Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Ganymed Stanek

Bluetella: A Java Application for New Mobile Phones

Student Thesis SA-2003.19
Summer Term 2003

Tutor: Matthias Bossardt

Supervisor:
Prof. Dr. Bernhard Plattner

4.7.2003

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Contents

Abstract **i**

1 Introduction **1**

2 Java Capabilities of Mobile Phones (J2ME CLDC MIDP) **3**
2.1 Java Editions for Mobile or Embedded Devices 3
2.2 Version Numbers 5
2.3 Visual User Interface 5
2.4 Keys 7
2.5 Data Storage 7
2.6 Sound 8
2.7 Special Features 8
2.8 New Features of MIDP 2 8
2.9 Connectivity in Java 9
2.9.1 IrDA 9
2.9.2 Bluetooth 9

3 Bluetella: Filesharing for Mobile Phones **13**
3.1 Selecting an Application 13
3.2 Bluetella Filesharing Scenario 15
3.3 Why Bluetella is not Gnutella 16

4 Bluetella Protocols **19**
4.1 The Necessity for Two Protocols 19
4.2 Field Format 21
4.3 Bluetella Source Routing Protocol 21
4.4 Bluetella File Sharing Protocol 23

5 Bluetella Implementation **25**
5.1 User Interface 25
5.2 Threads 27
5.3 Handling the Protocol Headers 29
5.4 Distance Abstraction Layer 30
5.5 Data Storage Revisited 30

6 Development Environment **31**
6.1 Sun ONE Studio 31
6.2 Mobile Phone Emulators 31
6.3 Rococo Impronto Simulator 31

7 Conclusion and Future Work **35**

A How to Compile and Execute Bluetella on Your PC **37**
A.1 Software Installation 37
A.2 Compiling and Running Bluetella 38

B Assignment of the Semester Thesis **41**
B.1 Introduction 41
B.2 Assignment 41
B.2.1 Objectives 41
B.2.2 Tasks 41
B.3 Deliverables and Organisation 42

C Timeline **43**

List of Tables

2.1 Elements that can be placed into a Form Screen 6
2.2 Derived classes from Screen which represent a complete Screen 7

4.1 Combination of BSR and BFS protocol message types used together 19
4.2 Bluetella Source Routing (BSR): common fields 22
4.3 Bluetella Source Routing (BSR): 'broadcast' specific fields . . 22
4.4 Bluetella Source Routing (BSR): 'unicast' specific fields . . . 23
4.5 Bluetella File Sharing (BFS): common fields 23
4.6 Bluetella File Sharing (BFS): 'searchRequest' specific fields . 24
4.7 Bluetella File Sharing (BFS): 'searchResponse' specific fields . 24
4.8 Bluetella File Sharing (BFS): 'packetRequest' specific fields . 24
4.9 Bluetella File Sharing (BFS): 'packetDelivery' specific fields . 24

List of Figures

2.1 Overview of the various Java versions [J2M] 4
2.2 Siemens SL45i, the first mobile phone with J2ME CLDC . . . 5
2.3 Simplified Bluetooth protocol stack 10

3.1 General filesharing scenario 15

4.1 Two in one sequence diagram: BSR and BFS protocols 20

5.1 Screen shown when beginning to use the Bluetooth functionality 26
5.2 Additional screens 27
5.3 Downloading a file 28

6.1 Sun ONE Studio 32
6.2 Rococo Impronto Simulator with Siemens SL55 Emulators . . 33

Abstract

New mobile phones provide a Java Virtual Machine for development of device independent software. Subject of this thesis is to demonstrate the possibilities and limitations of this JVM by implementing an application running on today's or near futures phones. A file share client similar to Gnutella, but using Bluetooth instead of the Internet protocol, was chosen and was therefore given the name Bluetella. The fileshare network involving several clients is as well a good demonstration of the newly specified Java API for Bluetooth Wireless Technology (JABWT), which is significantly increasing the possibilities of the Java Micro Edition. As no phones with this API were available at the time of writing, a third-party simulator had to be used, which turned out to bring several problems along. Beside that, the lack of an ad-hoc routing API on the MIDP platform required the implementation of a routing protocol. Because the focus of this thesis wasn't on advanced ad-hoc routing algorithms or their implementation, an own, more simple routing algorithm was defined based on source routing, but kept general. The actual file sharing protocol was then defined on top of that source routing protocol with strict layer separation in order to reuse both protocols independently, or for a later upgrade to an advanced ad-hoc routing protocol, which would reduce file transfer interruptions due to network topology changes.

Chapter 1

Introduction

A growing percentage of the mobile phones available today support the execution of Java software. The fact that millions of these phones are already sold and that Java is a device independent programming language bears a huge potential for new little software utilities that could change the way of our everyday life.

One of these applications that is missing on mobile phones today is a fileshare client similar to Gnutella or KaZaA available on personal computers. Implementing an ordinary fileshare client for mobile phones accessing these desktop filesharing networks via Internet would be possible[1] although not always a desirable solution. Data transfer rates for Internet access in mobile phones are still down near analog modem speeds while prices for the transferred megabytes are still impractically high at around 15 sFr[2]. Additionally, at some places, the mobile phone carrier network isn't available at all, or its reception quality is varying very strongly as in public transportation systems, especially trains. For these reasons, direct Bluetooth communication between mobile phones is an attractive alternative to an Internet connection. The combination of a fileshare client similar to Gnutella, but using an own Bluetooth network, is represented in the name Bluetella which was chosen for the application implemented in this thesis.

After getting an overview of the different editions of Java for mobile or embedded devices in section 2.1, the rest of chapter 2 concentrates on the most promising one and looks at its possibilities and limitations. Based on these capabilities, several possible applications are looked at in section 3.1 of which the fileshare client mentioned above was chosen. Chapters 4 to 5 deal with the protocols developed for this application and its implementation.

[1] Above version 2 of MIDP due to the required socket connections.

[2] Swisscom GPRS prices as of 25th July 2003 are 1.90sFr/100kB below 1 Megabyte a month and 1 sFr above one Megabyte

Chapter 6 presents the development environment in a general way.

Chapter 2

Java Capabilities of Mobile Phones (J2ME CLDC MIDP)

2.1 Java Editions for Mobile or Embedded Devices

Java entered the market of mobile or embedded devices with a one for all solution called PersonalJava, which contained almost the functionality of the JDK 1.1 but needed impractically powerful devices. In 1999 when all Java editions were renamed and regrouped, a new strategy was chosen for mobile or embedded devices [Relb]. The very different hardware capabilities and usage scenarios found in these devices ranging from watches to TV set top boxes or car navigation systems required a more versatile approach than the one for all solution of PersonalJava. Under the name *Java Micro Edition (J2ME)* two different so called *Configurations* were defined, the Connection Device Configuration (CDC) and the Connection Limited Device Configuration (CLDC) (see figure 2.1).

While the relatively powerful CDC was targeted to replace PersonalJava, CLDC approached a new market of even smaller devices like low to mid-end mobile phones. In summer 2001 Siemens was the first manufacturer to include J2ME CLDC into their mobile phone SL45i [KB] (see figure 2.1).

Today, CLDC is implemented in millions of phones worldwide while in the high end range the CDC was not yet able to substitute for PersonalJava. In fact, high end mobile phones such as the SonyEricsson P800 offer PersonalJava and J2ME CLDC, but no CDC despite the fact that SUN announced the death of PersonalJava back in 1999.

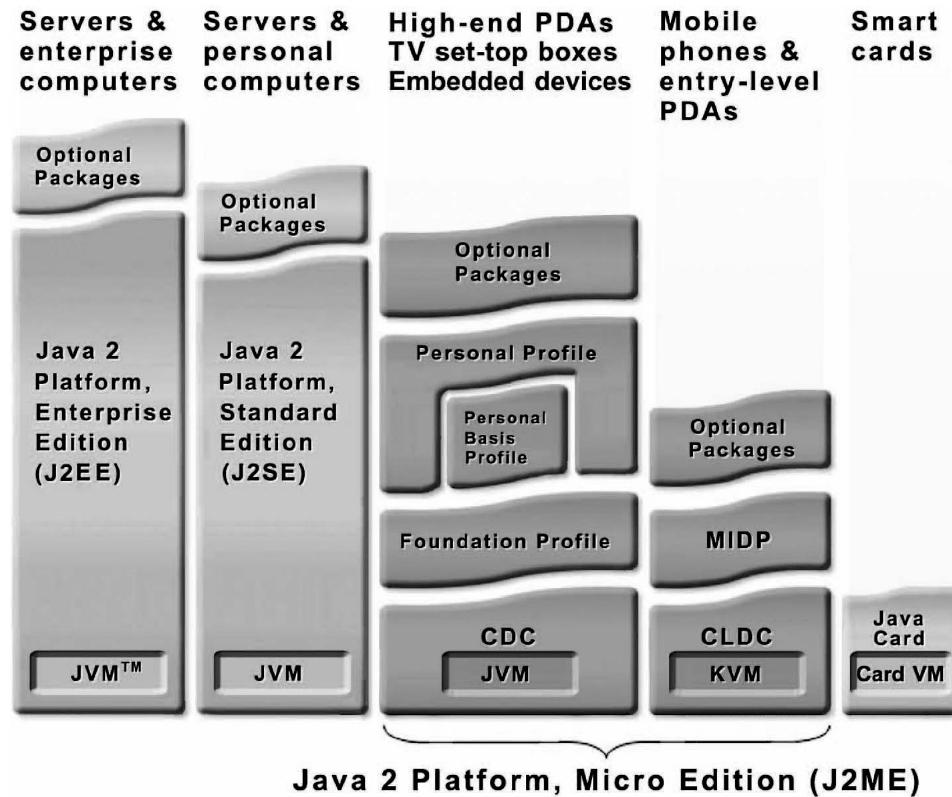


Figure 2.1: Overview of the various Java versions [J2M]

As J2ME isn't used just in mobile phones, it wouldn't make much sense to include phone book access functions from a mobile phone into devices without phonebooks. Therefore, an additional layer called *profile* was built on top of the *configuration* which includes all functionality that a group of devices has in common. Mobile phones for instance use the *Mobile Information Device Profile MIDP*.

Because there probably are still significant differences among this group of devices, again another, but optional layer called *packages* has been added on top of the *profile* layer. One of these optional packages for instance is Bluetooth, another is Web Services. For further information about the organization of the different Java editions refer to [J2M].

The vast majority of today's mobile phones uses the combination J2ME CLDC MIDP. Therefore, and because SUN has positioned it as the future programming API for mobiles, this thesis will concentrate on this set of Java APIs.

Figure 2.2: Siemens SL45i, the first mobile phone with J2ME CLDC

2.2 Version Numbers

The configurations, profiles and optional packages all have their own version numbers. The whole thesis except the outlook in section 2.8 is referring to version 1.0.4 of CLDC and version 1.0.3 of MIDP due to the following reasons:

- MIDP 2 is specified by sun, but not yet implemented by the mobile phone vendors.
- its features are not of specific use for the application implemented (see section 2.8).
- MIDP 2 is backward compatible to MIDP 1.x.
- the later mentioned and needed Rococo Impronto Simulator doesn't work yet with MIDP 2 due to a bug in the Impronto Simulator.

2.3 Visual User Interface

MIDP applications are built to run on many different devices without modification. This is particularly difficult in the area of the user interface because devices have screens of all sizes, in grayscale and in color. Furthermore, devices vary widely in their input capabilities, from numeric keypads to alphabetic keyboards, soft keys, and even touch screens. The minimum screen size mandated by MIDP is 96×54 pixels. As far as color depth is concerned, there is no restriction (thus including 1-bit monochrome). Devices are expected to have at least a keyboard or a touch screen.

From the programmers point of view, there are two ways of writing to the display:

Using the Canvas class: For being able to have full control over the display, the application has to discover the device characteristics first, such as the screen size of the mobile phone. This kind of user interface enables programming games and a lot of other visually appealing programs, but should just be used when really necessary. The code is getting bigger due to the discovery of the device characteristics and compatibility among devices is limited most often as additional assumptions about the device are made. The Canvas class isn't used in the implemented application, therefore this thesis is referring to [Esc03] and [Knu03] for a more detailed coverage of the the canvas api.

Using the various classes derived from the Screen class: Instead of specifying the exact position, size or font of a user interface element such as a textfield or button, the class `Form`, which is derived from class `Screen`, just allows to specify the order of the elements to be displayed on the screen. This way every implementation of the Java Virtual Machine can place the elements the best way, according to its knowledge of the display size, readability of fonts, colors matching the design of the device, etc. Table 2.1 is giving an overview of the various elements that can be placed into a form.

ChoiceGroup	A list with several rows, each beginning with a box that can be checked. Two types exist: EXCLUSIVE and MULTIPLE, specifying how many boxes can be checked
DateField	For user input of date and time. A calendar is displayed where the user can choose the date.
Gauge	Similar to a progress bar known from Windows. Additionally, it can be used as input by placing the bar to a certain position resulting in a linear extrapolation between some min and max values.
ImageItem	A container for an image to be displayed
StringItem	For outputting text to the display.
TextField	A field where the user can enter text. Input constraints can be defined.

Table 2.1: Elements that can be placed into a Form Screen

The other classes derived from `Screen` go even a step further in ab-

straction than the form class. Considering the small size of the display, the user interface has to be partitioned into so many screens, that lots of the forms would just contain one element as for instance a text box. Therefore special screens have been defined, which already include one type of user interface element. Table 2.2 lists these predefined subclasses of `Screen`.

TextBox	Similar to the TextField. Can be used to enter passwords in ***** mode.
List	The same as the choice group, but adding the type IMPLICIT, which is the same as EXCLUSIVE except that the user can't change his mind after selecting because an event is triggered immediately without displaying a checkbox.
Alert	Displays an alert message and depending on the alert type, waits for a certain period or until a key is pressed for getting back to the screen displayed before.

Table 2.2: Derived classes from Screen which represent a complete Screen

2.4 Keys

The keys available and their location on the mobile phone differs a lot from manufacturer to manufacturer. Following the way the visual user interface was divided into a standardized and less standardized approach, there are also two ways to use the keys. Actually, their use is associated with the visual user interface. When the displayed object is of type canvas, events can be specified for each key (therefore being laid out for a specific device).

When the displayed object is of type screen, keys can't be assigned events and are just used for navigating around the fields in the screen or for entering data. Which key exactly is used for a certain function is specified by the manufacturer of the mobile phone during the implementation of the Java Virtual Machine.

2.5 Data Storage

The Java Microedition doesn't offer access to the real mobile phone filesystem, instead it creates something like the application's own filesystem called RecordStore. A RecordStore consists of a collection of records which

remain persistent across multiple invocations of the MIDlet. A Record, which is the analogon of a file, contains a byte array. We can regard the RecordStore as a class instance, that survives the garbage collector and keeps its content (Records) during reboots or battery changes. RecordStores are created in platform-dependent locations, which are not exposed to the MIDlets. It is possible for a MIDlet to have more than one RecordStore. But most important, a RecordStore belongs to one MIDlet, so it is NOT accessible from other MIDlets. Thus we can use it to store settings for instance, but not for data interchange with other programs.

For being able to access the real mobile phone filesystem despite this limitation, some mobile phone manufacturers provide their own, unstandardized API as for instance Siemens, who offers the class *com.siemens.mp.io.File* for file access of their filesystem called MMC. [sie]. The older Siemens mobile phones available today (SL45, C-series) just allow access to the files in the same folder as the MIDlet itself. The new models (SL55 and S-Series) show an alert screen (see figure 5.1) when the currently running MIDlet tries to access a file and asks the user if the access to the specified directory is allowed, thus enabling access to all directories.

2.6 Sound

MIDP 1.x does NOT offer the ability to play sounds or any other media files as videos. MIDP 2, however, will use a subset of the future optional package JSR-135 Mobile Media API to play sound files.
Again, some mobile phone manufacturers have recognized the need for a media API already some time ago and provided their own such as Siemens (class `com.siemens.mp.media`).

2.7 Special Features

Most mobile phones include a vibrating motor, some offer different colors for display background lighting. Features like these could be well used in programs for getting the user's attention, but unfortunately are neither part of the J2ME CLDC MIDP api. Siemens provided again APIs to access these functions via the classes `com.siemens.mp.game.Vibrator` and `com.siemens.mp.game.Light`.

2.8 New Features of MIDP 2

Despite MIDP 2 isn't used in the application implemented, the following simplified list of new features in MIDP 2 [mid] was added for giving you

a feeling in which direction the J2ME CLDC MIDP platform is moving. All but the 3rd and last two points are directed towards providing needed features for commercial applications.

- A new security model needed for m-commerce applications.
- A standardized over the air (OTA) provisioning mechanism.
- Gaming enhancements.
- Connectivity standards such as serial ports and sockets.
- Push architecture so that midlets can be activated by a server, i.e. the carrier.
- A subset of the Multimedia API to play sounds.
- Some minor user interface improvements.

2.9 Connectivity in Java

The only type of connection that MIDP 1.x requires to be implemented by the Java Virtual Machine are http connections. Assuming Internet access was made available through some other software on the mobile phone, HTTP Get, Post and Head requests can be sent to a url such as `http://www.stanek.ch/ganymed/processData.jsp?firstParameter=red`.

Other types of connections like datagram or socket connections might be available on some mobile phones, but are not needed for MIDP 1.x compatibility. MIDP 2 offers several other connection mechanisms such as serial port, https, UDPDatagramConnection, and optionally, socket and secure socket (ssl) connections [Knu03].

2.9.1 IrDA

IrDA connections are not available in MIDP, but Siemens offers a manufacturer specific API `com.siemens.mp.io.Connection`. An API standardization process for IrDA similar to the one for Bluetooth described in the next section hasn't taken place till today.

2.9.2 Bluetooth

Much more versatile than IrDA is the use of Bluetooth as wireless transfer technology as it is not limited to line of sight connections. The mobile

phone industry, recognizing the potential of a standardized Bluetooth API, began working on the *Java API for Bluetooth Wireless Technology JABWT* in 2000. Often this API is referred to with the short form *JSR-82* which stands for *Joint Specification Request*, being the equivalent of a *Request For Comment (RFC)* in the Java world.

JABWT is an optional package, meaning it can be included into the implementation of the Java Virtual Machine by the manufacturer of the mobile phone depending if a Bluetooth module is available in the device. Despite the final specification was presented in 2002 and a substantial part of the mobile phones available would have built in Bluetooth hardware, this API is not yet available in any mobile phone on the market. The Bluetooth modules are so far just accessible via C for third party developers or are used in firmware software from the mobile phone manufacturer like for synchronization of addresses.

Having a standardized Bluetooth API can't be taken for granted. C for instance does not yet have such a standardized API, which results in code being dependent on the Bluetooth hardware module used.

For being able to use JABWT, it is useful to have a look at the simplified Bluetooth Protocol Stack shown in figure 2.3.

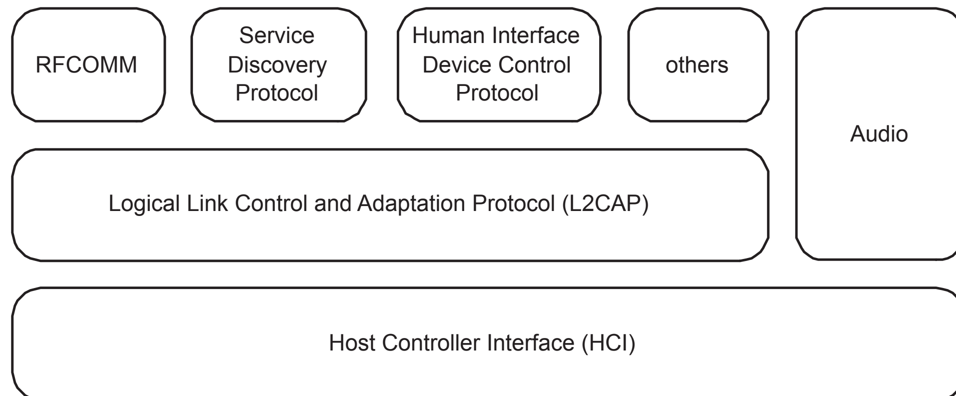


Figure 2.3: Simplified Bluetooth protocol stack

The Host Controller Interface layer is acting like a device driver for the various Bluetooth hardware modules. Data integrity isn't assured by this layer, but by the following L2CAP layer. Besides reliable transmission of data packets, L2CAP is offering multiplexion for higher protocols. All data is passing through this layer, except audio links not needing reliable transmission have direct access to the HCI layer (i.e. wireless mobile phone headsets).

The RFCOMM protocol sits one layer above L2CAP and is commonly known as wireless serial port or cable replacement protocol [HA03] because it simulates the functionality of a standard serial port by offering a stream without a maximum transfer unit.

There are a lot of other protocols that use the L2CAP layer, but we are not explaining them further as JABWT gives us just access to the L2CAP and RFCOMM protocols. This is limiting the usability of the JAVA - Bluetooth combination in two areas and forcing the programmer to use C++ or other languages:

- Signal strength measurement (used for distance calculation)
- Voice applications such as for instance voice recognition from a Bluetooth headset.

We conclude that all we are able to do with JABWT is transmit reliable data, either by wrapping it into L2CAP packets or by sending it over RFCOMM, the "virtual cable". But where do we send it to? We do not have an IP address of the other node and we probably don't have its Bluetooth address,[1] either.

The way we find the information for connecting to the other devices is called *Discovery*. But before we can discover, we first need to specify *Services*. A service is our partner software to which we like to connect to. Each service running on a device has an unambiguous port on which it is listening for incoming connections similar to sockets in the ip world, except that the port number is called service url here and is of the form `"btspp://localhost:00112233445566778899AABBCCDDEEFF;name=bluetella"`. The first part `btspp://localhost` specifies an RFCOMM connection and the 32 digit hexadecimal number is called *Universal Unique Identifier UUID* and should be unambiguous for the service. There is a list of already assigned UUIDs in [UUI]. The attribute `name` is optional and just for human readability.

Having some services waiting for connections on these service urls, we can start searching for these services in two different ways:

- When we need to know about all services available, we need to search first for all devices (`javax.bluetooth.DiscoveryAgent.startInquiry()`) and then search on each device individually for the service of interest (`javax.bluetooth.DiscoveryAgent.searchService()`).

[1] Bluetooth addresses are written into the Bluetooth chipset during production similar to a MAC address being written onto the Ethernet card and are 48 bits long.

- When we just need to get a connection to any one one of the services, we can directly search for services with the method `javax.bluetooth.DiscoveryAgent.selectServices()`, which cancels the search after the first matching service was found.

For each service found we receive a *serviceRecord* from which we can extract a *connection url* used for connecting to the host that offered the service. The connection url should not be mixed up with the service url which is the url used for advertising a new service. Having the connection url, we can decide if we like to open an L2CAP or RFCOMM connection to the service. In case of RFCOMM both, the client and server (the one who offered the service) get a Java StreamConnection to each other from which a InputStream and OutputStream can be opened as usual.[2] In case of L2CAP the client and server get a L2CAPconnection, which is a class from JABWT and can be used according to the JavaDoc of JABWT [jav].

Readers familiar to Bluetooth might wonder why multipoint/singlepoint capability of nodes isn't mentioned or why we didn't specify which node is taking the role of the master of a connection and which the slave, so that all nodes that like to talk to each other are able to.[3] All this is hidden behind the JABWT which groups Piconets into Scatternets the way it is most likely the most fortunate at each time and therefore cannot be influenced, making programming more simple but reducing also the amount of optimizations that can be taken.

[2] for instance when writing and reading to a file in the first week of learning how to program Java

[3] Just a master and slave can talk to each other, no other combination. There can be maximal 7 slaves connected to a master

Chapter 3

Bluetella: Filesharing for Mobile Phones

3.1 Selecting an Application

When we want our application to be portable between various mobile phones from different manufacturers, we need to restrict ourselves to the use of the pure J2ME CLDC MIDP API without any manufacturer specific APIs or future optional packages. This limitation is hard to live with, as our only way of communication would be the rather expensive way of http using the Internet connection provided by the mobile phone. Therefore, most applications available that restrict themselves to these APIs are single player games which do not use any form of communication and do not need to access the filesystem. For storing highscores, the RecordStore is sufficient.

The optional package for Bluetooth will increase the field of possible applications drastically enabling extensive mobile phone interaction which isn't used today. As the API specification is so recent and the first mobile phones will not appear on the market before the end of 2003, we decided to search for an application making use of this Bluetooth API.

When thinking about Bluetooth one often thinks about transferring large packets of data, ignoring the huge amount of applications that just need to deliver one or maybe a couple of bits of information where datarate is absolutely unimportant, but the low power consumption of Bluetooth is.

One of these groups of Bluetooth applications are the so called proximity applications. Analyzing the signal strength, they are able to know how far away other senders are. This knowledge is used for localizing people indoors where GPS isn't working or for localizing lost objects like wallets, keys or mobile phones. The signal strength could also be used to turn on

or off lamps in a house when walking around, or traffic lights to react on arriving cars, to name just a few of the uses. Unfortunately, this whole area of applications is out of reach for J2ME programs, as JABWT just offers methods for data transfer and no information about the signal strength, so far.

Another group of these low information uses of Bluetooth are key access systems. Having stored a secret key on the simcard or the memory of the mobile phone, one could imagine using it to open garage doors or adjust the seats and radio station in the car to the momentary driver.

Similar to the use above is the use of Bluetooth for payment. While mobile phones are used more and more for payment via sms or payment via a phone call, there will always be regions without network cover. The payment could then be done with extensive security mechanisms over Bluetooth and later, when being back in a place with network cover, synchronized with a bank account. In fact, among several other companies, VISA is working on payment via Bluetooth [Rela].

Of course the groups of applications described in the last two paragraphs would be possible to implement with the current JABWT, but they all make use of just one single connection a time between two partners. Much more interesting would be to have a peer-to-peer network of dozens of mobile phones to interact with each other. That's why we abandoned the ideas above.

Two technically similar applications would make much more use of Bluetooth connectivity: a dating application and a flea market. Both compare profiles - and in case of matches - inform the user. These applications would have the advantage over their Internet versions that the users would physically be not much further away than 10 meters, but of course also have the disadvantage that matches would be much less frequent.

After all, we decided on a fileshare client (described in the next section), as it simply demonstrates network functionality the best among the mentioned application ideas due to the fact that there are always several connections open at the same time to several different peers. Additionally, one of these connections is not just involving two nodes, but also all nodes in between that are necessary for routing the packets to their destination node. Finally, nobody published any information about an already existing implementation on the Internet so far, enlarging the chances that we are the first to implement a Bluetooth fileshare client.

3.2 Bluetella Filesharing Scenario

For better understanding of Bluetella, let's imagine sitting bored in one end of a train coach with Bluetella available on our mobile phone. Lets assume some of the other passengers also have Bluetella running on their mobile phone and that the range of each mobile phone is about 10 meters.[1] A situation like this is drawn in figure 3.1

If we now would like to check if one of the other users of Bluetella, including the ones sitting at the other end of the coach or maybe even train have some interesting files shared on their mobiles, all phones in between need to work as relay stations for our search request.

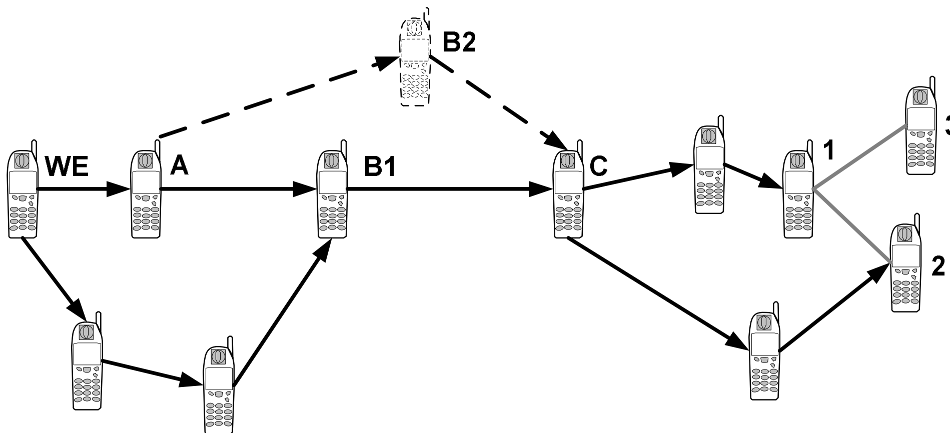


Figure 3.1: General filesharing scenario

We could imagine searching for ID3 tags of mp3s specifying a song title or user preference of the files among many other possibilities known from the desktop filesharing programs. For keeping the space-limited user interface simple and concentrating on the implementation of the network functionality, Bluetella is limited to searches for filenames or parts thereof. This way its still possible to list all files available by searching for an empty string "" or listing all textfiles by searching for ".txt".

Having chosen a file for download, the request must somehow reach the owner of the file so that he can send it back to us divided in packets. We can imagine a situation where we (marked `WE` in figure 3.1) are in the middle of a download and one of the passengers `B1`, working as relay station between

[1]This is referring to class 3 Bluetooth devices. Class 2 devices reach 20m and class 1 100m, but they are not built into mobile phones due to high energy consumption. [HA03]

other relay stations `A` and `C`, leaves the train at a stop interrupting the data transfer. For being able to resume the transfer, we can imagine a person `B2` that was sitting next to `B1` also being in range of `A` and `C` to take over the relay function between `A` and `C`. Maybe `B2` was not yet sitting beside `B1`, but entered the train some minutes after `B1` left. In both situations, it would be theoretically possible to resume the data transfer. For being able to recognize situations like this, we would need an advanced Ad-Hoc routing algorithm. Implementing one of these algorithms[2] has been excluded from this thesis in favor of a more simple self-defined Ad-Hoc source routing protocol (see section 4.3).

For not having our searches bother mobiles too far away, we limit the number of hops that continue spreading our search. In the example in figure 3.1 the search is limited to 5 hops. Still being node `WE`, we have no chance to find files from phone `3`, but files from `1` and `2` will be found by those search packets not going over more than 5 hops.

3.3 Why Bluetella is not Gnutella

The Gnutella protocol [gnu] is relying on an underlying IP layer, resulting in several differences to Bluetella:

Gnutella always needs a first IP address of another Gnutella client, for getting to know other clients to which it can send its search packets to. Blutella, being able to use Bluetooth service discovery, doesn't need a first address of another Bluetella client, just the service url mentioned in 2.9.2 for starting a service discovery. With the connection urls Bluetella gets back from the service discovery, it can build connections to all Bluetella clients within reach. Resulting from this, the next hops of a Bluetella client are within more or less 10 meters, while in Gnutella the next hops are spread all over the world.

Search packets in Gnutella are broadcasted without remembering the way they took. A computer that hosts the file being searched for (lets call him server) replies directly to the searching client's ip address. This answer packet as well doesn't have any information about the way between the client and the server, just the IP addresses of the two - leaving routing to IP. Most likely, the way, which the packets being sent directly between the client and the server take, is completely different from the way, on which the search packet reaches the server. Bluetella, not having the possibility of sending packets to a globally known address, needs to keep track itself of

[2]For instance the Dynamic Source Routing (DSR) protocol.

the way the search packets find the servers.

When Gnutella is deciding to talk to a new host whose IP address it got from one of the already connected hosts, the connection set up time will be in the order of milliseconds. Blutella, having to start a new service discovery for getting to know new hosts, needs in the order of 10 seconds to build up a new connection.

From time to time, Gnutella is sending out ping messages and replying to incoming ping messages with pong messages for updating its list of available Gnutella clients. As the Gnutella neighbors are physically very far and many ip hops apart from each other, lot of unnecessary traffic would be generated when a file transfer is started to a client that wasn't available anymore. Bluetella, being directly connected to its neighbors, doesn't need this functionality as it won't even be able to send data off to a lost partner as the latter is not sending any acknowledge messages to the underlying Bluetooth handshake mechanism. Additionally, Bluetella would not be able to send the packet on another way to its destination anyway, as it is relying on the hops collected during broadcast of the search packet.

Furthermore, Gnutella offers some functionality to work behind a proxy, omitted in Bluetella.

Chapter 4

Bluetella Protocols

4.1 The Necessity for Two Protocols

For being able to replace our limited ad-hoc source routing mechanism mentioned in section 3.2 at a later time, we like to separate delivery of the packets from the actual file sharing protocol. The protocol defined for delivering our packets in the Bluetooth network is called Bluetella Source Routing (BSR) and is described in detail in section 4.3 and the protocol taking care of the file sharing functionality is the Bluetella File Sharing (BFS) Protocol defined in section 4.4.

Both of these protocols consist of several message types, but the combination of message types used together is always the same. Table 4.1 shows all message types that exist and in which combination they are used together.

BSR message	**BFS message**
broadcast	searchRequest
unicast	searchResponse
unicast	packetRequest
unicast	packetDelivery

Table 4.1: Combination of BSR and BFS protocol message types used together

As the combination of messages used in conjunction is always the same, the sequence diagrams for the two protocols are joined in figure 4.1. Lets assume again the more or less one dimensional positions of mobile phones in a train (NOT the same as in figure 3.1) and the reception range to be again 10 meters. The distance between the mobiles is given in meters on top of the sequence diagram. The two sided arrows between the mobiles show which ones are in reach of each other (according to the distance). The broken ar-

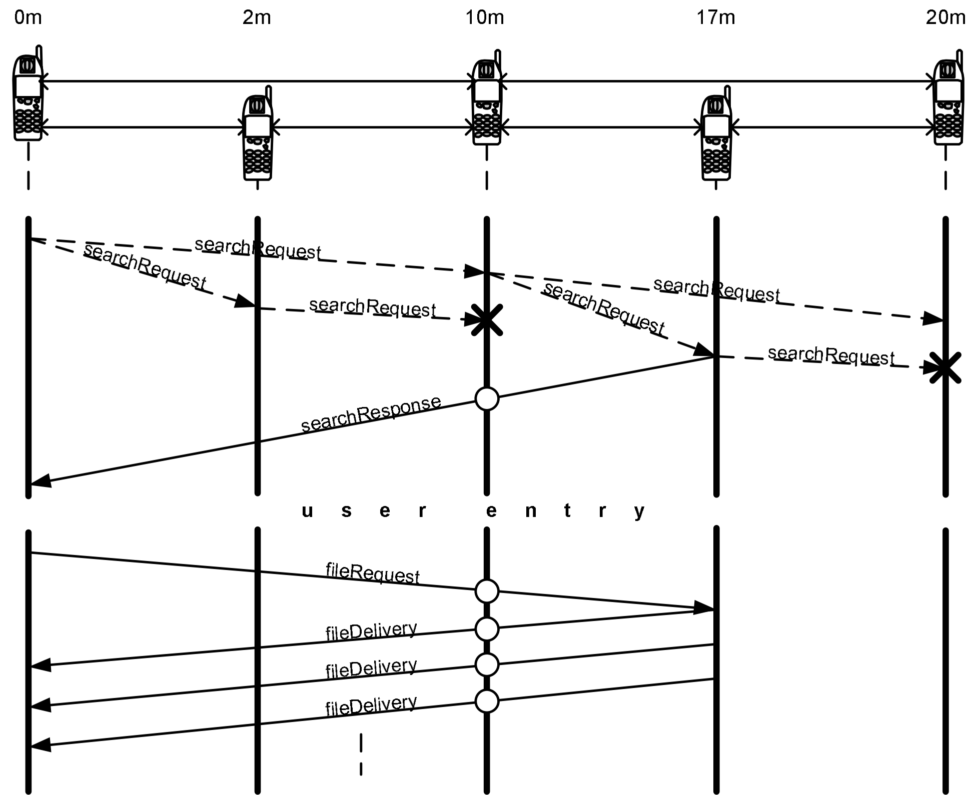


Figure 4.1: Two in one sequence diagram: BSR and BFS protocols

rows represent packets sent via BSR_Broadcast, the solid arrows packets sent via BSR_Unicast. The labels represent BFS packet names and the circles mark hosts, where just the BSR packet is looked at, but not the BFS packet.

When we follow the data flow beginning on the top left corner, we see that the searchRequest packet passing directly to the mobile phone at the 10m marker is further broadcasted while the one that arrived via the mobile phone at 2m was discarded. The decision, which one of the two is chosen to be forwarded is based on time, NOT number of hops. If the way via the additional hop had been faster, chances are high, it would be faster again in future, so being the way to choose in that case. Additionally, using time as criteria, the first packet can already be forwarded before the second packet has arrived.

The mobile phone at the 17m marker is the only one hosting the file we are looking for. We recognize this as no other phone is answering the search with a searchResponse packet, which is sent directly back to the client

searching for the file. Of course the file has to pass via the mobile phone at the 10m marker, but the payload of the BSR packet has not to be looked at, symbolized with the circle. The way the unicast searchResponse packet is taking back to the client who initiated the search is exactly the same as the one of the broadcast packet, that reached the server first. If a server hosts several files that match the search string, then it does not need to send back several searchResponse packets as it can put all answers into one packet.

In our example, the client receives just one searchResponse packet, but of course it is likely that a whole bunch of answers from different phones get back to the him. After the user chooses any one of them, his Bluetella client sends a fileRequest packet to the server, which then sends the file, divided into fileDelivery packets, back to the client. Again, the way these two packets take is the same as the one found with the broadcast packet and already used for the searchResponse packet.

4.2 Field Format

Before we can define the protocol fields in detail, we need to specify how each of this field is actually filled with data in case of number and text content:

Number fields consist of a certain number of bytes specified in the protocol specification. Inside a byte, the bits are set according to the big endian standard, as the bytes are written by Java in our implementation, which always uses big endian. If a number is bigger than one byte, a long for instance, then it is written as several bytes in a row, again ordered in big endian style.

Text fields do not have terminating characters, their length is always given in a number field preceding the text field.

4.3 Bluetella Source Routing Protocol

The Bluetella Source Routing Protocol consists of two different message types: broadcast and unicast. The beginning fields of both message types are the same, shown in table 4.2. The client Bluetooth address clientBTaddr is the address of the host who initially launched the broadcast packet into the net for building up the connection.

Lets first concentrate on the broadcast packet. At the time it is created on any one of the nodes, the *number of hops* field is set to 0 and no hop fields are following. The packet is sent out on each connection with a value of 7 in the *depth* field and a *packetID* being unambiguos for the packet's

clientBTaddr. Each hop receiving the packet checks if it is the first time a packet with this *packetID* from that *clientBTaddr* arrived. In case the packet was seen before, it is discarded. If it's new, then the *numberOfHops* field is incremented and the processing node adds it's Bluetooth address to the hoplist.

When one of the nodes wants to respond back to the client, all it needs to do for getting the unicast message header is to take the packet header, change the *messageType* to unicast, add its Bluetooth address to the *serverBTaddr* field instead of the hops list and set the LSB bit of the *flag* byte to 1 for signifying a backward direction (serverBTaddr, lastHop, ..., hopTWO, hopONE, clientBTaddr) and add the new payload. Later, when the client wants to send again a packet to the server, it just has to change the direction flag again to 0, add the new payload and send it off.

field name	**bytes**	**explanation**
protocolVersion	1	1
headerLength	2	just the BSR header (# of bytes)
payloadLength	4	BSR Payload (# of bytes)
messageType	1	0 = broadcast 1 = unicast
clientBTaddr	6	the one who initiated the connection with a broadcast
numberOfHops	1	client and server addr don't count as hops
hopONE	6	Bluetooth addr of hop one
hopTWO	6	...

Table 4.2: Bluetella Source Routing (BSR): common fields

field name	**bytes**	**explanation**
packetID	2	recognition of loops
depth	1	numberOfHops to fload

Table 4.3: Bluetella Source Routing (BSR): 'broadcast' specific fields

field name	bytes	explanation
serverBTaddr	6	the other end of the client
flags	1	LSB: direction 0 = forward 1 = backward further bits: future use

Table 4.4: Bluetella Source Routing (BSR): 'unicast' specific fields

4.4 Bluetella File Sharing Protocol

The Bluetella File Sharing Protocol is made of four message types. The common fields are analog to the common fields of the Source Routing protocol's common fields and shown in table 4.5. The only type with a payload is *packetDelivery*, which contains the file to be transferred. The other fields should be self-explaining together with the description in table 4.6 to 4.9.

field name	bytes	explanation
protocolVersion	1	1
headerLength	2	just the BFS header (# of bytes)
payloadLength	4	BFS Payload (# of bytes)
messageType	1	0 = search 1 = searchResult 2 = packetRequest 3 = packetDelivery

Table 4.5: Bluetella File Sharing (BFS): common fields

The *searchRequest* packet also contains a flag field of which just the LSB bit is used. If it is set to 1, it specifies if gateways, who receive the search-Packet should propagate the search in their networks and deliver the results back.

The number of parts and their size a file is partitioned into is up to the implementation to decide, as long as the length can be written into the BSR header, which is satisfied up to a payload length of about 4 GB ($2^{32} - BFS_headerLength$ bytes).

field name	**bytes**	**explanation**
flags	1	LSB: include Gateway to the Gnutella network 0 = false; 1 = true; other bits: future use
searchStringLength	1	
searchString	searchStringLength	maximal 256 characters

Table 4.6: Bluetella File Sharing (BFS): 'searchRequest' specific fields

field name	**bytes**	**explanation**
numberOfResults	1	maximal 255 hits
fileID	3	more than one file can have the same fileName
fileSize	4	maximal 4GB
fileNameLength	1	
fileName	fileNameLength	

Table 4.7: Bluetella File Sharing (BFS): 'searchResponse' specific fields

field name	**bytes**	**explanation**
fileID	3	more than one file can have the same fileName
partOfFileNumber	2	0 for request of all parts of the file 1-65535 for request of just one part

Table 4.8: Bluetella File Sharing (BFS): 'packetRequest' specific fields

field name	**bytes**	**explanation**
fileID	3	more than one file can have the same fileName
partOfFileNumber	2	which part of the file is being delivered now
payload	payloadLength	the actual part of file transmitted

Table 4.9: Bluetella File Sharing (BFS): 'packetDelivery' specific fields

Chapter 5

Bluetella Implementation

5.1 User Interface

In this section, we will explain the usage of all Bluetella menus and the type of visualization screen that was chosen for them. The order in which they are explained was chosen the way they would appear in a typical user session.

During startup of the Bluetella application, a first access to the Bluetooth module is made for initialization purposes. As the Siemens mobile phones require user confirmation for network access, the alert screen in figure 5.1 is displayed automatically by the operating system. It is the type of alert mentioned in 2.2 where a key needs to be pressed for continuing.

After confirming with *yes*, the main menu is displayed. It is a list screen containing the four menu entries:

1 - explore neighborhood Builds connections to all Bluetella clients by making a Bluetooth service discovery. If the network topology changed, this function can be launched again to get connections to possible new neighbors.

2 - search&download After having connections to the neighbors, a searchRequest can be sent to them. Several screens will follow when this entry is chosen.

3 - my shared directory Displays the files from the Bluetella shared directory (see figure 5.2, right). Just the files in this directory are visible to other Bluetella clients and all downloaded files will be put into this directory.

4 - settings So far there is just the *include Gnutella Gateway* option available in the settings menu (see figure 5.2, right).

Figure 5.1: Screen shown when beginning to use the Bluetooth functionality

As the small size of mobile phone displays leads to a large number of individual screens, Bluetella's menus contain numbers for ease of navigation. An example displays them best: Menu 214 would be reached by choosing the second entry in the main menu, the first in the following and ultimately the fourth menu entry. If a screen doesn't offer anything more to choose than a back and an ok button, then the following screen is also referenced by 1 as there are no other options.

Figure 5.3 displays, all the screens encountered when searching and downloading a file in their appearing order. The first picture displays the main menu, where we first need to choose the first entry to discover all neighbors and build connections to them. The mobile phones found are shown in the second picture.[1] Then we can choose the second option in the main menu and specify a part of the filename that we want to find. For this screen the form class was chosen, so that other fields could be added at a later time. The screenshots were taken in an example where the shared files had the names fileXoY where X stood for the last digit of the Bluetooth address and Y for the file number. When just searching for *file* we therefore get all files available in this Bluetella network. The fourth picture

[1] In the current implementation, the mobile phones found are written to the debug output and the names displayed on the mobile phone screen are dummy names.

Figure 5.2: Additional screens

displays the files found (list screen). For getting more information like size and number of hops the file is away, we can select one of the files and get to the fifth screen (also a form screen). When we are happy with the file, we are then at the point where we can start the file download, during which the sender and receiver mobile phone display the alert shown in the sixth picture (They both display the same alert). First, the server displays the alert, when Bluetella wants to read the file and second, the client displays the alert when Bluetella wants to write the arrived packets to the file system.

After the file is downloaded completely, it appears in the *my shared directory* menu.[2] Future versions could use the screen type *Gauge* to display the progress of the download as all the necessary information would be available in the Bluetella protocol.

5.2 Threads

This and the next section are particularly helpful for somebody, who is continuing working on the code. We will use the class names found in the code[3] and give an overview of how these classes work together.

Bluetella contains a lot of threads for making it possible that the user interface reacts quickly and that the whole bunch of connections that can be opened don't need to wait for another connection that might be blocked at the moment.

Every connection to a neighbor node is represented as an instance of the BiDirConnection[4] class and contains an outputstream for writing to the neighbor and it contains as well a thread called InputRdr which hosts the

[2]The SL55 emulator used had a bug in the method that displays the content of a directory. Used as in a documentation example, it did just show the contents of the root directory. A dummy file list is therefore displayed in the implementation.

[3]The complete source code is available on the CD added to this thesis

[4]The BiDirConnection class is one of the classes of Bluetella

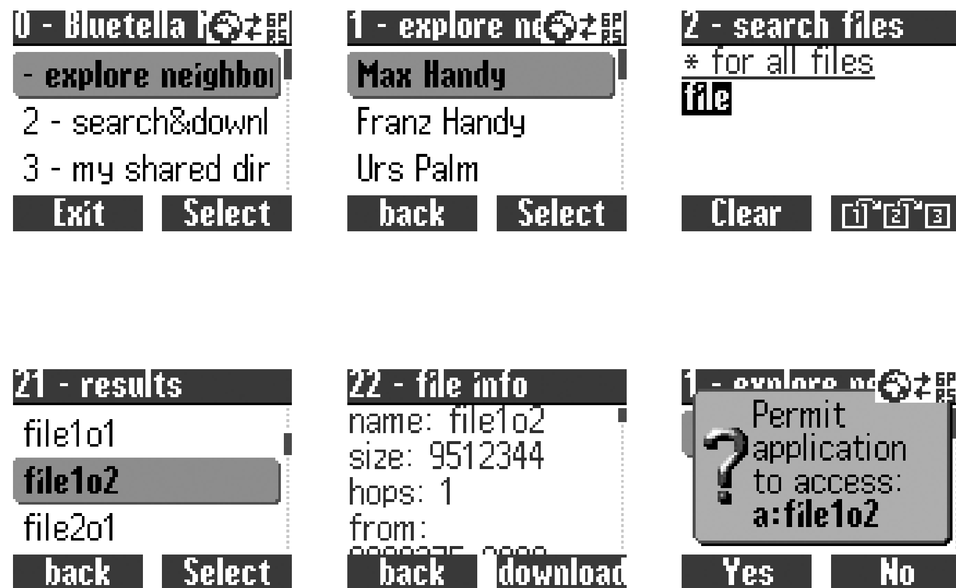


Figure 5.3: Downloading a file

inputstream belonging to this connection. The inputstream was put into a separate thread, as reading on the stream blocks the reading thread until data arrives and therefore making it impossible to send from that thread. We now have two threads for each connection to a neighbor. A BiDirConnection which contains an InputRdr. All these BiDirConnection threads to the various neighbors are organized by the ConnectionThreadController, which is a Thread himself. The latter has methods to send a bytearray to just one Bluetooth address, or to the whole group of its connections.

The question remains, where these connections originally come from. They all originate from the NeighborhoodDiscoverer thread which starts the DiscoverySearchMan, who was called like that due to a bug in Rococo Simulator. Whenever a DeviceDiscovery is started, the results are given back to a class that implements a Listener interface called DiscoveryListener. Our DeviceSearchMan is the class who implements this interface and therefore receives the results of the searches. Due to the bug, the method in our DeviceSearchMan just gets back the results of one single search (the found devices), but the second and and following searches don't deliver any results. The Rococo Knowledge Base [roc] knew from this bug and stated further, that every instance of the Listener can receive just the results from one serach. Therefore, we have to let the DiscoverySearchMan instance, who is implementing the listener interface, die and recreate him for the next search. The NeighborhoodDiscoverer is doing this.

Having found a device, we need to start the ServiceSearchMan, in order to get the connection url of a Bluetella application, if there is one on that device. The ServiceSearchMan has the same problem as the DeviceSearchMan, he is implementing the DiscoveryListener interface and just receives the results for one search. He therefore needs to be reinstantiated for every search on each device, which is done by the DeviceSearchMan.

The DeviceSearchMan and ServiceSerachMan don't need to be threads as they don't block while waiting for the results thanks to being just listeners for the implemented DiscoveryListener interface.

In the end, when all these ServiceSearchMan instances have found Bluetella clients, they inform the ConnectionThreadController about the new neighbor who is then being added to the ConnectionThreadController's list of BiDirConnections.

5.3 Handling the Protocol Headers

Optimizing code for execution speed or human readability often are contradicting tasks. In Bluetella this question turned up when the protocol headers had to be read in, analyzed, changed and written out again. This happens for instance when a BSR packet had not yet reached its destination and had to be forwarded to the next hop.

The fast way would be to read in the byte array, keep it as byte array and build several functions working on the array for changing the fields. The slower, but more general way of doing it is parsing the string and saving all its information in an object. After all changes have been made to the objects variables, build the byte array completely new. If just a few changes are made to the bytearray, the latter is definately wasting cpu power and memory. If a lot of fields get longer inside the array, enlarging the array gets very complex without unnecessary copying of the array - and even might be slower than the object variant.

Bluetella parses the header always when it arrives, because human readability for further programmers was more a concern than speed and the header is never getting longer than a few kilo bytes anyway.

The Bluetella Source Routing header and Bluetella File Sharing header are both represented by the separate classes BSR_header and BSF_header, respectively. Each class has two constructors, one getting an input stream

from which they read and fill their protocol field variables and another one having the message type as argument from which they fill the field variables for a new empty protocol header. Both classes also offer a toByteArray() method which writes all field variables in the correct order and with the necessary helping fields into a new byte array. For instance, the search string in the object is written out as a length field plus the actual characters of the string. For programmers convenience a Bluetooth address, that is compressed into a 6 byte subarray in the array is saved as a string in the header object. Other fields offering a similar way for simplification such as numbers of type long are stored analog.

Separating both protocols as well in the implementation after they have been separated in the protocol specification allows for easy exchange of a singular protocol in Bluetella's source code at a later time.

5.4 Distance Abstraction Layer

As the Rococo Simulator doesn't simulate the range of Bluetooth nodes, service discovery finds all running Bluetella clients, which is rather unrealistic. Most likely, some of them would be too far away of each other. For being able to test our protocol, we can't live with this simplification as every node would have a direct connection to all the others, never having to route its packets over hops in between. The class *DistanceAbstractionLayer* is simulating a distance between the mobile phones. It offers a method that returns true if a mobile phone couple is within reach of each other by just allowing predefined mobile phones (Bluetooth addresses) to build connections between them.

5.5 Data Storage Revisited

For accessing the filesystem of the mobile phone, the Siemens specific class *com.siemens.mp.io.File* is used. As mentioned in section 2.5, this is the only way for accessing the filesystem. This use of the Siemens API is the only ocurrance in the code of a non-standard Java API and is limited to Bluetella's *FileSystem* class.

Chapter 6

Development Environment

This chapter is just giving a brief overview of the third-party software used. For a detailed instruction on how to install and integrate Sun One Studio, the mobile phone emulators and Rococo Impronto Simulator please refer to appendix A.

6.1 Sun ONE Studio

Basically, there exist three major development environments for J2ME development: Borland JBuilder, Metroworks CodeWarrior and Sun ONE Studio (see figure 6.1). As Sun ONE Studio was the only version available for free, it was practically decided to be chosen for this work. Being the only one to be sure that doesn't include third party libraries or code contributed to the decision.

6.2 Mobile Phone Emulators

Access to the mobile phone's file system was the main criteria why emulators of Siemens mobile phones were chosen. For having the new type of file access mentioned in section 2.5 we needed one of the newer Siemens mobile phones. The SL55, one of the new color phones, satisfied all needs, except that it lacks an mp3 player. The SX1 will be the first Siemens mobile phone with the new file access method based on user alerts and an mp3 player, but neither the emulator nor the phone were available during the thesis (the SL45i uses the old file access).

6.3 Rococo Impronto Simulator

The software used for simulating the JSR-82 API and Bluetooth environment is from Rococo [weba]. This company offers a free Linux version which

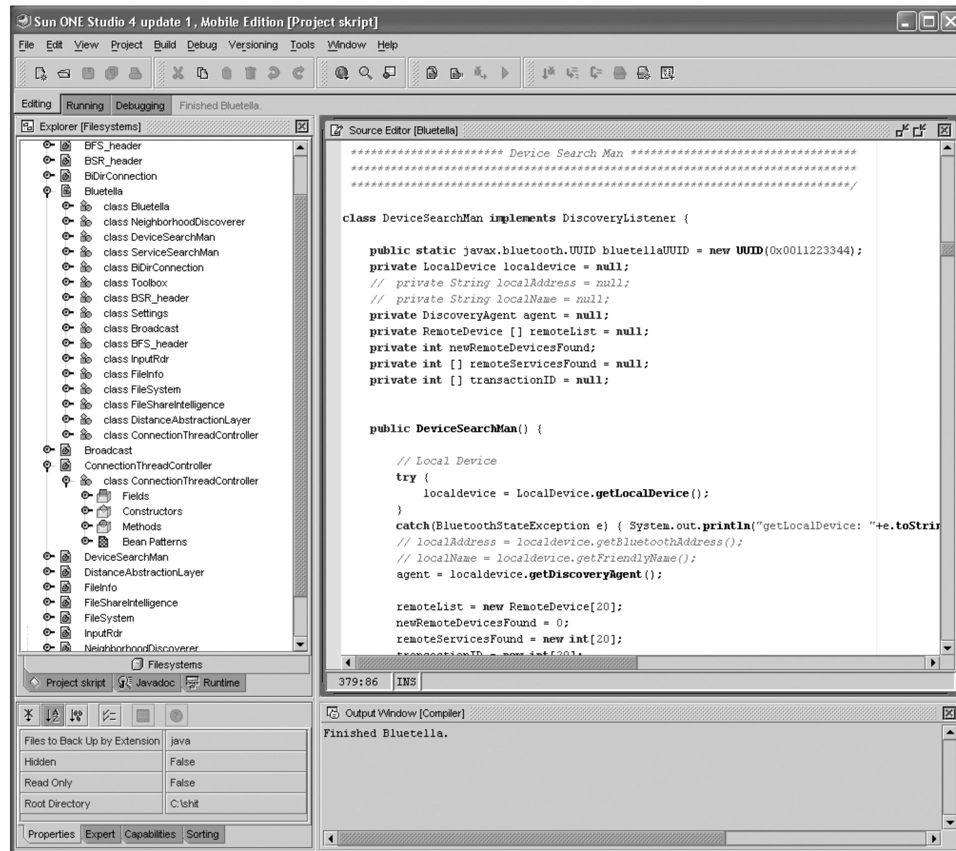


Figure 6.1: Sun ONE Studio

needs a hardware Bluetooth stack and a 1000$ Windows version which simulates a Bluetooth stack (see figure 6.2), therefore getting along without any hardware at all as the mobile phone is emulated as described in section 6.2. At the time of writing, Rococo Impronto Simulator was the only available software that simulated the Bluetooth stack. For development of Bluetella, the evaluation version of the Windows version of Rococo Impronto simulator was used. Its features are equal to the full version, just its use is limited to 30 days by a license file that can be generated on Rococo's webpage [weba].

There is a further bug in the simulator besides the ones mentioned in the knowledge base of Rococo [roc]. While demonstrating Bluetella from the notebook without network connection, an IOException is thrown inside one of the Rococo Simulator's classes, which is not caught. It appears that the Rococo Simulator just works if the computer it is running on has got an ip address, which is not the case if the computer gets the address by DHCP and the network cable is not connected.

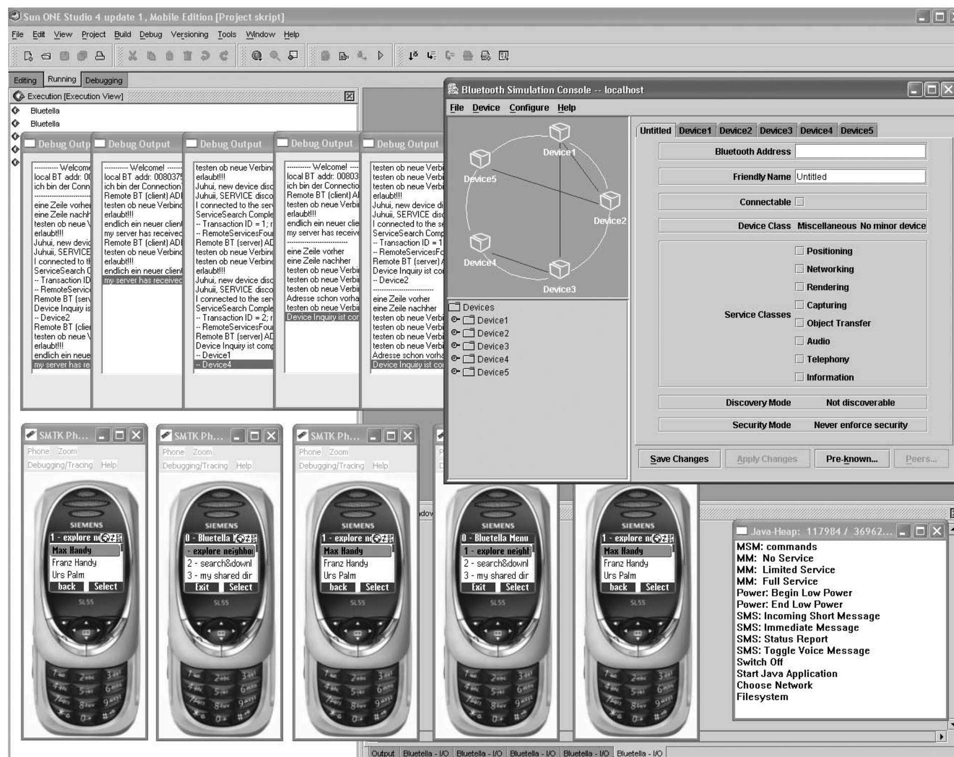


Figure 6.2: Rococo Impronto Simulator with Siemens SL55 Emulators

Chapter 7

Conclusion and Future Work

The implementation of Bluetella and the demonstration of a complete and error free download of a file over several hops showed that it is possible to build a file share client for mobile phones using the Java Microedition and the newly available API for Bluetooth Wireless Technology.

Having an implementation of Bluetella that is optimized for speed and that deals with all possible breakdowns of connections would need further time for development. In a future work, one should be able to recognize when neighbors become unavailable and close the transfer gracefully. In the current implementation of Bluetella, the reading and writing threads to a neighbor who disappeared block till the other side writes to the buffer or reads from it, respectively. For terminating these threads instead of having them suspended till Bluetella is exited could be done by timers that are reset whenever a byte was sent or received.

Another likely situation that should be dealt with is when one of the intermediate hops disappears. The client and server need to recognize this in order to interrupt the transfer instead of leaving suspended threads behind.

An even better way of dealing with nodes that leave the Bluetella network and new ones that appear would be exchanging the Bluetella Source Routing protocol layer with a more advanced ad-hoc routing protocol by either improving BSR or replacing it completely.

Shall the user be blessed with more search results, a gateway to one of the desktop file sharing clients like Gnutella or KaZaA could be implemented. Most likely, the gateway would be running on a personal computer who has both an Internet and Bluetooth connection and could automatically search in one network, when it sees a search in the other. Actually, the Bluetella File Sharing protocol already has a field in the searchRequest packet, where

can be specified whether such gateways should be included in the search or not.

For making Bluetella compatible to all manufacturers of mobile phones, future versions should exchange the *FileSystem* class as soon as a standardized filesystem access gets available in the J2ME APIs.

As mentioned in chapter 2.3, several user interface improvements are recommended. During download a gauge screen should inform the user about the current state of the download. Screen 1 is a dummy screen and should be filled with the found Bluetooth addresses instead of the names displayed. Screen 3 displaying the shared directory content is as well just displaying dummy data, but can't be implemented till the the directory listing bug was removed from the SL55 emulator.

Appendix A

How to Compile and Execute Bluetella on Your PC

A.1 Software Installation

All installation files being mentioned in this section can be found on the CD enclosed to this thesis in the directory `3rdPartySoftware`. It is recommended you begin further development of Bluetella with the program installation files from the CD instead of downloading the latest versions. Bluetella isn't particularly sensitive to different third-party software versions, but some version combinations of the development environment programs themselves will not work properly as stated below.

Microsoft Windows 2000 or XP Both versions were used during development of Bluetella and worked fine with all third-party software.

J2SE Developer Kit (JDK) 1.3.1 It is crucial to have exactly this version (not a newer or older one) installed, otherwise Rococo Impronto Simulator will fail already during installation with a message saying "could not load jvm.dll". In case you see this error message despite having installed JDK 1.3.1, try uninstalling all other JDK's and Java Runtime Environments (JRE). Remember that Windows might itself have a JRE in its system directories. If you see the error message "Installer User Interface / Mode not supported" then the installer hasn't found the JDK at all. [roc]

Rococo Impronto Simulator As mentioned in 6.3, the evaluation version of Rococo Impronto Simulator is sufficient for running Bluetella. Install `simulator.exe` and follow the on-screen instructions. After completion you need to download a license file from Rococo's webpage [weba] which will enable you to use the simultor for 30 days counting from the day the license was downloaded. Copy the (`License.txt`) file into the subdirectory `config\` of Rococo Impronto Simulator.

Siemens Mobile Phone Emulator Run `siemens_smtkSL55_00_4089.exe` (from [webb] / CD) to install the Siemens Mobility Toolkit for SL55 mobile phones.

SUN Studio One Development Environment Install `ffj_me_win32.exe` and follow the on-screen instructions till the installation completes. To tell SUN Studio where to find the emulator we installed earlier, run SUN Studio and select the *Editing* tab. Select *Runtime* from the appearing subtabs to show the Runtime tree and open it to the branches *Runtime / Device Emulator Registry / Installed Emulators*. Right click on *Installed Emulators* and select *Add Emulator*. Guide the appearing "file open" window to the directory where the emulator is installed and click *add* (If you haven't changed the path names during installation it will be `c:\Siemens\SMTK\SL55`).

A.2 Compiling and Running Bluetella

Open the *Project Manager* in the *Project* menu bar of SUN Studio. Choose *new project* and give it a name at your discretion, then choose *CLDC/MIDP* and click *Finish*.
For using Rococo Impronto Simulator to simulate the JSR-82 API and Bluetooth hardware, all we need to do is adding it to the classpath by adding a jar library to our project: Click *File*, then *Mount Filesystem* and choose *Archive (JAR,ZIP)*. Guide the appearing explorer window to `lib\isim_midp.jar` in the Rococo Impronto Simulator directory and click *Finish*.
Prepare a source code directory on your harddrive at your discretion and mount it also into your project by clicking: *File*, then *Mount Filesystem* and choose *Local Directory*. Guide the appearing explorer window to your source code directory that you created before and click *Finish*.
The quickest and safest way to get Bluetella running is by creating a helloMidlet and overwriting its source code by the one from Bluetella. It showed to be the only possibility to get the Siemens emulator to accept all Bluetella classes without classpath problems. Create a new midlet by clicking *File* then *New*. Choose the branches *MIDP / HelloMidlet* and continue with *Next*. Select your source code directory, change `<default name>` to `Bluetella` and click *Finish*. You should see now the source code of a hello world application. Feel free to try it out by compiling it with the key *F11* and running it with *F6*. If the Siemens SL55 mobile phone appeared on your screen and you see the words `test string` in it, then everything worked properly and you can exchange the helloWorld source code with the one from the CD found in the `Bluetella` directory. Compile it again with *F11*. Before running it, we need to start the management console of Rococo Impronto Simulator by executing `bin\manager.bat` in the Rococo Impronto Simulator directory and wait till the window *Bluetooth Simulation Console*

appears. Now go back to SUN Studio and run Bluetella by pressing *F6*.

Appendix B

Assignment of the Semester Thesis

B.1 Introduction

Mobile phones of the latest generation feature a Java virtual machine (J2ME). In this project, we will analyze the capabilities of those phones. Based on this analysis, an application is defined and then implemented. In particular, we encourage the student to come up with ideas for interesting applications and discuss them with the supervisors.

B.2 Assignment

B.2.1 Objectives

New mobile phones feature different communication protocols, such as GPRS, IrDA, Bluetooth, and others, which can be programmed using the Java Virtual Machine (JVM) provided by those phones. Since processing and storage resources on mobile phones are scarce, the functionality of the JVM is limited.

In this project, the JVM functionality of mobile phones is evaluated. Based on the evaluation, an application is defined and implemented. The application should exploit as many phone features as possible.

B.2.2 Tasks

- Get familiar with the JVM and related APIs provided by the latest models of mobile phones.
- Compile a list of ideas for applications. Discuss them with your supervisors and select an appropriate application for detailed definition and implementation.

- Evaluate development environments for mobile phones and choose an appropriate one.
- Define the selected application in detail.
- Implement the selected application.
- Define and set up a demonstration scenario of the application.
- Document your work in a detailed and comprehensive way. We suggest you to continually update your documentation. New concepts and investigated variants must be described. Decisions for a particular variant must be justified.

B.3 Deliverables and Organisation

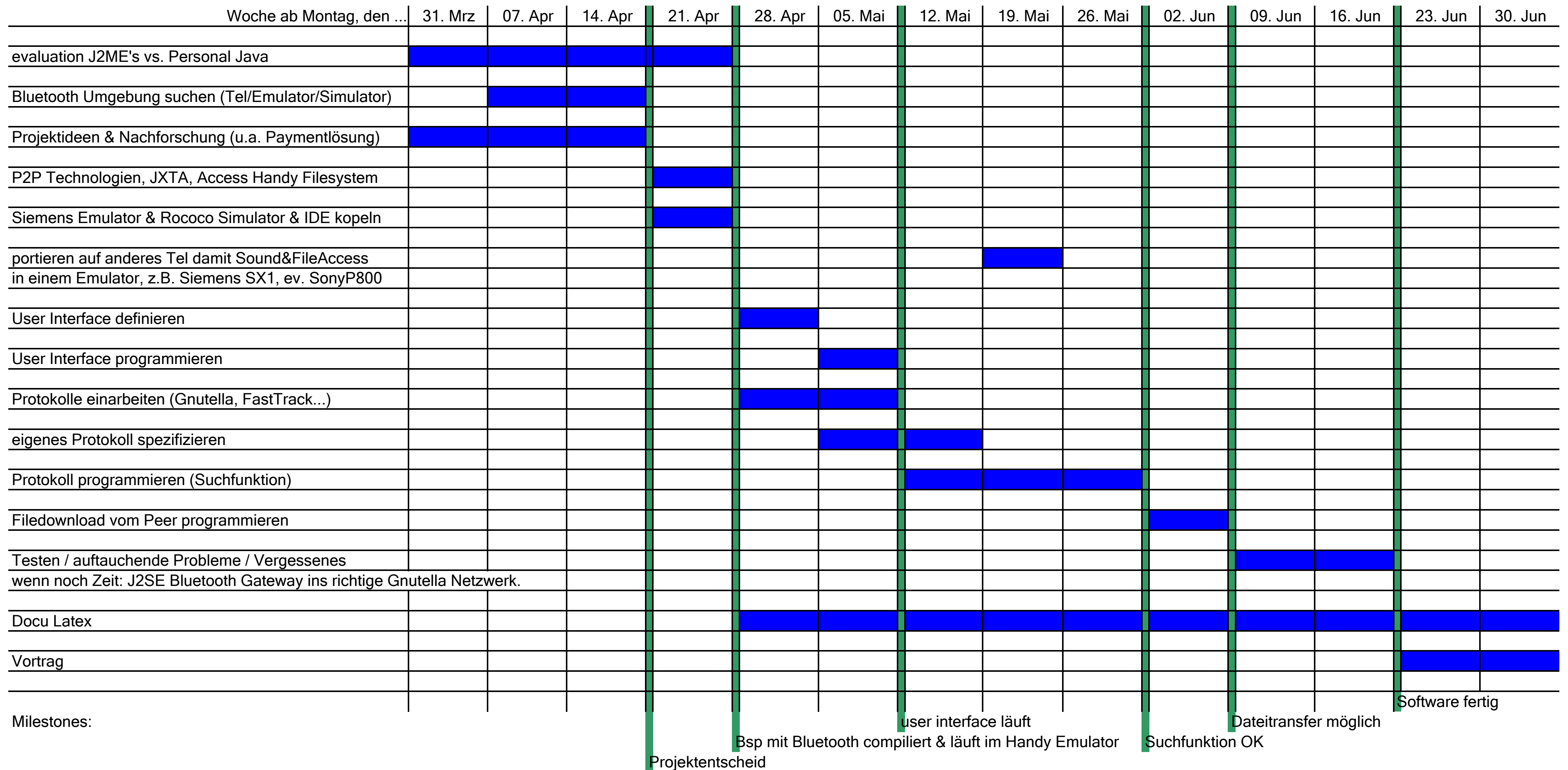
- If possible, students and advisor meet on a weekly basis to discuss progress of work and next steps. If problems/questions arise that can not be solved independently, the students may contact the advisor anytime.
- At the end of the third week, a detailed time schedule of the semester thesis must be given and discussed with the advisor.
- At half time of the semester thesis, a short discussion of 15 minutes with the professor and the advisor will take place. The student has to talk about the major aspects of the ongoing work. At this point, the student should already have a preliminary version of the table of contents of the final report. This preliminary version should be brought along to the short discussion.
- At the end of the semester thesis, a presentation of 15 minutes must be given during the TIK or the communication systems group meeting. It should give an overview as well as the most important details of the work. Furthermore, it should include a small demo of the project.
- The final report may be written in English or German. It must contain a summary written in either English or German, the assignment and the time schedule. Its structure should include an introduction, an analysis of related work, and a complete documentation of all used software tools. Related work must be referenced correctly. See http://www.tik.ee.ethz.ch/flury/tips.html for more tips. Three copies of the final report must be delivered to TIK.
- Documentation and software must be delivered on a CDROM.

Appendix C

Timeline

Zeitplan Semesterarbeit: Java application for new mobile phones

Ganymed Stanek, Stand 10. Mai 2003



Bibliography

[Esc03] Sebastian Eschweiler. *Java 2 Micro Edition - Das Einsteigerseminar*. bhv-verlag, 2003.

[gnu] The gnutella protocol specification v0.4. http://www.clip2.com.

[HA03] Bruce Hopkins and Ranjith Antony. *Bluetooth for Java*. apress, 2003.

[J2M] Datasheet: Java 2 plattform, micro edition. http://java.sun.com/j2me/j2me-ds.pdf.

[jav] Jsr-82 javadoc. http://jsr82.devzone.possio.com/javadoc/.

[KB] Sun Mirosystem Klaus Bergius, Enterprise IT Architecht Wireless. Java macht mobil: Neue spezifikationen und endgeraete. http://www.linecity.de/pdfs/Artikel%5FJavaSpektrum.pdf.

[Knu03] Jonathan Knudsen. *Wireless Java: Developing with J2ME*. apress, 2003.

[mid] What's new in midp 2.0. http://java.sun.com/products/midp/whatsnew.html.

[Rela] Nokia Press Release. Nokia demonstrates electronic mobile payment services with visa and meritanordbanken. http://press.nokia.com/PR/200002/775312

[Relb] Sun Microsystems Press Release. Sun announces java(tm) 2 platform, micro edition. http://java.sun.com/pr/1999/06/pr990615-01.html.

[roc] Rococo knowledge base. http://www.rococosoft.com/devcorner/knowledge.html.

[sie] Mobile information device profile (midp) and siemens extension apis. html - just available in one of the Siemens Mobile Developer Kits found at http://www.siemens-mobile.com.

[UUI] The bluetooth assigned numbers document. http://www.bluetooth.org/assigned-numbers/sdp.htm.

[weba] Rococo website. http://www.rococosoft.com.

[webb] Siemens Mobile Developer website. http://www.siemens-mobile.com.