooth address of your Bluetooth device.

3. Start the Service Discovery Daemon
   `/usr/sbin/sdpd`

4. Try to establish a connection typing:
   `pand --role PANU --search --service NAP --persist`

5. check the kernel log file for the following line:
   ... New connection from 99:88:77:66:55:44 bnep0
   This indicates the successful establishment of a connection.

6. The `bnep0` interface is configured automatically using `ifcfg-bnep0` script.

7. The last step is to configure the routing table. Each connection will be routed through the Bluetooth connection.
`/sbin/route add default gw 192.168.0.1` (as root)

8. Now you should be able to ping any host.

C.3 Resources

- BlueZ: Please refer "How To PAN[1]"
- Linux: Please refer [12], pp 768

[1]http://bluez.sourceforge.net/contrib/HOWTO-PAN

Appendix D

Installing and Configuring Impronto Developer Kit For Linux

Guide how to install and configure Impronto Developer Kit For Linux v1.1. There exists two versions of Rococo's Impronto Developer Kit for Linux depending on the processor architecture it is compiled for (x86 or ARM). This is necessary because of Impronto brings along some native code.

D.1 Impronto v1.1 for Linux (x86)

- make sure that you have a kernel of version 2.4.20
- make sure that BlueZ and the following components are installed on your computer (see Appendix B)
  - BlueZ-SDP 1.2
  - BlueZ-Utils 2.3
  - BlueZ-Libs 2.4
- make sure that the following libraries are installed on your computer:
  - `libbluetooth.so.1`
  - `libsdp.so.2`
- make sure that Java is installed on your computer (1.3.1 or later)
- to build the examples you need also Ant[1]
- copy the `impronto-1.1-15.i386.rpm` to your computer
  - `/afs/ethz.ch/inf/proj/wlab/bluetooth/impronto-1.1/\-impronto-1.1-15.i386.rpm`
- to install Rococos Bluetooth Development Kit type the following command: `rpm -ivh --nodeps impronto-1.1-15.i386.rpm`
- the install routine creates the following files and directories:

[1] http://ant.apache.org/

  - `/usr/share/doc/impronto-1.1/` containing `user_guide.pdf`, examples and JavaDocs
  - `/usr/share/java/idev_bluez.jar`
  - `/usr/lib/libimpronto.so`

- copy the `LinuxLicense.txt` into the folder `/usr/share/java`. **This folder must be included in the classpath to run any applications!**
  - `/afs/ethz.ch/inf/proj/wlab/bluetooth/impronto-1.1/\-LinuxLicense.txt`

- Before developing your own Java Bluetooth Applications check the programming hints (see Section D.3)

Verify your installation

Test Service Inquiry and Discovery

- to verify the installation build and run the examples:

```
cd /usr/share/doc/impronto-1.1/examples/browser
ant -f build-example.xml
java -cp /usr/share/java: \-
     /usr/share/java/idev_bluez.jar:classes \-
     com.rococosoft.impronto.examples.browser.Browser
```

Testing L2CAP Connections (HelloWorld)

- see Section D.2: Testing L2CAP Connection

Testing RFCOMM Connections (Simple Chat)

- see Section D.2: Testing RFCOMM Connections

D.2 Impronto v1.1 for ARM

Installation

- Make sure that there is a JVM installed on your iPAQ (see the corresponding tutorial [2] written by Andreas Frei)
- Make sure that BlueZ is installed on your iPAQ (see Appendix B)
- Copy the following files to the iPAQ (e.g. into the root-folder `/root`):
  - `/afs/ethz.ch/inf/proj/wlab/bluetooth/impronto-1.1/\-impronto_1.1_arm.ipk`
  - `/afs/ethz.ch/inf/proj/wlab/bluetooth/impronto-1.1/\-LinuxLicense.txt`

[2] http://ik4.inf.ethz.ch:8999/projects/Wiki.jsp?page=Af_ipaq_opiejava

- Install the `impronto_1.1_arm.ipk` package:
  `cd /root`
  `ipkg install -force-depends impronto_1.1_arm.ipk`
  Make sure that the following libraries are installed on your iPAQ: `libbluetooth` and `libsdp`.
- During the installation the following files are created:
  - `/usr/lib/libimpronto.so` - The ARM native library
  - `/usr/share/idev_bluez.jar` - The Java runtime library
- When you are running Java Bluetooth application you need to ensure that the file `LinuxLicense.txt` is included in the classpath. A program call may look like this:
  `java -Djava.library.path=/usr/lib -cp /usr/share/idev_bluez.jar:\-`
  `<directory including LinuxLicense.txt>:\-`
  `<directory including your classes> my.prog.Start`
- Before developing your own Java Bluetooth applications check the programming hints (see Section D.3)

Testing L2CAP Connection (HelloWorld)

The test application consists of a client and a server. The client and the server must run on two different machines. The client looks for a server and establishes a L2CAP connection to the server. The server sends a "hello world" to client, that's it. It is assumed that you already have installed Rococo's Bluetooth Development Kit on a desktop computer otherwise see Section D.1.

iPAQ as server

- **on the iPAQ:**
  - Copy the following files to the iPAQ (e.g. into the root folder /root):
    - `/afs/ethz.ch/inf/proj/wlab/bluetooth/impronto-1.1/hello.jar`
  - make sure that the `LinuxLicense.txt` is on your iPAQ (e.g. into the root folder `/root`)
  - type the following command

```
cd /root
java -Djava.library.path=/usr/lib \-
     -cp /usr/share/idev_bluez.jar:/root/hello.jar:. \-
     ch.ethz.inf.iks.dk.bluetooth.helloWorld.HelloServer
```

  It is expected that the server runs without any exception or other output.

- **on your desktop computer:**
  - make sure that BlueZ is installed on your computer (see Appendix B)
  - make sure that Java is installed on your computer (version 1.3.1 or later)
  - make sure that Rococos Bluetooth Development Kit for Linux is installed on your computer (see Section D.1)

- make sure that the `LinuxLicense.txt` is in your home (<your home>)
  - `/afs/ethz.ch/inf/proj/wlab/bluetooth/impronto-1.1/\-LinuxLicense.txt`
- copy the file `hello.jar` into your home
  - `/afs/ethz.ch/inf/proj/wlab/bluetooth/impronto-1.1/hello.jar`
- type the following command

```
cd <your home>
java -Djava.library.path=/usr/lib \-
     -cp /usr/share/java/idev_bluez.jar:\-
     <your home>/hello.jar:<your home> \-
     ch.ethz.inf.iks.dk.bluetooth.helloWorld.HelloClient
```

- expected output:

```
Searching for Devices...

New Device discovered : 0002C7178A54
Service Name : HelloServer
Connection url :btl2cap://0002C7178A54:1001;\-
     authenticate=false;encrypt=false;master=true
Connect to: btl2cap://0002C7178A54:1001;authenticate=false;\-
     encrypt=false;master=true

SERVICE_SEARCH_COMPLETED

waiting for a message
Hello World
received the message
```

iPAQ as client

- **on your desktop computer:**
  - same procedure as above but type the following command on your desktop:

```
cd <your home>
java -Djava.library.path=/usr/lib \-
   -cp /usr/share/java/idev_bluez.jar:\-
   <your home>/hello.jar:<your home> \-
   ch.ethz.inf.iks.dk.bluetooth.helloWorld.HelloServer
```

  - It is expected that the server runs without any exception or other output.

- **on the iPAQ**
  - same procedure as above but type the following command on your iPAQ:

```
cd /root
java -Djava.library.path=/usr/lib \-
     -cp /usr/share/idev_bluez.jar:/root/hello.jar:. \-
     ch.ethz.inf.iks.dk.bluetooth.helloWorld.HelloClient
```

  - expected output:

```
Searching for Devices...

New Device discovered : 0002C7178A54
Service Name : HelloServer
Connection url :btl2cap://0002C7178A54:1001;\-
     authenticate=false;encrypt=false;master=true
Connect to: btl2cap://0002C7178A54:1001;authenticate=false;\-
     encrypt=false;master=true

SERVICE_SEARCH_COMPLETED

waiting for a message
Hello World
received the message
```

Testing RFCOMM Connections (Simple Chat)

- get the file chat.jar from:
  `/afs/ethz.ch/inf/proj/wlab/bluetooth/impronto-1.1/chat.jar`
- start the chat server:
  - on a desktop computer:
    `java -Djava.library.path=/usr/lib -cp ./chat.jar:.:\-`
    `/usr/share/java/idev_bluez.jar ChatServer`
  - on a iPAQ:
    `java -Djava.library.path=/usr/lib -cp /root/chat.jar:.:\-`
    `/usr/share/idev_bluez.jar ChatServer`
- Expected output: **none**
- start the chat client:
  - on a desktop computer:
    `java -Djava.library.path=/usr/lib -cp ./chat.jar:.:\-`
    `/usr/share/java/idev_bluez.jar ChatClient`
  - an a iPAQ:
    `java -Djava.library.path=/usr/lib -cp /root/chat.jar:.:\-`
    `/usr/share/idev_bluez.jar ChatClient`

- expected output:

```
Searching for Devices...

New Device discovered : 0002C7178A54
Service Name : ChatServer
Connection url :btspp://0002C7178A54:2;authenticate=false;\-
     encrypt=false;master=true

SERVICE_SEARCH_COMPLETED
```

now you can start chatting!

D.3 Java Bluetooth Programming Hints

Starting any program cause a "ServiceRegistrationException"

- **Exception:**

```
Exception in thread "main" javax.bluetooth.BluetoothState\-
      Exception: Device initialization failed; errno=19
   at com.rococosoft.impronto.impl.LocalDeviceImpl.<init>
   at aL.a(Unknown Source)
   at com.rococosoft.impronto.ImprontoLocalDevice.\-
      getImprontoLocalDevice(Unknown Source)
   at javax.bluetooth.LocalDevice.<init>(Unknown Source)
   at javax.bluetooth.LocalDevice.getLocalDevice
   at javax.microedition.io.protocol.btspp.\-
      ProtocolHandler.open(Unknown Source)
   at javax.microedition.io.Connector.open(Unknown Source)
   at javax.microedition.io.Connector.open(Unknown Source)
   at javax.microedition.io.Connector.open(Unknown Source)
   ...
```

- **Reason:** There is no Bluetooth device connected to your computer.
- **Solution:**
  - Make sure that your Bluetooth device is connected properly to your computer.
  - Make sure that the HCI daemon is running.
  - Publishing Bluetooth Services: Make sure that the SDP daemon is running.

Starting any program a "ServiceRegistrationException" is thrown

- **Exception:**

```
Exception in thread "main" javax.bluetooth.\-
   BluetoothStateException: the device is not connectable
   at com.rococosoft.impronto.ImprontoLocalDevice.\-
      a(Unknown Source)
   at t.acceptAndOpen(Unknown Source)
   ...
```

- **Reason:** HCI daemon is not started.
- **Solution:** Start the HCI daemon typing: `hcid` (as root)

Starting a program offering a service a "ServiceRegistrationException" is thrown

- **Exception:**

```
Exception in thread "main" javax.bluetooth.\-
      ServiceRegistrationException: registerRecord: -111
   at aA.update(Unknown Source)
   at com.rococosoft.impronto.impl.ServiceDatabaseImpl.\-
      updateRecord(Unknown Source)
   at R.updateRecord(Unknown Source)
   at com.rococosoft.impronto.ImprontoLocalDevice.\-
      a(Unknown Source)
   at t.registerServiceRecord(Unknown Source)
   at t.acceptAndOpen(Unknown Source)
   ...
```

- **Reason:** The SDP daemon is not started.
- **Solution:** Start the SDP daemon typing: `/usr/sbin/sdpd` (as root)

The library "libimpronto.so" is not in the library path of Java

- **Exception:**

```
Exception in thread "main" java.lang.UnsatisfiedLinkError: \-
      no impronto in java.library.path
   at java.lang.ClassLoader.loadLibrary(ClassLoader.java:143)
   at java.lang.Runtime.loadLibrary0(Runtime.java:788)
   at java.lang.System.loadLibrary(System.java:832)
   at com.rococosoft.impronto.impl.LocalDeviceImpl.\-
      <clinit>(Unknown Source)
   at aL.a(Unknown Source)
   at com.rococosoft.impronto.ImprontoLocalDevice.\-
      getImprontoLocalDevice(Unknown Source)
   at javax.bluetooth.LocalDevice.<init>(Unknown Source)
   at javax.bluetooth.LocalDevice.\-
      getLocalDevice(Unknown Source)
   at javax.microedition.io.protocol.btspp.ProtocolHandler.\-
      open(Unknown Source)
   at javax.microedition.io.Connector.open(Unknown Source)
   at javax.microedition.io.Connector.open(Unknown Source)
   at javax.microedition.io.Connector.open(Unknown Source)
   ...
```

- **Reason:** The JSR-82 implementation of Rococosoft calls some functions of a native code library. Make sure that the directory that contains the `libimpronto.so` is within the environment variable `LD_LIBRARY_PATH`.
- **Solution:** Add the following option to the statement starting your Java application `-Djava.library.path=/usr/lib`. You can also set an environment variable containing the library path. To do this add the following shell script `lib.sh` to `/etc/profile.d` directory

  ```
  export LD_LIBRARY_PATH="/usr/lib:$LD_LIBRARY_PATH"
  ```

  after restarting your XServer the `LD_LIBRARY_PATH` is set correctly.

The file "LinuxLicense.txt" is not included in the classpath

- **Exeption:**

```
Exception in thread "main" javax.bluetooth.\-
      BluetoothStateException: License not found
   at com.rococosoft.impronto.impl.LocalDeviceImpl.\-
      <init>(Unknown Source)
   at aL.a(Unknown Source)
   at com.rococosoft.impronto.ImprontoLocalDevice.\-
      getImprontoLocalDevice(Unknown Source)
   at javax.bluetooth.LocalDevice.<init>(Unknown Source)
   at javax.bluetooth.LocalDevice.\-
      getLocalDevice(Unknown Source)
   ...
```

- **Reason:** `LinuxLicense.txt` is not included in the classpath.
- **Solution:** make sure that LinuxLicense.txt is in the classpath. You can get it from:
  `/afs/ethz.ch/inf/proj/wlab/bluetooth/impronto-1.1/LinuxLicense.txt`

Having an "EOFException"

This error can occur having a L2CAP connection between a desktop computer and an iPAQ.

- **Exeption:**

```
Exception in thread "main" java.io.EOFException
   at com.rococosoft.impronto.impl.L2CAPConnectionImpl.\-
      receive(Unknown Source)
   at ch.ethz.inf.iks.dk.bluetooth.simpleChat.Client.\-
      <init>(HelloClient.java:83)
   at ch.ethz.inf.iks.dk.bluetooth.simpleChat.HelloClient.\-
      main(HelloClient.java:15)
```

- **Reason:** Your computer is too *fast*. Probably the send statement running on your desktop machine does look like this:

```
...
L2CAPConnection con = ...
...
con.send( data );
..
con.close();
```

- **Solution**: Insert a **wait** statement after the send command:

```
...
L2CAPConnection con = ...
...

con.send( data );
...
try {
   synchronized (this) {
      wait(100);
   }
} catch (InterruptedException e) {
   e.printStackTrace();
}
...
con.close();
```

D.4 Environment Variables

Every Java Bluetooth application using Impronto needs to have access to the `libimpronto.so` library. Therefore the folder containing this library must set in a environment variable. This environment variable `LD_LIBRARY_PATH` can be set on the shell that executes the program or passed as an argument invoking java (`-Djava.library.path`). The first method is the preferred one because it can be automated to take place at start time. The following instruction show how to set the `LD_LIBRARY_PATH` at start time.

- `cd /etc/profile.d`
- create as root a new text file called `lib.sh` containing the following line:

  ```
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib
  ```

- `chmod 755 path.sh`
- after restarting the XServer (Log out and press ALT + CTRL + BACKSPACE) these directory is included in the `LD_LIBRARY_PATH` environment variable.

Appendix E

Developing Java Bluetooth Applications

E.1 Using Multiple Bluetooth Devices

It is possible to connect multiple Bluetooth devices to a single computer. But a Java Bluetooth application can only operate on one device. The device, which an application uses, can be specified by setting an environment variable (`bluez_device`). If this variable is empty the default device `hci0` is accessed by any application. Since it is not possible to modify any environment variables out of a Java application that has to be initialized before the application is started. Use the bash's `export` statement to set the `bluez_device` variable.

E.1.1 Developing with Eclipse

Developing Java Bluetooth applications using Eclipse[1] the variable `bluez_device` has to be initialized before Eclipse is launched. You can do this by setting the `bluez_device` in shell where Eclipse is going to be launched. Having multiple devices the Integrated Development Environment (IDE) has to be launched once per used device since it is not possible to modify environment variables out of Java programs.

Using Eclipse there are a few more pitfalls. Having more than one instance of Eclipse operating on a single workspace can cause irreversible crashes. To avoid this problem create the needed number of copies of your workspace, so each eclipse is able to work on its own workspace. The drawback of this solution is that the source code becomes replicated and changes in the sources are not propagated to the copies created. This problem can be solved by linking the original workspace's source code folder into the copies of the workspace. Make sure that your sources are not located in the root folder of your project's folder which is located in the folder `workspace`. It is strongly recommended to create sub folders in your project folder containing the source files such as `src` or `junit` (check the documentation of Eclipse how to create such source folders). Delete all of these source folders in the copied workspaces. Create symbolic links to link the original workspace's sources to its copies using the corresponding command `ln -s` (root privileges are required). After having created these copies you can start one instance of Eclipse per workspace. The workspace, that Eclipse is working on, can be specified using the option `-data` while launching Eclipse. At the end of the day you have got replicated workspaces whereas all of them have the same source files.

[1] http://www.eclipse.org

Listing E.1: Shell script to start Eclipse

```
1  export bluez_device=hci1
2  eclipse -data ~/myWorkspace
```

Listing E.1 contains a start script which launches Eclipse using the specified workspace (`myWorkspace` in the users home folder) and Bluetooth device (`hci1`).

Appendix F

Enabling Bluetooth Security

Bluetooth offers three security mechanisms: Authorization, authentication and encryption. Rococo's Java/Bluetooth stack offers authentication and encryption. Authorization is applied by the underlaying BlueZ Bluetooth stack. Such a mechanism can be requested by one of the two devices participating. To see the details of the security mechanisms JABWT and Rococo's DevKit for Linux offer, see [17] pages 53 – 56. The described settings take place in the following classes of the messaging system implemented:

ch.ethz.iks.jxme.bluetooth.impl.BTEndpointClient

and

ch.ethz.iks.jxme.bluetooth.impl.BTEndpointServer

Keep in mind that e.g. encryption requires authentication otherwise a *BluetoothConnectionException* will be thrown [17].

Authentication is based on challenge/response mechanism that requires a common link key. This link key is generated during the first time a connection between two devices is established based on a PIN code. To set the PIN code edit `/etc/bluetooth/bluepin` file. This setting becomes active after restarting the HCI daemon.

BlueZ can be configured to ask the PIN establishing a connection. Thereto a so called *PIN helper* is used. Check out the BlueZ documentation how to install and configure such a PIN helper. BlueZ comes along with a graphical PIN helper which is called `bluepin` and is located in the `/bin` folder. Using `bluepin` on RedHat 9 causes errors. This bug has been fixed by Maxim Krasnyansky. A fixed release of `bluepin` was released in July 2003. If you have any problems with `bluepin` download the newest version from the BlueZ web page [1] and check the corresponding message in the mail archive[2].

[1] http://bluez.sourceforge.net/download/bluepin

[2] http://sourceforge.net/mailarchive/message.php?msg_id=5687993

Appendix G

Benchmarks

This chapter contains some additional information about the benchmarks and the configuration of systems that were used.

G.1 Configuration

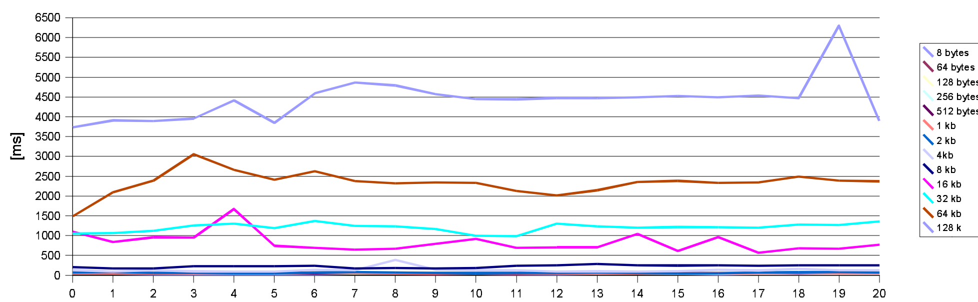
The Bluetooth devices have been placed in a distance of 1 m. The configuration of the involved computers is given in Table G.1. During the measurements the IBM T30 played the role of the server and AMD Athlon 1400 played the role of the client. Both machine's operating system was RedHat 9. The Bluetooth support was given by the recompiled 2.4.20-8 kernel delivered by RedHat. This kernel is already patched and could be used out of the box.

Table G.1: Configuration of the test systems

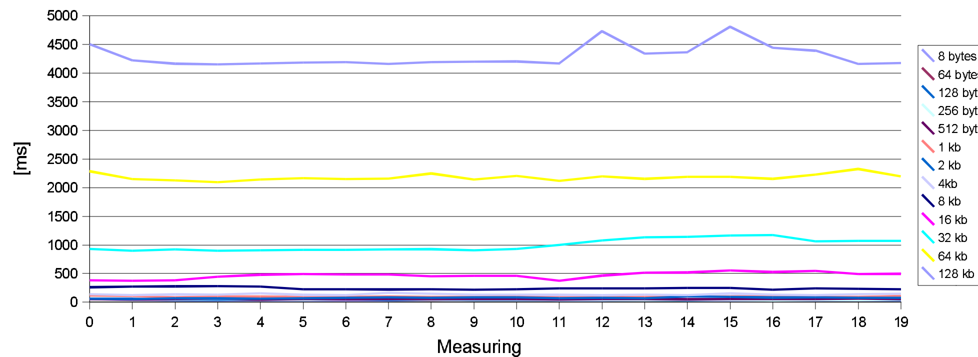
	IBM T30	**AMD Athlon 1400**
Processor	Intel P4 mobile 2.0 GHz	AMD Athlon 1.4 GHz
RAM	512 MB	512 MB
Bluetooth Device	IBM Daughter Card	Acer BT-500
OS	RedHat 9	RedHat 9
Kernel	RedHat 2.4.20-8	RedHat 2.4.20-8
Java	1.4.1	1.4.1
Java BT Stack	Impronto v1.1	Impronto v1.1
BlueZ Library	v2.4	v2.4
BlueZ Utils	v2.3	v2.3
BlueZ PAN	v1.1	v1.1
BlueZ SDP	v1.2	v1.2

G.2 Data Ascertainment

This section includes the raw data of the measurement and some figures that have not been shown in Chapter 4. Table G.2 and Table G.3 are containing the raw data of the measurements. The following figures illustrate the measured data.



(a) RFCOMM



(b) TCP/IP

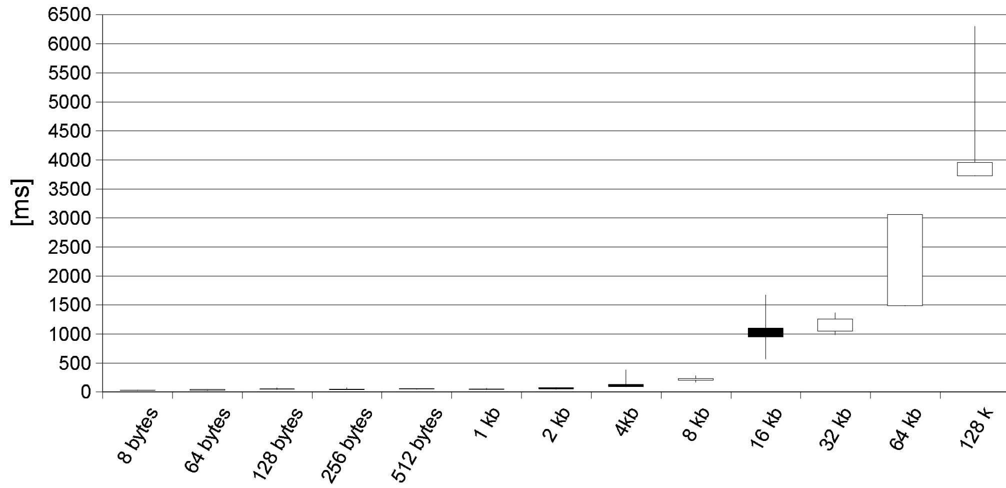
Figure G.1: Measured time [ms] by given amount of data

[ms]	8 bytes	64 bytes	128 bytes	256 bytes	512 bytes	1 kb	2 kb	4 kb	8 kb	16 kb	32 kb	64 kb	128 kb
	34	34	53	50	58	52	75	130	205	1098	1050	1490	3730
	35	33	53	53	47	52	81	88	171	839	1066	2092	3908
	29	34	73	51	59	49	61	126	172	958	1119	2390	3889
	30	50	55	49	55	44	53	97	230	951	1258	3057	3954
	27	43	47	45	56	45	48	90	230	1673	1301	2658	4414
	28	49	48	64	56	56	52	91	229	743	1186	2413	3847
	29	33	45	65	52	68	71	135	241	692	1370	2629	4589
	37	41	49	74	62	63	82	129	173	644	1242	2376	4861
	35	38	57	70	62	47	70	384	180	670	1238	2316	4792
	33	30	51	43	57	49	62	135	173	789	1171	2342	4567
	34	35	56	52	51	48	52	126	179	915	999	2335	4448
	27	28	54	56	58	54	60	125	235	694	988	2124	4441
	30	37	56	44	54	51	55	90	248	701	1298	2013	4474
	34	41	52	45	56	41	68	98	285	701	1231	2144	4465
	31	35	59	50	57	48	53	90	246	1041	1196	2356	4492
	34	46	49	47	56	54	49	102	245	613	1216	2381	4528
	28	33	37	47	60	51	48	133	248	962	1210	2334	4491
	35	34	41	46	57	46	70	121	242	570	1196	2345	4534
	33	48	40	55	52	61	74	161	249	684	1279	2487	4471
	36	49	41	57	61	51	83	126	251	666	1264	2389	6300
	33	38	39	52	49	44	67	122	248	774	1360	2371	3900
Min	27	28	37	43	49	41	48	90	173	570	988	2013	3847
Max	37	49	59	74	62	68	83	384	285	1673	1370	2658	6300
Avg	32.00	38.52	50.24	53.10	55.95	51.14	63.52	128.52	222.86	827.52	1201.81	2335.33	4433.1
Std. Dev.	3.13	6.68	8.32	8.64	4.03	6.63	11.65	61.92	34.33	243.82	106.28	292.49	538.22
ms/byte	4.00	0.60	0.39	0.21	0.11	0.05	0.03	0.03	0.03	0.05	0.04	0.04	0.03

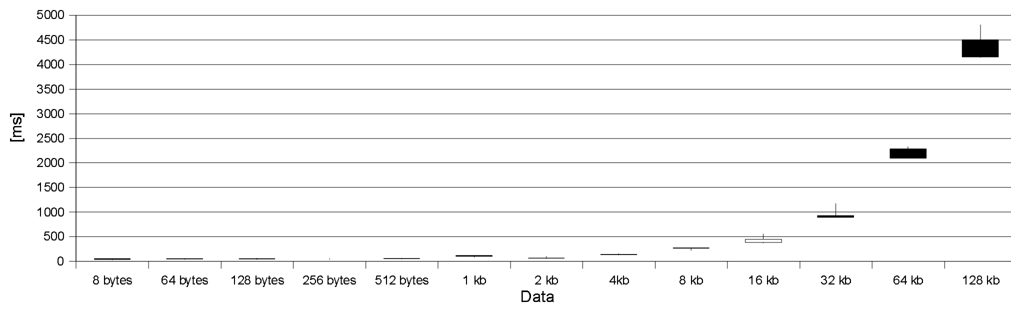
Table G.2: RFCOMM Data Ascertainment

Table G.3: TCP/IP Data Ascertainment

[ms]	**8 bytes**	**64 bytes**	**128 bytes**	**256 bytes**	**512 bytes**	**1 kb**	**2 kb**	**4 kb**	**8 kb**	**16 kb**	**32 kb**	**64 kb**	**128 kb**
	55	53	60	52	53	116	61	141	261	381	931	2287	4503
	38	44	62	44	49	106	56	127	275	375	896	2151	4222
	30	47	40	44	51	87	71	136	277	382	920	2129	4163
	32	44	41	52	59	101	65	132	277	447	897	2097	4149
	33	43	44	49	47	97	61	154	271	479	904	2142	4166
	31	43	42	42	57	91	69	128	224	493	915	2163	4181
	34	46	53	52	58	94	74	127	227	483	911	2149	4190
	32	45	43	51	51	103	85	160	222	486	919	2160	4158
	38	39	43	45	56	91	74	144	222	450	925	2248	4190
	38	43	38	43	54	92	88	134	218	459	903	2142	4199
	38	45	38	46	57	100	83	133	223	459	927	2207	4203
	34	46	38	38	46	101	71	128	242	374	1000	2120	4168
	32	43	41	38	57	88	75	127	238	464	1079	2198	4730
	41	46	35	41	58	91	72	132	241	514	1134	2153	4337
	32	48	34	40	49	100	99	127	250	518	1141	2186	4366
	33	43	49	36	55	101	94	152	249	552	1161	2187	4810
	30	37	36	34	53	104	80	131	219	526	1172	2154	4442
	34	39	35	37	55	94	85	125	240	543	1065	2227	4391
	40	50	39	41	61	94	76	139	234	492	1070	2326	4161
	33	40	39	45	50	107	78	145	223	495	1067	2195	4172
	38	42	43	49	48	95	77	134	222	497	1070	2330	4172
Min	30	37	34	34	46	87	56	125	218	374	896	2097	4149
Max	55	53	62	52	61	116	99	160	277	552	1172	2330	4810
Avg	35.52	44.10	42.52	43.76	53.52	97.76	75.90	136.00	240.71	469.95	1000.33	2188.14	4289.19
Std. Dev.	5.56	3.74	7.63	5.57	4.30	7.16	10.78	9.95	20.59	53.58	100.49	64.01	191.20
ms/byte	4.44	0.69	0.33	0.17	0.10	0.10	0.04	0.03	0.03	0.03	0.03	0.03	0.03



(a) RFCOMM



(b) TCP/IP

Figure G.2: High and low water marks

Figure G.3: Comparison of RFCOMM vs. TCP/IP

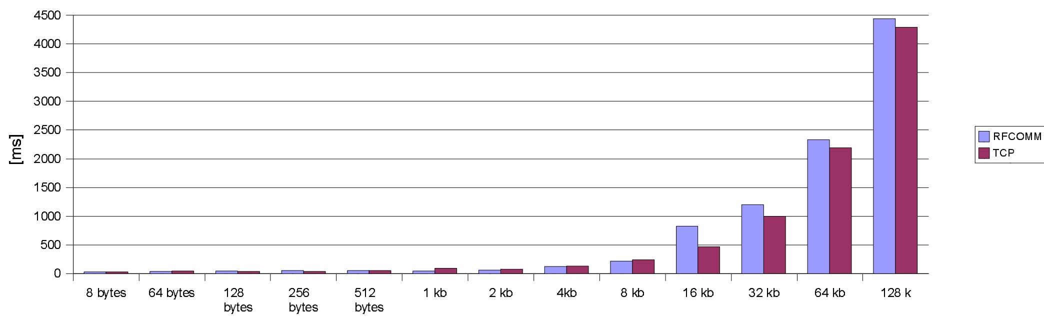


Figure G.4: Comparison of the average transmission time

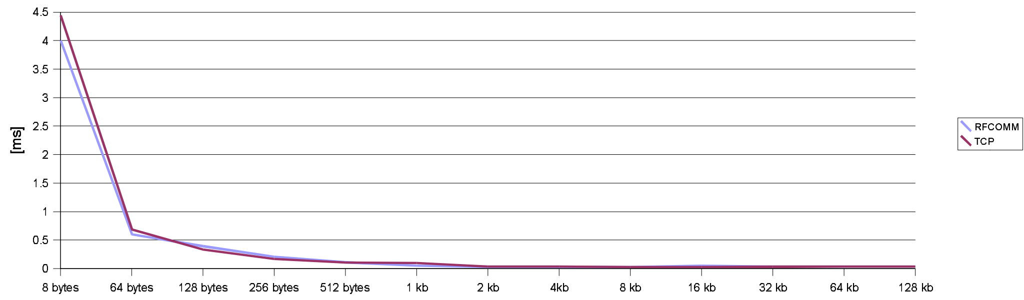
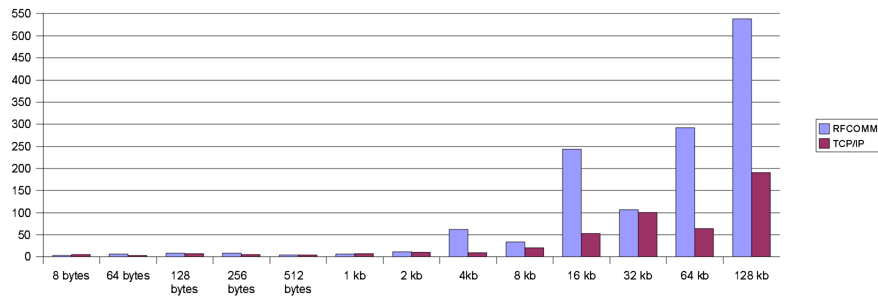


Figure G.5: Comparison of the Standard Deviations



Appendix H

Compiling CLDC Personal Basis Profile

H.1 Installing J2ME Personal Basis Profile (binary, Redhat 9)

This section describes how to install the precompiled Personal Basis Profile on a Linux box. It has been tested on a RedHat 9 installation.

- Download the following files:
  - `/afs/ethz.ch/inf/proj/wlab/j2me/cdc-personal-basis/\-cdc-personal-basis-i686.tar.gz`
  - `/afs/ethz.ch/inf/proj/wlab/j2me/cdc-personal-basis/\-microwindows-i686.tar.gz`
- Untar and install these archives

```
tar -xzvf cdc-personal-basis-i686.tar.gz
tar -xzvf microwindows-i686.tar.gz
mv cdc /usr/local
mv microwin /usr/local
```

- Verify the installation running the HelloWorld sample:

```
cd /usr/local/cdc
./bin/cvm -Dawt.toolkit=java.awt.MWToolkit \-
     -Djava.awt.graphicsenv=java.awt.MWGraphicsEnvironment \-
     Xbootclasspath=./lib/basis.jar:./democlasses.zip \-
     HelloWorld
```

H.2 Compiling J2ME Personal Basis Profile (RedHat 9)

- Before you can start installing J2ME Personal Profile you need to download and build Microwindows (**Sun Modified Edition**)
  - download Microwindows(**modified**)[1]
  - `mkdir /usr/local/mircowin`
  - `cd /usr/local/microwin/`
  - save the file `microwin-sun-11_jun_2002.tar.gz` in the created folder
  - `tar -xzvf microwin-sun-11_jun_2002.tar.gz`
- J2ME Personal Basis Profile
  - download J2ME Personal Basis Profile[2]
  - `unzip j2me_pb-1_0-fcs-src-b45-linux-i686-15_jul_2002.zip`
  - if you are using `bash` you have to edit `basis/build/share/defs.mk` and replace the line containing `SHELL=ksh -e` by `SHELL=bash -e`
  - `cd basis/build/linux-i686`
  - make the Personal Basis Profile executing the following command:

```
make J2ME_CLASSLIB=basis AWT_IMPLEMENTATION=microwindows \-
     JDK_HOME=/usr/java/j2sdk1.4.1_02/bin/ \-
     CVM_GNU_TOOLS_PATH=/usr/bin \-
     CVM_JVMDI=false CVM_DEBUG=false
```

  - **NOTE:** I wasn't able to build the Personal Basis Profile using **j2sdk1.3.1** as recommended in the release notes and in the documentation. **j2sdk1.4.1** worked perfectly fine.
  - to verify your installation run the `HelloWorld` sample:

```
cd basis/build/linux-i686/democlasses
../bin/cvm -Dawt.toolkit=java.awt.MWToolkit \-
     -Djava.awt.graphicsenv=java.awt.MWGraphicsEnvironment \-
     -Xbootclasspath=../lib/basis.jar HelloWorld
```

[1] http://experimentalstuff.sunlabs.com/Technologies/mwpbp/download.html
[2] http://wwws.sun.com/software/communitysource/j2me/pbp/download.html

Appendix I

Mail about Proxyless JXME

```
Subject: Re: [JXTA discuss] Re.[JXTA discuss] Project Proposal:
         Bluetooth asatransport protocol forJXME

From:    Kuldipsingh Pabla <Kuldipsingh.Pabla@...>

Date:    Wed, 16 Jul 2003 19:18:29 -0700

To:      Andreas Frei <frei@inf...>

CC:      Daniel Kaeppeli <danielka@student...>

Hi Andreas,

Thanx for your reply. As you may know JXTA is an open source initiative for
P2P framework. It is a place to exchange ideas and collaborate on
implementations. Like you, I too have a prototype for Infrastructure-less
P2P framework for small devices - basically proxyless JXME.

Currently, I am in the process of designing a full fledge small footprint
proxyless JXME based on my learnings from the prototype. I would like to
invite you to contribute to the proxyless jxme effort based on your
experience with BT and UDP. The community plan is to implement BT as well
and that's where your contributions will be highly useful.

-Kuldip

Andreas Frei wrote:

>
> Hi Kuldeep,
>
> thanks for your interest, I didn't know that there was already an
> announcement at JavaOne in this direction.
> I think the implementation we are working on right now is very
> preliminary and can be improved a lot.
>
> I have to say that a master student, Daniel Kaeppeli, is currently
> implementing the bluetooth jxme. There are still some implementation
> work ongoing and certainly a lot of documentation.
>
> Until we get the implementation to a more stable version I could give
> you some information about the infrastructure.
>
> 1) We decided to take over almost all of the methods from the jxme
> implementation and moved it to an interface, this allows to implement
> all interfaces for different transport layers, beside bluetooth, we have
> implemented a UDP Layer which just uses multicast in a private Lan.
>
> 2) The bluetooth implementation uses the rococo layer to have the JSR82
> as a standard API for communication. Rococo itself uses the bluez stack.
>
```

```
> 3) Currently we have to decide which node is the master and does the
> inquire in bluetooth and who is the slave.
>
> 4) We have not yet compiled the code with cldc or cdc, but we would like
> to go in this direction and are looking at the implementation to use
> that libraries.
>
> 5) We have not looked yet at the concept of Pipes and Groups. We
> certainly think groups would be a concept which is also needed in ad hoc
>   env. But do we need also pipes? Couldn't pipes be simulated by this
> groups and broadcasting messages in that group?
>
> It would be nice if we could collaborate
> We would like to checkin our version on jxta.org a little later, until
> we are on a more stable version.
>
> cheers,
>
> Andreas
>
>> Kuldipsingh Pabla wrote:
>>
>> Hi Andreas,
>>
>> I am very excited to know that you have been using the JXME API to build
>> your own vesion of JXTA/JXME. At JavaOne we had announced that we would
>> work on a infrastructure-less JXME that wouldn't relay on any proxies,
>> relays and rdvs. I am in the process of putting together a desin and may
>> announce it by the end of this week. We may also announce the new API.
>>
>> Since, you have already done some work, I was wondering if we can
>> collaborate and work together going forward, so that the community can
>> leverage from your work and you can benefit from the community efforts.
>>
>> Looking forward to your contributions to JXME,
>>
>> -Kuldeep
>>
>> Andreas Frei wrote:
>>
>>
>>> Hi Vidya
>>>
>>> this project sounds very intersting, as I understand you wan't to use
>>> the JXME infrastructure on top of bluetooth in a scatternet.
>>> My question is now how do you decide which node is the master and should
>>> therefore take over the full jxta functionality for the jxme nodes,
>>> would this be static or a dynamic change? An other question which arises
>>> what's the behaviour of a node which swiches from master to slave in the
>>> bluetooth infrastructure.
>>>
>>> We are also working in this direction, we looked at the JXME classes and
>>> decided to take over some of its method names into a core interface for
>>> "JXME" nodes. Actually we are not building a JXME node in the common
>>> sense, where it needs a full jxta node to communicate. We have built a
>>> "JXME" node which is standalone and may communicate with other "JXME"
>>> nodes over bluetooth. We are also interested in doing scatternet over
>>> this framework, but have not yet found time to do so.
>>>
>>> cheers,
>>>
>>> Andreas
```

Bibliography

[1] aboutIT. Bluetooth: Die Zukunft kabelloser Kommunikation. http://www.aboutit.de/view.php?ziel=/01/25/07.html, July 2001.

[2] Bluetooth Special Interest Group (SIG). *Specification of the Bluetooth System, Version 1.1*. Bluetooth Special Interest Group, 2001.

[3] Ericsson Inc. Bluetooth History. http://www.ericsson.com/bluetooth/companyove/history-bl/, 2003.

[4] A. Frei, A. Popovici, and G. Alonso. Event based systems as adaptive middleware platforms. In *Workshop of the 17th Europeean Conference for Object-Oriented Programming*, July 2003.

[5] Heise News Ticker. Bluetooth 1.2 kommt. http://www.heise.de/newsticker/data/jk-17.06.03-004/, 2003.

[6] M. Holtmann. Bluetooth hardware support for bluez . http://www.holtmann.org/linux/bluetooth/devices.html, February 2002.

[7] M. Holtmann. BlueZ User Mailing List. http://sourceforge.net/mailarchive/message.php?msg_id=5846139, August 2003.

[8] M. Holtmann. CSR BlueCore specific information. http://www.holtmann.org/linux/bluetooth/csr.html, June 2003.

[9] M. Holtmann. Von Pinguinen mit blauen Zhnen. 2003.

[10] Internet Engineering Task Force (IETF). A UUID URN Namespace. http://www.ietf.org/internet-drafts/draft-mealling-uuid-urn-00.txt, October 2002.

[11] D. Klammer, G. McNutt, B. Senese, and J. Bray. *Bluetooth Application Developer's Guide: The Short Range Interconnect Solution*. Syngress Publishing Inc., first edition, 2002.

[12] M. Kofler. *Linux: Installation, Konfiguration, Anwendung*. Addison-Wesley, 2002.

[13] Open Source. Project BlueZ web site. http://bluez.sourceforge.net, 2000.

[14] Open Source. Project JXTA web site. http://www.jxta.org, 2001.

[15] Open Source. Project JXME web site. http://jxme.jxta.org/servlets/ProjectHome, 2002.

[16] J. Oraskari. Bluetooth versus WLAN IEEE 802.11x.

[17] Rococo Software. User Guide: Impronto Developer Kit For Linux, 2001-2003.

[18] B. Siddiqui. JXTA4J2ME Implementation Architecture, November 2002.

[19] J. Stuphorn. Was ist Bluetooth. http://www.holtmann.org/lecture/bluetooth/bt_slides_021031.pdf, October 2002.

[20] D. Suvak. Irda and bluetooth: A complementary comparison. 2000.

[21] A. S. Tannenbaum. *Computer Networks*. Prentice Hall, Inc., third edition, 1996.

[22] R. Wattenhofer. Lecture: Mobile Computing, 2003.

List of Figures

1.1 Illustration of the Terminology at Different layers 5

2.1 Piconet Overview 7
2.2 Scatternet Overview 8
2.3 Bluetooth Protocol Stack Overview 9
2.4 Fragment of the BlueZ Bluetooth Protocol Stack Overview . . . 14

3.1 UML Diagram illustrating a PeerNetwork 19
3.2 Data Processing of Incoming Data 20
3.3 Data Processing of Outgoing Data 20
3.4 UML Diagram of a Message and its Elements 21
3.5 Message Header 23

4.1 Overview of RFCOMM and TCP/IP Stack 26
4.2 Architecture Overview of Benchmark Environment 26
4.3 Illustration of the measured time 27
4.4 Dependency Graph of the Benchmark Environment 28
4.5 Measured time by given amount of data 29
4.6 Comparison of the Standard Deviation 30
4.7 Comparison of the average round trip time per byte 31

5.1 BAD2 User Interface 33
5.2 Message Sequence Diagram 34

G.1 Measured time by given amount of data 65
G.2 High and low water marks 68
G.3 Comparison of RFCOMM vs. TCP/IP 68
G.4 Comparison of the average transmission time 69
G.5 Comparison of the Standard Deviations 69

List of Tables

2.1 Bluetooth Radio Power Classes 6
2.2 Brief Comparison of IrDA, Bluetooth, and WLAN 12
2.3 Brief Comparison of Different Stacks 15

3.1 Predefined Namespaces 22
3.2 Description of the Message Header 22
3.3 Description of a Message Element 23
3.4 Description of a Message Element (user defined mime type) . . . 23

5.1 Example of an Initial Message 34
5.2 Example of a Notification Message 35
5.3 Example of an Acknowledgment 35

G.1 Configuration of the test systems 64
G.2 RFCOMM Data Ascertainment 66
G.3 TCP/IP Data Ascertainment 67

Listings

3.1 Sample Message 21
3.2 Code Snippet of Class *BTConnectionHandle* 24
B.1 Shell script to create the devices (`create_dev`) 43
E.1 Shell script to start Eclipse 62