Seminarausarbeitung
zur Lehrveranstaltung 183.263
“Seminar (mit Bachelorarbeit)”
im SS 2008

Bluetooth - KNX Gateway

bearbeitet von

Dominik Windhab
Matrikelnummer: 0301096 Studienkennzahl: 535

Technische Universität Wien
Fakultät für Informatik
Institut für Rechnergestützte Automation

Lehrveranstaltungsleiter: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner

Contents

1 Abstract

Home and building automation becomes more and more important today. It refers to a kind of the intelligent home where devices and systems are used to improve energy efficiency, comfort, flexibility and security. The main focus of home and building automation is on building networks with different devices. One well established technology to create these kind of networks is KNX. KNX is an open and standardized network communication protocol for intelligent buildings. It is the successor of the the EIB (European Installation Bus), the EHS (European Home Systems) and the BatiBus standards.

The aim of this project was it to create a software which makes the user capable of controlling KNX devices with a mobile device like a mobile phone or a notebook. An important aspect was to use wireless communication technology for this task. One of the most important and well established technology today is Bluetooth. In nearly any portable device a Bluetooth adapter can be found. It works on the ISM (Industrial Scientific and Medical) wave band which makes it possible to use Bluetooth all over the world.

The result of this project is a Java based software which consists of three parts. A configuration tool with a user interface which allows the user to create an XML based description of a KNX network. The other two parts are a server and a client for controlling the KNX devices. The server acts as a gateway between the mobile device and the KNX network. The client runs on the mobile device. It connects to the server, displays KNX devices and provides the user interface.

This document gives an introduction to the Bluetooth and the KNX technology. It also focuses on how to develop Java applications for Bluetooth. The last part is an explanation of the particular parts of the project and how they are designed.

2 Task and Objectives

The initial plan of this project was it to create a system where a user can control his KNX devices, for example the lights, attached to a network with his handheld device (cf. Figure 2.1).

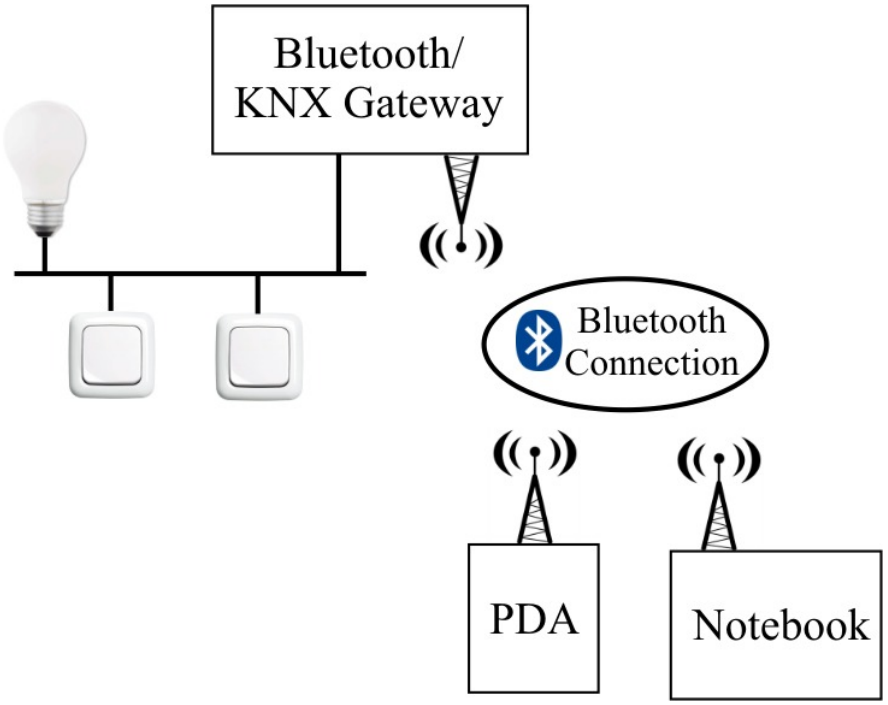

Figure 2.1: Initial plan

2.1 Project Setup

There was already a test environment available for this project. It consists a working KNX network with an electric bulb attached to it. It can be controlled via a switch which was also part of the network. A PC was connected to the KNX network. Windows and Linux was running on this PC. As handheld part was an iPaq hx2790 pocket PC was used running Windows Mobile 5.0. A USB Bluetooth stick was also part of the setup.

2.2 System Overview

There are three major parts in this project: a configuration tool, a server- and a client part.
With the configuration tool it is possible to create definitions of KNX datapoints that can be controlled with the Bluetooth devices. The server part is the software which runs on the PC that acts as a gateway to the KNX network. The server part of the project is also be applicable for embedded PCs (e.g. embedded Linux). The third part in this project is the client software. This software is runnable on a PDA as well as on standard PCs like a notebook. It is possible for the

user to connect to the mentioned server. In addition a user interface is available to handle the connection to the server and to control the devices in the KNX network.

3 Introduction to Bluetooth

Bluetooth is an open standard for wireless communication over short distances up to a maximum of 100 meters. Today there are wireless standards which are optimized for data transmission and others which are optimized for multimedia data. Bluetooth focuses on both of them. A mobile phone is an example of how Bluetooth can be used in different ways (e.g. headset). The transmission of voice from the mobile phone to the headset is treated differently than the transmission of data, like images for example, from one mobile phone to the other.

During the development of the Bluetooth technology one focus was to create a wireless standard which can be used all over the world without interfering occupied wave bands. Bluetooth is also optimized for the use in portable devices characterized by a low power consumption, a small but robust design and the possibility for cheap production. Most of today's cell phones and notebooks have a Bluetooth controller on board and the market is still growing [MT02]. The Bluetooth specification is available in version 2.1 [Blub].

3.1 History

The first efforts to create a new wireless standard started in 1994. The company "Ericsson Mobile Communications" started a study to find a new cheap and energy-saving way of wireless communication. In 1998 the companies IBM, Intel, Ericsson, Nokia and Toshiba founded the "Bluetooth Special Interest Group" or " Bluetooth SIG" [MT02].

Today the Bluetooth SIG is a nonprofit federation of commercial and industrial enterprises. The SIG itself does not produce or sell any Bluetooth devices. The SIG has more than 10.000 member companies that are leaders in telecommunication, computing, automotive, music, apparel, industrial automation, and network industries. The headquarter of the SIG is in Bellevue but there are also offices in Hong Kong or Malmo [SIG].

Figure 3.1: Bluetooth logo

There are 3 types of memberships within the Bluetooth SIG: Promoter member companies, Associate and Adopter member companies. The Promoter member companies are Ericsson, Intel, Lenovo, Microsoft, Motorola, Nokia, and Toshiba which form the core of the Bluetooth SIG. The Associate members pay annually a certain amount and are allowed to contribute to future releases of the Bluetooth specification. The membership for Adopter companies is free. They are allowed to use the Bluetooth technology [MT02].

Companies which produce devices with the Bluetooth brand have to fulfill major points of the Bluetooth specification and have to run through a Bluetooth qualification program.

> "The Bluetooth Qualification Program Reference Document (PRD) is the primary reference document for the Bluetooth Qualification Program and defines its requirements, functions, and policies. The PRD is available on the Bluetooth Web site. Passing the Bluetooth Qualification Process demonstrates a certain measure of compliance and interoperability, but because products are not tested for every aspect of this Bluetooth Specification, qualification does not guarantee compliance. Passing the Bluetooth Qualification Process only satisfies one condition of the license grant. The member has the ultimate responsibility to ensure that the qualified product complies with this Bluetooth Specification and interoperates with other products."[Blub]

3.2 Network Topology

An important part to make Bluetooth wireless communication possible is the way how networks are formed. Because there is no need for plugging in cables, the wireless networks can be formed autonomously. This way of creating networks is called ad-hoc networking. It is possible for every Bluetooth device which is in reach to join an already established ad-hoc network (cf. Figure 3.2). Any time a Bluetooth wireless link is formed, it is within the context of a piconet. In a Piconet there is one master and up to seven active slaves. Master and slave in this context are only the names of the roles. In general it does not matter which devices takes the role of the master. The specification defines that the device which initializes the communication is the master. It is responsible for the frequency hopping sequence (cf. Section 3.3.2) and the synchronization of the different participants of the network. Hence there can only be one master in a piconet.

When piconets overlap it is possible for a device to participate in more than one piconet (cf. Figure 3.2). This case is referred to as scatternet. Each piconet uses its own hopping sequence (cf. Section 3.3.2) defined by the master. Therefore it is not possible for one device to be master in two or more piconets. In Figure 3.2 the slave 3A/4B participates in two piconets [MT02].

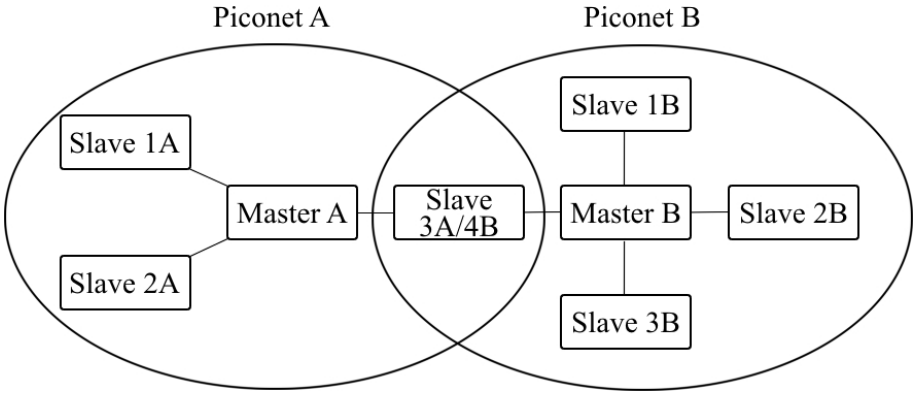

Figure 3.2: Scatternet

3.3 Bluetooth Protocol Stack

The Bluetooth protocol stack is based on a layered architecture (cf. Figure 3.3). For interoperability of applications it is necessary to agree on a common base of this protocol architecture. It

is up to the developer to choose a suitable subset of the protocol stack for his application. In the next chapters we will discuss the most important layers of the stack and what they are used for [Mul01].

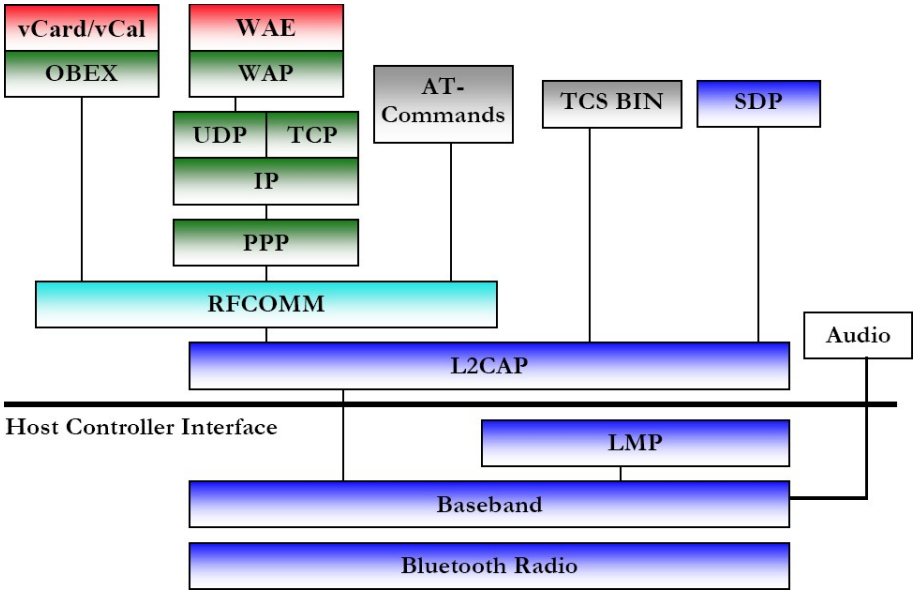

Figure 3.3: The Bluetooth protocol stack [Met99]

3.3.1 Bluetooth Radio

The lowest layer in the protocol stack is the Bluetooth radio. It represents the interface to the "air" and is responsible for modulation and frequency hopping.

Bluetooth communication makes use of the 2,4 GHz ISM-Band (**I**ndustrial, **S**cientific and **M**edical). This band is reserved for devices of industrial, scientific or medical applications and can be used without a license. To use this waveband and match the standard, some conditions have to be fulfilled. According to ETSI EN 300 328 one condition is to use some kind of spread spectrum modulation of the signal. The modulation has to make use of at least 15 different non-overlapping channels which are also referred as to hopping positions. The dwell time per channel shall not exceed 0,4 s [Com01]. For Europe the ETSI (European Telecommunications Standard Institute) released the standard for data transmission equipment operating in the 2,4 GHz ISM band [MT02].

3.3.2 Spread-Spectrum-Frequency-Hopping

Code-Division-Multiple-Access (CDMA) is an important channel access method utilized by various radio communication technologies. In contrast to Frequency-Division-Multiple-Access (FDMA) and Time-Division-Multiple-Access (TDMA), it is based on a Spread-Spectrum technique. That means the division of the channels is not based on time or frequency. Every participant is sending at the same time and in the same wave band. The basic principle of signal spreading is to transform a narrowband signal to a broadband signal. But how is it possible for communication partners to find the correct signal when everyone sends at the same time? The trick is that each sender and receiver pair knows the spreading code of the partner and hence can filter the correct signal out of other signals which will be treated like noise.

Frequency-Hopping is also a Spread-Spectrum technique which is used for Bluetooth communication. To use this method, the given bandwidth is divided into equidistant wave bands. Sender and receiver use the same wave band for transmission. During the communication sender and receiver change the wave band used for data transmission at the same time. The order of the used wave bands is pseudo-random. This pseudo-random code has to be known by the sender and the receiver to stay synchronized during the transmission. In Figure 3.4 the spectrum of a Frequency-Hopping signal is outlined.

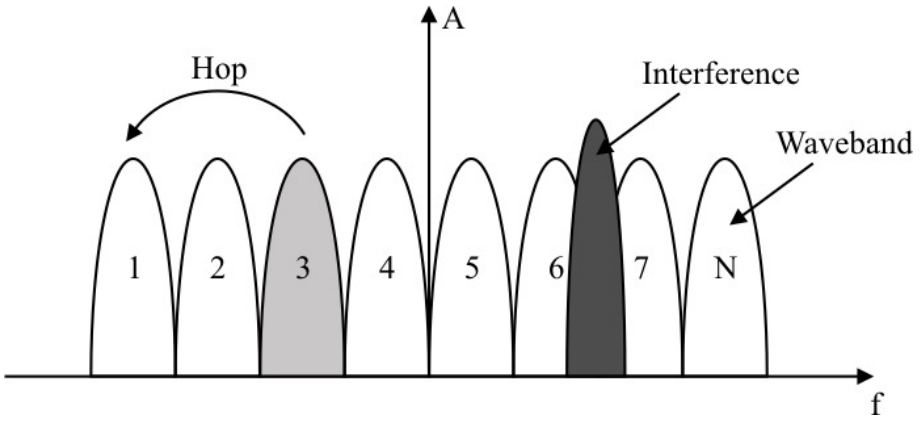

Figure 3.4: Spectrum of a Frequency-Hopping-Spread-Spectrum signal with interference

But what are the advantages of this method? First of all the transmission is resistant against narrowband interference (cf. Figure 3.4). Because many different wave bands are used, the transmission is only interfered for a short time by the unwanted signal. Another important point is security. Only with the knowledge of the hopping sequence, the receiver is able to filter the correct signal.

According to the ETSI EN 300 328-1 V1.3.1 standard, if a Frequency-Hopping technique is used for data transmission, at least 15 different frequencies have to be used and the hop-time must not be larger than 400 ms. Bluetooth uses 79 different wave bands with a bandwidth of 1 MHz. Because other systems like WLAN (IEEE 802.11) that also work in the 2,4 GHz ISM band it might be possible that these systems interfere each other. Thats why frequency hopping was invented. There is only a small time window where systems interfere each other because they both follow another hopping sequence. To avoid interference with other radio systems a guard band was invented. That means a 2 MHz guard band at the lower edge and a 3,5 MHz guard band for the upper edge of the frequency spectrum in order to comply with out-of-band regulations in each country [Blub]. In fact the whole ISM band (2,4 - 2,4835 GHz) is used for Bluetooth communication. There are 1600 frequency hops per second, that means a hop to another frequency every 625µs. The timing and the order of the frequency changes is managed by a master device which initiates the communication [MT02].

3.3.3 Baseband

One of the most important parts of the Bluetooth stack is the baseband layer. It controls the hardware and provides the received data to upper layers in the stack. The most important tasks of the baseband layer are:

- Controlling the physical connections
- Packing and unpacking of data packets

- Error correction
- Addressing

[MT02]

3.3.4 Packeting

Data is transmitted over the air in packets. The structure of a packet is shown in Figure 3.5

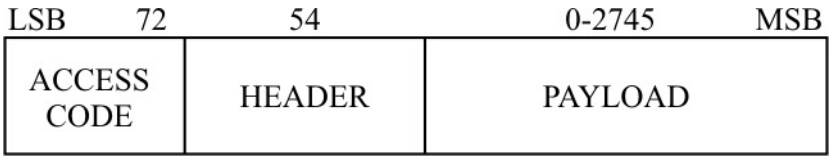

Figure 3.5: Basic packet format

Every packet starts with an access code which is used for synchronization and identification. A Bluetooth device checks the access code of every packet and discards the packet if the access code is not recognized. Each piconet has its own access code. There are three types of access codes:

Channel Access Code(CAC): identifies the corresponding piconet

Device Access Code (DAC): for special signaling purposes

Inquiry Access Code(IAC): needed for the recognition of other Bluetooth devices around

The header contains link control (LC) information and consists of 18 bits and is encoded with a forward error correction code which makes 54 bits total.
The payload depends on the type of the connection. There are two types of connections and therefore two basic types of packets: Synchronous Connection Oriented (SCO)- and Asynchronous Connection-Less (ACL) connections. The SCO link is only used for time critical applications like voice or video transmission. Lost packets will not be retransmitted. The counterpart of SCO links, are ACL links where lost packets are retransmitted. There are lots of different types how the payload field of a packets is structured depending on if the connection is SCO or ACL based [MT02].

3.3.5 Link Manager Protocol - LMP

The task of the link manager is it to set up logical communication links between Bluetooth devices. This is done via the Link Manager Protocol (LMP). With this protocol the link managers of the devices communicate with each other.

> "To perform its service provider role, the Link Manager draws upon the functions provided by the underlying Link Controller (LC), a supervisory function that handles all the Bluetooth baseband functions and supports the Link Manager. It sends and receives data, requests the identification of the sending device, authenticates the link, sets up the type of link (SCO or ACL) and determines what type of frame to use on a packet-by-packet basis." [Mul01]

The corresponding LMP messages are filtered out and hence will not reach higher layered protocols. The LMP does not transport data of the application layers. It only communicates

with the Link Mangers of other devices or it sends control data to lower layer like the baseband or the radio layer. Although there is no direct acknowledgment message of the communication partners required, there are two message types specified which are used to give the sender of the message information if the receiver accepted or declined the message. 55 types of LMP messages are specified, the most important are:

Clock Offset Request: The clock offset is the difference between the slave's clock and the master's clock. When a slave receives this message the difference between the master's and the slave's clock is computed and sent back.

LMP-Version Request: This request is necessary because there are differences between the different versions of the LMP. The response contains a version number, a company ID (eg. 0 for Ericsson Mobile Communication or 1 for Nokia etc.) and a sub-version number.

Features Request: Because there are several optional features specified in the Bluetooth specification there has to be a way to determine which features are supported on the device.

Name Request: Every Bluetooth device has a user friendly name in addition to its unique address. To get this name, a name request is carried out.

Connection Establishment: After the sender has received all needed data it is possible to start the connection. This is done with a special PDU which can be accepted or neglected by the receiver. Then, other link parameters which are needed for a correct transmission of data can be negotiated.

3.3.6 HCI (Host Controller Interface)

The HCI is the interface between the host-software and the hardware which provides the functionality of the baseband and the radio layer. Each Bluetooth module has this interface and can be accessed by special HCI-commands. There are different commands for different purposes. The communication from the host to the controller for example is managed with HCI-Command-Packets whereas the signaling in the other direction is managed with HCI-Event-Packets. To transmit data special HCI-Data-Packets are used (cf. Figure 3.6).

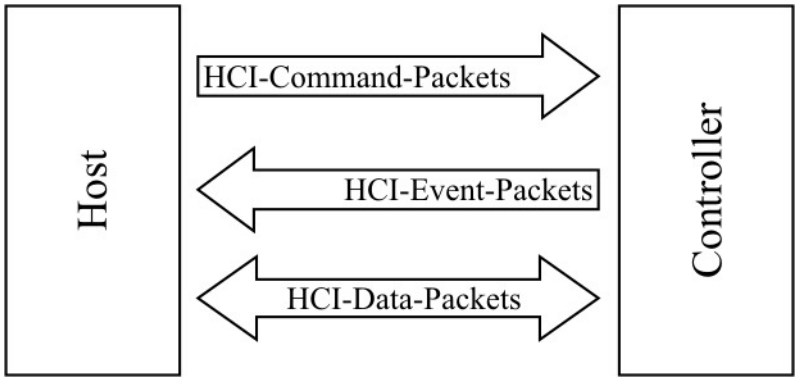

Figure 3.6: HCI communication [MT02]

At the moment there are three possibilities for the physical connection between the host and the Bluetooth hardware: USB (Universal Serial Bus), PCMCIA(PC-Card) or a UART connection.

HCI-Command-Packet: This type of packet is used to send commands to the Bluetooth controller. After the completion of the command the controller answers with a command complete event to inform the host.

HCI-Event-Packet: The HCI-Event-Packets are used for signaling events to host.

HCI-Data-Packet: For transmitting data between host and controller HCI-Data-Packets are used. The data packets for ACL and SCO connections are slightly different but the most important part of them is the *connection handle* field. It represents an id which identifies a virtual connection channel. This is necessary because more than one channel can be open at a time.

[MT02]

3.3.7 L2CAP (Logical Link Control and Adaptation Protocol)

L2CAP is a link management protocol which runs on the host. It receives data from the upper layers and forwards it to lower layers in the stack. It is a packet based protocol which is designed to transmit data and no audio. As shown in Figure 3.3 there is a separate layer for audio transmission in the Bluetooth stack. That is necessary because the transmission of audio is a time critical task. There is for example no need for the retransmission of a packet if one is lost.

> "In applications like voice over IP for example, data which run over the L2CAP protocol contain audio data but will be treated like any other data and not as time critical data."[MT02]

The size of the packets can be negotiated from 672 to 65,535 bytes after a connection was established. L2CAP can be compared to the common Internet protocol UDP (User Datagram Protocol). UDP is also a packet based approach with some small differences. One of them is that L2CAP forces order delivery of the packets which is not the case for UDP where packets can be received orderless. Another major difference is that UDP makes use of a best-effort strategy. L2CAP can be configured to different levels of reliability. The default level is to retransmit until success or total connection failure [HR07].

3.3.8 RFCOMM (Radio Frequency Communication)

For the developers of the Bluetooth technology it was important to re-use existing protocols. The major advantage of that re-use is that these already existing protocols are well established and hence can be used for Bluetooth purposes directly or with only slight changes.

An example of such a re-use is RFCOMM. It is one of the most important and widely-used Bluetooth protocols. It emulates the serial cable line settings and status of an RS-232 serial port and is used for providing serial data transfer. RFCOMM connects to the lower layers of the Bluetooth protocol stack through the L2CAP layer. Because serial connection is often used for PDAs or mobile phones it is also called "cable replacement protocol". Since it emulates an RS-232 interface the status and controlling signals which are provided by RS-232 are also available for the RFCOMM protocol. It supports up to 60 parallel connections [MT02].

3.3.9 SDP (Service Discovery Protocol)

> "The service discovery protocol (SDP) provides a means for applications to discover which services are available and to determine the characteristics of those available

services.

Service Discovery in the Bluetooth environment, where the set of services that are available changes dynamically based on the RF proximity of devices in motion, is qualitatively different from service discovery in traditional network based environments.

The service discovery mechanism provides the means for client applications to discover the existence of services provided by server applications as well as the attributes of those services. The attributes of a service include the type or class of service offered and the mechanism or protocol information needed to utilize the service." [Blub]

To gain information about the services, a request-response model is used. The information is saved in *service records*. A service records consists of a list of service-attributes which are id/value pairs. Each attribute describes the service being offered (cf. Figure 3.7). One of the most important attribute is the Service Class ID. All services which are available for Bluetooth devices can be divided into different classes. According to the ISO/IEC 11578:1996 standard these classes are defined in form of an 128 bit UUID (Universal Unique Identifier).

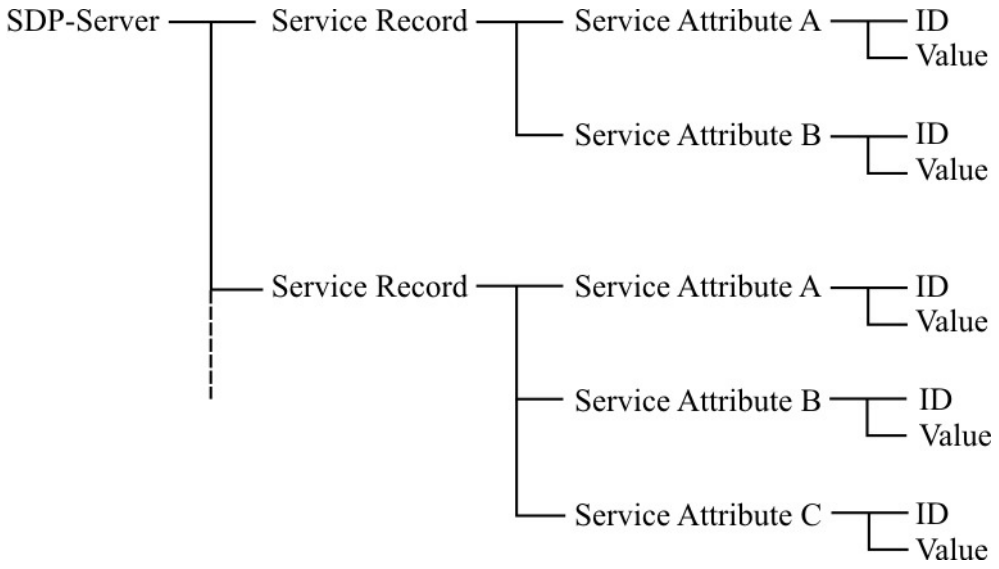

Figure 3.7: Simplified structure of a SDP-Database [MT02]

Some common attributes are:

Service ID: A single UUID identifying the specific service.

Service Name: A text string containing the name of the service.

Service Description: A text string describing the service provided.

[HR07]

The Service Discovery Protocol is different from the other protocols which are based on L2CAP. It is not designed as a basis for higher protocols, it provides functionality to discover services which are running on the corresponding Bluetooth host [MT02].

3.3.10 TCS BIN (Telephony Control Specification Binary)

One important part of the Bluetooth technology is the ability to support audio or voice connections. This is used for example in a Bluetooth headset for a mobile phone. One problem is, that an audio connection does not have the same signaling capabilities like a phone connection (e.g. ringing-functions).

> "TCS BIN is a bit-oriented protocol that defines the call control signaling for setting up speech and data calls between Bluetooth devices. It also defines mobility management procedures for handling groups of Bluetooth TCS devices. TCS BIN is based on Recommendation Q.931 issued by the International Telecommunications (ITU-T), an agency of the United Nations, which coordinates standards for global telecom networks and services. Q.931 is the ITU-T specification for basic call control unter ISDN (Integrated Services Digital Network)." [Mul01]

3.4 Basic Communication Concepts

Before Bluetooth devices can communicate with each other, a connection has to be established. The developer has to distinguish between incoming and outgoing connections. Devices initiating an outgoing connection need to choose a target device, a transmission protocol and a service before establishing a connection and transferring data. Devices accepting an incoming connection need to choose a transmission protocol and a service, and then listen before accepting a connection and transferring data (Figure 3.8).

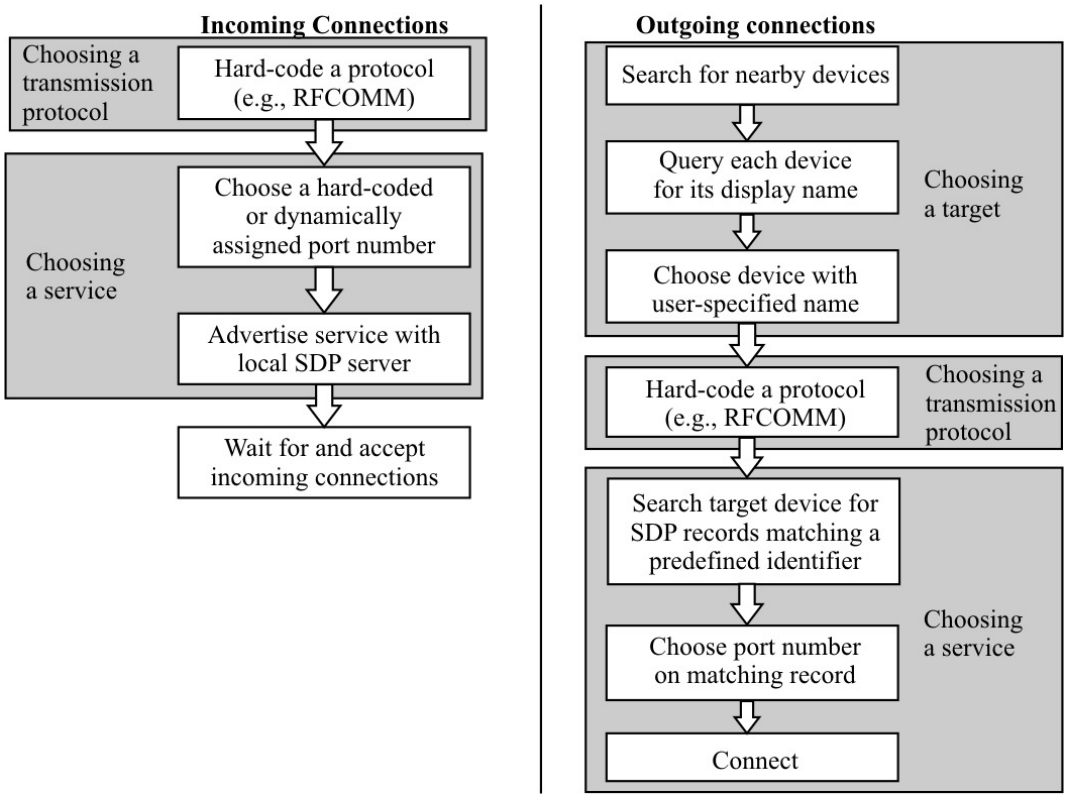

Figure 3.8: Incoming and outgoing connections

3.4.1 Choosing a Target

The task of getting a connection to a nearby Bluetooth device consists of two steps: The inquiry scan which determines all available devices around and the page scan which makes sure that a connection can be established.
The first step to initiate communication via a Bluetooth wireless link is to find a nearby device which is also capable of the Bluetooth technology by starting an inquiry scan. Every Bluetooth module has a globally unique 48-bit address imprinted. These addresses are (like the MAC addresses for Ethernet) managed by the IEEE Registration Authority.
To start a communication it is necessary to retrieve these addresses of other Bluetooth devices around because these addresses are used in all layers of the stack. In addition to this address there is also a configurable human-readable name for each device. This is referred to as the display- or user-friendly name. The display name does not have to be unique which can cause confusion if more devices have the same name.

To get knowledge of the nearby devices a "discovery" message must be sent *(inquiry procedure)*. The reply of this message consists of the address and the general type of the answering device (e.g. mobile phone, headset, etc.). To get more detailed information each device has to be contacted separately. Since Bluetooth 2.1 it is possible to get the most common information, like the human readable name, directly with the inquiry response.

> "Bluetooth devices use the inquiry procedure to discover nearby devices, or to be discovered by devices in their locality. The inquiry procedure is asymmetrical. A Bluetooth device that tries to find other nearby devices is known as an inquiring device and actively sends inquiry requests. Bluetooth devices that are available to be found are known as discoverable devices and listen for these inquiry requests and send responses." [Blub]

The second step to establish a connection is the page scan. Every device which is able to, or wanted to accept incoming connections listens and answers page scan message. Unlike the inquiry message, the page message contains the address of a specific receiver. That means that the procedure is targeted, so that the page procedure is only responded to by one specified Bluetooth device [Blub].

To get the display name each device has to be contacted separately. For reasons of privacy and power consumptions there are two options on each Bluetooth device which control the discoverability and the connectability. The Inquiry Scan options controls the former and the Page Scan option the latter (table 3.1).

3.4.2 Choosing a Transmission Protocol

An important part for the application developer is the selection of the right protocol for the application. In 3.3.8 we discussed the RFCOMM protocol for data based transmissions. In addition to RFCOMM other transmission protocols exist that may be chosen by the application (e.g., audio ...) [HR07].

3.4.3 Service Discovery

The functionality of a user application that is implemented by the application developer is called a service. A service includes the business logic of the application. The device hosting the application might act as a server and waits for incoming connections, or it acts as a client trying to

Table 3.1: Inquiry and Page Scan options [HR07]

Inquiry Scan	Page Scan	Interpretation
On	On	The local device is detectable by other Bluetooth devices, and will accept incoming connection requests.
Off	On	Although not detectable by other Bluetooth devices, the local device still responds to connection requests by devices that already have its Bluetooth address.
On	Off	The local device is detectable by other Bluetooth devices, but it will not accept any incoming connections. This is mostly useless.
Off	Off	The local device is not detectable by other Bluetooth devices, and will not accept any incoming connections. This could be useful if the local device will only establish outgoing connections.

connect to another service. Whatever the plan of a developer is, he will create a service as the highest layer of the Bluetooth stack.

It is possible and desired that more than one service runs on a Bluetooth device. That is where ports are used.

> "A port is used to allow multiple applications on the same device to simultaneously utilize the same transport protocol. Almost all Internet transport protocols in common usage are designed with the notion of port numbers. Bluetooth is no exception, but uses slightly different terminology. In L2CAP, ports are called Protocol Service Multiplexers (PSM), and can take on odd numbered values between 1 and 32,767. Don't ask why they have to be odd-numbered values, because you probably won't get a convincing answer. In RFCOMM, channels 1 - 30 are available for use." [HR07]

As in other protocol types like TCP or UDP there are also well-known or reserved port numbers in Bluetooth protocols. In TCP ports 1 - 1024 are reserved for special applications. Port 80 is used for web traffic for example. In L2CAP the ports 1-1023 are reserved. The SDP for example uses port 1, or RFCOMM connections are multiplexed through L2CAP port 3. RFCOMM itself has no reserved ports.

The question for a developer is now how his application is able to find the correct service and the correct port number it is registered on. That is where SDP comes in. Every application can register itself at the SDP-server on the Bluetooth device. After registration it gets a dynamic port number by this server. When another device wants to connect to a running service it has to send a request with a description of the service to the SDP-server (cf. Figure 3.9). The main advantage of that technique is that there are no port conflicts [HR07].

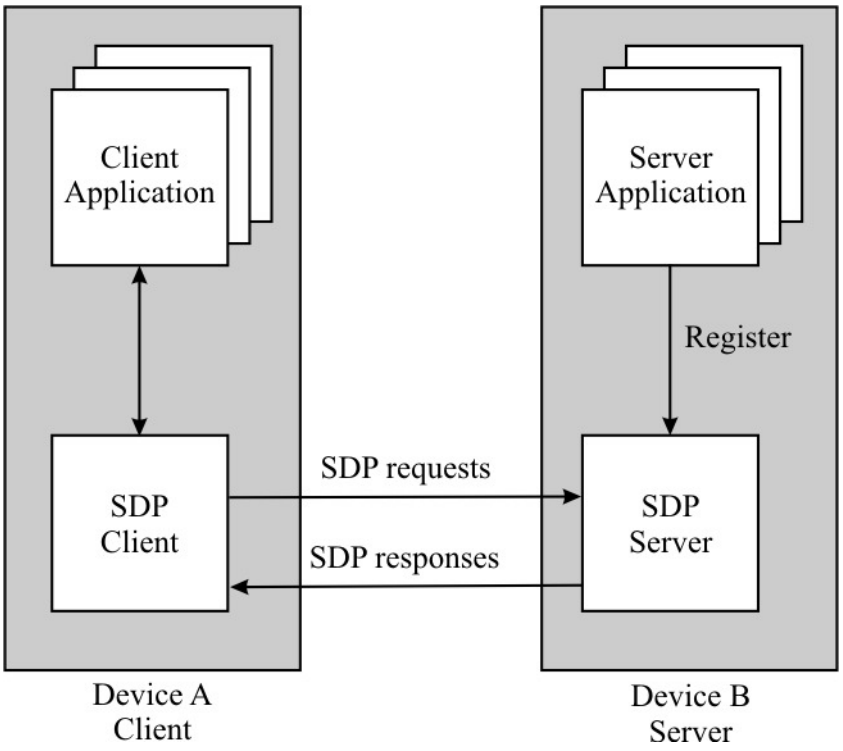

Figure 3.9: SDP client-server interaction [Blub]

4 Implementations of Bluetooth Stacks

An implementation of a Bluetooth stack is a collection of drivers, libraries and tools which make a developer capable of creating Bluetooth software. Typically there is one dominating stack for each of the most popular operating systems. An important point is that applications which are written for one special stack are not compatible to another stack which makes multi platform development rather difficult. [HR07]

Table 4.1 shows an overview of the most important Bluetooth stacks for different operating systems.

Table 4.1: Bluetooth stacks for different OS

Stack	OS	Comment
Widcomm	Windows	Widcomm was the first Bluetooth stack for the Windows operating system. The stack was initially developed by a company named Widcomm Inc., which was acquired by Broadcom Corporation in April 2004.
Microsoft	Windows	Windows XP includes a built-in Bluetooth stack starting with the Service Pack 2 update, released on 2004-08-06.
BlueZ	Linux	BlueZ is the official Bluetooth stack for Linux. As of 2006, the BlueZ stack supports all core Bluetooth protocols and layers.
OS X	OS X	Mac OS X implementation of a Bluetooth stack built into Mac OS X version 10.2 and later.

4.1 Bluetooth Development With Java - JSR-82

JSR-82 (Java Specification Request 82) is a standard defined by the Java Community Process for providing a standard to develop Bluetooth applications in Java. It is an open and non-proprietary standard for developing Bluetooth applications. The JSR-82 API hides the complexity of the Bluetooth protocol stack by exposing a simple set of Java APIs.

One of the major advantages of using JSR-82 for Bluetooth development is that the API is independent of the underlying hardware and operating system. It is possible to develop and test applications on one platform (i.e., on Win-32, Mac OS X, or Linux) and deploy it on another (a PDA or mobile phone). The goals of the JSR-82 specification are:

> "The overall goal of this specification is to define a standard set of APIs that will enable an open, third-party application development environment for Bluetooth wireless technology. The API is targeted mainly at devices that are limited in processing power and memory, and are primarily battery-operated. These devices may be manufactured in large quantities, meaning that low cost and low power consumption will be primary goals of the manufacturers." [Tho05]

There are some basic requirements specified in the JSR-82.

"This API is designed to operate on devices characterized as follows:

- 512K minimum total memory available for Java 2 platform (ROM/Flash and RAM). Application memory requirements are additional.
- Bluetooth communication hardware, with necessary Bluetooth stack and radio.
- Compliant implementation of the J2ME Connected Limited Device Configuration (CLDC) or a superset of CLDC APIs, such as the J2ME (Java 2 Micro Edition) Connected Device Configuration (CDC)" [Tho05]

CLDC (Connected Limited Device Configuration) is another standard (JSR-139) which describes a highly portable, minimum-footprint Java application development platform. The minimum requirements for a device which should be capable of the CLDC are at least 192 kB of total memory budget available for the Java platform and a 16- or 32 bit processor. It is designed for low power consuming devices which are often operating with battery power. [SM03]

"The requirements of the underlying Bluetooth system upon which this API will be built are:

- The underlying system shall be "Qualified" in accordance with the Bluetooth Qualification Program (...)
- The following layers are supported as defined in the Bluetooth specification version 1.1, and the implementation of this API has access to them.
  - Service Discovery Protocol (SDP)
  - RFCOMM (type 1 device support)
  - Logical Link Control and Adaptation Protocol (L2CAP)"[Tho05]

JSR-82 Implementation: Blue Cove

BlueCove is a LGPL licensed JSR-82 implementation on Java Standard Edition (J2SE) that currently interfaces with the Mac OS X, WIDCOMM, BlueSoleil, the Microsoft Bluetooth stack and the Linux Bluetooth stack BlueZ. It was originally developed by Intel Research and is currently maintained by volunteers. Figure 4.1 shows how the JSR-82 implementation of BlueCove interacts with other parts of the stack and the final application [Blua].

The layers which can be found in the Bluetooth controller and in the Bluetooth stack included in the operating system were already discussed. We will now focus on the part which is running inside the Java Virtual machine on the device.

BlueCove JNI - Java Native Interface The main advantage of Java is its platform independence. But there are some cases where it is inevitable to connect to platform dependet system interfaces. With the traditional Java programming approach it is not possible to call methods and functions outside the Java Virtual Machine. The developer is "imprisoned" in his Java based box. That's where the Java Native Interface (JNI) comes in.[Oft05]

"The JNI is a native programming interface. It allows Java code that runs inside a Java Virtual Machine (VM) to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly."[SM]

In BlueCove JNI is the interface to the native implemented Bluetooth stack of the operation system. It represents the lowest layer in the JVM.

CLDC (GFK) The GFK (Generic Connection Framework) are a set of classes and interfaces which extend the the CLDC. It was introduced because the standard I/O packages java.io or java.net did not match the given memory constraints of mobile devices. The GFK makes a generic approach of creating connections of many types possible. The developer is able to manage I/O of HTTP, datagram or streams with the same set of classes.

> "As the name implies, the GCF provides a generic approach to connectivity. It is generic because it provides a common foundation API for all the basic connection types - for packet-based (data blocks) and stream-based (contiguous or sequence of data) input and output."[Ort03]

JSR-82 API The JSR-82 implementation is the API on which the Bluetooth communication in the application relies on. It provides classes and methods that make it easier for a developer to use the Bluetooth adapter of the system.

Applications The highest layer of the Bluetooth stack is the application. The developer uses the given Java classes of the JSR-82 and the GFC to write his applications. In most of the cases these applications will be registered as services at the SDP-server. With this high level of abstraction the Java APIs provide, the developer can focus on his application and does not have to take care about the procedures in lower layers.

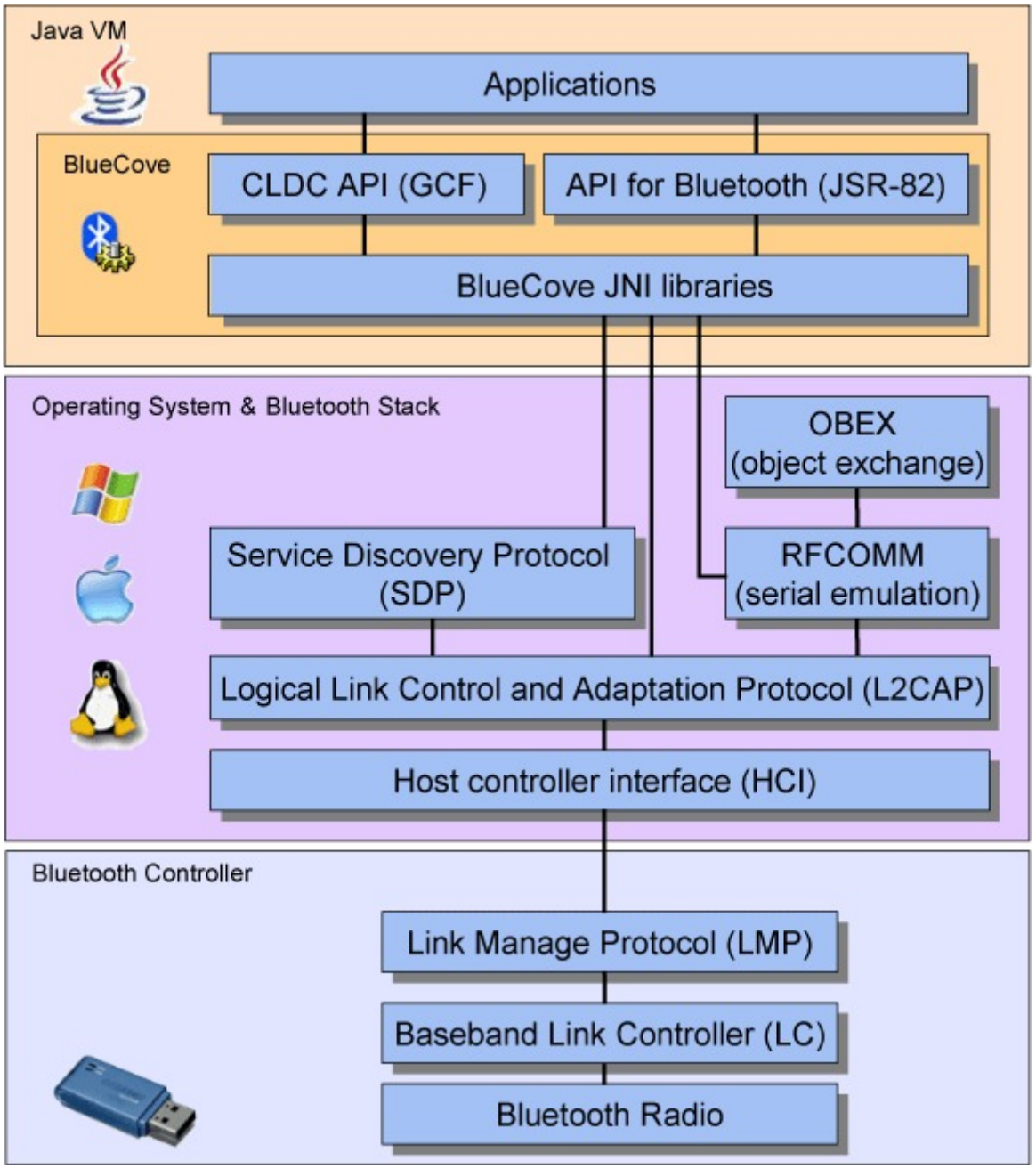

Figure 4.1: BlueCove integration in the Bluetooth stack [Blua]

5 Introduction to KNX

5.1 History

In the year 2002 the KNX Association introduced the first version of the KNX-specification. At this time three bus-standards were commonly used in Europe: EIB (European Installation Bus), the EHS (European Home Systems) and BatiBUS. The KNX specification is the result of a combination of all of them. It was the aim to create a standardized technology platform for all sectors of home and building automation. In 2003 the standard became EN 50090 ratified by the CENELEC (European Committee for Electrotechnical Standardization). Later on the KNXnet/IP specification was added. It describes how the KNX protcol is implemented using the commonly used Internet Protocol (IP). The advantage of this technique is that communication can also take place beyond the KNX network. It is for example possible to do a remote configuration or remote control of the network.
Today the Konnex association which cares for the specification has over 100 member companies from different sectors of home and building automation such as energy supply, lightning/shading or HVAC (Heating, Ventilating and Air Conditioning) [kon].

5.2 How it Works

A KNX system consists of three main parts: sensors, actuators and the bus. Sensors can be switches or sensors for temperature or wind. They send commands or status changes to the bus. Actuators receive these messages and react to them. Actuators can be light bulbs, an HVAC system or window blinds. The bus provides the infrastructure for the communication. It connects sensors and actuators with each other.

The whole system has a decentralized organization. That means there is no need for central processing unit which manages the system. The running peer to peer network algorithm makes sure that there is neither a bottleneck in the network nor that there is a single point of failure. Because of that a KNX system is very flexible and can be adjusted to the need of the customer very easy.

Bus Connection

The common method to transmit data is to use a twisted pair (TP) wire. The transmission speed is 9,6 kBit/s. The connections should be shielded against electromagnetic interference, thermal and mechanical stress to be robust against any kind of interference.

Other possibilities to connect devices to a KNX bus are:

- Powerline
- Radio
- Infrared

- Ethernet

[knx]

Group Communication

Every device has an individual 16 bit address following the Format in Figure 5.1.

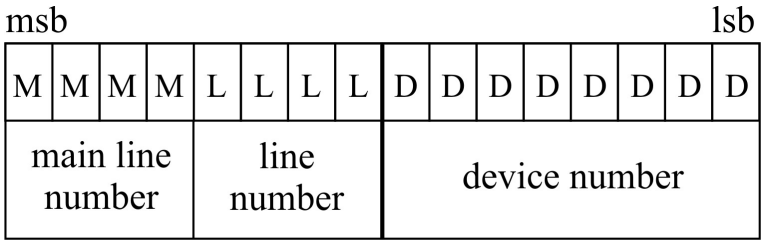

Figure 5.1: KNX addressing

It consists of the main number, the line number and the device number. The addressing of the devices is derived from the topology of a KNX network. A network may consist of one backbone line, different main lines and lines. There can be up to 15 main lines numbered from 1 to 15 connected to the backbone line via backbone couplers. Lines are subordinate to main lines. They are also numbered from 1 to 15 and connected to the main lines via line couplers. An area is a main line with all lines connected to it. KNX devices can be connected anywhere in the topology. They are numbered from 1 to 255 (Figure 5.2).

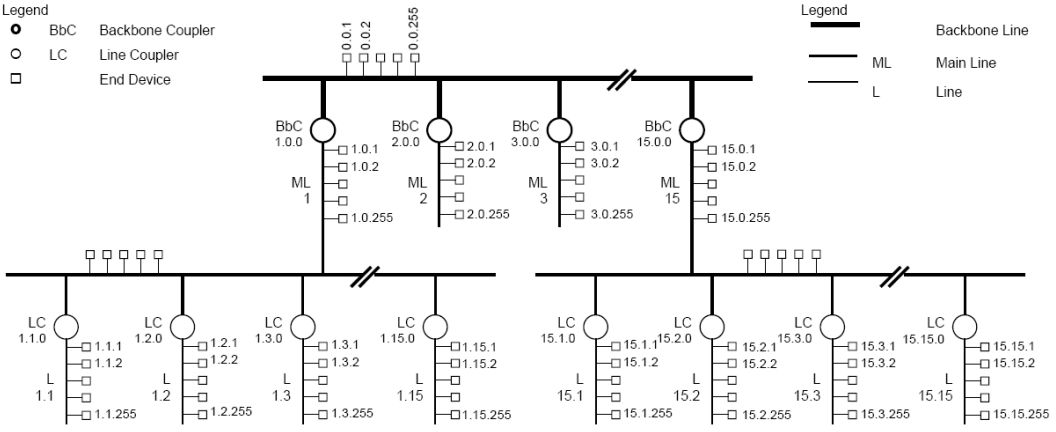

Figure 5.2: KNX network topology [Ass]

The address is programmed during the system installation by the engineer. He has to take care to use an ideal structure for the network and the addressing of the devices.

Exchanging process data in a KNX network is done within so called communication groups. The communication groups follow the producer-consumer model. That means some nodes write data on the bus (producer) and other nodes listen to the messages on the bus (consumer). Every node owns a table containing the logical addresses it is interested in. If a node sends data on the bus it sends a message as well as the corresponding logical address called group address. If the address is listed in the table of the receiving device the message will be forwarded to the application. The sender is also referred to as data source and the consumer to as data sink. An important point is that there is no knowledge about which device receives and processes the message [KN05].

5.3 Bus Access With Calimero

Calimero[1] is a set of Java libraries which was developed for KNX bus access. Calimero is a middleware that hides the complexity of the underlying communication mechanisms from the developer whose focus should be at the application. Another important point is the small footprint of Calimero. It only requires Java 2 Micro Edition APIs which makes the library perfect for embedded solutions. The easiest way of getting access to a KNX network is over an IP connection with an appropriate KNX/IP router. The KNX communication protocol over IP networks is called KNXnet/IP.

Once there is a connection to the configured KNX network, it is easy to send and receive messages to the bus. Using Calimero's API, it is possible to send messages to the KNX bus. To receive process data, a callback mechanism is available. It gets called whenever a message was produced by a node in the network. It's up to the developer to process it in a correct way.

The Calimero library works with a datapoint structure to represent a KNX network. Datapoints are saved in an XML file with a certain syntax. Because J2ME has no capabilities for reading and processing XML files Calimero brings its own XML read and write mechanisms.

[1] http://calimero.sourceforge.net/

6 Bluetooth - KNX Gateway

As already mentioned in Chapter 2 the task of this project was to develop a gateway with the corresponding client to control devices which are connected to a KNX network via a Bluetooth capable device. The name of the whole project is Priscilla according to one of Calimeros friends in the cartoons of Calimero. In Figure 6.1 the general setup of the final application is outlined.

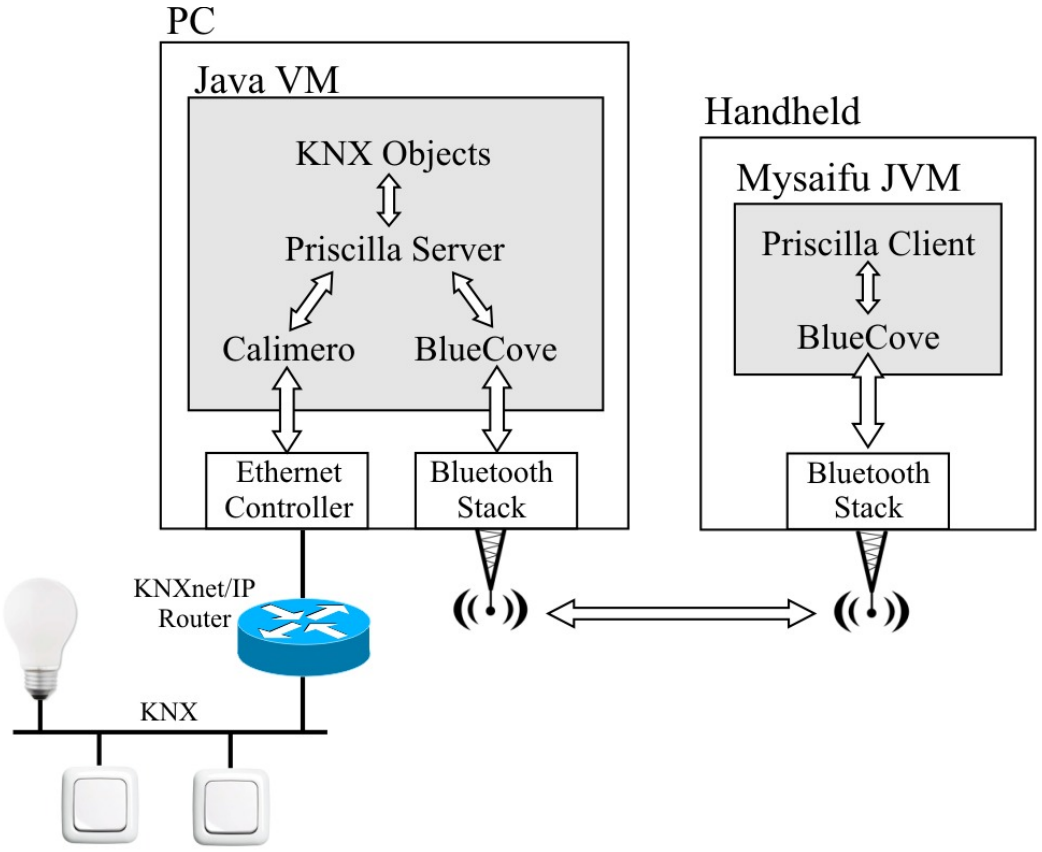

Figure 6.1: Project assembly

The basic setup consists of three parts: a KNX network, a PC and a handheld device. The KNX network in Figure 6.1 consists of two switches and a light bulb. To this network a PC is attached via a KNXnet/IP router. The PC acts as gateway. It communicates over two different interfaces. On the one hand it sends and receives commands from the KNX network with an Ethernet controller, on the other hand it communicates via a Bluetooth interface with connected clients. These clients display the status of the KNX devices in a user friendly way. It is also possible to change the status of the KNX devices, for example switch on the lights.

6.1 KNX Network Configuration Tool

The first part of the Priscilla project is a configuration tool with a user interface.

The KNX datapoints that have to be controlled using the client have to be described in an XML file. The syntax of the XML is based on the XML specification of the Calimero project [KMN07].

The whole file describes a set of datapoints that have to be controlled. The root XML element which occurs in every KNX XML file is `<datapoints>` :

```
<datapoints>
...
</datapoints>
```

The next step is to describe every datapoint inside the `<datapoints>` tag. This is done with the `<datapoint>` tag. In this tag the user can set important attributes about the device.

name Unique name of the KNX datapoint

stateBased Set to "true" if variable is state based, false if it is event based. An example for a state based variable would be a light with the states "on" and "off". The same light could also be controlled with an event based variable which could be controlled with a command like "increase the brightness of the light by 10 percent".

dptId Describes the type of the datapoint. The DPT numbers can be found in the KNX specification. DPT numbers have the form [mainNumber].[subNumber].

mainNumber Contains the main number of the class of the datapoint. This attribute is not needed if the dptId is set.

priority Priority of messages from devices in a KNX network. This attribute can be set to low, normal, urgent or system and specifies the priority of the message on the KNX bus.

```
<datapoints>
    <datapoint name="device1" stateBased="true" mainNumber="0" dptID="1.002" priority="low">
...
    </datapoint>
</datapoints>
```

Beside the already mentioned attributes there are other settings which the user can change. These settings are made via child tags of the `<datapoint>` tag. `<knxAddress>` describes the main group address of the datapoint which is used as destination address when sending changes of the datapoint's value. Multiple updating group addresses can be defined within the `<updatingAddresses>` tag. The datapoint's value is updated on incoming group messages with one of these updating addresses.
In Figure 6.2 an example of a small KNX network is outlined. There is one electric bulb and 2 switches. The bulb has the address 0/0/1 and uses this address to send messages. On the other side, there are two switches with the address 0/0/2 and 0/0/3. With these switches the light can be updated. The light has these addresses in the updating address table saved. Every time a switch is pressed it sends a message with including its address on the bus. The light recognizes the message and looks up the address in the mentioned table. If there is a match it can react on the message and turn the light "on" or "off".

A complete example of such a file with two devices could look like the following:

```
<datapoints>
    <datapoint name="device1" stateBased="true" mainNumber="0" dptID="1.002" priority="low">
        <knxAddress type="group">2306</knxAddress>
        <updatingAddresses></updatingAddresses>
```

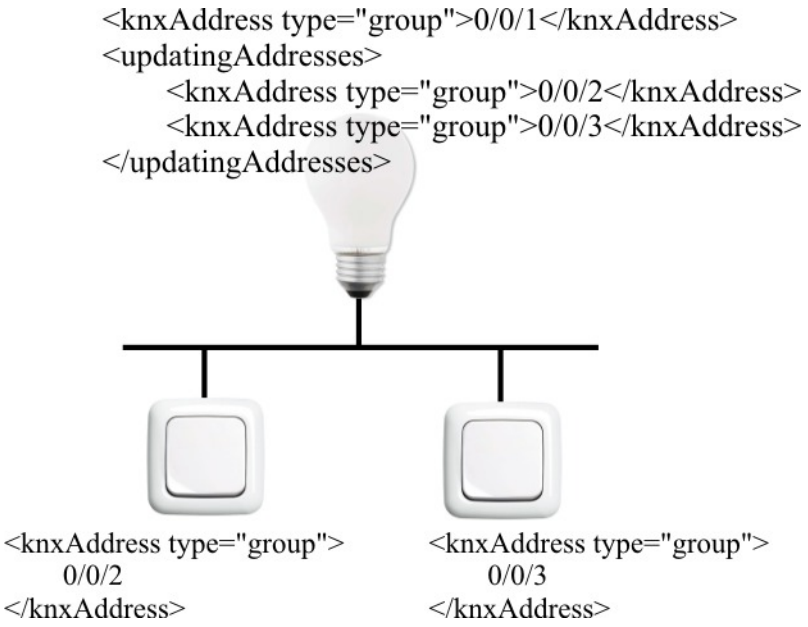

Figure 6.2: Updating address of an electric bulb

```
        <invalidatingAddresses></invalidatingAddresses>
    </datapoint>
    <datapoint name="device2" stateBased="true" mainNumber="0" dptID="5.001" priority="low">
        <knxAddress type="group">2563</knxAddress>
        <updatingAddresses>
            <knxAddress type="group">2819</knxAddress>
        </updatingAddresses>
        <invalidatingAddresses></invalidatingAddresses>
    </datapoint>
</datapoints>
```

The configuration tool was implemented in Java and the user interface package swing. An important part of this tool is the Calimero package. It was used to read and write the XML file. There are also methods for converting a given XML file to Java objects.

The result is a tool which allows the user to create a KNX network configuration with a simple user interface as it can be seen in figure 6.3. The features of this tool are as follows:

- Load and save XML files from the file system.
- Create as much datapoints as needed.
- Change of datapoint attributes (e.g., updating addresses)
- Choose between three different datapoint types: boolean, switch and scaling.

6.2 Bluetooth Server/KNX Connector

The server part of the Priscilla project is also entirely written in Java. It is a console application without user interface running on a PC. As shown in Figure 6.1 the server has two interfaces to other systems. On one hand there is an Ethernet interface which is connected to a KNX/IP router. On the other hand is a Bluetooth device. In our case it was a BlueFritz! USB stick running with a Microsoft Bluetooth stack on the PC.

For a proper start of the server four command line arguments are needed:

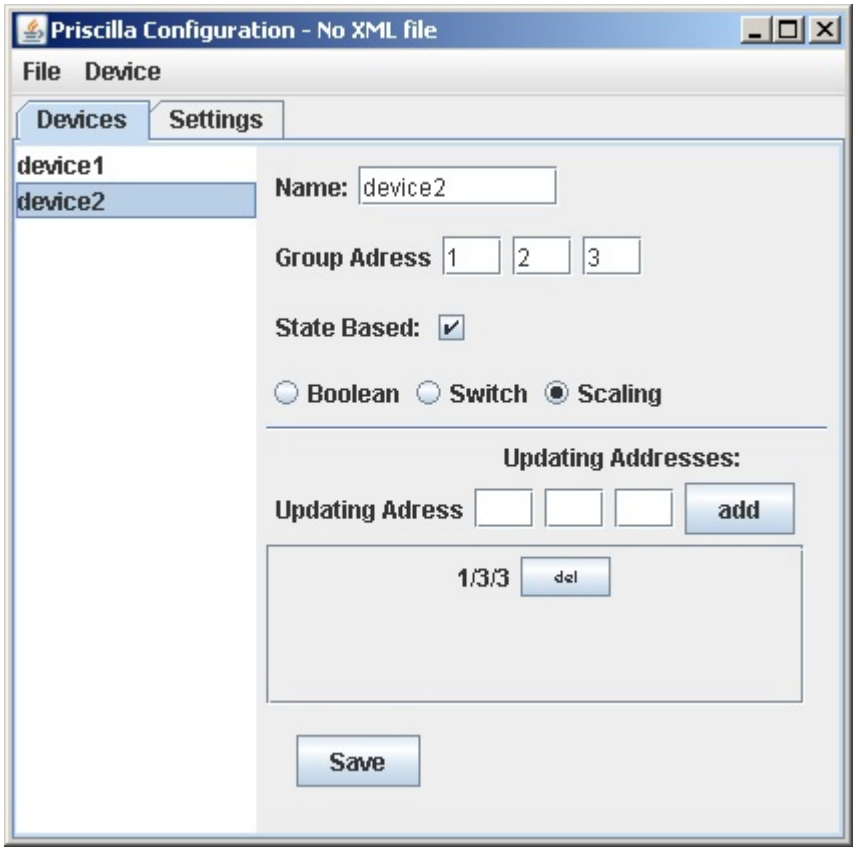

Figure 6.3: Screenshot: Configuration Tool

1. Local IP of the PC
2. Remote IP of the IP/KNX bridge
3. Used port of the IP/KNX bridge
4. Path to the XML file which describes the datapoints that have to be controlled by the clients.

Figure 6.1 outlines how the communication between the layers inside the Java application works.

6.2.1 Calimero

The intention of this project was to use the Calimero Java libraries for KNX bus access (cf. Section 5.3). In the server part of the Priscilla project, Calimero was used for two purposes. One was the connection to KNX network as well as sending and receiving data from this network, and the other one was the handling of the XML file where the datapoints are described.

The Java classes which interact with the Calimero API can be found in the package `tuwien.auto.priscilla.knx`. The class which handles the Connection is the `KnxConnection` class. This class implements the singleton pattern.In software engineering, the singleton pattern is a design pattern that is used to restrict instantiation of a class to one object. A common way to do this is to make the constructor of an object private and use a method, for example with

the name `getInstance()` to get an instance of this object. A typical use case of this technique would be an object which is responsible for writing outputs into a file. In our case, this singleton object is needed to open the connection and write data on the bus. Before an instance of a `KnxConnection` can be acquired, an `initialize` method which takes the local IP, the remote IP, the port and the path to the XML description of the KNX network to establish the connection to the bridge has to be called. The KnxConnection contains two other important functions: an `init()` function to establish the communication and a `writeData()` function which takes an address and the data which should be sent on the bus.

Another important part of the KNX package is the `MyProcessListener` class. It is initialized with the connection to the KNXnet/IP router. This class implements `ProcessListener` and has the callback method `groupRead()`. It is invoked whenever a new KNX message is available on the bus. The callback provides a `ProcessEvent` object as argument. This object contains information about the data (e.g., source address, user data) written on the bus. In the Priscilla project it listens for changes of the status of the datapoints of interest i.e., the datapoints that are specified in the XML file. Every time a status change is received the server part gets notified and can react on the change. Figure 6.4 outlines how the components work together to establish a connection to the KNX network.

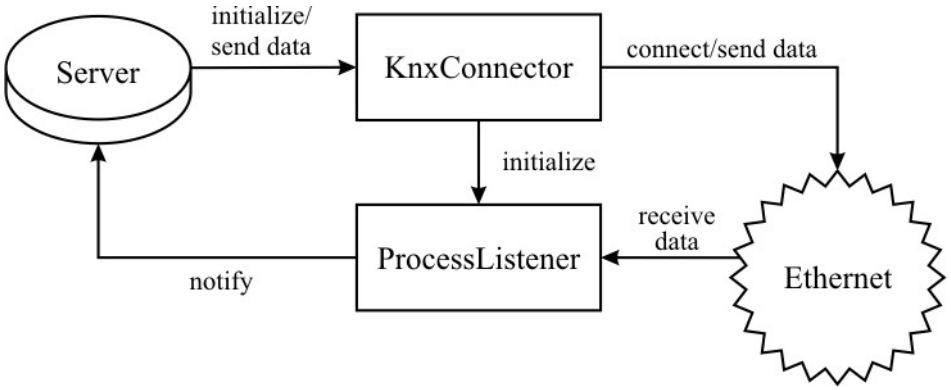

Figure 6.4: Relation of KNX classes with the server and the Ethernet

6.2.2 BlueCove

The second interface is the Bluetooth interface. In Section 4.1 we discussed the JSR-82 and one of its implementations: BlueCove. This implementation was used in this project to gain access to the Bluetooth interface.

The Priscilla server makes use of the SDP and its benefits (cf. Section 3.3.9). In particular a service was implemented and registered at the SDP server. The service itself uses the RFCOMM protocol.

> "Establishing an incoming connection requires first defining a URL string, which specifies some basics of both how to listen for incoming connection requests and the service record to advertise. There is a header followed by a list of attribute-value pairs.
>
> The URL string for RFCOMM always starts with "btspp://" (Bluetooth Serial Port Profile) and is followed by the address of the local Bluetooth adapter to use, or "localhost" if it doesn't matter. This is followed by a colon and the Service Class ID

> to advertise in the service record (more UUIDs can be added later). Next, up to five optional parameters can be specified. They appear as "attribute=value" pairs, each preceded by a semicolon."[HR07]

In the BlueCove apidoc [bcA] several UUIDs are described. The UUID 0x1101 is the one for the Serial Port which is used in this project to communicate. The only attribute specified in this case is the name of the service to make it easier for the client to find the correct service immediately.

> "`StreamConnectionNotifier` is the equivalent of a listening socket, and is required to accept incoming connections. A URL string, such as the one defined above, is passed to the static method `Connector.open` to instantiate the listening socket. (...) A program waits for and accepts an incoming connection using the `acceptAndOpen` method of the instantiated `StreamConnectionNotifier` object. (...) The method `acceptAndOpen` blocks and waits for an incoming connection. When a new connection has been accepted and established, it returns a `StreamConnection` object that can be used to exchange data with the remote Bluetooth device."[HR07]

The Java code to open a RFCOMM port and register it at the SDP looks as follows:

```
String url = "btspp://localhost:" + new UUID(0x1101).toString() + ";name=knxGateway;
service = (StreamConnectionNotifier) Connector.open(url);
StreamConnection streamCon = (StreamConnection) service.acceptAndOpen();
```

6.2.3 Server Logic

In the last sections, the interfaces of the server and how connections are established have been described. What's missing now is a link between these two interfaces. This link is represented by the server logic. It is responsible for mapping the data from one interface to another. For this reason a kind of management of all connections is needed. In Section 6.2.1 the management of the connection from the server to the KNX network was described. A singleton pattern was used. This was possible because there was only one connection available, the connection to the KNXnet/IP router over the Ethernet adapter. The management of the Bluetooth connections is a little bit more complicated. The problem is, that more than one Bluetooth device could connect to the service the server offers and each connected client has to be treated separately (cf. Figure 6.5).

The classes which are used for the connection management for Bluetooth can be found in the package `tuwien.auto.priscilla.bluetooth`. The server starts the Bluetooth service with the `BluetoothService` class. During initialization the Bluetooth service is registered as mentioned in Section 6.2.2. Then the server waits for connecting clients as explained in Section 6.2.2. The other important task of this class is the management of the connecting clients. Every time a client connects, a new object of type `BluetoothClient` is instantiated. This class contains a thread which is responsible for handling incoming requests. Since the method for reading from an input stream are blocking functions, it was necessary to spawn a new thread for each connected device. Only if data is received by the thread it reads the data from the stream and sends it back to the `BluetoothService` which will handle the data correctly. If the client for example wants to change the status of a KNX datapoint, the `BluetoothService` gets an instance of the `KnxConnection` and sends data to the KNX network.

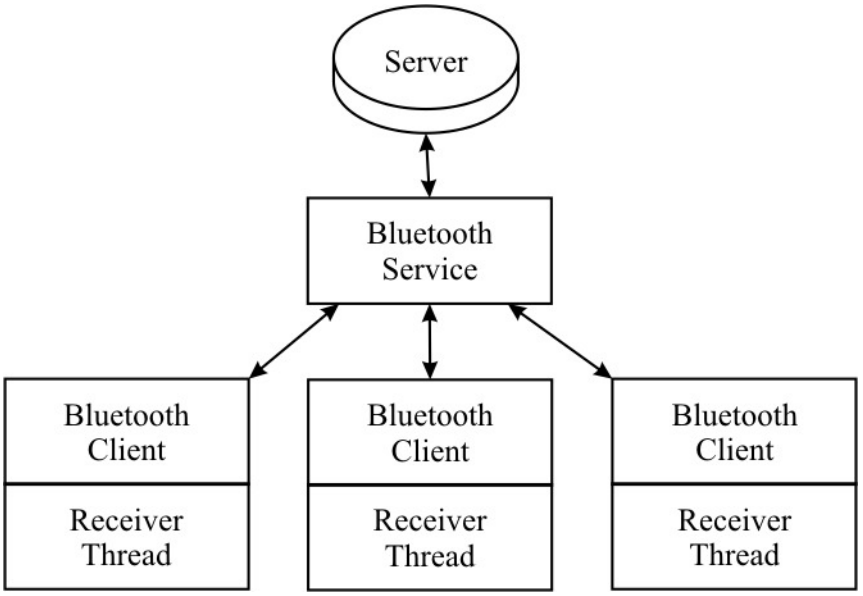

Figure 6.5: Management of Bluetooth connections

For purposes of user interaction another type of thread is running. It waits for an input on the console. If the user for example wants to stop the server he only have to input a "q" and the server releases all resources and stops itself.

6.3 Client

The client part of the Priscilla project should run on a handheld device like a PDA. The client software makes it possible for a user to control KNX devices attached to a KNX network. The test device in this case was an HP iPaq hx2790 running Windows Mobile 5.0 (cf. Figure 6.6) in combination with the Mysaifu JVM (cf. Section 6.6). The software running on the client can

Figure 6.6: HP iPaq hx2790

be divided into two parts. The first one consists of searching for nearby devices and connecting to the correct service. The second part is to retrieve data from the server to display the KNX devices on the user interface.

6.3.1 Connecting to the Server

Figure 6.7 presents a screenshot of the user interface of the client software. In this part of the software the user is able to search for active Bluetooth devices around. With the "connect" button it is possible to connect to the service which makes communication with the server and the KNX devices possible.

Figure 6.7: Screenshot of the client UI

Section 3.4 explained how devices are able to find each other. With the `DiscoveryAgent` this process is mapped into a Java class.

> "The process of detecting nearby devices can be outlined as follows:
>
> 1. Get a reference to the singleton instance of `LocalDevice`.
> 2. Get a reference to the `DiscoveryAgent` of the `LocalDevice`.
> 3. Use the `DiscoveryAgent` to start a device discovery via its `startInquiry` method.
> 4. Each time a device is detected, the `deviceDiscovered` method of a `DiscoveryListener` is invoked, with the details of the detected device as parameters.
> 5. When the device discovery has finished, the `inquiryComplete` method of a `DiscoveryListener` is invoked." [HR07]

The `deviceDiscovered` method adds the device to a list. The `inquiryComplete` method triggers the display of this list in the user interface. In Figure 6.7 two devices were found and can be selected to connect to one of them.

When the user decides to connect to a device the software has to connect to the correct service which is running on the device. The technique used for this is the Service Discovery. It works similar to the Device Discovery with some minor differences.

```
UUID uuids[] = new UUID[1];
uuids[0] = new UUID(0x1101);
int attridset[] = new int[1];
attridset[0] = 0x0100;

LocalDevice.getLocalDevice().getDiscoveryAgent()
.searchServices(attridset, uuids, device, this);
```

In this piece of code a service discovery is started. The `searchServices` method invoked at the end takes four parameters.

attridset Section 3.3.9 explained that the SDP returns attribute value pairs. In JSR-82 only some attributes are returned by default. If the developer needs other attributes he has to specify them. This is done with the first parameter of the `searchServices` method. In this case the Service Name with attribute `0x0100` is requested.

uuids This parameter is an array of UUIDs to search on. `searchServices` will only find services matching to every UUID in this array. In this case the only UUID is `0x1101` which describes the serial port.

device This object represents the device where the services should be searched for. This device was returned by the `deviceDiscovery` function.

this This parameters refers to a `DiscoveryListener` which contains the callback method `inquiryComplete()`. If a service was found or the inquiry is completed one of these methods get called and can treat the received data. The `servicesDiscovered` method for example checks if the correct service is running on the device by checking the name of the service.

If the connection establishment was successful the user interface displays it in the bottom line. With the disconnect button it is possible to release all resources and disconnect from the server properly.

6.3.2 Retrieve Data and Display Devices

After the connection has been established, the client is able to retrieve the available KNX datapoints from the server. Figure 6.8 show a corresponding screenshot.

It it possible to update the device list whenever the user wants to update the list. If the user wants to turn on the light he has to press one of the buttons or use the slider if a dimmer was installed. Another feature is that the client receives any status change of the KNX devices. If for example a light switch is pressed, the UI of the client will react on it and change the button status of the corresponding UI part to "on" or "off".

At the moment three different datapoint types are supported:

- Boolean
- Switch
- Scaling

These datapoints can be displayed with the current version but the UI can easily be extended for other devices.

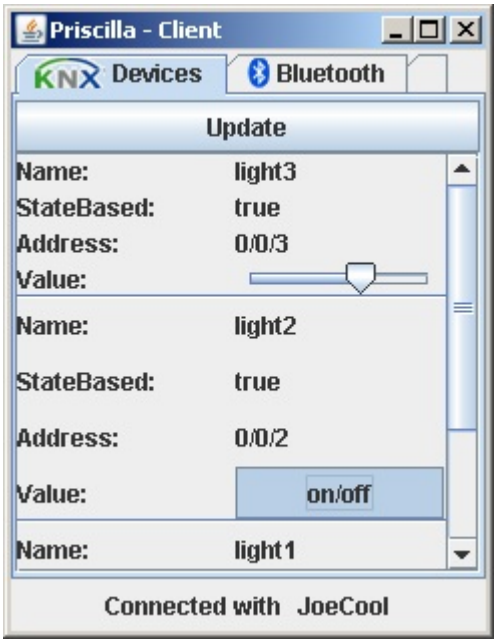

Figure 6.8: Presentation of the KNX devices in the UI

6.4 Communication protocol

For the communication from the server to the client and vice versa a small protocol was implemented. Whenever the client or the server wants to communicate with the other part this protocol is used.

The basic idea is that it has a layered structure. Each layer is separated by a ":" and the command ends with an ":EOF" as indication that the data was received completly. Every command is built this way:

```
<commmand>:<address>:<value>
```

Some commands do not need all parameters. The following commands are sent from the client to the server:

getDevices Request the server to send back all available KNX datapoints.

getStatus:1/1/1 Request the server to send back the current value of the datapoint with the KNX address 1/1/1. The address is arbitrary.

setStatus:1/1/1:true This command is sent when the user changes a value in the user interface. Depending on the type of the device a the new value will be sent to the server. In this example a boolean device button was turned on.

The following commands are sent from the server to the client:

devices:XML:EOF This is the answer of the `getDevices` command. The XML description of the KNX network will be sent to the client.

setStatus:1/1/1:true If there are status changes of KNX devices from outside the Priscilla Client software the server notifies the client with this command. The client will immediately change the status of the corresponding button.

6.5 Swing

Every user inteface in this project is done with help of Swing.

> "If you poke around the Java home page (http://java.sun.com/), you'll find Swing described as a set of customizable graphical components whose look-and-feel(L&F) can be dictated at runtime. In reality, however Swing is much more than this. Swing is the next-generation GUI toolkit that Sun Microsystems created to enable enterprise development in Java"[LEW+02]

But what does look-and-feel (L&F) exactly mean? L&F means the style of the user interface of an operating system for example. Every operating system GUI has its own L&F. Swing is capable of emulating these L&F settings of the operating system. There is a L&F setting for Windows, Linux or a "native" L&F which comes with Java. The advantage is that Swing programs plug in seamless to the operating system.

For the developer Swing brings other helpful features. The most important one is the great variety of components. There are ready to use tables, trees, sliders, progress bars, buttons etc. which can be easily be used by the programmer.

6.5.1 Eclipse Visual Editor

The Eclipse Visual Editor is a tool for GUI development which is embedded in the Eclipse platform.

> "For one thing, it generates high-quality code, comparable to what an experienced GUI developer would develop by hand, with no special artifacts that make modifications difficult. For another, its powerful parsing abilities allow full round-tripping of code, so changes made to the source code are reflected nearly immediately in the graphical editor.
>
> One of the most tedious tasks in building a Swing application manually is managing the placement of components using layout managers. Because it's a WYSIWYG graphical editor, Visual Editor makes it easy to get the appearance and behavior you want in your user interface." [Gal]

Today the Visual Editor is available in version 1.2. In Figure 6.9 a screenshot of a class being developed in the Visual Editor can be seen. The developer has the possibility to use the WYSIWYG facilities as well as changing the source code as usual. On the left side is a list with some components the developer can choose during development. He can simply put it via drag and drop on the stage and can immediately access the components for his purpose.

Every GUI in this project was developed with help of the Visual Editor.

6.6 Mysaifu - JVM

On the portable device a Java Virtual Machine was needed. The Mysaifu JVM is a Java Virtual Machine which runs on Windows Mobile. It is a free software under the GPLv2 (GNU Public License Version 2). An important point why this JVM was used is that BlueCove has been tested on the Mysaifu JVM [mys].

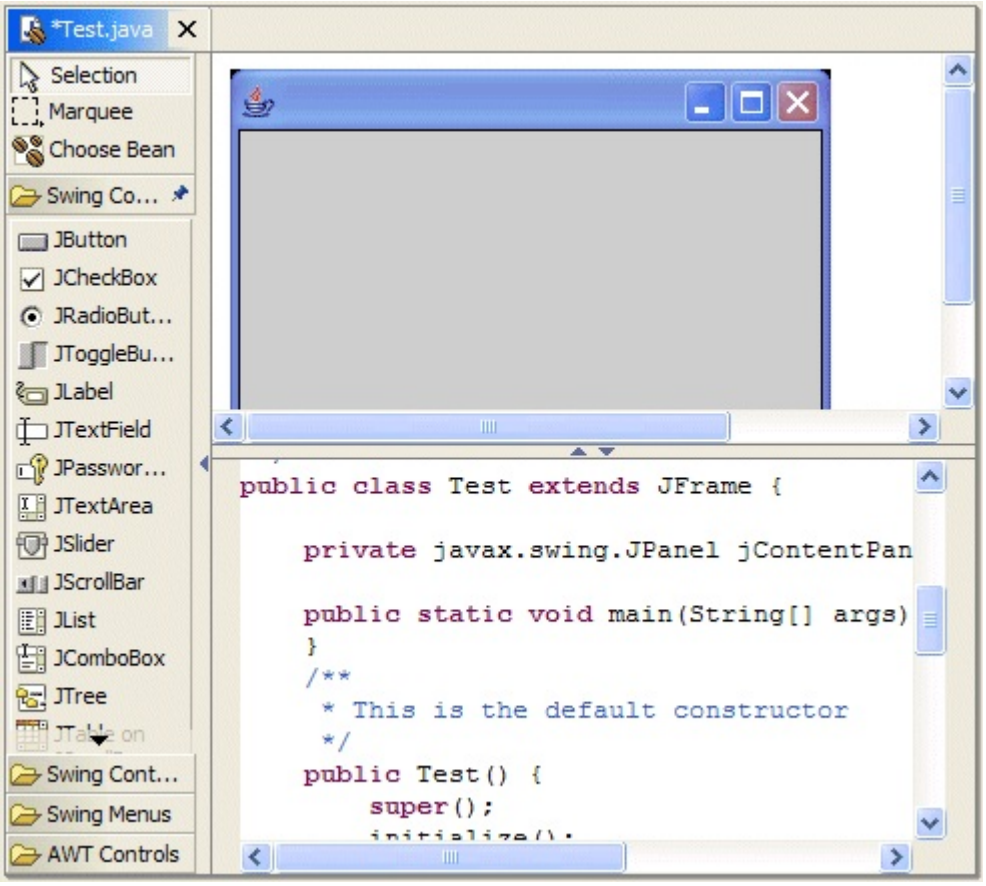

Figure 6.9: Editing a Swing class with the Visual Editor [Gal]

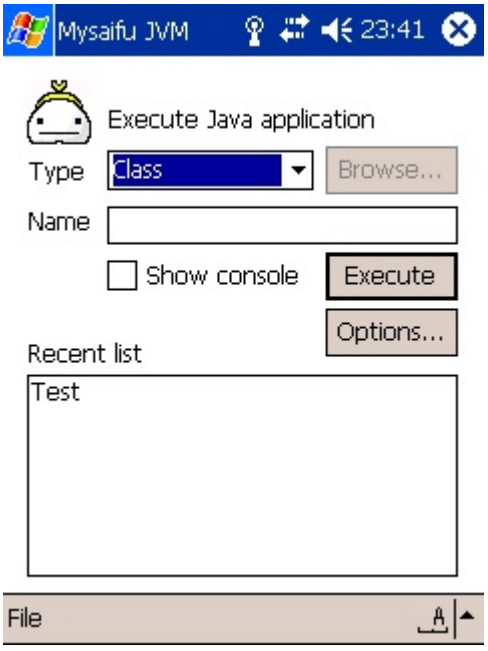

Figure 6.10: Input dialog of the Mysaifu JVM

In Figure 6.10 the main screen of the JVM can be seen. The user has to enter a Java application class name or jar file name to execute and set the correct classpath that the Java classes can be found on the system.

7 Conclusion

At the moment there is a trend to communicate wireless. Every notebook or mobile phone has adapters embedded which makes it possible to send and receive data without cable. One big representative is Bluetooth. It is developed since 1999 and is now one of the best established wireless technologies. Nearly each mobile phone or notebook has a Bluetooth adapter on board. It brings a lot of advantages, but also some disadvantages. The energy consumption is low for example, but the transfer rate compared to the 802.11 Wlan standard is also relatively low.

To completely understand the Bluetooth technology and all of its features it needs a lot of technical knowledge. The Bluetooth specification has about 1,400 pages and contains everything a developer needs to know if he wants to create a Bluetooth adapter. The good news are that there are some tools which makes it easy for software developers to make use of the Bluetooth technology. The JSR-82 Java API for example introduces a high level abstraction of the Bluetooth functionality. The advantage is that developers do not have to know every detail about how communication is initialized and how the whole low level functionality works. With only a little effort a small client-server application which communicates via Bluetooth can be developed.

The result of this project is an example of such a client-server architecture. On the one hand, a server which acts as a gateway between Bluetooth and the KNX has been developed, on the other hand a client that sends and receives data to the server. With this version of the software the user is able to control three different types of datapoints with his PDA which are attached to a KNX network. Besides the mentioned client and server a configuration tool which makes it possible for the user to specify the datapoints of interest with a user interface was developed. The result of this definition is an XML file which is required by the server.

Bibliography

[bcA] *BlueCove Apidoc*. http://www.bluecove.org/apidocs/index.html. [Online; accessed 10-June-2008].

[Blua] *BlueCove Homepage*. http://code.google.com/p/bluecove/. [Online; accessed 21-May-2008].

[Blub] *Bluetooth Specification 2.1*. http://www.bluetooth.com/Bluetooth/Technology/Building/Specifications/Default.htm. [Online; accessed 09-May-2008].

[Com01] COMMITTEE, ETSI TECHNICAL: *ETSI EN 300 328-1 V1.3.1*, 12 2001.

[Gal] GALLARDO, DAVID: *Build GUIs with the Eclipse Visual Editor project*. http://www.ibm.com/developerworks/opensource/library/os-ecvisual/.

[HR07] HUANG, ALBERT S. and LARRY RUDOLPH: *Bluetooth Essentials for Programmers*. Cambridge University Press, New York, 2007.

[KMN07] KASTNER, WOLFGANG, BORIS MALINKOWSKY and GEORG NEUGSCHANDTNER: *Calimero: Next Generation, Konnex Scientific Conference*. 2007.

[KN05] KASTNER, WOLFGANG and GEORG NEUGSCHANDTNER: *EIB: European Installation Bus*. CRC Press, 2005.

[knxa] *KNX information for developers*. http://www.knx.de/entwickler/index.html. [Online; accessed 26-May-2008].

[knxb] *Konnex Association, KNX specification 1.1*.

[kon] *Konnex Association*. http://konnex.org. [Online; accessed 09-October-2008].

[LEW+02] LOY, MARC, ROBERT ECKSTEIN, DAVE WOOD, BRIAN COLE and JAMES ELLIOTT: *Java Swing*. O'Reilly, 2002.

[Met99] METTALA, RIKU: *Bluetooth Protocol Architecture*. technical report 1.C.120/1.0, Bluetooth SIG, 1999.

[MT02] MERKLE, ANDREAS and ANESITS TERZIS: *Digitale Funkkommunikation mit Bluetooth*. Franzis, 2002.

[Mul01] MULLER, NATHAN J.: *Bluetooth Demystified*. McGraw-Hill, 2001.

[mys] *Mysaifu JVM Homepage*. http://www2s.biglobe.ne.jp/~dat/java/project/jvm/index_en.html. [Online; accessed 26-November-2008].

[Oft05] OFTERDINGER, ADRIAN: *Java Native Interface ab JS2SE 1.4*. JavaSpektrum, 5:32–35, 2005.

[Ort03] ORTIZ, C. ENRIQUE: *The Generic Connection Framework*. Sun Developer Network, 2003.

[SIG] *About the Bluetooth SIG*. `http://www.bluetooth.com/Bluetooth/SIG/Default.htm`. [Online; accessed 12-May-2008].

[SM] SUN MICROSYSTEMS, INC.: *Java Native Interface Specification*. `http://java.sun.com/j2se/1.5.0/docs/guide/jni/index.html`. [Online; accessed 26-November-2008].

[SM03] SUN MICROSYSTEMS, INC.: *Connected Limited Device Configuration Specification Version 1.1*. `http://jcp.org/en/jsr/detail?id=139`, 2003. [Online; accessed 26-November-2008].

[Tho05] THOMPSON, TIM: *Java APIs for Bluetooth Wireless Technology (JSR 82)*. `http://jcp.org/en/jsr/detail?id=82`, 2005. [Online; accessed 26-November-2008].