Tuukka Puranen

Implementing a Lecture Feedback System Using Semi-Ad-Hoc Wireless Networks

Master's Thesis
in Information Technology
October 19, 2007

UNIVERSITY OF JYVÄSKYLÄ
DEPARTMENT OF MATHEMATICAL INFORMATION TECHNOLOGY

Jyväskylä

Author: Tuukka Puranen
Contact information: tupepura@jyu.fi
Title: Implementing a Lecture Feedback System Using Semi-Ad-Hoc Wireless Networks
Työn nimi: Luentopalautejärjestelmän toteutus langattomia semi-ad-hoc verkkoja hyödyntäen
Project: Master's Thesis in Information Technology
Page count: 119
Abstract: This thesis presents a system for assisting learning: a prototype of a mobile device based electronic questionnaire and feedback system for mass lectures. Technical details regarding the protocols, architecture, and implementation of such system are provided. The system utilizes short range radio technologies, and this thesis examines and compares means to form and use these semi-ad-hoc wireless networks for this purpose. The study emphasizes Bluetooth technology but covers also a slightly broader area. Using Bluetooth's built-in methods for connection establishment, different network establishment procedures can be implemented. Several alternatives are presented and evaluated in a given context and against given requirements.
Suomenkielinen tiivistelmä: Tässä tutkielmassa esitellään mobiilaitteille suunniteltu luentopalautejärjestelmä, jolla pyritään edesauttamaan oppimista massaluennoilla. Työssä tarkastellaan tällaisen järjestelmän prototyypin arkkitehtuuria, protokollia ja teknistä toteutusta. Järjestelmä hyödyntää lyhyen kantaman radiotekniikkaa, ja tutkielmassa tarkastellaan keinoja langattomien verkkojen muodostamiseen tätä varten. Tutkimus keskittyy Bluetooth-tekniikkaan, mutta sivuaa myös muita langattomia kommunikaatioteknologioita. Bluetooth-verkko voidaan muodostaa soveltaen Bluetoothin sisäänrakennettuja proseduureja eri tavoin. Tässä tutkielmassa tarkastellaan ja arvioidaan useita tällaisia tapoja annetussa kontekstissa.
Keywords: wireless network, connection establishment, lecture feedback, PI, peer instruction, InSitu, GPRS, mobile, semi-ad-hoc, WLAN, Bluetooth, BlueZ, Petri nets
Avainsanat: langaton verkko, yhteydenmuodostus, luentopalaute, PI, adaptiivinen opetus, InSitu, GPRS, mobiililaite, semi-ad-hoc, WLAN, Bluetooth, BlueZ, Petriverkot

Preface

This thesis is an extension to author's Bachelor's thesis on the subject. The reader does not have to be familiar with the previous work in order to fully comprehend the contents of this study.

This study is a result of two years of work at the Department of Mathematical Information Technology at the University of Jyväskylä. The author has been employed at the department half time as a research assistant during years 2005–2007. The novelty of research has given a great deal of independence and responsibility, but the task can also be characterized as struggle against unstable and immature environment and numerous invisible variables. There has been, for example, one complete rewrite of server program, one *attempt* of complete rewrite of mobile client program, and three complete rewrites of the router program during these two years.

This thesis is, to some extend, first on the subject at this depth. It must be noted, to be fair, that at least the author was surprised by vastness of the task; considerable amount of work is still left for future research. That said, the author expresses his wish for continuity of research and development in one form or another.

It has been a sudden leap — not only to world of software engineering and mobile systems — but also that of scientific method, academic papers, seminars and presentations, which now culminates to this study, before the author leaves this particular research for new challenges.

Acknowledgements

A book is never written by a single person; I am not sure whether this is true when it comes to master's thesis, but enormous amount of group work was done before the actual writing process, and cannot be left without acknowledgement.

First I would like to thank Vesa Lappalainen for providing an opportunity to work in academia, and also for invaluable advice during both the research work and the writing of this thesis. It has been a pleasure to work at the Department of Mathematical Information Technology.

A great deal of background work was shared with me by Mikko Tyrväinen, with whom I worked closely during these two years. Of the topics covered in this thesis, especially the development of system protocols and overall architecture was a joint operation; special thanks should also be made for working on the architectural figures seen in this thesis.

Advice from staff members at the department was much appreciated; thanks to Jonne Itkonen, Antti-Juhani Kaijanaho, Pekka Neittaanmäki, Tommi Kärkkäinen, and few others for various reasons.

I would like to thank the following persons, who contributed to study by attending to performance tests and by providing devices: Peter Ciszek, Sauli Korhonen, Pauli Kujala, Antti Lahtinen, Olli Rautiainen, Mikko Savolainen, and Janne Silvendoinen.

A special mention should also go to staff at upper secondary school of Pieksämäki and that of the University of Joensuu for giving a curious young student a first introduction to information technology studies during years 2001 and 2002.

To my parents, Pasi and Kaisa, for buying us the Amiga 500 — and for few other things along the way — *thank you*.

Jyväskylä, October 2007

Tuukka Puranen

Contents

Appendices

List of Figures

List of Tables

Glossary

100Base-T. One standard for Ethernet over twisted pair. – *Cf.* ETHERNET

3G. 3rd generation of mobile network standards.

APMC. All to Predefined Module Connection; a possible connection scheme used in the InSitu system in the future.

APPC. Adapted Piconet Physical Channel; one of the physical channels in Bluetooth. – *Cf.* BLUETOOTH

AWT. Abstract Window Toolkit; part of the Java programming language. – *Cf.* JAVA SE

BNMC. Buffered Notify Module Connection; a possible connection scheme used in the InSitu system in the future.

Bluetooth. A specification for wireless personal area networks.

BPPC. Basic Piconet Physical Channel; the default physical channel in Bluetooth communication. – *Cf.* BLUETOOTH

CLDC. Connected Limited Device Configuration; a specification of a framework for Java ME. – *Cf.* JAVA ME

Ethernet. A collection of networking technologies for local area networks. – *Cf.* LAN

GIAC. General Inquiry Access Code; the default access code for Bluetooth inquiry. – *Cf.* BLUETOOTH; LIAC

GPRS. General Packet Radio Service; a mobile service for data transfer. – *Cf. next*

GSM. Global System for Mobile communications; a standard for mobile phones.

Hello Point. *See* NOTIFICATION MODULE

IBC. Inquiry Based Connection; a connection scheme used by the InSitu system.

IEEE. Institute of Electrical and Electronics Engineers; an international non-profit organization for advancement of technology.

IMP. InSitu Mobile Protocol; a protocol for server-client communication in InSitu system.

IMPv2. *See previous*

InfoNode Docking Windows. An open-source Java UI framework. – *Cf.* UI; JAVA SE

IP. Internet Protocol; a protocol for communicating across a packet-switched network. – *Cf.* TCP

IPP. InSitu Projector Protocol; a protocol for server-projector communication in InSitu system.

IRP. InSitu Router Protocol; a protocol for server-router communication in InSitu system.

Java SE. Java Standard Edition; widely used general platform for programming in the Java language. – *Cf. next*

Java ME. Java Micro Edition; a subset of the Java platform for small devices. – *Cf. previous*

JCommon. An open-source general purpose Java class library.

JFreeChart. An open-source Java charting library.

JSR. Java Community Process; a formalized process for defining the future versions of the Java platform. – *Cf.* JAVA SE

Joda Time. An open-source Java date and time management library.

L2CAP. Logical Link Control and Adaptation Protocol; a protocol within Bluetooth stack. – *Cf.* BLUETOOTH

LAN. Local Area Network; a computer network typically covering a small local area. – *Cf.* WLAN

LC. Link Controller; a protocol within Bluetooth stack. – *Cf.* BLUETOOTH

LIAC. Limited Inquiry Access Code; an access code for Bluetooth inquiry. – *Cf.* BLUETOOTH; GIAC

MDC. Module Discovery Connection; a connection scheme used by the InSitu system.

MIDP. Mobile Information Device Profile; a specification for the use of Java on embedded devices.

NMC. Notify Module Connection; a connection scheme used by the InSitu system.

Notification Module. A Bluetooth module available for client initiated connection establishment, used for reporting a presence of a device.

PAC. Predefined Address Connection; a connection scheme used by the InSitu system.

PI. Peer Instruction; a methodology for enhancing learning during lectures.

Piconet. An Ad-Hoc network of Bluetooth devices. – *Cf.* BLUETOOTH; SEMI-AD-HOC

RFCOMM. Radio Frequency Communication; a protocol within Bluetooth stack. – *Cf.* BLUETOOTH

SDP. Service Discovery Protocol; a protocol for searching services from Bluetooth devices. – *Cf.* BLUETOOTH

Semi-Ad-Hoc. An Ad-Hoc network with some known static components.

SIG. Special Interest Group; a consortium with a particular interest in a specific technical area.

TCP. Transmission Control Protocol; a transport layer protocol for communicating packet data. – *Cf.* IP

UI. User Interface.

Ultra-wideband. Technology for short-range high-bandwidth communications.

USB. Universal Series Bus; a serial bus standard mainly for interface devices.

UTF-16. 16-bit Unicode Transformation Format; a variable-length character encoding.

UUID. Universally Unique Identifier; an identifier standard for identifying entities. *Here* Used to identify Bluetooth services. – *Cf.* BLUETOOTH

WLAN. Wireless Local Area Network. – *Cf.* LAN

1 Introduction

What I hear, I forget. What I hear and see, I remember a little. What I hear, see and discuss, I begin to understand. What I hear, see, discuss, and do, I acquire knowledge and skill. What I teach to another, I master.

— Mel Silberman, adapted from [40]

InSitu is a prototype of an electronic mass lecture feedback and questionnaire system implemented by utilizing students' mobile devices. This thesis examines the system, reasons behind its development, and its possible implications.

1.1 Structure of the Thesis

This thesis provides a sketch for implementation of the lecture feedback and questionnaire system by presenting an overall view of domain. Due to novelty of the InSitu system, one objective is also to introduce and stabilize terminology. This study emphasizes inspection of Bluetooth connection establishment and communication but takes also a slightly broader view, since this is among the first papers on the subject. The system utilizes also ordinary local area networks, but the issues are mainly trivial and will, therefore, be examined briefly. There will be an introduction to the system and its goals to provide insight to typical usage and usage environment and to provide a general view to evaluate and compare different alternatives and methodology presented.

In Chapter 2, some problems in mass lectures are addressed; a solution is proposed; and effects of the solution are briefly evaluated as motivation. In Chapter 3, technical background and theoretical foundation are examined in detail in order to provide means to analyze and measure different methods for solving the research problem. Chapter 4 gives broader view on the technical aspects of the current implementation of the system and gives detailed context in which the solution is applicable. It also presents the most relevant requirements, in the scope of this thesis and in this particular context, that guide the analysis of different methods. In Chapter 5, four different methods for solving the research problem are identified, analysed and measured.

1.2 Research Problem

This thesis introduces the feedback system, outlines an architecture and an example implementation, and attempts to find out whether this implementation is viable in a given context. Thus, some general remarks about implementability of such system will be made. This thesis will also examine whether it is possible, and how, in an auditorium or a classroom, to form a Bluetooth network of a single server and several dozens of mobile devices in an acceptable time frame. This study will, however, present only estimates, since limited hardware supplies[1] prevent exhaustive examination with actual volumes. The primary method in the empirical part of the study is measurement of the performance of the prototype.

In other words, this thesis provides two main scientific contributions. The first is to introduce a wireless mobile device based lecture feedback system for assisting learning and provide outline for its implementation. The other is to evaluate whether a Bluetooth can be utilized for this purpose from theoretical and empirical point of view, and how this can be done by constructing a semi-ad-hoc[2] network.

Directions for further research will also be provided. These include evaluating effects of the actual radio traffic with mathematical modelling and simulations. That said, network simulations, cognitive effects of the final system, and, for example, physical radio traffic measures are outside the scope of this thesis.

In general, problems that may prevent large number of Bluetooth devices from functioning in same environment (i.e., overloading of radio frequencies) will not be addressed in detail in this thesis; instead, examination of hardware and software limitations, connection algorithms, performance, and concurrency issues will be performed.

Notes

[1]We have been able to acquire about a dozen mobile devices for day to day testing and an additional half a dozen for larger intermittent tests.

[2]In this context, the term *semi-ad-hoc* network means a wireless network that has some known elements but whose structure is not fully determined beforehand.

2 Overview

The mind is not a vessel to be filled, but a fire to be kindled.

— Plutarch [2]

This chapter presents overview of the scope of this thesis. Section 2.1 presents background, and Section 2.2 history of the system. In Section 2.3, some problems in mass lectures are addressed and a solution is proposed in form of a software system. Effects of the solution are briefly evaluated as motivation in Section 2.4.

2.1 Background

In 1996 a staff member of the University of Jyväskylä was attending a lecture in Maryland, USA. The lecture was given, in physics, by Eric Mazur who used a questionnaire system while lecturing. Mazur's system was part of his teaching strategy that was later published under term *Peer Instruction*[1] (PI). A central part of the PI method are so called *Concept Tests* [36, 133–134] in which lecturer asks conceptual multiple choice questions and directs the lecture based on the answers. The effects of the approach have been studied since (see, for example, [38], [10], and [33]), and Mazur summarizes his ten year experience in [12]. The pedagogical results are, indeed, encouraging.

Apart from the system presented in this thesis, other similar systems (see, for example, [11] and [17]) have been developed and their implications are under active research. Many (but not all[2]) implementations of PI rely on electronic questionnaire systems, and many have adopted either radio or infrared based communication.

It must be, however, noted that while the InSitu system has initially been based on idea of Peer Instruction, it has not been developed solely for the PI *per se*; the system can be used to assist lectures in any way teachers find suitable[3].

2.2 History

First attempts to implement an electronic lecture feedback system at the University of Jyväskylä were done in 1997. The system was based on custom made radio devices. The devices communicated with a server program running on a computer at

auditorium. A first set of 30 devices was tested successfully in the end of 1990s, but it was quickly noted that the system was impractical for several reasons. Firstly, it was unpleasantly immobile, in a sense that devices themselves were large, not to mention the fact that there should have been 200 of them in the final system. Secondly, it was costly to build 200 devices from ground up. Thirdly, the usability of the system was dictated by the usability of the radio devices, which in turn was dictated by the capabilities of the electronic parts available. The first version was, indeed, primitive [31], and a more sophisticated system would have cost even more.

Nevertheless, the system was tested about once in a year in 2000–2005 in a programming course, and feedback was positive. It was pointed out, for example, that this method encourages students to think actively during the lecture, and it also eases answering the lecturer's questions, since no one[4] is able to see whether you answer correctly or not.

To attack the issues on the radio system, a different implementation approach was taken in 2005. Short range radio started to become common in mobile phones, and an idea to use students' mobile devices and Bluetooth was brought up. This resulted in two years of research and prototype development, which has lead to this particular paper.

2.3 Overview And Purpose of InSitu System

InSitu is a system designed to aid communication at mass lectures. The problem is usually lack of feedback to a lecturer and consequent inability to adjust teaching according to the level of students. This is especially true with basic studies where students with different knowledge levels are present. This is also a matter of cultural environment; it is more difficult to get Finnish students to interact during lecture than, for example, those from southern Europe.

A way to assist communication is to provide an electronic query and feedback system where the lecturer's actions include, but are not limited to, sending questions to students and receiving and analyzing answers immediately. The system is implemented by providing a software for mobile devices with short range radio capabilities, such as Bluetooth. Many students have mobile phones with support for technologies like Java ME and Bluetooth, and this has been seen strength compared to other electronic feedback methods; the lecturer does not have to provide devices; they are brought and maintained by the students.

Typical usage environment of InSitu system is an auditorium or a classroom of an educational establishment. There are typically several dozens and at most few

hundred devices present at a time. The devices vary widely in properties; mobile phones come from several different manufacturers, have dozens of models and versions and utilize various different communication technologies.

When compared to other systems, a system based on students' mobile devices costs practically nothing. Also — in general — the devices have charged batteries, they are familiar to users, and they are not broken in a way that requires attention from the institutional side. It is, on the other hand, more difficult to ensure that the system functions properly with every available mobile device, but as the technical environment matures, this will likely become less laborious.

2.4 Possibilities and Effects

The overall goal of the system is to provide means to adapt teaching and assist learning with modern mobile technology. One obvious usage is the Peer Instruction method mentioned in Section 2.1. The system can also be used to measure understanding in more traditional teaching schemes by, for example, allowing students to state their opinion about whether they are familiar with the current topic or not. Other usage scenarios include collecting detailed background information, gathering opinions or even performing exams.

General effect of the Peer Instruction method is that it enhances learning. Results from Crouch and Mazur show that students taught with PI significantly outperform the students taught traditionally, averaging 7.4 out of 10 compared to 5.5 out of 10 [12]. They also state that all measures indicate that students' quantitative problem-solving skills are comparable to or better than those achieved with traditional instruction.

A question is raised whether a lecture should be technology assisted at all; and that lecturers must be able to teach without technical devices. Indeed, systems of this kind must not be substitute for bad teaching. Also some concerns are expressed about the passivating effect of simple button pressing during lectures, but to be fair, it must be noted that sitting on lectures in general can be passivating by itself. Also, in some cases, an electronic system has even activated students to debate about the answers[5].

Notes

[1]In 1997, Eric Mazur published a book [32] that provides details on his teaching strategy.

[2]The electronic system is in PI mainly used to collect answer data from students, which can also be implemented, for example, using simple pen and paper method. This is not as costly as electronic approach, but lacks possibility to accurately record and analyse the results.

[3]Examples of usage outside Peer Instruction methodology include charting backgrounds in seminars and measuring understanding by directly querying subjective opinions in real time.

[4]Lecturer may want to know how particular individual answers, and this possibility is included in the system, but in most cases the teacher is interested only in general notion.

[5]The author attended an example lecture in spring 2006, in which some intense debates regarding answers to programming related questions and their justifications emerged.

3 Wireless Communication

In this chapter the technical background and theoretical foundations are examined in detail in order to provide means to analyze and compare different methods presented in subsequent chapters. Section 3.1 presents Bluetooth technology, its background and implementation. In Section 3.2, an overview of WLAN technology is rendered, and in Section 3.3 GPRS technology is briefly examined.

3.1 Bluetooth

Bluetooth is a radio standard (also known as IEEE 802.15.1) and a short range communications protocol designed mainly for mobile and handheld devices. Data transmission is based on spread spectrum and utilizes frequency-hopping technique. The Bluetooth technology has been available almost a decade and has been adopted widely, especially among mobile phone industry.

3.1.1 Background

Development of Bluetooth began in 1994 by a telecommunications equipment manufacturer Ericsson, and first publicly available specification appeared in 1999. Today, Bluetooth is developed by *Bluetooth SIG*, founded in 1998. The group consists of 9000 member companies and is responsible for developing Bluetooth standards and licensing Bluetooth technologies and trademarks.

Major Bluetooth versions and their main additions and modifications to previous versions are listed in Table 3.1 [7].

3.1.2 Implementation and Diffusion Levels

There are far over a billion Bluetooth devices in the world today [8], and even there exists some controversy whether Bluetooth will be capable of handling future demands placed on the wireless communication, no signs of saturation is yet seen; new Bluetooth chips are being shipped at a rate of 10 million per week.

Of the Nokia phones released after 2004 and still actively sold in Finland, over 80%[1] have support for Bluetooth, and almost 85%[2] of the Nokia's future releases

Rev	Date	Additions including
v2.1 + EDR	July 2007	Encryption Pause and Resume, Erroneous Data Reporting, Extended Inquiry Response, Secure Simple Pairing, Security Mode 4
v2.0 + EDR	October 2004	Enhanced Data Rate (EDR)
v1.2	November 2003	Faster Connection, Adaptive frequency hopping, Extended SCO Links, Enhanced error detection, flow control, synchronization capability and flow specification
v1.1	February 2001	Error fixes
v1.0B	December 1999	Interoperability with WAP, Test Control Interface, Bluetooth Audio, Baseband Timers and Optional Packing Scheme.
v1.0a	July 1999	First public release

Table 3.1: Major Bluetooth versions.

will support Bluetooth as of August 2007 [34].

Bluetooth is used, apart from mobile phones, for example, in headsets, printers, laptops, hands-free devices, input devices, watches, clock-radios, televisions, and gaming consoles.

Current version (2.1) of Bluetooth was released in July 2007 but has not yet been widely adopted. Version 2.0 was released in 2004 and is extensively used today. 2.0 is the mainly used version in the InSitu system as well. Next version (3.0) of Bluetooth will include, for example, support for ultra-wideband radio technology, enabling data transfer rates up to 480 Mbit/s.

3.1.3 Technology

Two Bluetooth devices can communicate with each other whenever they are within transmission range. Three levels of transmission power are defined [5, 31], and they are presented in Table 3.2. Each Bluetooth radio device belongs to one of these classes.

The range varies depending on, for example, antenna and transmission path attenuations; estimates of the maximum range vary greatly. Hongfeng Wang [44] uses A. Kamerman's indoor propagation model and estimates that especially in open environments Bluetooth can reach as far as 30 meters with class 3, 50 meters with class 2, and 300 meters with class 1. Michael Hayoz [20] on the other hand presents

	Maximum Permitted Power (mW)	**Maximum Permitted Power (dBm)**	**Approximated range**
Class 1	100	20	50-300
Class 2	2.5	4	10-50
Class 3	1	0	0.1-30

Table 3.2: Three classes of transmission power levels.

typical ranges of 10, 20, and 100 meters.

Bluetooth operates on 2.400–2.4835 GHz frequencies. The actual radio channel frequencies f_k are defined in the Bluetooth specification [5, 29] by equation

$$f_k = 2402\ \mathtt{MHz} + \ k, k = 0, ..., 78. \tag{3.1}$$

Bluetooth communication is based on four physical channels. Two of them are reserved for actual data transfer and the other two for connection establishment. The physical channel is partly characterized by pseudo-random hopping between the frequencies f_k. The hopping is defined by the physical address and clock offset of the host device. One data transfer channel, BPPC, utilizes all of the 79 frequencies defined, whereas the other, APPC, can skip, for example, frequencies with high load. APPC is, however, guaranteed to use at least 20 of the frequencies f_k [5, 70–75]. Frequency hopping density in data transfer channels is 1600 hops per second.

During communication the Bluetooth devices are identified by a 48-bit device address. A typical representation of Bluetooth address is seen in Table 3.3.

`00:0A:3A:51:72:C1`

Table 3.3: An example of a Bluetooth address.

A Bluetooth address is usually displayed in a user friendly form, that is, as an arbitrary name defined by user. The user friendly name is sent to another device as soon as its presence is known. To form a connection, there is no need to know the address of the remote device due to nature of the Bluetooth technology, but if the address is available, it can be used to speed up the connection establishment. This issue and its effects on user interaction will briefly be revisited in Section 5.3; examination of connection establishment in general will be performed in Section 3.1.4.

3.1.4 Connection Establishment

There are two distinct phases in connection establishment of Bluetooth devices: inquiry and paging [4, 51–53]. During inquiry, a device looks for discoverable Bluetooth devices within range. This must usually be performed prior to paging, during which the actual connection is established. A separate physical channel is reserved for both inquiry and paging procedures. Due to nature of the channels, the specification [5, 70–75] defines a hopping density of 3200 hops per second. During inquiry, the inquiring device actively sends inquiry messages. The devices that have been set to discoverable mode will respond with inquiry response using the same physical channel.

Inquiring device iterates through all inquiry channel frequencies; the device sends inquiry message to each channel and listens for responses. Devices share knowledge of inquiry access codes, which can be used to restrict inquiry to certain types of devices. [4, 37–38] Two of the access codes are GIAC and LIAC, General Inquiry Access Code and Limited Inquiry Access Code, respectively. GIAG is the default access code used during inquiry, and LIAC is used for temporary inquiries, for example, when devices are planning to be visible for a limited amount of time. The InSitu utilizes the existence of multiple different access codes in different connection establishment schemes; these are examined in detail in Chapter 5.

A device can perform paging procedure when the address of target device is known; typically, it is the inquiry that provides address and clock of the target device. The paging device iterates through frequencies of physical channel reserved for paging, it sends page request to them, and listens for possible responses. The access code of paging channel is derived from the address of the paging device. The paging device can utilize knowledge of the clock of the device being paged to speed up the synchronization and thereby the connection establishment.

A connection is established between two devices when paged device has sent page response. The devices share a common physical channel and have synchronized clocks. The devices are now, by definition [4, 51–53], part of same piconet and can perform data transfer. At this point, the paging device is able to perform a search among the services provided by the paged device.

Service Discovery Protocol is used to search services from paged device. Services contain attributes, which can be used to identify them. Typical attributes include `ServiceID`, `ServiceName`, and `ServiceDescription`. Paging device performs search among paged device's services and after finding appropriate, usually based on `ServiceID` attribute, opens an actual software level connection to the service.

3.1.5 Communication

Of the two physical channels defined for data transfer, BPPC is the default. Data transfer channels are always associated with specific piconet, thus when devices share a common physical channel, they belong to same piconet. A piconet consists of one master and one to seven slave devices, and therefore, the maximum number of devices in a piconet is restricted to

$$d_{pico} = 8. \tag{3.2}$$

Existence of the piconets is a determinative feature of Bluetooth communication, especially when building large Bluetooth networks.

If expressed in more detail, Bluetooth physical channels are characterized by combination of pseudo-random hopping sequences between frequencies f_k, slot timing of the transmissions, the access code, and packet header encoding. Frequency hopping is used to change frequency periodically to, for example, reduce the effects of interference. [5, 69]

Phase in the pseudo-random hopping sequence is defined by the Bluetooth clock. The physical channels are divided into time slots whose length depends on the type of the channel. When a piconet is formed, the clock of the master device is communicated to the slave devices. The master clock is used for all timing and synchronizing activities.

It is possible that multiple Bluetooth devices communicate using same physical channel, which may lead to *physical channel collision*. To reduce the effects of this event, each transmission on a physical channel begins with an access code that can be used to identify the channel. This channel code is a property of the physical channel and is present at every packet transmitted. [5, 69] It is, however, discovered that multiple piconets do interfere with each other, and this interference is the is key factor limiting the throughput of the network [15].

Several papers have addressed the issue of interference (see for example [21]), and solutions have, indeed, been proposed. Cordeiro and Agrawal present a dynamic packet segmentation algorithm to resolve this matter in [14] and [13]. It is, however, not clear — without measurement in this particular context — at which point this interference prevents the InSitu system from functioning properly[3].

3.2 WLAN

A wireless LAN or WLAN is a wireless local area network, which is the linking of two or more computers without network cables. WLAN utilizes spread-spectrum modulation technology based on radio waves and operates, in most cases — like Bluetooth — on 2.4–2.5 GHz frequencies [41].

3.2.1 Background

Foundation of spread-spectrum wireless networks was laid in 1970s at the university of Hawaii when ALOHANET, the first low-cost computer communication network, was developed [1].

First IEEE Workshop on wireless local area networks was held in May 1991 and subsequent workshops in 1996 and 2001. First version of the IEEE 802.11 standard series was released in 1997.

While wireless networks were formerly used mainly as substitute for cables, in places where cables were not practical, e.g., due to their high cost, the WLANs are nowadays widely used in parallel with LANs for their convenience. But, though many consumer laptops contain a wireless network card, for mobile phones the WLAN is not yet widely available.

3.2.2 Technology

Since WLAN and Bluetooth operate on the same frequencies, they interfere with each other [28] and, perhaps counterintuitively, *WLAN throughput* is the property that degrades. Presented solutions include *Bluetooth Interference Avoidance Scheduling* and *WLAN Rate Scaling* [19], which take into account the existence of the other network and adjust their operation accordingly.

From the InSitu system's point of view, a possibility to run a client software on a personal computer or a laptop is not highly relevant[4]; and, indeed, it might not be especially interesting, that is, any regular TCP/IP connection is sufficient for communication. Nevertheless, while this alternative may yet not be suitable to mass lectures, it can be used for testing purposes or for questioning, for example, in a computer classroom. Another interesting option will be WLAN support that is beginning to appear in mobile phones.

The question of throughput with 200 devices remains also within WLAN. A new standard (IEEE 802.11n) is under development and plans include enhancing WLAN throughput — in theory — up to 600 Mbps [22]. The new standard is scheduled to

be out in September 2008 [23].

As mentioned, WLAN utilizes TCP/IP, which enables client to connect directly to server. After the connection is established, the server software handles devices connected via TCP/IP in the same way as devices connected via Bluetooth.

3.3 GPRS

General Packet Radio Service is a mobile data service and a packet switched data link layer protocol accessible to users of Global System for Mobile Communications (GSM).

3.3.1 Background

In 1999 and 2000, network operators placed different trials and commercial contracts for GPRS infrastructure with the incorporation of GPRS infrastructure into GSM networks. In the summer of 2000, the first trial GPRS services became available. In 2001, the basic GPRS capable terminals began to be available in commercial quantities, and network operators launched GPRS services commercially. [43]

Today, GPRS techniques are widely used, and 3rd generation of mobile networks (3G) keep expanding the capabilities of GSM-based data transfer. From the InSitu system's point of view, many otherwise suitable mobile device models (that is, having Java support), do not have support for either Bluetooth, or its programming interface. Among these mobile device models, GPRS is, however, quite common, and even if GPRS is considered as legacy support (again from the system's point of view), it is an option that has been rather straightforward to implement and, therefore, an easy way to expand device support.

3.3.2 Technology

The mobile devices that provide possibility to communicate via GPRS use TCP/IP, which, from the server's standpoint, is similar to WLAN; the client devices connect directly to the server.

Unlike short range radio, WLAN or Bluetooth, GPRS usually requires fee for data transfer. The IMPv2 specification (see Appendix C) defines that communication is based on UTF-16 encoded characters, which signifies 2 bytes for each character. GPRS traffic is based on TCP/IP, which implies header data of 28 [42] + 24 [25] bytes per packet. A typical IMPv2 packet vary in length between 11 and 150 char-

acters, yielding typical packet length of 22 to 300 bytes. Length of typical TCP/IP packet, therefore, varies between 74 and 352 bytes.

The greatest impact on the data transfer have the keep alive packets, which are sent to ensure that connection has not been lost. Keep alive packet requires 74 bytes of data transfer. Default value for keep alive interval is 2.0 seconds, but this should be configured to a larger value if network has GPRS devices. The client informs the server of its connection method in the login response. This information can be used to configure keep alive interval from the server side per device basis minimizing the data transfer to GPRS devices. Client software is, however, not able to distinguish between GPRS and WLAN connection, which results also in longer keep alive interval for WLAN clients.

Notes

[1]A sample of 89 phone models.

[2]A sample of 13 phone models.

[3]The term *properly* depends naturally on the requirements, which should be expressed in detail. This is not, however, done in this thesis since measurements of required magnitude cannot be performed.

[4]Some users do use a laptop during lectures, and while this provides a more useable approach, the option is — and probably will remain — in margin, at least in near future. On the other hand, in the long run, it is likely that the importance of WLAN will increase.

4 Implementation

I can call spirits from the vasty deep.
Why so can I, or so can any man; but will they come when you do call for them?

— Shakespeare, King Henry IV, Part I [18]

The modern magic, like the old, has its boastful practitioners: "I can write programs that control air traffic, intercept ballistic missiles, reconcile bank accounts, control production lines." To which the answer comes, "So can I, and so can any man, but do they work when you do write them?"

— Frederick P. Brooks [18]

This chapter will provide a view to current implementation of the system prototype. Section 4.1 presents some of the relevant functionality, and Section 4.2 examines the essential non-functional requirements for the system. Section 4.3 examines system at the architectural level, and Section 4.4 introduces the communication protocols used between different parts of the system.

4.1 Main Functionality

This section presents relevant requirements, especially for connection establishment and communication.

4.1.1 General

Typical functionalities are related to asking questions and analyzing the results. This section deals with major system features that are not related to connection and communication *per se* but bear some implications to the resulting design.

Analyzing. The system should be able to record and store data gathered during sessions and enable analyzing this data, preferably over several years.

Interoperability with other systems. The system should be able to acquire existing and available necessary data from other systems, for example, to minimize manual work in user information management.

Querying. The system should be able query users with different types of questions, ranging from multiple choice questions to free text answers, and from instant[1] questions to continuous ones.

Real time observing. Lecturer should be able to observe and interpret the answers as they arrive to system and react to them in a way he or she finds suitable.

Presentation. The lecturer should be able to present the overall results to students after the question has been asked.

4.1.2 Connection Establishment

This section examines requirements directly related to connection establishment.

Simultaneous connecting. Multiple clients should be able to perform connection procedure simultaneously, that is, initiate connection once and wait for successful establishment, even if there are other connections pending from other users.

Adding and removing clients in arbitrary times. System should be able to add and remove clients during session freely without affecting other clients.

Reconnecting. Clients should be able to reconnect to the system during session in case of disconnection.

4.1.3 Communication

This section presents requirements related to communication between server and clients.

Authorization. The system should be able to identify and require authorization of users, if the lecturer chooses to demand this.

Detection of communication errors. The system should be able to detect communication errors and react to them in an appropriate way[2].

4.2 Non-Functional Requirements

This section presents the major non-functional requirements that have implications on the overall design of the system.

Heterogeneous environment. The server should be able to operate in heterogeneous environment in both client and server side, that is, on multiple devices, on varying operating systems, and different physical deployment configurations.

Usable with prior knowledge of client device only. Users, with prior knowledge only of their mobile device, should be able to use client software after a single

instruction lection.

Scalability. The system should scale up to 200 simultaneous users.

Worst case performance. 200 users should be able to form a network in less than 15 minutes[3].

Utilizing existing infrastructure. The system should utilize existing infrastructure to prevent need for large hardware acquisitions.

Maintainability. Architecture at package level, especially in Teacher's App, should not contain cyclic dependencies between the elements.

4.3 System Architecture

InSitu system is essentially a client-server application where the server software is operated by lecturer, and the clients are used by students attending the lecture. Since there exists no free Bluetooth interface for Microsoft Windows operating system, and to ensure portability of the server software, the part of the server handling the Bluetooth connections has been detached to a small Linux program.

4.3.1 Overall

An example configuration of the overall system architecture can be seen in Figure 4.1. Notation for the figures in this chapter (except for the informal 4.1) is presented in Appendix D. The system consists of the following pieces of software:

- server (*InSitu Teacher's App*),
- Bluetooth router (*InSitu Router*),
- projector for results (*InSitu Projector*),
- mobile device clients (*InSitu Leaf*), and
- PC clients (*InSitu PCLeaf*).

As with the hardware requirements, one design objective has been ability to minimize need for hardware acquisitions. Most of the required infrastructure should be available in an average auditorium. The system requires the following hardware:

- USB hub(s)[4],
- Bluetooth modules,

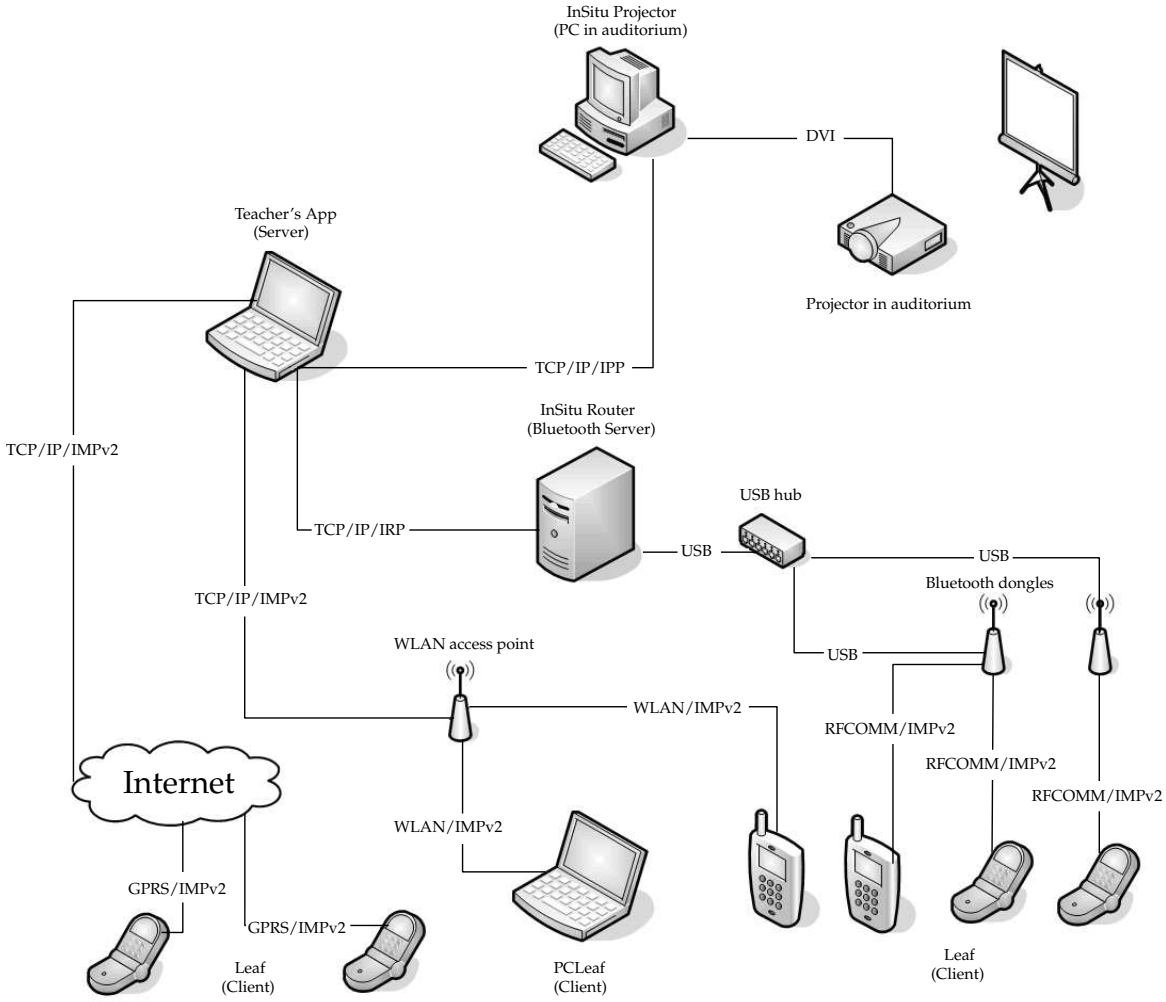

Figure 4.1: Overview of InSitu system.

- Linux PC for Router,
- PC for Teacher's App,
- PC for Projector[5],
- video projector, and
- WLAN access points[6].

The server software communicates with the router with LAN-connection, and the router communicates with the clients via Bluetooth. Due to Bluetooth technology, each Bluetooth antenna[7] can be connected to seven other Bluetooth devices; thus a number of antennas is required.

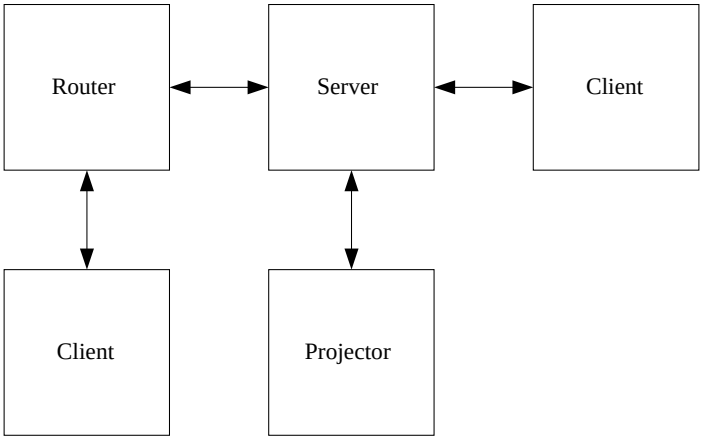

Figure 4.2: Logical structure of InSitu system.

Formally, the system consists of five logical elements, which communicate with each other as presented in Figure 4.2. The router is responsible for managing Bluetooth connections and communicates this data to the server. The server, on the other hand, communicates with clients directly or through the router[8]. The server directs operation of the projector software, which is used to display results using a video projector.

Server, Router, and Projector are logically different elements, but can also run in a single machine. Figure 4.3 presents the default physical layout for server elements. This has some usability implications, such as that InSitu Router must run on Linux, and a video projector typically[9] presents the same information that is on monitor, which may not be desirable from the lecturer's point of view; that is, lecturer may want to examine results or answers of the questions without students seeing them. This requirement for two interfaces was the primary reason for creating two different programs.

Depending on the hardware availability and OS options, different configurations can be used. Each of the three pieces of software can either run on an individual PC or on the same machine.

4.3.2 Teacher's App

Teacher's App is responsible for administering the server system as a whole. It keeps track of the system state and takes care of handling data. The application operates as a primary interface to the lecturer enabling management of questions, answers and users; as well as querying and analyzing answers.

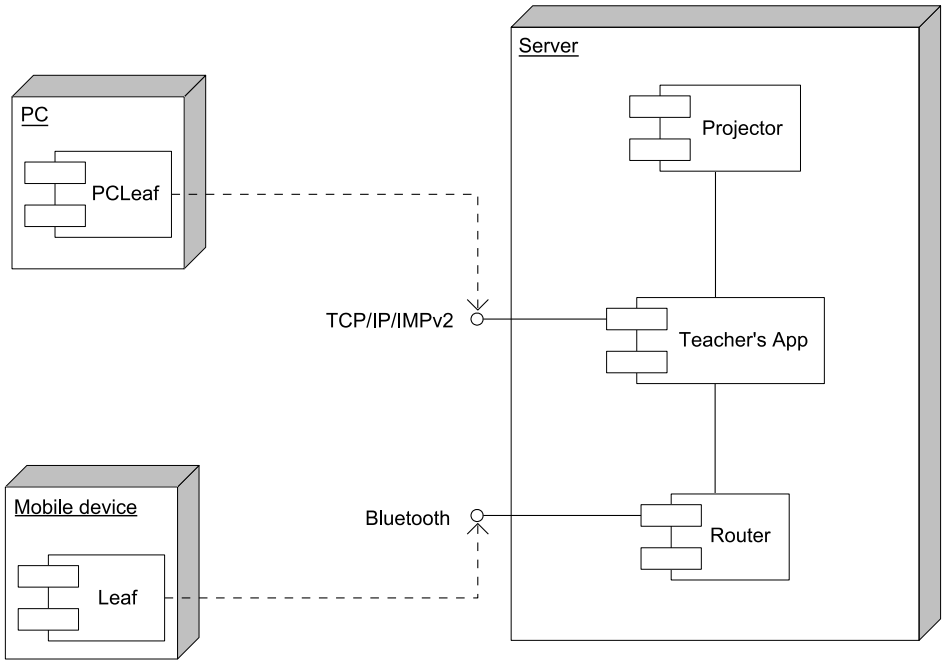

Figure 4.3: Default physical layout of InSitu system.

Logically, the program is divided into four parts, which are presented in Figure 4.4. *Core* administers the operations of the system and coordinates communication among the other parts of the system. User interface (*UI*) presents the graphical interface to user and translates user's actions and wraps them into *Commands*, and related events, for other parts of the system to process. *Data* contains storage for runtime data and program state and defines file structures and communication protocols. *Network* element provides application-specific services for accessing network.

The Teacher's App has a typical design of a multithreaded server. The process graph for Teacher's App is presented in Figure 4.5. The runtime structure of Teacher's App consists of main thread (*Core Thread*), user interface thread (*Java AWT Thread*) and several network threads (*Listener Thread n*). Main thread is responsible for managing data, coordinating user interface thread to prevent modification of data during update, creation (and deletion) of network threads and sending data between parts of the system. User interface thread operates on the *UI* element seen in Figure 4.4, main thread on *Core* and *Data* parts, and network threads operate on *Core* part.

Network threads push incoming data to queue, which is periodically checked by main thread. User interface thread is responsible for updating the interface, whenever requested by main thread, and initializing commands and events based on

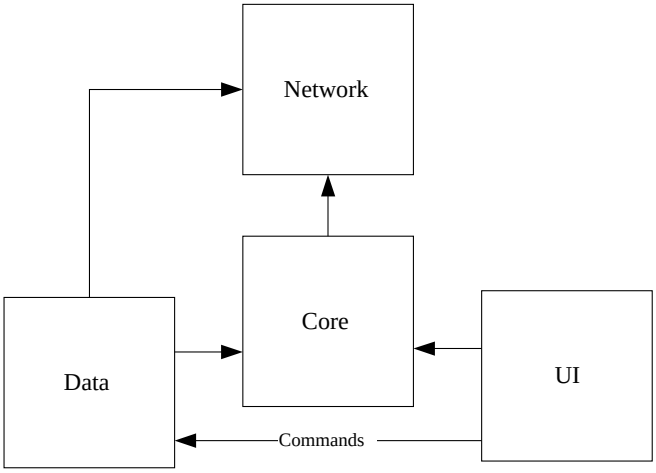

Figure 4.4: Logical structure of Teacher's App.

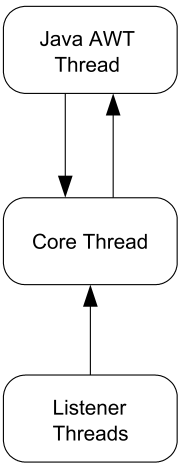

Figure 4.5: Processes in Teacher's App.

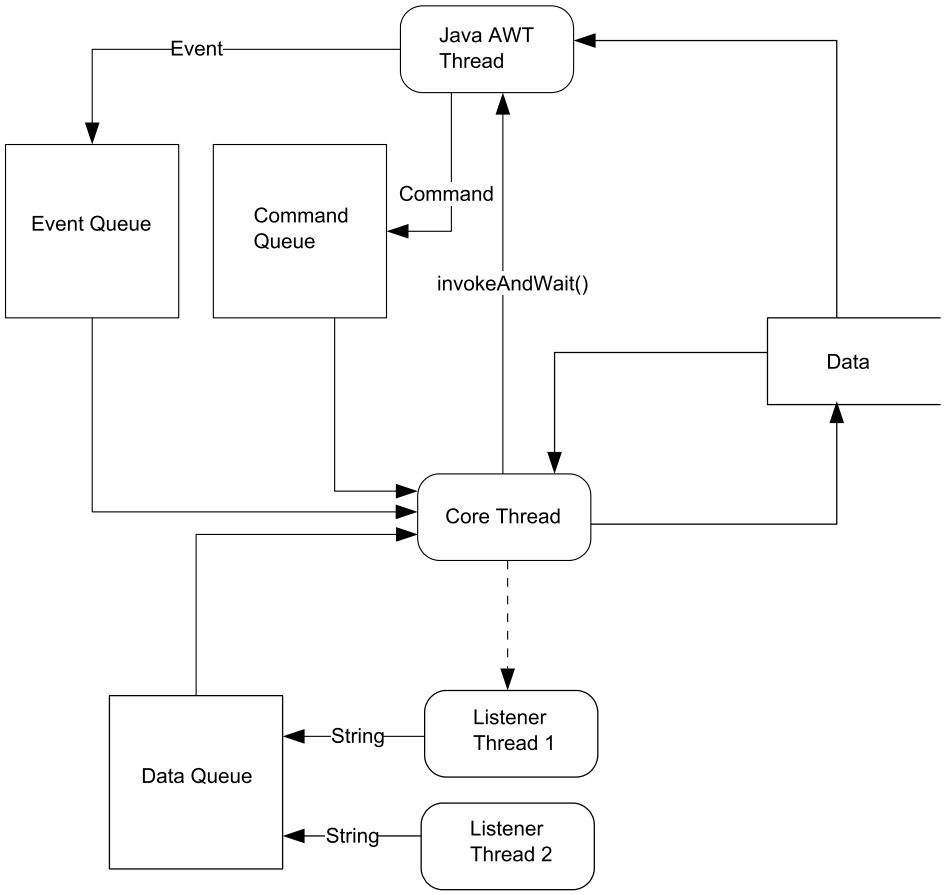

Figure 4.6: Processes and their communication in Teacher's App.

user's actions. These commands and events are pushed to appropriate queues as seen in Figure 4.6. Main thread issues a special update request (*InvokeAndWait()*) and waits for its completion whenever user interface thread needs to access *Data* to update the user interface. Only the main thread is allowed to modify the runtime data. If the AWT thread detects a need to update, a request to do so can be sent to main thread via *Event queue*.

The multithreaded approach to networking aims to ensure that the server software is able to meet the performance requirements of 200 simultaneous users. Differentiating the user interface thread from processing thread ensures that the user interface remains operational during lengthy operations.

InSitu Teacher's App is implemented with Java SE 6 and utilizes the following open source libraries and frameworks:

- InfoNode Docking Windows 1.5.0 [24],
- Joda Time 1.4 [29],

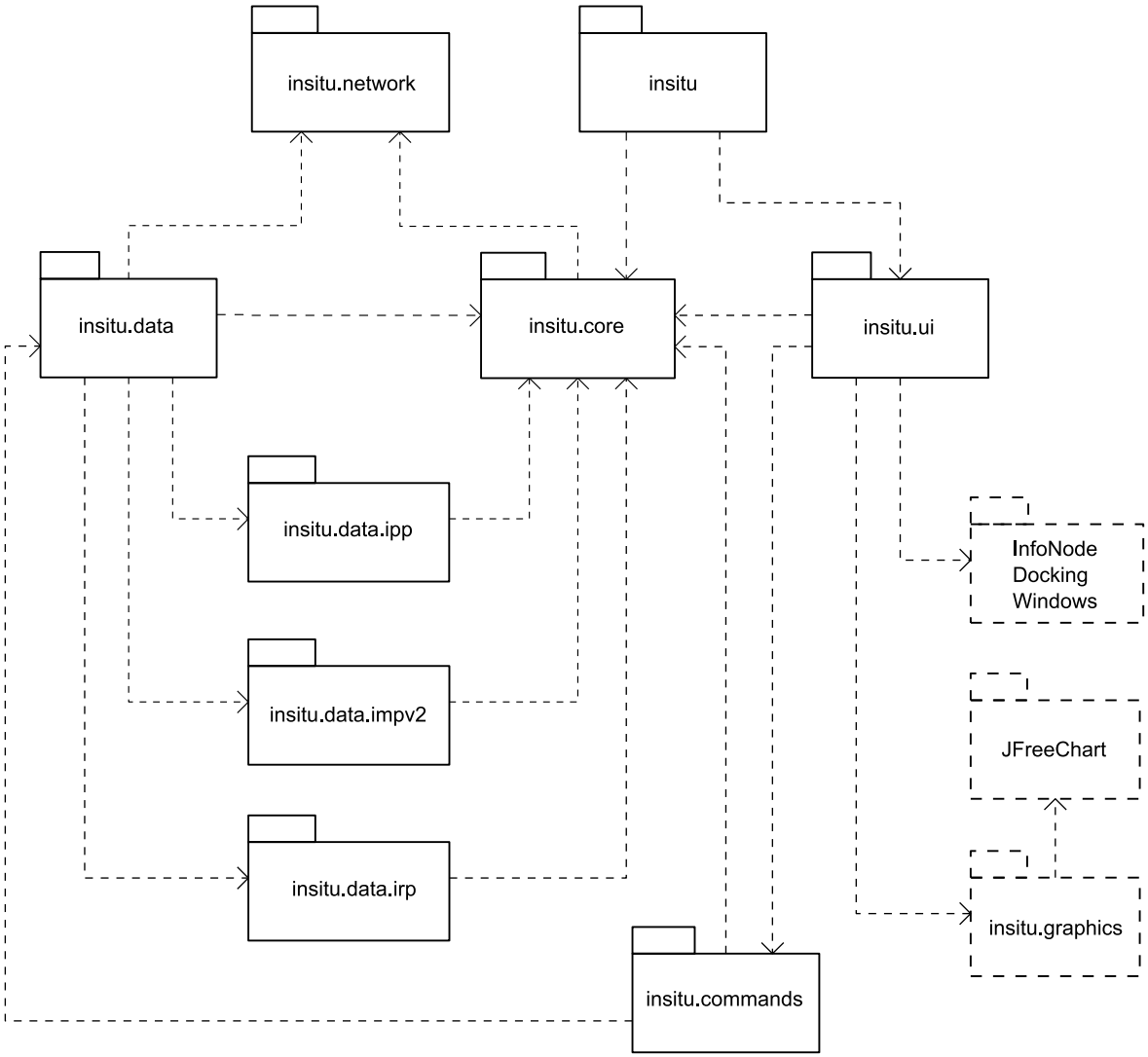

Figure 4.7: Development structure of Teacher's App

- JFreeChart 1.0.3 [27], and
- JCommon 1.0.8 [26].

Figure 4.4 presented four main parts, which map to the development structure seen in Figure 4.7 as follows: the core element is basically the *insitu.core* package, user interface the *insitu.ui* package, and network is handled by *insitu.network*. *Data* element is divided to actual data and to different protocols. Commands are bundled to *insitu.commands*. User interface utilizes two external libraries (JFreeChart via insitu.graphics and Docking Windows), whereas JCommon (omitted from the figure) is required by the JFreeChart. Joda Time library (also omitted) is used throughout the program wherever calendar and time calculations are needed[10]. The *insitu.graphs*

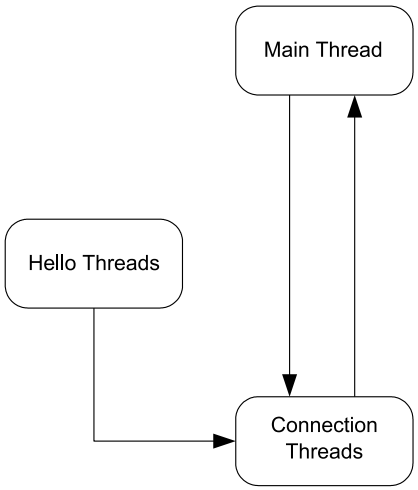

Figure 4.8: Processes in Router.

is implemented as an external library to provide the required functionality for drawing graphs for all the necessary parts of the system; most notable part being naturally the projector. It is notable that no cyclic dependencies are present in the Figure 4.7.

4.3.3 Router

During normal communication, Router essentially maps Bluetooth connections to virtual device identifications. The server sends a packet that contains an identification number and the actual packet. The identification number is provided by Router during connection establishment. Similarly, the incoming packet from Bluetooth device is wrapped to another packet and sent to the server along identification.

During connection establishment, however, the router operates in a more sophisticated manner; details, naturally, depending on the connection scheme used. Figure 4.8 presents typical processes and their interaction in router during connection establishment. A number of connection threads is created. The creator and creation time depending on the context and the particular connection establishment procedure. The number of hello threads depends typically on the number of *notification modules*[11] — if any — present in the system.

In one of these schemes (see Section 5.1.3 for more details), a connection to a visible notification module is made by a client device, to which in turn another module connects from the server side. To enable as many simultaneously connection establishment procedures as possible, each of these notification modules is operated under a distinct *Hello Thread*, and a new *Connection Establishment Thread* is spawned

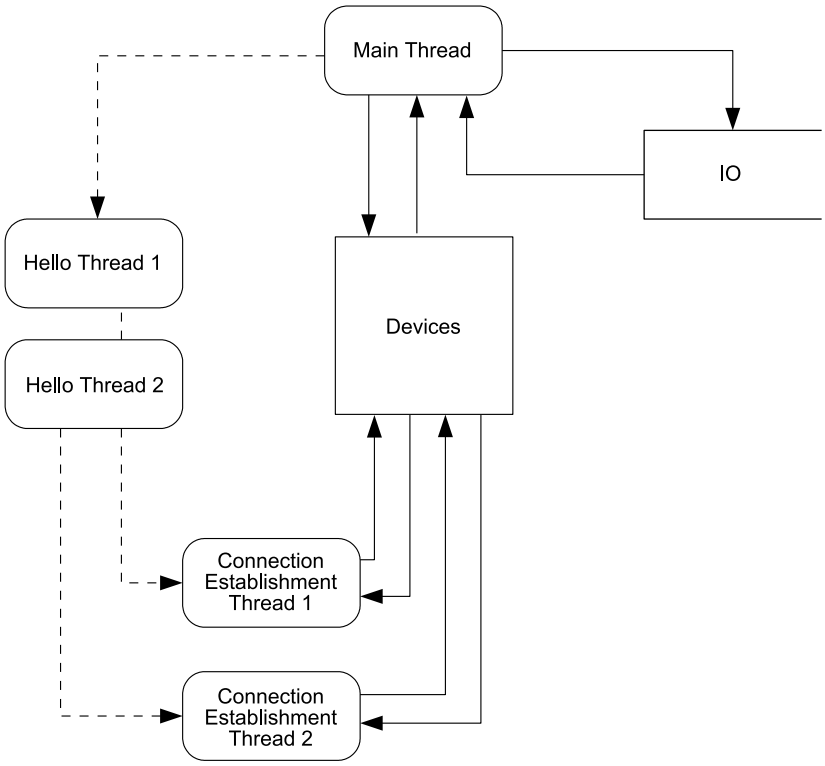

Figure 4.9: Processes and their communication in Router.

for each connection establishment procedure.

Figure 4.9 presents the threads operating in one of the connection schemes. *Main Thread* is responsible for administering the client data (*Devices*) and multiplexing the network traffic to server and clients (*IO*). The *Devices* data structure is synchronized using mutexes. Notice that even if several simultaneous connection establishment threads should be able to operate independently, hardware seems to place additional restrictions to their operation. This issue will be examined in detail in Section 5.1.

The router is designed to be as lightweight as possible, partly due to lower level programming required. It consists only of two major modules presented in Figure 4.10. *Router* module handles network IO, manages runtime data structures, and program state as well as parses and interprets packets; *BTManager* administers the Bluetooth connection establishment procedures and notification modules. In general, all operations requiring Bluetooth functions from *Bluetooth* library of Linux kernel should go to *BTManager* module. Due to nature of the program, both modules put Linux threading utilities *PThread* to use.

The router requires currently Linux Kernel version 2.6.20[12] due to built-in support for Bluetooth.

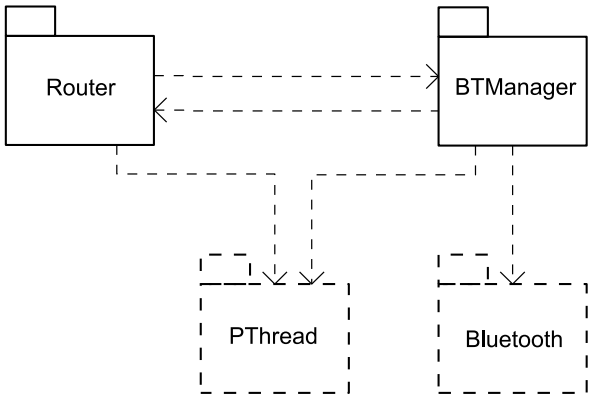

Figure 4.10: Development structure of Router.

4.3.4 Leaf

InSitu Leaf is the main client software in the system. It is implemented in Java ME, supports both Bluetooth and GPRS connections, and operates basically in any mobile phone with support for the following Java Specification Requests:

- Connected Limited Device Configuration (CLDC) 1.0 for Java ME (JSR 30),
- Java APIs for Bluetooth (JSR 82), and
- Mobile Information Device Profile (MIDP) 2.0 for Java ME (JSR 118).

The Leaf is divided to a core package *leaf*, custom-made user interface components *leaf.components*, and a Bluetooth management module *leaf.bluetooth* as rendered in Figure 4.11[13]. Leaf can be compiled without the *insitu.bluetooth* package to ensure compatibility with phones without JSR 82 (that is, phones with GPRS and Java). A client side package of IMP, *insitu.impv2*, is an external library and shared by both Leaf and PCLeaf (see Section 4.3.5). The *insitu.impv2* can, therefore, be compiled to either desktop or mobile Java.

4.3.5 Others

Other elements in the system include *InSitu PCLeaf* and *InSitu Projector*. PCLeaf is a PC equivalent of InSitu Leaf; and is essentially a rather straightforward Java client application which connects directly to the server via TCP/IP.

Projector is a thin client operated by the server for displaying a portion of the server's user interface to audience. The projector is not meant to be used locally and, therefore, requires no interactive user interface. The processes in the projector

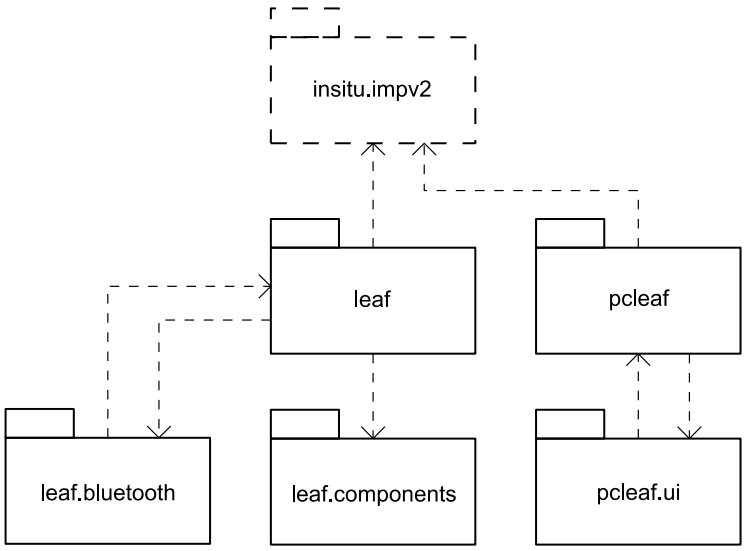

Figure 4.11: Development structure of Leaf and PCLeaf.

are rendered in Figure 4.12 and they present quite a simple structure. *Listener Thread* pumps network traffic to *Core Thread*, which in turn adjusts the operation and program state accordingly and further translates the messages to user interface thread *Java AWT Thread*, which typically renders a graph of some sort.

The Projector consists of *projector.core*, which is responsible of processing and storing data and program state; *projector.ui*, which manages the user interface; and *projector.network*, which handles network traffic and protocol parsing. The package *insitu.graphs* is an external library shared by the Projector and the Teacher's App. It contains the functionality required to render the graphs used in the system. The elements are presented in Figure 4.13.

In the future, usability of the system is to be enhanced by introducing a small program for operating the server remotely using, for example, lecturer's mobile phone. Current sketch is known as *InSitu Remote*.

4.4 System Protocols

There exist protocols for low-level communication, connection establishment, and service discovery in Bluetooth. There are, however, no application level protocols available for lecture feedback and questionnaire systems. As to TCP/IP-based communication between different parts of the server, most of it could have been implemented using, for example, remote method invocation of some kind, but essentially, the heterogeneity of the operating environment requires an implementation inde-

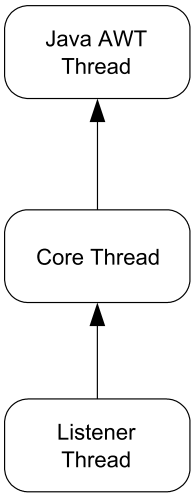

Figure 4.12: Process structure of InSitu Projector.

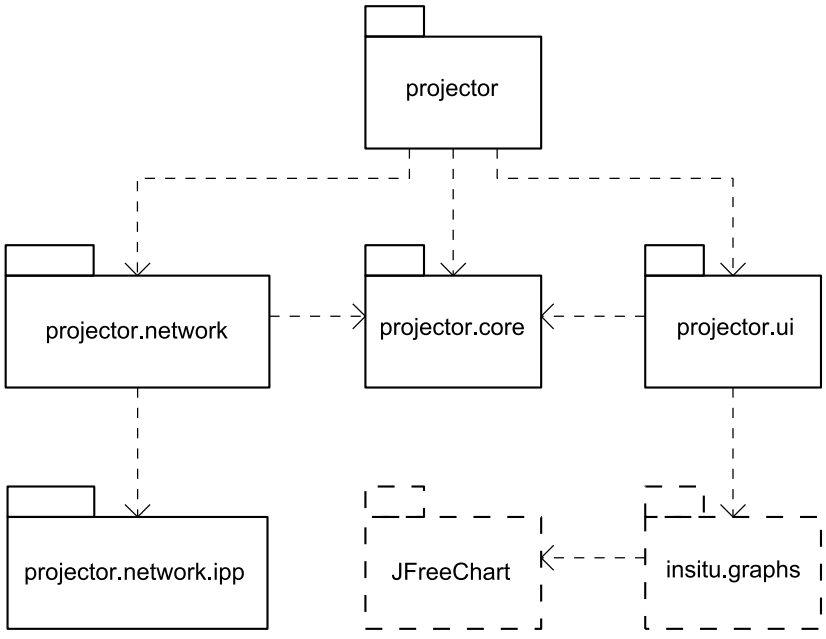

Figure 4.13: Development structure of InSitu Projector.

pendent solution. Thus, a set of text-based protocols were developed.

System architecture, as seen in Figure 4.2, requires four kinds of communication:

- server-router,
- router-client,
- server-client, and
- server-projector.

Communication between the server and the router is based on regular TCP/IP and consists mainly of administering Bluetooth connections and sending and receiving question and answer data. InSitu Router Protocol (IRP) is defined for this purpose. Communication between the server and the clients is done either directly or via router. From the client's point of view the actual route is transparent; both connections utilize InSitu Mobile Protocol (IMPv2) for the communication. Server-projector communication is based on TCP/IP and InSitu Projector Protocol[14] (IPP).

4.4.1 IRP

InSitu Router Protocol is defined for communication between the server and the router. The main purpose of the protocol is to provide transparency by serving as carrier to higher level protocols, such as IMPv2 (see Section 4.4.2). IRP also provides means to communicate the mapping between actual Bluetooth clients and the session-specific identification number.

Full protocol stack for the server-router communication is seen in Table 4.1.

IMPv2[15]
IRP
TCP
IP
Ethernet
100BASE-T

Table 4.1: Full protocol stack for server-router communication.

IRP is a text-based protocol, and it's packets are divided to header section and data section. The header consists of packet length and a packet-specific command. The data section contains the actual data as specified for each specific packet. The packets specified by IRP can be divided into three categories:

- administering clients,
- communicating with clients, and
- administering router.

Category for handling clients contains *User connected*, *User disconnected*, *Connection list*, and *Disconnect user* packets, which notify the server side of newly connected user, report a lost or ended connection, initiate a connection procedure to given addresses, and disconnect a given user respectively. Apart from the *Connection list*, these packets contain the session-specific identification(s) for the particular client(s), which in turn can be used, for example, to send data to — and identify messages from — the client(s). *User connected* packet contains also the Bluetooth address of the connected device.

Packets for sending data to clients and receiving data from them form the second category. This category contains *Send to* and *Received from* packets. Both packets contain the identification number(s) of the client(s) and the actual data received or sent in form of an IMP packet. *Send to* packet may contain more than one identification number, which can be useful when sending same packet to multiple Bluetooth clients.

The third category contains packets for administering the router and consists of *Router settings*, *Disconnect*, and *Keep alive* packets. *Router settings* is sent by the router immediately after a connection is made to it. The packet contains information of the capabilities of the router. *Disconnect* packet is sent by the server, and it indicates that all connections must be closed and the server itself should go idle and wait for a new connection from the server. *Keep alive* is sent periodically by the server to ensure that the connection to the router has not been lost.

Full IRP specification is included in Appendix B.

4.4.2 IMPv2

InSitu Mobile Protocol (IMP) is used for communication between the server and the clients. IMP was used in first prototype[16], and current system utilizes the second version, IMPv2, which includes, for example, more details about questions, enabling more comprehensive usage. The router processes the IRP packets and forwards the contained IMPv2 packets to mobile devices using L2CAP/RFCOMM. IRP is not dependent of IMPv2, but the router has to be able to parse IMPv2.

The full protocol stack for communication between the clients and the router after the connection is established is shown in Table 4.2 [6, 181]. Lower parts of

the stack are Radio, Link Controller, and Link Manager Protocol and these three are sometimes grouped into a subsystem known as the *Bluetooth controller* [4, 21]. Logical Link and Control Adaptation Protocol (L2CAP) interfaces higher level protocols such as Radio Frequency Communication (RFCOMM) and Service Discovery Protocol (SDP).

IMPv2
RFCOMM
L2CAP
LMP
LC
Radio

Table 4.2: Full protocol stack for router-client communication.

It is also possible for a client to connect directly to the server. IMPv2 has to be transported via TCP/IP, since the server has no means to accept Bluetooth connections. All client devices regardless of the platform or network type use IMPv2 for communication.

The full protocol stack for server-client communication is presented in Table 4.3. There is no need to use other protocols like SDP, and connection establishment and communication is straightforward.

IMPv2
TCP
IP
Ethernet
100BASE-T[17]

Table 4.3: Full protocol stack for direct server-client communication.

Like IRP, IMPv2 is a text-based protocol whose packets are divided to header section and data section. The header consists of packet length and a packet-specific command. The packets specified can be divided into five categories:

- session handling,
- messaging,

- querying,
- responding, and
- administering client.

Session handling contains *Login request*, *Login response*, *Login reject*, and *Login success* packets. These packets are sent after connection has been established. The packets contain information related to the authentication process such as allowance of anonymous login in *Login request* and reason for rejection in *Login reject*. *Login response* is sent by the client; the others, by the server. The authentication process itself starts with login request. *Login response* contains, apart from the user identification and password, information about the client software, its connection type, and version.

Messaging contains *Message* packet, which can either be sent by the client or the server. This packet contains essentially a textual message of arbitrary length (see Section 4.4.3 for implementing an informal *subprotocol* with this packet).

Querying consists of different question related packets that are sent by the server. These include *Question select*, *Question order*, *Question input*, and *End question*. The packets ask a multiple choice question, ask an ordering question, ask an arbitrary input, and end a specified question, respectively. The packets contain a session-specific number identifying the question the packet is referring to.

Multiple choice questions require user to select one or more[18] options from given list. Ordering questions require user to put given elements, or a subset of them, to proper order, and arbitrary input can contain any textual answer. Questions can either be instant or continuous[1].

Responding contains *Answer* packet, which is sent by the client as a response to a question. It contains the answer and the session-specific number identifying the question in question.

Administering client consists of *Disconnect* and *Keep alive* packets. The former is used to notify the client that it should disconnect from the server for whatever reason, and the latter to ensure that the connection to client has not been lost. Both packets are sent by the server.

Full IMPv2 specification is included in Appendix C.

4.4.3 Others

Other protocols include the forthcoming InSitu Projector Protocol (IPP) and possible protocols, for example, for *InSitu Remote* (see Section 4.3.5). Full protocol stack

for server-projector communication is presented in Table 4.4. IPP will most likely include packets, for example, for displaying questions and graphs of answers, in addition to necessary administrative packets.

IPP
TCP
IP
Ethernet
100BASE-T

Table 4.4: Full protocol stack for server-projector communication.

The *Message* packet mentioned in Section 4.4.2 has been used to implement an informal subprotocol for debugging purposes. This protocol violates the independence of IRP and IMPv2, but is not planned to be formally supported at least in its current form; hence the term *informal*. The router interprets the IMPv2 *Message* packets, checks whether header of debugging protocol is in place, and sends its client-specific debugging data to the server by masking it as a traffic from the client itself.

This debugging protocol is a subprotocol, since it is sent as a message of IMPv2 and interpreted only if debugging has been turned on in both the server and the client. The packets in the debugging protocol are, for example, used to communicate clock differences between systems[19] and query recorded timestamps of predefined events during connection establishment. This method is used during the measurement of performance examined in Section 5.4.

Notes

[1]*Instant question* refers to questions that can be answered only once, whereas *continuous questions* can be answered multiple times, to the point where lecturer chooses to end the question.

[2]The system may want to notify user of disconnect or attempt to reconnect silently.

[3]15 minutes for connection establishment may appear to be rather long, but as will be seen in Section 5.4, a performance requirement this broad may be required. 15 minutes is generally speaking acceptable if we take into account the fact that lectures of 200 students are in this context very large and quite infrequent.

[4]USB hubs are most likely required due to large number of Bluetooth modules.

[5]The system does not necessarily require three different PCs, see Figure 4.3 for more details.

[6]WLAN is not necessary for the operation, but it does enable usage of, for example, laptop clients without Bluetooth capabilities.

[7]The terms Bluetooth antenna, Bluetooth module and Bluetooth dongle are used interchangeably in this study.

[8]Hence the name *router*; other appropriate ones might have been *Bluetooth server* or *Bluetooth connection manager*.

[9]Multiple monitors and multiple (different) views can be used, for example, in Mac OS X, but this is not necessarily the case with other operating systems.

[10]Joda Time is preferred over Java Calendar due to better usability (during development) and more predictable performance.

[11]Term *hello point* is also used.

[12]Older versions may also work, but have not been tested. Especially 2.4 series may have problems with newer (2.0) Bluetooth modules.

[13]Names in the figure may not reflect the exact names in the actual up-to-date source code. Some elements are under heavy (re)development and subject to change.

[14]As of August 2007 no formal specification of IPP exists.

[15]IRP packet does not necessarily contain an IMPv2 packet. This is the case when the server talks directly to the router or vice-versa.

[16]First Bluetooth prototype dates back to 2004.

[17]The two bottom layers vary depending on the communication medium. WLAN, for example, may be used in some parts of the communication channel.

[18]The number of choices user can or must select can be specified for each question.

[19]Not to be confused with Bluetooth clock, which is used within the lower levels of Bluetooth radio communication.

5 Bluetooth Connection Establishment

It is common sense to take a method and try it. If it fails, admit in frankly and try another. But above all, try something.

— Franklin D. Roosevelt [37]

In this chapter, different Bluetooth connection establishment schemes are studied. In Section 5.1, a number of connection establishment schemes are introduced, formalized and analyzed. Some notes on the implementation and usability aspects are given in Sections 5.2 and 5.3, respectively. Some of the schemes were measured using methods presented in Section 5.4; results and conclusions are expressed in Section 5.5. Finally, further development guidelines are rendered in Section 5.6.

5.1 Connection Alternatives

Several different connection establishment schemes can be defined. These include

- Inquiry Based Connection (IBC),
- Predefined Address Connection (PAC),
- Notify Module Connection (NMC), and
- Module Discovery Connection (MDC).

Each of these alternatives utilizes Bluetooth connection establishment procedures presented in Section 3.1.4. The approaches vary mainly in usage and in the initiator of different phases. This section discusses the parts of the connection establishment prior to service discovery, which is performed identically in each alternative.

The alternatives are not mutually exclusive; they can be used interchangeably or simultaneously depending on the situation. Different aspects of defined connection establishment schemes are summarized in Table 5.1.

Number of devices affects the overall connection delay and dictates the number of Bluetooth modules required to build the network. Let d_{pico} be the maximum number of devices in single piconet and N_{dev} the overall number of client devices in the system, the number of Bluetooth modules N_m required is defined as

$$N_m = \lceil N_{dev}/(d_{pico} - 1) \rceil. \tag{5.1}$$

Aspect	IBC	PAC	NMC	MDC
Delay	Largest	Smallest	Acceptable	Possibly large
Scalability	For small networks	Can be used in large networks	Can be used in large networks	Possibly for medium-sized
Flexibility	Flexible implementation laborious	Flexible enough	Flexible enough	Flexible enough
Traffic	Additional traffic	No additional traffic	Some additional traffic	Additional traffic
Usability	Simplest	Requires address database	Rather simple to use	Simple to use

Table 5.1: Summary of connection establishment schemes.

5.1.1 Inquiry Based Connection

In Inquiry Based Connection the server first performs an inquiry. Client devices within range of the module are then found and can be connected to. This requires that the client devices are discoverable and run client software as connectable Bluetooth service. According to Siegemund et al. [39] inquiry is, however, a slow process: although, in theory [5, 400], it is possible to perform inquiry up to 255 found devices, one inquiry cannot find all of these. They measured that in 10.24 seconds that specification [5, 165] recommends, only about 20 of 50 available devices were found.

If already found devices are not set to non-discoverable mode during inquiry, for example, by establishing a connection to them, discovering the same devices repeatedly will be a problem. Paging and service discovery, presented in Section 3.1.4, between inquiries cause additional delay and requires additional actions on software level. Furthermore, if minimizing network traffic should be taken into account, then inquiry should be used dynamically on demand rather than continuously. This requires ability to control the inquiry programmatically by server-side users, and results in increased delay in connection establishment.

IBC approach requires both inquiry and paging. Paging is required for each device in the system, but should, in theory, be performed simultaneously by each

Bluetooth module. Assuming that number of devices found with a single inquiry from devices available N_{dev} is N_f, the average overall time T_{ibc} for connecting all devices is

$$T_{ibc} = I_{avg}\frac{N_{dev}}{N_f} + P_{avg}\frac{N_{dev}}{N_m} \tag{5.2}$$

, where I_{avg} and P_{avg} are the average times of inquiry and paging, respectively. T_{ibc} can further be reduced by performing paging to found devices while still inquiring devices that have yet not been found.

Therefore, in theory, as can be seen from equation 5.2, IBC can perform linearly in proportion to number of devices in the network. In practice, however, no assumption about N_f can be made due to finding of same devices. It is defined, for example, in JSR-82 [30] that device's discoverability can be programmatically altered, but the Bluetooth Control Center of the device prevents changing this property in some implementations. This prevents effective implementation of IBC for large networks.

5.1.2 Predefined Address Connection

Predefined Address Connection is based on a database of device addresses. As stated in Section 3.1.4, paging requires knowledge only of the address of the paged device. It is, therefore, quite straightforward to iterate a list of addresses and form a connection to them. Paging takes usually only about one second [44], and as such is notably faster than Inquiry Based Connection. This applies only if device in question is within range of the Bluetooth module, and the software is running and ready to accept connections. If the device is not available, the connection attempt fails by default after 5.12 seconds [5, 385], but timeouts as high as 20.48 seconds were measured by default on BlueZ [9], a Linux Bluetooth implementation.

The Predefined Address Connection scheme consists of multiple consecutive paging procedures. Each module in the network can perform paging procedure simultaneously, and thus *when all the devices are present*, the time T_{pac} required for connecting N_{dev} devices is

$$T_{pac} = P_{avg}\frac{N_{dev}}{N_m}, \tag{5.3}$$

which is, given by equation 5.1, equivalent to

$$T_{pac} = P_{avg}(d_{pico} - 1). \tag{5.4}$$

PAC, therefore, requires essentially a constant time under the assumption that each device (present or not) needs to be connected once, i.e., no disconnects occur during the connection establishment. Formally, PAC can be expressed as in Figure 5.1. The

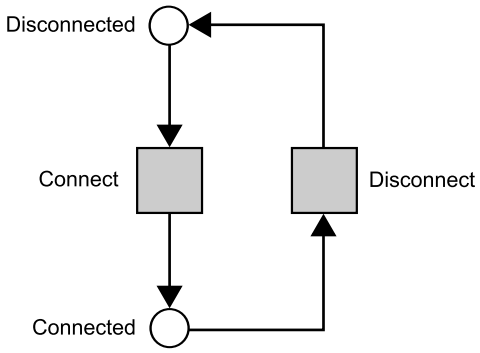

Figure 5.1: Formalization of PAC scheme.

formalization is done using Petri nets [16]. The tokens in the net represent the mobile Bluetooth devices, and there are initially N_{dev} tokens in place *Disconnected*.

Firing rates of *Connect* and *Disconnect* transitions in the formalization are the rates in which the devices are connected to router, and disconnects occur during the connection establishment, respectively. The exact values are context dependent and some results are presented in Section 5.5.

PAC is executed by the server by sending router a request to connect to given Bluetooth addresses. The router initiates connection establishment procedures individually (using distinct threads) for each device as presented in Sections E.1.1 and E.1.2 in the appendix.

5.1.3 Notify Module Connection

Unlike in pure *ad-hoc* networks, the connection establishment can also be based on a known static component of the network. Difference to the preceding schemes is that in this alternative the initiator is the client device. Client users must, therefore, have an address of one or more *notification modules*.

The client device establishes a connection to this notification module as the server does in the Predefined Address Connection scheme. When connection has been established, the server has knowledge of the device, its address and Bluetooth clock. The connection is then immediately disconnected and re-established by the server via some other module. This approach is slower than PAC but requires no knowledge of client devices in advance. First part of the connection establishment takes approximately as long as in PAC, but in addition another is required; though, it may utilize knowledge of the device clock. This scheme is, in theory, both flexible and scalable.

Notify Module Connection requires one paging procedure from the client-side

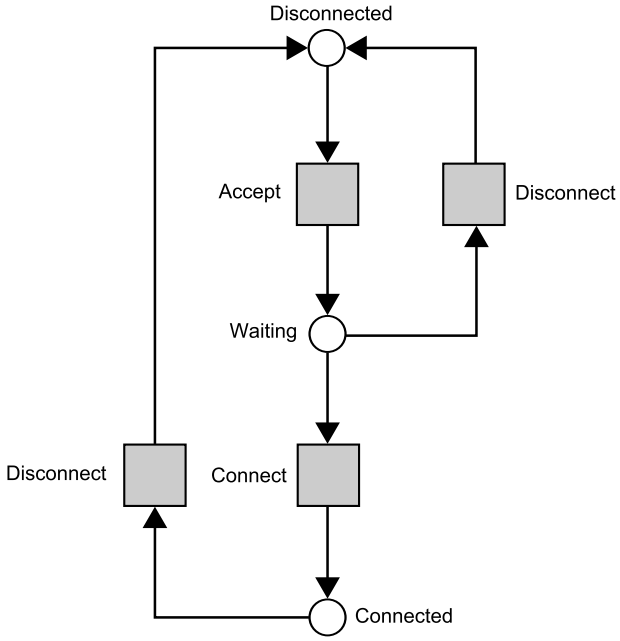

Figure 5.2: Formalization of NMC scheme.

and another from the server-side. This scheme requires at least one notification module and with larger networks possibly multiple modules. The first paging is performed by client devices and as such can be done almost simultaneously. The connections are, however, accepted at a rate based on the number of notification modules N_{hp} and their ability to handle incoming connections. Typically, each Bluetooth module can accept one connection at a time. Similarly, each normal module can execute a single connection establishment procedure to a single remote device at a time, thus the system can perform N_m simultaneous outgoing connection establishment procedures, where N_m is the number of normal modules. Therefore, we state that

$$T_{nmc} = \sum_{i=1}^{N_{dev}} \frac{T_{in_i}}{N_{hp}} + \sum_{i=1}^{N_{dev}} \frac{T_{out_i}}{N_m}, \tag{5.5}$$

where T_{in_i} and T_{out_i} are times required for *accepting the connection from* and *connecting back to* device i, respectively.

From the formalization presented in Figure 5.2 we see that the firing rate of *Connect* is limited by the firing rate of *Accept*, and the throughput of the whole net is limited by the firing rate of *Connect*. There are initially N_{dev} tokens in place *Disconnected*. If we assume approximately constant rate of firing, which is reasonable, we can state that

$$T_{in_i} = T_{in_{i+1}}, T_{out_i} = T_{out_{i+1}} \forall i, 0 < i < N_{dev}. \tag{5.6}$$

Since both *Accept* and *Connect* are performed simultaneously, *whichever of the addends in equation 5.5 is larger, dictates the overall throughput*, and we may determine the optimal number of modules as follows: assume optimal situation by stating that

$$\sum_{i=1}^{N_{dev}} \frac{T_{in_i}}{N_{hp}} = \sum_{i=1}^{N_{dev}} \frac{T_{out_i}}{N_m}, \tag{5.7}$$

which under the assumption 5.6 yields

$$N_{dev} \frac{T_{in}}{N_{hp}} = N_{dev} \frac{T_{out}}{N_m}, \tag{5.8}$$

and, therefore,

$$\frac{T_{out}}{T_{in}} = \frac{N_{hp}}{N_m}, \tag{5.9}$$

under the assumption that the devices are equally distributed to different HPs.

NMC scheme performs linearly in proportion to the number of devices if we assume, again, that each device needs to be connected only once, in other words, no disconnections occur.

The router listens each hello point, as presented in Section E.2.1 in the appendix, and initiates connection establishment immediately for each device accepted. As mentioned, each module can perform a single connection establishment at a time, thus even though there is a separate thread for each active connection procedure, they can be executed at the rate dictated by the number of Bluetooth modules. This scheduling is, in practice, done with mutual exclusion. The exact mechanics are presented in Section E.2.2 in the appendix.

5.1.4 Module Discovery Connection

The fourth scheme is a more user friendly version of Notify Module Connection. Module Discovery Connection is mainly analogous to NMC, but the user does not have to know the address of the notification module; the client device uses inquiry to locate it.

The inquiry phase in this scheme should be less difficult than in Inquiry Based Connection because there is a notably smaller amount of devices to discover. The problem found by Siegemund et al. [39] was that all inquiry responses were not received by the inquiring device due to large amount of responses. This is especially true with devices with different transmission power, such as mobile devices of different generations: transmission from newer devices tends to dampen signal from older ones.

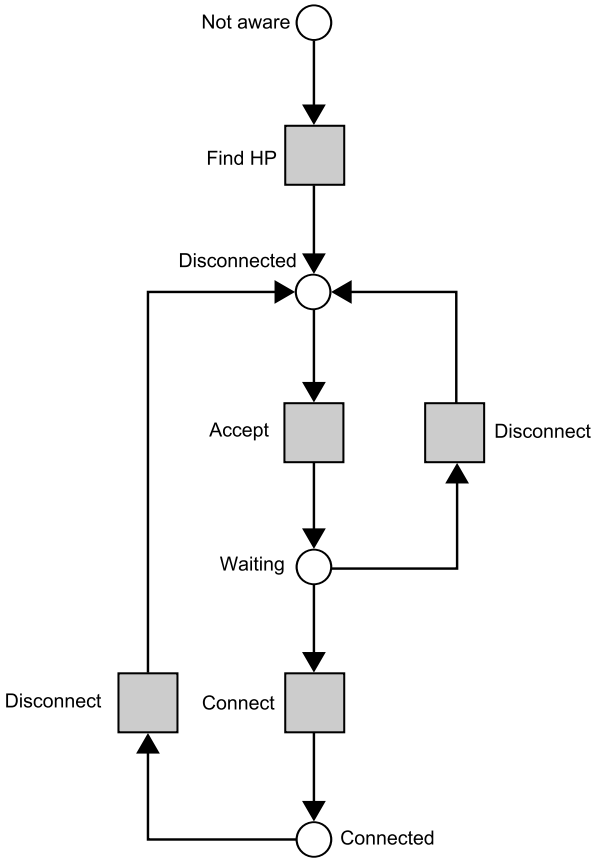

Figure 5.3: Formalization of MDC scheme.

As seen from formalization in Figure 5.3, the Module Discovery Connection scheme is similar to the Notify Module Connection approach but adds an additional inquiry to the process. Therefore,

$$T_{mdc} = I_{avg} + T_{nmc}, \tag{5.10}$$

but since throughput is limited again by the acceptance and connect rates in NMC, the impact of I_{avg} is in practice insignificant. In this scheme there are initially N_{dev} tokens in place *Not aware*.

From the implementation point of view, router operates like in Notify Module Connection, but the notification modules must have a special Inquiry Access Code (IAC) set, to limit the inquiry, and they must be visible to other devices. After the notification module has been found, the process continues as in NMC even if *Disconnects* occur. There is no need to perform another inquiry since client's Bluetooth device cache store the address of the hello point for further use.

5.2 Implementation Considerations

Bluetooth modules typically belong to class 1 of Bluetooth transmission power level (see Table 3.2). Although mobile phones usually belong to class 2, this should still be enough for an open auditorium environment. According to a proof of concept test, two Nokia 6630 mobile phones were able to communicate via Bluetooth from one end to another in a mid-sized auditorium. Radio coverage can further be reinforced with appropriate placement of Bluetooth antennas. This requires different adjustments to different schemes in order to be used efficiently, and therefore, to fully implement scalable network, it is not enough to have static components; the topology of the known components must also be available. Figure 5.4 presents one possible antenna arrangement.

It should be possible to link regular modules to notification modules, in order that system can determine which regular module should be used to connect to found device. In Figure 5.4 notification module N1 is linked to regular modules R1 and R2, forming *cluster* 1, and notification module N2, to modules R3 and R4, forming cluster 2. This adds some burden to client device users in both Notify Module Connection and Module Discovery Connection, which is analyzed in Section 5.3.

In both Predefined Address Connection and Inquiry Based Connection, coverage by clustering is harder to arrange. One solution could be that modules are laid out correspondingly, and a connection attempt from both (all) clusters is done if device could not be found in previous the attempts. This naturally increases the overall

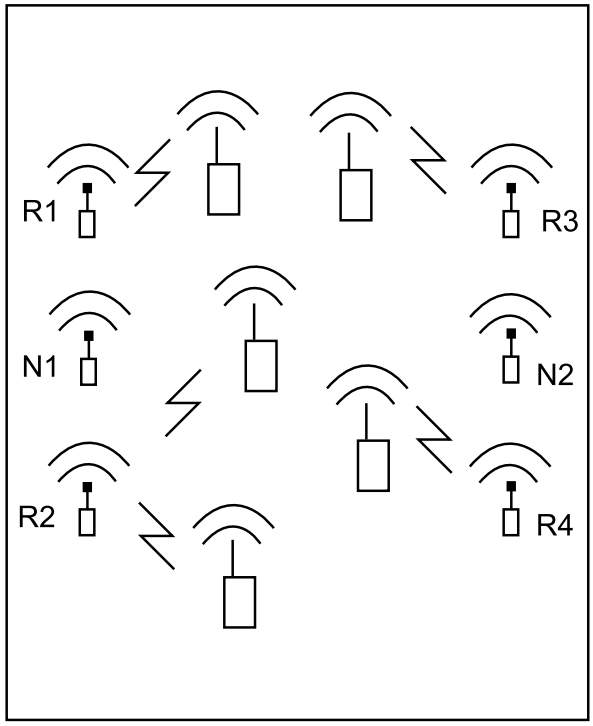

Figure 5.4: An example of a module layout ensuring coverage of the network.

time required.

Notify Module Connection and Module Discovery Connection can also be combined to form a single connection procedure. A large number of devices inquiring in MDC can cause too much radio traffic preventing effective communication. This can be resolved by saving addresses of notification modules to the devices. Those users who have address saved, can perform NMC only[1], while those not having the address can use MDC. As mentioned, it is also possible to ensure radio coverage during connection establishment and radio communication with both of these approaches using clusterization.

It is also possible to adjust visibility of notification modules during the MDC procedure, which may prevent piling up the connections to single cluster, or to single notification module. There is, however, no empirical data at the moment regarding the effects of the usage of this option.

While implementing the connection schemes, it was occasionally noted that specifications available are quite limited: not all mobile phones work within InSitu system, even if the phone's public specification is identical to another that does work. For example, Nokia 6600 phone has problems with authentication[2] and some others cannot simply run the client application. In addition, mobile device emulators do

not work quite like their actual counterparts, and especially debugging is tedious using real devices. Implementations also restrict their usage; Java implementation in mobile devices prevents devices from using other access codes than the General Inquiry Access Code (GIAC) and the Limited Inquiry Access Code (LIAC), even though there exists several other access codes. In InSitu system this has lead to "abuse" of LIAC, which should, according to specification, only be used for a limited time.

The connection establishment in each scheme ends when a connection has been made to the service in client device. Bluetooth services are identified during service discovery using Universally Unique Identifiers (UUID). UUIDs used within the system are listed in Table 5.2.

Router	`38BE1FB8-9980-11D9-8BDE-F66BAD1E3F3A`
Leaf	`17F68C6C-FA09-11DA-93F6-0050DA44E4F7`

Table 5.2: Universally Unique Identifiers for Bluetooth services in the system.

The Two-way paging used in both NMC and MDC may seem a bit complex, especially since there is a master-slave switching specified in Bluetooth specification. Not all implementations, however, support master-slave switching, and it would also be tedious to distribute devices to piconets without coordination from the server (router) side; but, see Section 5.6 for details regarding this option.

5.3 User Interaction and Usability

Due to wide range of users, the user interaction should remain relatively simple, especially during connection establishment, to ensure fluent forming of the network, for example, in the beginning of a lecture. The procedures presented in Section 5.1 can be divided in two categories: inquiry-based and address-based; Inquiry Based Connection and Module Discovery Connection belonging to the former, and Predefined Address Connection and Notify Module Connection to the latter.

Inquiry-based approaches are the easiest for the client device users; starting the client software is enough to start the connection establishment.

In the address-based procedures, user has to input a raw 48-bit hexadecimal address, that is, 12 hex numbers (see Section 3.1.3 for an example), at least when connecting for the first time. In Predefined Address Connection, on the other hand, this is required only on the server-side, but have to be done to every device wishing to join the network. A database can be formed to assist this approach.

Typing a single address in Notify Module Connection may not be the trickiest task, but larger networks may require multiple notification modules, which means multiple addresses in address-based procedures. This depends on the actual number of antennas and layout of the notification modules. The users must also know which module is located where, in order to fully take advantage of module layout, i.e., the clusterization. In addition, this depends on the position of the client device in proportion to module clusters, which may vary between different sessions even if clusters' locations do not change.

Norman [35] argues that user's actions in a given situation should be obvious. If the user is required to input one of the several 48-bit addresses given, depending on the positioning in proportion to cluster positions, the actions are far from obvious.

As stated in Section 5.1, Notify Module Connection and Module Discovery Connection can be combined. This eases user interaction; the user can input address of a notification module in simple cases like in networks with one or two notification modules, or use an inquiry to find a module whenever there are multiple ones. The user can then select the appropriate module among the notification modules found and save the address in a user friendly form (see Section 3.1.3) for future use. The saved address can then be used with simple NMC.

5.4 Performance measurement

Every approach presented in Sections 5.1.1–5.1.4 can be evaluated using the prototype system introduced in Chapter 4. The procedure is simple and follows a real situation. The results are obtained in two forms: as performance data recorded by the system and as system logs. This section presents the measurement procedure, protocol for communicating performance data, and the measurements performed for this thesis.

5.4.1 Procedure

Measurement of performance in nominal case consists of five phases:

1. Connect to server
2. Synchronize system clocks
3. Perform the connection establishment being measured
4. Collect the performance data

5. Start again from phase 3

To synchronize the system clocks for comparing data from different devices, the router must have knowledge of the devices. This is achieved by first connecting, using any of the schemes, to the server and storing devices' addresses and system clocks. Synchronization of system clocks is done using debugging subprotocol (see Section 5.4.2) which consists of *Time request*, *Time response*, *Timestamp request*, *Timestamp response*, and *Module response* packets.

Time request is sent by the server to all clients and indicates that the system clock should be sent immediately to server. The system time is communicated using *Time response*. The difference between system clocks is determined by calculating time difference between request and response, dividing it with two and comparing it to received timestamp. Maximum error caused by this operation is in the worst case the round-trip time, which is typically 0.1–0.3 seconds. In practice, the actual error is considerably smaller since it takes approximately equal time to send the request to client as send the response back to server.

Connection establishment is performed as presented in Section 5.1. The Inquiry Based Connection has not been measured using this procedure — but would be done similarly — whereas, Predefined Address Connection uses a slightly more straightforward method. PAC is initiated by the server and requires, therefore, no knowledge of the timestamps from clients. The server simply measures the time difference between connection request and actual connection.

Apart from *Time request* and *Time response* used in system clock synchronization, other packets of the debugging subprotocol are used to gather data recorded during connection establishment. The timestamps are written instantly and collected centralized after the procedure to ensure that the measurement itself has a minimal impact on the results. *Timestamp request* is sent by the server to request timestamp data from router and clients. The clients respond normally with *Timestamp response* using IMPv2 packets. The router sends timestamp data to server, also using IMPv2, by masking them to appear to be coming from clients, as seen in Section E.1.5 in the appendix. The device associated with data is concluded from the data of the enclosing IRP *Received from* packet.

In addition to *Timestamp response*, the router sends the module data using *Module response*, which contains knowledge of the hello point and module used during the connection. The router selects the module m from the set of modules M for the new device on the fly using a module selection algorithm, which is in essence as follows: the router keeps track of the number of currently running connection establishment procedures E^m and the number of active connections C^m for each module $m \in M$,

and selects the module m where $E^m \leq E^i \forall i \in M \wedge C^m < d_{pico} - 1$. The algorithm is portrayed in detail in Section E.2.2 in the appendix.

5.4.2 Protocol

The protocol used to communicate timestamp data is specified using augmented Backus-Naur Form (see, for example, Appendix B for detailed definition) as follows:

time-request =	”TIME”
time-response =	”TIME” time
timestamp-request =	”TIMES”
timestamp-response =	”TIMESTAMP” time timestamp-name
module-response =	”MODULES” hp-number module-number
time =	hour ”:” minutes ”:” seconds ”.” milliseconds
hour =	2DIGIT
minutes =	2DIGIT
seconds =	2DIGIT
milliseconds =	3DIGIT
timestamp-name =	3DIGIT
hp-number =	3DIGIT
module-number =	3DIGIT

It must be noted that it was surprisingly straightforward to implement the debug functionality to Teacher's App due to the centralized message passing system, which allowed effective hooking to events invoked by the core module presented in Section 4.3.2.

Each timestamp response contains an identifier for the timestamp. The identifier specifies the event that has been logged. List of timestamps is presented in Table 5.3. Identifiers of the form `Cxx` are recorded by the client, identifiers of the form `Rxx` by the router, and identifiers `PA0` and `PA1` are used within the server while measuring the PAC scheme.

The results obtained by analyzing the timestamp data for NMC and PAC are examined in Section 5.5.

5.4.3 Measurements

Measurements for this thesis were carried out for different connection schemes, for different number of devices, and for different number of modules. Predefined Address Connection, Notify Module Connection, and Module Discovery Connection

C00	User initiated the connection from client.
C10	Client beginning to connect to Hello Point.
C15	Client started to connect to Hello Point.
C20	Client connected to hello point.
C30	Client accepted connection from router.
R00	Router accepted connection from remote device.
R10	Router beginning to create SDP connection.
R15	Router listing services from remote device.
R20	Router connected to remote device.
PA0	Connection request sent by server.
PA1	Device connected to server.

Table 5.3: Timestamp identifiers.

were measured with 6, 12, and 18 devices and with 2 or 4 notification (or regular, as in PAC) modules.

The measurements were made in two different locations. Connection establishment with 18 devices in different schemes was measured in an mid-sized auditorium, with distance between modules and devices ranging from 1 to 10 meters in most cases. Measures performed with 6 and 12 devices were done in ”laboratory conditions”, i.e., in an open room with range of approximately 2 meters. That said, it is not exactly clear how much the environment affects the results, but the general trend can nevertheless be seen.

When measured with 6 devices, the following phone models were used:

- 4 $\times$ Nokia 6630, and
- 2 $\times$ Nokia 6680.

When measured using 12 devices, the following phones were used:

- 4 $\times$ Nokia 6630,
- 3 $\times$ Nokia 6680,
- 2 $\times$ Nokia 9500,
- 2 $\times$ Nokia 6620, and
- 1 $\times$ Nokia 7610.

During the larger test with 18 phones, the following phones were present:

- 4 × Nokia 6630,
- 3 × Nokia 6680,
- 2 × Nokia 9500,
- 2 × Nokia 6620,
- 1 × Nokia 7610,
- 1 × Nokia N80,
- 1 × Nokia N76,
- 1 × Nokia E60,
- 1 × Nokia E70,
- 1 × Nokia 9300, and
- 1 × Sony-Ericsson 990.

The measurement was automated as far as possible. Clients were run in *automated mode*, in which they, for example, react to *Disconnect* packet with reconnect after few seconds. If no abnormal disconnects were detected, no human interaction was necessary within nor between connection establishment *cycles*. A cycle is essentially a sequence of connection establishment procedures that end in a successful simultaneous connection of all devices; i.e., is formed by the phases 3–4 of the procedure presented in Section 5.4.1.

5.5 Results

In this section, the results from the measurements introduced in Section 5.4.3 are presented with conclusions. Summary of data is given in Appendix A.

Generally speaking, the results are related to two main topics: the time required in each connection scheme and configuration[3], and number of disconnects in different phases of the connection establishment.

5.5.1 General

When formulating the connection schemes and analyzing their performance in Section 5.1, this was generally done under the assumption that each device must be connected exactly once. The measurements proved that this is not the case: measures with different device amounts resulted in different amounts of disconnects during the connection establishment.

There were three kinds of communication errors: connection establishment attempts that failed for some reason, devices which lost already acquired connection during the connection establishment, and devices that lost connection while sending the performance timestamps. Of these three, the latest reflects the fragility of the Bluetooth network in general, not only during the connection establishment. When analyzing the numbers, it must be noted that automatic connection retries were allowed from the client side in PAC and NMC schemes, and were not counted as a failed connection establishment, as long as one of the attempts succeeded.

The impact of reconnects on time per device T_{avg} value has been eliminated from the results by defining T_{avg} as presented in equation A.2, that is, calculating the time to first attempt to request the timestamps and dividing it with the number of devices plus reconnects. This is not perfectly accurate measure, but as seen from relatively small deviations, accurate enough for general analysis. Although disconnects have been taken into account, we do not know *how much* of the additional delay seen in results is from larger interference from larger amount of devices[4], and how much from the disconnects themselves.

The data presented in Appendix A contains numbers for disconnects, but some of the disconnects included were client crashes[5]; their exact number is uncertain, but presumably around 1% of total number of connection establishment procedures. These disconnects are not result from the Bluetooth radio technology and should not be counted towards failed connection attempt.

In order to simulate the network using formalizations presented in Section 5.1, we need to determine the rates, at which the transitions are fired, i.e., the models must be *configured*. We may attempt to determine the variation of, for example, P_{avg} in different schemes and situations using the results from subsequent sections. If we can achieve that, we may be able to simulate the actual throughput of the connection establishment with larger amounts of devices. The challenge is, as said, that the rates vary considerably from situation to situation. One solution might be to model the network mathematically, simulate it, and determine from probabilities of collisions and disconnects the firing rates, which could then be used to configure the formal

model. In this thesis, however, no further calculations using the formalizations are performed.

5.5.2 Predefined Address Connection

Both PAC 4, which is Predefined Address Connection using four modules, and PAC 2, with two modules, were measured using 6 and 12 devices. PAC 4 was also measured using 18 devices. The results indicate that even with PAC 2 with 12 devices, which is the second challenging of the cases, no disconnects were detected. However, while measuring with 18 devices, PAC 4 suffered from disconnects in about 3% of the cases.

Performance measures of PAC 2 indicate that with 6 devices, each device takes on average approximately 2.0 seconds and with 12 devices approximately 1.6. The reason may be the smaller overhead per device present with 12 devices. On the other hand, there was relatively high deviation ($\sigma = 0.23$) in this case for some reason.

A sample device by device analysis of PAC 2 with 12 devices, seen in Figure 5.5, offers a partial explanation. The first devices are quickly connected by the two available modules, but the later ones seem to suffer from piling to queue.

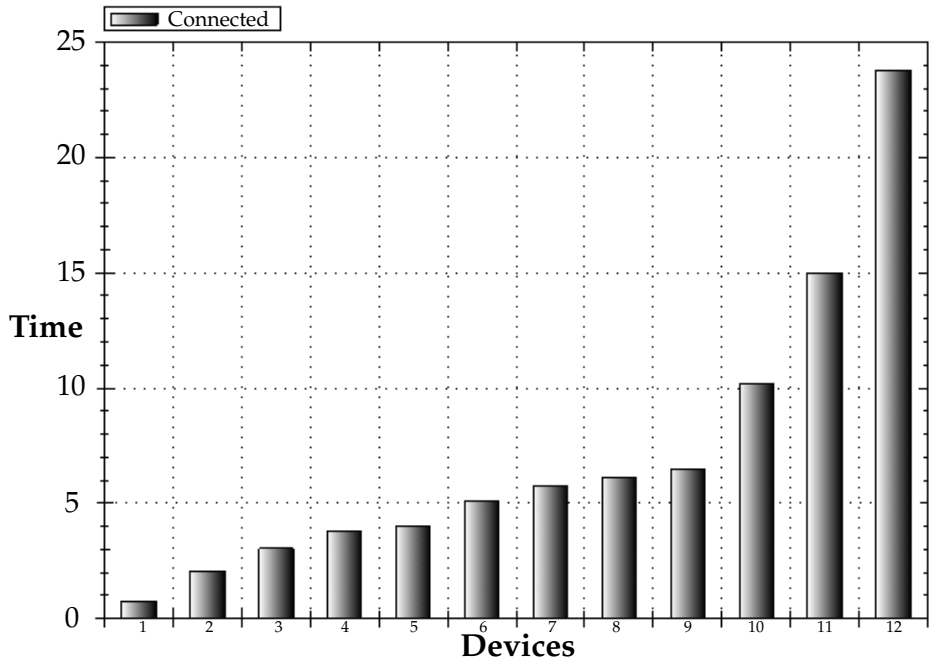

Figure 5.5: Performance of PAC 2 with 12 devices.

Performance measures of PAC 4 indicate that PAC 4 is approximately 2.7 times and 1.6 times faster than PAC 2 with 6 and 12 devices, respectively. This is due to the

two additional modules, which should, in theory, double the throughput. As seen from Figure 5.6, this sample of PAC performs in a more linear fashion. Interesting is that the first 6 devices are connected in 5 seconds in both PAC 2 and PAC 4. The sudden increment of PAC 2 may be result of some retries[6] needed in this particular case.

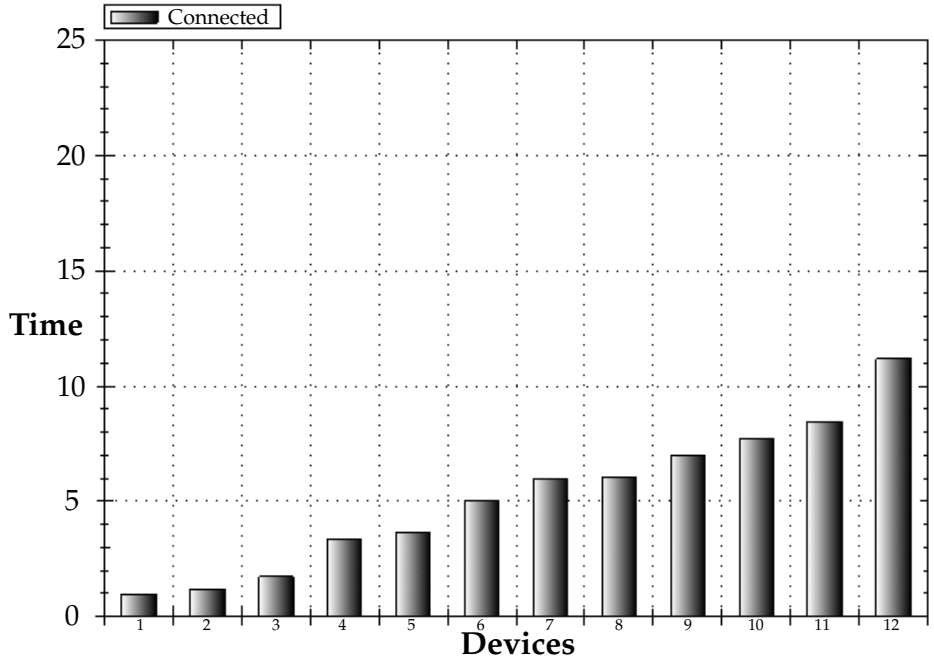

Figure 5.6: Performance of PAC 2 with 12 devices.

PAC 4 with 18 devices has considerable higher deviation ($\sigma = 1.13$), which is true to all measures performed with 18 devices regardless of the scheme. There are several factors in play: disconnects, of which impact cannot be fully nullified, larger interference due to larger amount of devices, and longer distance due to different measurement location. PAC 4 with 18 is also over two times slower than PAC 4 with 12 for these reasons.

5.5.3 Notify Module Connection

Whereas in PAC the time per device was the most relevant factor, for NMC it is the number of disconnects. NMC 1 to 4 (that is, one notification module and four others) was not able to handle 6 devices without disconnects. The number of disconnects while sending timestamps, varied from 5.5% to 7.5% in all cases of NMC 1 to 4. The more interesting result is the rate of *Disconnects while connected*, which increases as the number of devices increase. There were 1.7%, 5.8%, and 18.5% disconnect

rates while connected in cases of 6, 12, and 18 devices, respectively. While using 18 devices, disconnects during connection also began to appear at the rate of 1.9%.

NMC 2 to 4 (i.e., two notification modules) yields similar but arguably better results. The number of disconnects while sending timestamps, varied between 1.5%, and 2.5% in all cases. The rates of disconnects while connected, were 0.0%, 0.0%, and 10.0%, with 6, 12, and 18 devices, respectively, i.e., NMC 2 to 4 was able to handle 12 devices. Disconnects during connection appeared at rate of 2.2% with 18 devices.

These numbers reveal clearly that NMC cannot, in this form, cope with required device amounts; especially cases of 18 devices yield extremely high disconnect rates, and the trend seems to be overlinear. What is peculiar is the fact that number of disconnects while sending the timestamps — that is, while all devices were already connected — was notably higher with 1 to 4 configuration.

Results with 6 and 12 devices indicate connection establishment performance of 2.0–3.0 and 4.0–5.0 seconds per device for NMC 2 to 4 and 1 to 4, respectively. Probably for same reasons as in PAC, as mentioned in Section 5.5.2, the corresponding results from 18 devices are almost twice as high as those from 6 and 12 devices.

When analysed separately, the results indicate acceptable connection time per device. But when the number of disconnects (and reconnects needed) is taken into account, the theoretically linear performance slips to polynomial form due to the ascending rate of disconnects in proportion to the number of devices.

Regardless of the unsatisfactory results, it is possible, for the sake of it, using the timestamp data gathered, to analyse NMC in more detail by dividing the connection establishment to several phases.

As seen[7] from Figure 5.7, NMC is divided to five phases in this case. First phase (*Starting*) consists of starting the connection establishment, and its end is indicated by the timestamp `C10` presented in Table 5.3. Then either follows the phase *Waiting server* or phase *Paging server*, their ending marked by `C15` and `R00`, respectively. The phase depends on whether the device in question was able to connect to a notification module or was timed out. As can be seen from the figure, this delay is the most significant when determining the overall connection establishment time. Time taken by *Paging server*, *Paging client*, or *Connecting to device* bears no visible dependency to the order in which the devices are connected.

Figure 5.8 presents a quite similar procedure of 2 to 4, but with less long *Waiting server* phases. In fact, this seems to be the only significant difference to NMC 1 to 4.

The number of modules and hello points is, indeed, a significant factor: from the performance point of view, there is no use for eight hello points if there are, say, only four other modules. The number of normal modules is dictated by the

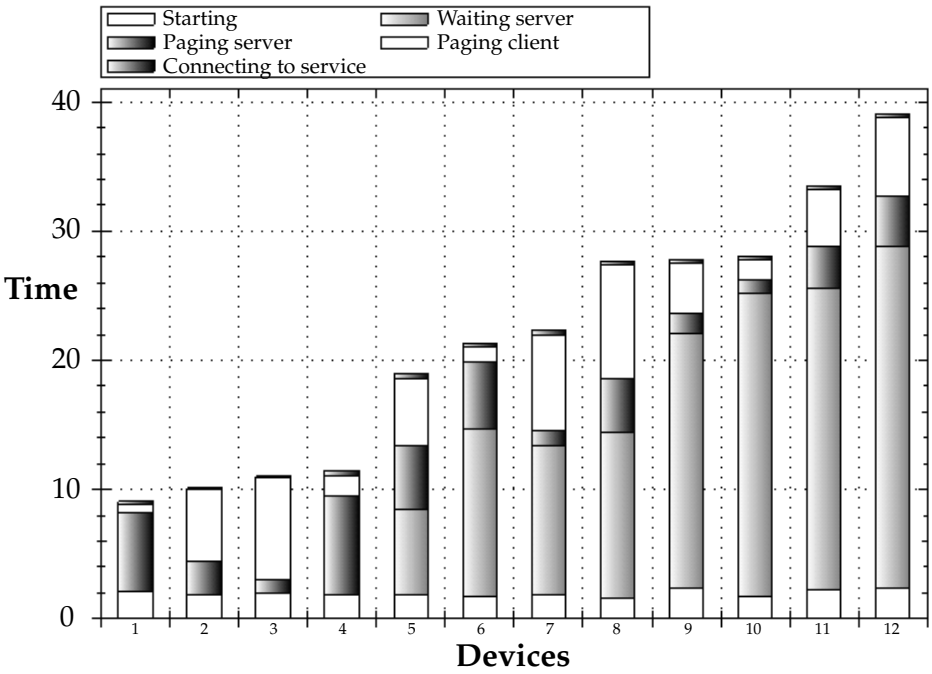

Figure 5.7: Performance of NMC 1 to 4 with 12 devices.

number of devices in the network, which can then be used to determine the suitable number of notification modules. It seems that larger module count produces better performance.

It can be seen from figures 5.7 and 5.8 that the bottleneck is *Accept* rate in both situations (the *Waiting server* indicates time taken by failed attempts to page the server). If there was, for instance, a 2 to 2 configuration, the bottleneck would most likely be the *Connect* (see Figure 5.2), whereat there would be another phase[8] between the *Paging server* and *Paging client* phases.

It might be interesting to note that if paging time is affected by, e.g., interference or larger distance, this affects the optimal number of modules in NMC (see equation 5.9), since, as paging the server is performed simultaneously and paging of clients is not, these factors affect differently the paging times T_{in} and T_{out}. This relationship is, however, subtle and has unlikely a practical significance.

5.5.4 Module Discovery Connection

MDC was not measured as extensively as Predefined Address Connection and Notify Module Connection. The Bluetooth device cache of the devices stores the recently inquired devices, and thus only first cycle of MDC measurements yields reasonable results. In MDC, the number of disconnects at connection is the number

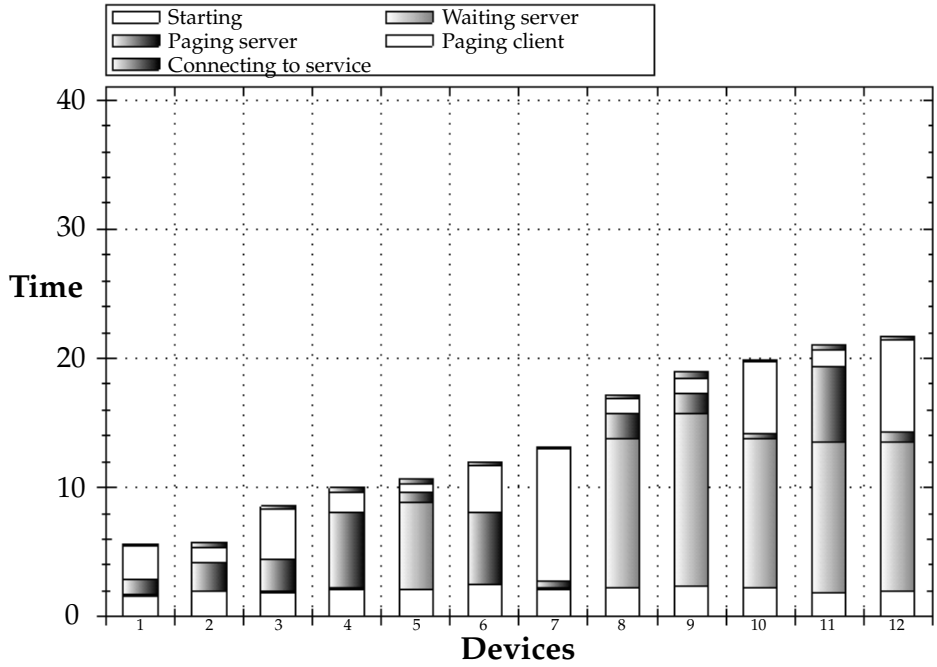

Figure 5.8: Performance of NMC 2 to 4 with 12 devices.

of devices that did not find hello point during first inquiry attempt. MDC scheme is not completely implemented: it could use multiple sequential inquiries to ensure better coverage, but in essence MDC after inquiry is identical — or at least not better due to traffic generated by inquiry — in performance to corresponding NMC. Since NMC was not a viable solution in its current state, MDC will not be either. The results from the quick tests indicate that even with as small amounts as 6 devices, there were some which could not find the hello point with single inquiry for single notification module. The number of failures increases as the number of devices increases. This is understandable due to the fact that each device performs inquiry only once.

It is notable that with two visible modules the failure rate is considerable smaller. This may suggest that using enough modules and multiple sequential inquiries MDC may cover all the devices. This could be used as base for further improvements presented in Section 5.6.

5.5.5 Others

Inquiry Based Connection was not measured in detail, but couple of proof of concept tests were performed. The problem with measuring inquiry based schemes is the Bluetooth device cache which prevents practical performing of iterative tests. As

pointed out earlier, IBC cannot guarantee the finding of all the Bluetooth devices present. However, during the laboratory tests with 12 devices within 2 meters, all the devices were found with single inquiry performed by USB module attached to Linux PC[9]. That said, development and measurement of IBC has not been a major concern, but the scheme *might* provide some additional support if used jointly with other approaches.

5.6 Improving performance and robustness

There are at least two other approaches that have not yet been implemented, which might provide better performance and more robust network. There are also some possible improvements to existing schemes.

Due to relatively good performance and high robustness of Predefined Address Connection approach, a hybrid of NMC and PAC might provide a solution. It may be that performing connections causes the heaviest interference to other piconets, which in turn causes disconnects among the already connected devices. This hybrid, Buffered Notify Module Connection (BNMC), is in essence a Notify Module Connection where after the initial paging to notification modules, the server waits for the flow of new incoming connections to stop and performs then the connection establishment to all present devices at once, much like in PAC. This approach is even slower than NMC but might work fast enough if the number of disconnects can be kept low.

Second option is to list all piconets (i.e., normal modules), and distribute their addresses to mobile device users in a way that each piconet receives no more than seven clients. This approach, All to Predefined Module Connection (APMC), then requires only one paging from the client side, which should be fast. The process of distributing the addresses manually should be arranged somehow.

A general improvement to all approaches is either to introduce *ping* functionality also to client side or *timeout to ping* to either side. In addition, there should be functionality which enables reconnects after connection has been lost, without bothering the user and without losing any data. In practise this requires monitoring from both the client and the server side, for example, in cases where a question was sent while some device was disconnected. The question should, in cases like this, be automatically sent after reconnect.

One possibility to improve robustness of the network, is to introduce scheduling: the router could forward data to clients in small bulks, say, to one piconet at a time. This should decrease the interference between different piconets.

Notes

[1]The address might be stored independently from device cache, which stores addresses in a temporary fashion.

[2]An assumption based on the observed behaviour during testing; not verified.

[3]A configuration of a scheme is essentially determined by the number of modules and hello points used.

[4]Or how much does the auditorium environment affect the results, for that matter.

[5]Client crashes are most likely result of threading related software defects in In-Situ Leaf.

[6]Server retries are not counted towards failed attempts, if some of them eventually succeed.

[7]In figures 5.7 and 5.8 the legend is interpreted in a way that phases listed are being executed first from left to right and then from top to bottom. In *bars*, the phases are presented from bottom to top.

[8]The phase has been omitted from the figures for clarity; it was in both cases practically zero.

[9]It must be strongly expressed that there was no way, the author was aware of, to ensure that these devices were not in device cache. Mobile devices usually clear their cache after a while unless the remote device is explicitly stored, but this cannot be guaranteed to be the case with BlueZ kernel module.

6 Conclusion and Further Research

InSitu is a prototype of an electronic mass lecture feedback and questionnaire system having its roots in *Peer Instruction* method. To solve the usual high cost of electronic questionnaire systems, it is implemented using students' mobile devices as answering devices. The overall goal is to improve learning by adjusting teaching as necessary, by activating students to participate and by encouraging interaction during various education situations.

A central part of the system is based on wireless network technology, especially Bluetooth, which is a short range radio communication protocol, now widely available in mobile devices. Other possible communication means include WLAN and GPRS. GPRS has the disadvantage of being probably costly, and WLAN has not been widely adopted yet. On the other hand, there is no certainty whether any of these radio technologies will function when brought to same environment together in orders of magnitude required in this context.

A system prototype was developed during years 2005–2007. It consists of five distinct parts, of which three play key role while developing and evaluating radio technologies and different connection schemes. The system is divided to server, router, mobile clients, PC clients, and projector. The server part coordinates the operation of other parts and provides primary interface to lecturer, router manages Bluetooth clients, mobile and pc clients serve as interface to students during the lecture, and projector displays gathered results to audience as necessary.

Four different connection schemes were introduced and three of them measured. Inquiry Based Connection was not suitable due to incompleteness of Bluetooth inquiry, Predefined Address Connection was difficult due to the Bluetooth address database required, Notify Module Connection and Module Discovery Connection were, in theory, suitable, but tend to suffer from disconnects during the connection establishment.

PAC, NMC, and MDC were tested in both real environment and in laboratory conditions with different module counts and different device amounts. Results indicate that none of the approaches were sufficient. The connection schemes came off in linear time — to a certain point — as suspected, but suffered from disconnects in ascending rate, which in essence indicates polynomial results in practice. With given requirements the polynomial result is not acceptable.

Further research and development can be divided to four major categories: developing better connection establishment schemes, elaborating the software system, researching physical radio communication, and studying the cognitive effects of lecture questionnaires.

New connection establishment schemes that might provide an alternative to those defined and measured in detail in this thesis, include All to Predefined Module Connection (APMC) and Buffered Notify Module Connection (BNMC). These approaches attempt to minimize the traffic needed in order to prevent connection establishment from interfering with devices already connected. The research also includes verifying the actual reasons behind the failing connection establishment: is it piconets interfering with each other alone or are there other factors in play.

Elaborating the software system further might include adding additional safety measures to clients to ensure detection of connection loss and adding automatic reconnection functionality. Shifting functionality from router to server might also be beneficial since the server is easier to maintain and develop. From a broader point of view, future research could include developing an open standard format for questions, results etc. to provide ways to share resources between lecturers using different systems.

Radio communication in general could be examined by evaluating effects of radio traffic with mathematical modelling and simulations. This might give answer to how WLAN and GPRS interoperate together, and with Bluetooth. WLAN and piconet interference are presumably the major issues in systems of this magnitude.

Peer Instruction and lecture feedback in general should be subject to study in future. The cognitive effects, especially in programming lectures, would be reasonable fruitful area of research.

As a general observation it must be pointed out that current heterogeneity and immaturity of technological environment prevents effective software development in mobile environments. Implementations are not entirely conformable to standards, emulators do not work quite like the actual devices, debugging is tedious in real situations, and implementations are in many cases too restrictive in their usage. Overall, there is clearly a need for better developbility [sic] in mobile environments; there is pressure to develop systems more complex.

This thesis presented a glimpse to software development for mobile devices, radio communication, lecture feedback, and related topics. Utilizing wireless networks for assisting lectures proved to be more challenging than first anticipated. Further research — and development — may be needed, but some basis has now been laid out.

References

[1] Norman Abramson, *Development of the ALOHANET*, in *IEEE Transactions on Information Theory*, volume 31, pp. 119–123, March 1985

[2] Frank Babbitt (ed.), *Moralia*, volume 1, Loeb Classical Library, 1927

[3] Len Bass, Paul Clements and Rick Kazman, *Software Architecture in Practice*, Addison-Wesley, 2006, ISBN 978-0321154958

[4] *Bluetooth Specification Version 2.0 + EDR*, volume 1, November 2004

[5] *Bluetooth Specification Version 2.0 + EDR*, volume 3, November 2004

[6] *Bluetooth Specification Version 2.0 + EDR*, volume 4, November 2004

[7] *Bluetooth Specification Version 2.1 + EDR*, volume 0, July 2007

[8] Bluetooth SIG, *Bluetooth Wireless Technology Surpasses One Billion Devices*, November 2006

[9] *BlueZ Official Linux Bluetooth protocol stack*, 2007, `http://www.bluez.org/`, Referenced August 2007

[10] Yu-Fen Chen, Chen-Chung Liu, Ming-Hung Yu, Sung-Bin Chang, Yun-Chen Lu and Tak-Wai Chan, *Elementary Science Classroom Learning with Wireless Response Devices – Implementing Active and Experiential Learning*, in *WMTE'05*, IEEE Computer Society, 2005

[11] *Classtalk, The Classroom Communication System*, `http://www.bedu.com/classtalk.html`, Referenced August 2007

[12] Catherine Crouch and Eric Mazur, *Peer Instruction: Ten years of experience and results*, in *American Journal of Physics*, pp. 970–977, American Association of Physics Teachers, Harvard University, September 2001

[13] Carlos de Morais Cordeiro and Dharma Agrawal, *Employing Dynamic Segmentation for Effective Co-located Coexistence between Bluetooth and IEEE 802.11 WLANs*, in *IEEE GLOBECOM*, pp. 195–200, 2002

[14] Carlos de Morais Cordeiro and Dharma Agrawal, *Mitigating the Effects of Intermittent Interference on Bluetooth Ad Hoc Networks*, in *PIMRC*, pp. 496–500, IEEE, 2002

[15] Carlos de Morais Cordeiro and Djamel Sadok, *Piconet Interference Modeling and Performance Evaluation of Bluetooth MAC Protocol*, in *IEEE Transactions on Wireless Comm.*, pp. 2870–2874, 2001

[16] Jörg Desel and Gabriel Juhás, *What Is a Petri Net? Informal Answers for the Informed Reader*, in *Unifying Petri Nets, LNCS 2128*, pp. 1–25, 2001

[17] *ENJOY Audience Response System*, `http://www.enjoy-ars.com/1-Overview.htm`, Referenced August 2007

[18] Jr. Frederick P. Brooks, *The Mythical Man-Month*, Addison-Wesley, 26th edition, 2005, ISBN 978-0201835953

[19] N. Golmie and O. Rebala, *Techniques to Improve the Performance of TCP in a mixed Bluetooth and WLAN Environment*, in *International Conference on Communications*, volume 2, pp. 1181–1185, IEEE, 2003

[20] Michael Hayoz, *The Bluetooth Wireless Technology, An Overview*, 2005, Department of Informatics, University of Fribourg, `http://diuf.unifr.ch/ds/michael.hayoz/docs/hayozm_blatand.pdf`, Referenced August 2007

[21] Ivan Howitt, *Mutual Interference Between Independent Bluetooth Piconets*, in *IEEE Transactions on Vehicular Technology*, volume 52, pp. 708–718, IEEE, 2003

[22] *IEEE 802.11n Report*, 2007, `http://grouper.ieee.org/groups/802/11/Reports/tgn_update.htm`, Referenced September 2007

[23] *IEEE 802.11 Official Timelines*, 2007, `http://grouper.ieee.org/groups/802/11/Reports/802.11_Timelines.htm`, Referenced September 2007

[24] *Infonode Docking Windows*, 2007, `http://www.infonode.net/index.html?idw`, Referenced August 2007

[25] *RFC 791: Internet Protocol*, September 1981, `http://tools.ietf.org/rfc/rfc791.txt`, Referenced August 2006

[26] *JCommon*, 2007, `http://www.jfree.org/jcommon/`, Referenced August 2007

[27] *JFreeChart*, 2007, http://www.jfree.org/jfreechart/, Referenced August 2007

[28] Jung-Hyuck Jo and Nikil Jayant, *Performance Evaluation of Multiple IEEE 802.11b WLAN Stations in the Presence of Bluetooth Radio Interference*, in *International Conference on Communications*, pp. 1163–1168, 2003

[29] *Joda Time*, 2007, http://joda-time.sourceforge.net/, Referenced August 2007

[30] *JSR-000082 Java(TM) APIs for Bluetooth Specification 1.0 Final Release*, 2006, http://jcp.org/aboutJava/communityprocess/final/jsr082, Referenced August 2006

[31] Vesa Lappalainen, *In Situ -vastauslaitteen käyttöopas*, 2000, User manual for In Situ answering device, http://www.mit.jyu.fi/~vesal/insitu/palikka.htm, Referenced September 2007

[32] Eric Mazur, *Peer Instruction: A User's Manual*, Benjamin Cummings, 1996, ISBN 978-0135654415

[33] David Nicol and James Boyle, *Peer Instruction versus Class-wide Discussion in Large Classes: a comparison of two interaction methods in the wired classroom*, in *Studies in Higher Education Volume 28, No. 4*, Society for Research into Higher Education, October 2003

[34] *Nokia Suomi - Kaikki puhelimet*, 2007, Nokia Finland – All Phones, http://www.nokia.fi/A4312003, Referenced August 2007

[35] Donald Norman, *The Design of Everyday Things*, Basic Books, New York, 1988, ISBN 978-0465067107

[36] Edward Redish, *Teaching Physics with the Physics Suite*, Wiley, 2003, ISBN 978-0471393788

[37] Franklin D. Roosevelt, *The Public Papers and Addresses of Franklin D. Roosevelt*, volume 1, Random House, 1938

[38] Perry Samson, Stephanie Teasley, Ben van der Pluijm and Peter Knoop, *Using Handheld PCs and Peer Instruction to Improve Science Teaching and Learning in Higher Education*, in *ICLS'06*, pp. 980–981, 2006

[39] Frank Siegemund and Michael Rohs, *Rendezvous layer protocols for Bluetooth-enabled smart devices*, February 2003, Springer-Verlag London Limited

[40] Mel Silberman, *Active Learning: 101 Strategies to Teach Any Subject*, Allyn & Bacon, 1996, ISBN 978-0205178667

[41] IEEE-SA Standards Board, *ANSI/IEEE Std 802.11: Wireless LAN Medium Access Control and Physical Layer Specifications*, June 2003

[42] *RFC 793: Transmission Control Protocol*, September 1981, `http://www.ietf.org/rfc/rfc793.txt`, Referenced August 2006

[43] Ltd. VOCAL Technologies, *GPRS White Paper*, 2002, `http://www.vocal.com/white_paper/GPRS_wp1pdf.pdf`, Referenced August 2007

[44] Hongfeng Wang, *Overview of Bluetooth Technology*, July 2001, Department of Electrical Engineering, State College

A Performance Data

Summary[1] of data gathered during InSitu performance tests is presented here. Results from various connection schemes are rendered, and their exact nature is presented in detail in Chapter 5. *Disconnects at connection* N_{dc}^{conn} indicate number of disconnects that occurred during the actual connection establishment, *Disconnects while connected* N_{dc}^{wconn} the number of disconnects after successful connection, and *Disconnects at timestamps* N_{dc}^{ts} the number of disconnects during gathering of timestamp data.

Number of connection establishment procedures N_{con} indicate the number of connection procedures applied for this connection scheme during performance testing. The number of connection establishment procedures is given by

$$N_{con} = N_{dev} N_{cycles}, \tag{A.1}$$

where N_{dev} is amount of devices and N_{cycles} number of connection establishment cycles. Thus, the number of reconnects resulting from disconnects or failed attempts is not included in N_{con}. *Average time per device* T_{avg} presents the time taken by each device[2] from the initiation of connection establishment to an established connection, and *Standard deviation* σ, the root mean square deviation of average times of different connection establishment cycles.

N_{dc}^{ts} was not measured in PAC since timestamps were not send in this scheme; all timing was done by the server. In MDC, N_{dc}^{ts} and N_{dc}^{wconn} were not measured due to Bluetooth restrictions regarding device caches. For the same reason, T_{avg} and the corresponding σ were not measurable.

In NMC and PAC schemes

$$T_{avg} = \frac{1}{N_{cycles}} \sum_{i=1}^{N_{cycles}} \frac{T_{tot_i}}{N_{dev} + N_{dc_i}^{wconn} + N_{dc_i}^{conn}}, \tag{A.2}$$

where T_{tot_i} is the time to first attempt to send the timestamp requests, i.e., when all devices are connected for the first time in cycle i.

Notes

[1]The actual data gathered from single performance test session is several hundreds of pages of XML data and several hundreds of pages of system logs. These are not included here for practical reasons.

[2]More detailed partitioning to different phases is presented in Chapter 5, but the raw data behind this analysis is omitted from this thesis.

Connection scheme	**Number of devices** N_{dev}	**Disconnects at connection** N_{dc}^{conn}	**Disconnects while connected** N_{dc}^{wconn}	**Disconnects at timestamps** N_{dc}^{ts}
NMC 1 to 4	6	0	1	4
NMC 1 to 4	12	0	7	7
NMC 1 to 4	18	1	10	4
NMC 2 to 4	6	0	0	1
NMC 2 to 4	12	0	0	2
NMC 2 to 4	18	2	9	2
MDC 1 to 4	6	2	—	—
MDC 1 to 4	12	7	—	—
MDC 1 to 4	18	9	—	—
MDC 2 to 4	6	0	—	—
MDC 2 to 4	12	3	—	—
PAC 2	6	0	0	—
PAC 2	12	0	0	—
PAC 4	6	0	0	—
PAC 4	12	0	0	—
PAC 4	18	3	0	—

Table A.1: Summary of performance measurement results. *cont.*

Connection scheme	**Number of devices N_{dev}**	**Number of connection establishment procedures N_{con}**	**Average time per device T_{avg}**	**Standard deviation σ**
NMC 1 to 4	6	60	4.79	0.93
NMC 1 to 4	12	120	4.14	0.38
NMC 1 to 4	18	54	9.86	2.57
NMC 2 to 4	6	60	2.82	0.44
NMC 2 to 4	12	120	2.68	0.48
NMC 2 to 4	18	90	5.38	1.01
MDC 1 to 4	6	6	—	—
MDC 1 to 4	12	12	—	—
MDC 1 to 4	18	18	—	—
MDC 2 to 4	6	6	—	—
MDC 2 to 4	12	12	—	—
PAC 2	6	60	2.04	0.09
PAC 2	12	120	1.61	0.23
PAC 4	6	60	0.74	0.10
PAC 4	12	120	1.02	0.15
PAC 4	18	90	2.26	1.13

Table A.1: *cont.* Summary of performance measurement results.

B IRP Specification

1 Introduction

1.1 Purpose

The InSitu router Protocol (IRP) is an application level protocol for InSitu system. It defines means for managing Bluetooth connections in InSitu router. It is also used to transport InSitu Mobile Protocol packets to clients via router.

See InSitu Mobile Protocol version 2 specification for details regarding background of IRP.

1.2 Requirements

The key words ”MUST”, ”MUST NOT”, ”REQUIRED”, ”SHALL”, ”SHALL NOT”, ”SHOULD”, ”SHOULD NOT”, ”RECOMMENDED”, ”MAY”, and ”OPTIONAL” in this document are to be interpreted as described in RFC 2119. An implementation is not compliant if it fails to satisfy one or more of the MUST or REQUIRED level requirements for the protocols it implements. An implementation that satisfies all the MUST or REQUIRED level and all the SHOULD level requirements for its protocols is said to be ”unconditionally compliant”; one that satisfies all the MUST level requirements but not all the SHOULD level requirements for its protocols is said to be ”conditionally compliant.”

1.3 Terminology

This specification uses a number of terms to refer to the roles played by participants in, and objects of, the IMP communication.

IRP InSitu Router Protocol

IMPv2 InSitu Mobile Protocol version 2

Connection A transport layer virtual circuit established between two programs for the purpose of communication.

Client A client device with program that is able to communicate with server using IMPv2.

Router A system managing Bluetooth connections and forwarding IMPv2 packets to clients.

Server A program to be able to communicate with clients using IMPv2. Server MAY be able to communicate with router(s) using IRP.

Packet Single sequence of big-endian UTF-16 characters for single purpose in terms of communication; one entity of those defined in Section 3.3.

Sender Program sending a specified packet.

Receiver Program receiving specified packet.

Session Time scope from connection establishment between server and router to disconnection of this connection.

1.4 Scope

This specification defines communication after connection has been established between two programs, server and client. Connection establishment and general network properties are not considered in this paper.

1.5 Encoding

All communication is based on plain text sequence of characters represented in big-endian UTF-16 encoding.

2 Notational Conventions and Generic Grammar

All of the mechanisms specified in this document are described in both prose and an augmented Backus-Naur Form (BNF). Implementers will need to be familiar with the notation in order to understand this specification.

2.1 Augmented BNF

The augmented BNF includes the following constructs and is partly presented here based on presentation in RFC 2616:

name = definition The name of a rule is simply the name itself (without any enclosing ”<” and ”>”) and is separated from its definition by the equal ”=” character. White space is only significant in that indentation of continuation lines is

used to indicate a rule definition that spans more than one line. Certain basic rules are in uppercase, such as CHAR, DIGIT, etc. Angle brackets are used within definitions whenever their presence will facilitate discerning the use of rule names.

"literal" Quotation marks surround literal text. Unless stated otherwise, the text is case-sensitive.

rule1 | rule2 Elements separated by a bar ("|") are alternatives, e.g., "yes | no" will accept yes or no.

(rule1 rule2) Elements enclosed in parentheses are treated as a single element. Thus, "(elem (foo | bar) elem)" allows the token sequences "elem foo elem" and "elem bar elem".

***rule** The character "*" preceding an element indicates repetition. The full form is "<n>*<m>element" indicating at least <n> and at most <m> occurrences of element. Default values are 0 and infinity so that "*(element)" allows any number, including zero; "1*element" requires at least one; and "1*2element" allows one or two.

[rule] Square brackets enclose optional elements; "[foo bar]" is equivalent to "*1(foo bar)".

N rule Specific repetition: "<n>(element)" is equivalent to "<n>*<n>(element)"; that is, exactly <n> occurrences of (element). Thus 2DIGIT is a 2-digit number, and 3CHAR is a string of three characters.

; comment A semi-colon, set off some distance to the right of rule text, starts a comment that continues to the end of line. This is a simple way of including useful notes in parallel with the specifications.

2.2 Basic Rules

The following rules are used throughout this specification to describe basic parsing constructs.

```
CHAR =   <any big-endian UTF-16 character>
DIGIT =  <any big-endian UTF-16 digit character "0".."9">
HEX =    DIGIT | "A" | "B" | "C" | "D" | "E" | "F" |
         "a" | "b" | "c" | "d" | "e" | "f"
```

It is notable that definition of CHAR includes DIGITs.

3 IMPv2 Packets

A packet is a single plain text sequence of characters for single purpose in terms of communication.

3.1 Overall Operation

Both sender and receiver MUST treat packet as an atomic entity. When data arrives to receiver, a packet is defined by the length sequence of the packet, and any further data arriving MUST be treated as a part of the next packet. Packet length SHALL NOT exceed 2048 bytes.

3.2 General packet composition

A packet consists of header and data.

packet = header data

Header is divided to length sequence and command sequence. First five characters SHALL represent the length of the packet NOT INCLUDING the first five characters.

header = packet-len command
packet-len = 5DIGIT

Command defines the purpose of the packet. Specifics about different packets are defined in Section 3.3.

```
command =   router-settings-cmd |       ; Router settings
            user-connected-cmd |        ; User connected
            user-disconnected-cmd |     ; User disconnected
            send-to-cmd |               ; Send to
            received-from-cmd |         ; Received from
            connection-list-cmd |       ; Connection list
            disconnect-user-cmd |       ; Disconnect user
            disconnect-cmd |            ; Disconnect
            keep-alive-cmd              ; Keep alive
```

Data consists of arbitrary number of characters defined by length of the packet.

```
data =  *CHAR
```

3.3 Packets

Specific packets are sent either by server, client or both. This is declared separately for each packet in the subsequent sections. Each section includes example packet represented in quotation marks, which are not included in the actual data.

3.3.1 Router settings

Router settings packet MUST be send by router immediately after server has established the connection to it. Packet contains information of capabilities of the router.

```
router-settings-packet =  packet-len router-settings-cmd max-num-clients
router-settings-cmd =     "ROUTER"
max-num-clients =         3DIGIT
```

Example data: "00009ROUTER007"

3.3.2 User connected

User connected packet is sent by router to indicate that new client has connected to router. Packet contains an id assigned by router to device. This id is used to identify the particular device during current session. Router MUST use unique id to identify devices but MAY reuse any id that has been used by client that is no longer connected. Packet also contains a Bluetooth address of the device, which is represented as hex values.

user-connected-packet =	packet-len
	user-connected-cmd
	device-id
	device-address
user-connected-cmd =	”USRCON”
device-id =	3DIGIT
device-address =	12HEX

Example data: ”00021USRCON003000A3A53D4C1”

3.3.3 User disconnected

User disconnected packet is sent by router to indicate that an existing client with specified id has disconnected.

user-disconnected-packet =	packet-len user-disconnected-cmd device-id
user-disconnected-cmd =	”USRDCN”
device-id =	3DIGIT

Example data: ”00009USRDCN003”

3.3.4 Send to

Send to packet is sent by server. The packet contains inner packet that is sent to devices with specified ids.

send-to-packet =	packet-len
	send-to-packet-cmd
	id-count ids
	inner-packet
send-to-packet-cmd =	”SENDTO”
id-count =	3DIGIT
ids =	id | id ids
id =	3DIGIT
inner-packet =	*CHAR

The number of ids is equal to parsed value of id-count.

Example data: ”00057SENDTO00200500600037QUESTS0102005What?000030600203Yes02No”

3.3.5 Received from

Received from packet is sent by router. It contains inner packet that has been received from client with specified id.

received-from-packet =	packet-len received-from-cmd device-id inner-packet
received-from-cmd =	”RECVFR”
device-id =	3DIGIT
inner-packet =	*CHAR

Example data: ”00024RECVFR00600010003ANSWER1”

3.3.6 Connection list

Connection list packet is sent by server to order the server to establish connection to specified addresses.

connection-list-packet =	packet-len connection-list-cmd address-count addresses
connection-list-cmd =	”CONLST”
address-count =	3DIGIT
addresses =	address | address addresses
address =	12HEX

The number of addresses is equal to parsed value of address-count.
Example data: ”00021CONLST001000A3A51D4C1”

3.3.7 Disconnect user

Disconnect user packet is sent by the server to disconnect client(s) from router.

disconnect-user-packet =	packet-len disconnect-user-cmd id-count ids
disconnect-user-cmd =	”DCUSER”
id-count =	3DIGIT
ids =	id | id ids
id =	3DIGIT

Example data: ”00015DCUSER002001003”

3.3.8 Disconnect

Disconnect packet is sent by server to disconnect all users, disconnect server from router and reset the router to accept new connections.

disconnect-packet = packet-len disconnet-cmd
disconnect-cmd = ”DISCON”

Example data: ”00006DISCON”

3.3.9 Keep alive

Keep alive packet is sent by the server to ensure that router is still connected. Router SHOULD determine that connection has been lost if no packet has arrived in certain time. Default timeout value is 5.0 seconds and keep alive interval 2.0 seconds.

keep-alive-packet = packet-len keep-alive-cmd
keep-alive-cmd = ”KEEPAL”

Example data: ”00006KEEPAL”

3.4 Communication restrictions

Router MUST send router settings to server immediately after connection establishment prior sending or receiving any other packets. Server MUST NOT send packets to router before receiving router settings packet.

4 Acknowledgments

Sections 1.2 and 2 have been edited after a number of Network Working Group RFCs noted in sections in question. Specification development was greatly assisted by Mikko Tyrväinen and Vesa Lappalainen.

C IMPv2 Specification

1 Introduction

1.1 Purpose

The InSitu Mobile Protocol (IMP) is an application level protocol for InSitu system. The first IMP was defined for early prototypes for Bluetooth communication between InSitu system and Bluetooth enabled mobile devices. It defined basic functionality for asking single questions, session handling and authorization.

There was need for clear separation between InSitu Router Protocol (IRP) and InSitu Mobile Protocol (IMP), to ensure extensibility to different network schemes. IRP was separated from IMP by isolating all Bluetooth management specific operations to independent specification. Name IMPv2 was introduced at this point to create distinction to previous version. IMPv2 was further extended to include more detailed information regarding questions and answers as well as packets for better session management.

IRP and IMPv2 are syntactically similar but semantically independent. IRP can be used to manage router without IMPv2 and is able forward any kind of packets to clients connected to a server supporting IRP. IMPv2 can be used for communication between server and any IMPv2-compatible client regardless of the network architecture.

1.2 Requirements

The key words ”MUST”, ”MUST NOT”, ”REQUIRED”, ”SHALL”, ”SHALL NOT”, ”SHOULD”, ”SHOULD NOT”, ”RECOMMENDED”, ”MAY”, and ”OPTIONAL” in this document are to be interpreted as described in RFC 2119. An implementation is not compliant if it fails to satisfy one or more of the MUST or REQUIRED level requirements for the protocols it implements. An implementation that satisfies all the MUST or REQUIRED level and all the SHOULD level requirements for its protocols is said to be ”unconditionally compliant”; one that satisfies all the MUST level requirements but not all the SHOULD level requirements for its protocols is said to be ”conditionally compliant.”

1.3 Terminology

This specification uses a number of terms to refer to the roles played by participants in, and objects of, the IMP communication.

IRP InSitu Router Protocol

IMP InSitu Mobile Protocol

IMPv2 InSitu Mobile Protocol version 2

Connection A transport layer virtual circuit established between two programs for the purpose of communication.

Client A client device with program that is able to communicate with server using IMPv2.

Router A system managing Bluetooth connections and forwarding IMPv2 packets to clients.

Server A program to be able to communicate with clients using IMPv2. Server MAY be able to communicate with router(s) using IRP.

Packet Single sequence of big-endian UTF-16 characters for single purpose in terms of communication; one entity of those defined in Section 3.3.

Sender Program sending a specified packet.

Receiver Program receiving specified packet.

Session Time scope from first connection of any client to disconnection of last connected client

1.4 Scope

This specification defines communication after connection has been established between two programs, server and client. Connection establishment and general network properties are not considered in this paper.

1.5 Encoding

All communication is based on plain text sequence of characters represented in big-endian UTF-16 encoding.

2 Notational Conventions and Generic Grammar

All of the mechanisms specified in this document are described in both prose and an augmented Backus-Naur Form (BNF). Implementers will need to be familiar with the notation in order to understand this specification.

2.1 Augmented BNF

The augmented BNF includes the following constructs and is partly presented here based on presentation in RFC 2616:

name = definition The name of a rule is simply the name itself (without any enclosing "<" and ">") and is separated from its definition by the equal "=" character. White space is only significant in that indentation of continuation lines is used to indicate a rule definition that spans more than one line. Certain basic rules are in uppercase, such as CHAR, DIGIT, etc. Angle brackets are used within definitions whenever their presence will facilitate discerning the use of rule names.

"literal" Quotation marks surround literal text. Unless stated otherwise, the text is case-sensitive.

rule1 | rule2 Elements separated by a bar ("|") are alternatives, e.g., "yes | no" will accept yes or no.

(rule1 rule2) Elements enclosed in parentheses are treated as a single element. Thus, "(elem (foo | bar) elem)" allows the token sequences "elem foo elem" and "elem bar elem".

***rule** The character "*" preceding an element indicates repetition. The full form is "<n>*<m>element" indicating at least <n> and at most <m> occurrences of element. Default values are 0 and infinity so that "*(element)" allows any number, including zero; "1*element" requires at least one; and "1*2element" allows one or two.

[rule] Square brackets enclose optional elements; "[foo bar]" is equivalent to "*1(foo bar)".

N rule Specific repetition: "<n>(element)" is equivalent to "<n>*<n>(element)"; that is, exactly <n> occurrences of (element). Thus 2DIGIT is a 2-digit number, and 3CHAR is a string of three characters.

; comment A semi-colon, set off some distance to the right of rule text, starts a comment that continues to the end of line. This is a simple way of including useful notes in parallel with the specifications.

2.2 Basic Rules

The following rules are used throughout this specification to describe basic parsing constructs.

CHAR = <any big-endian UTF-16 character>
DIGIT = <any big-endian UTF-16 digit character ”0”..”9”>

It is notable that definition of CHAR includes DIGITs.

3 IMPv2 Packets

A packet is a single plain text sequence of characters for single purpose in terms of communication.

3.1 Overall Operation

Both sender and receiver MUST treat packet as an atomic entity. When data arrives to receiver, a packet is defined by the length sequence of the packet, and any further data arriving MUST be treated as a part of the next packet. Packet length SHALL NOT exceed 2048 bytes.

3.2 General packet composition

A packet consists of header and data.

packet = header data

Header is divided to length sequence and command sequence. First five characters SHALL represent the length of the packet NOT INCLUDING the first five characters.

header = packet-len command
packet-len = 5DIGIT

Command defines the purpose of the packet. Specifics about different packets are defined in Section 3.3.

```
command =   login-request-cmd |      ; Login request
            login-response-cmd |     ; Login response
            login-reject-cmd |       ; Login reject
            login-success-cmd |      ; Login success
            message-cmd |            ; Message
            question-select-cmd |    ; Question select
            question-order-cmd |     ; Question order
            question-input-cmd |     ; Question input
            answer-cmd |             ; Answer
            end-question-cmd |       ; End question
            disconnect-cmd |         ; Disconnect
            keep-alive-cmd           ; Keep alive
```

Data consists of arbitrary number of characters defined by length of the packet.

```
data =   *CHAR
```

3.3 Packets

Specific packets are sent either by server, client or both. This is declared separately for each packet in the subsequent sections. Each section includes example packet represented in quotation marks, which are not included in the actual data.

3.3.1 Login request

Login request is sent by the server as soon as the connection has been established.

```
login-request-packet =   packet-len login-request-cmd allow-anonymous
login-request-cmd =      "LGNREQ"
allow-anonymous =        "0" | "1"
```

In allow-anonymous token, value "0" represents false and "1" represents true. Example data: "00007LGNREQ0"

3.3.2 Login response

Login response is sent by the client in response to login request packet.

login-response-packet =	packet-len login-response-cmd username-len username password-len password connection-type client-version
login-response-cmd =	”LGNRSP”
username-len =	2DIGIT
username =	*CHAR
password-len =	2DIGIT
password =	*CHAR
connection-type =	2DIGIT
client-version =	2DIGIT

Number of characters in username token is equal to parsed value of username-len token. Number of characters in password token is equal to parsed value of password-len token.

In connection-type token value ”00” represents Bluetooth, ”01” represents TCP/IP and ”02” represents GPRS/WLAN.

Example data: ”00026LGNRSP080123456704999900001”

3.3.3 Login reject

Login reject is sent by the server in response to Login response if login fails.

login-reject-packet =	packet-len login-reject-cmd reason
login-reject-cmd =	”LGNREJ”
reason =	DIGIT

Example data: ”00007LGNREJ2”

3.3.3.1 Reason: Unknown user

Login reject reason ”Unknown user” is represented by value ”0”. The server responds with this reason if it was not able to find the username specified in Login response.

3.3.3.2 Reason: Invalid password

Login reject reason ”Invalid password” is represented by value ”1”. The server responds with this reason if the password and username specified in Login response did not match.

3.3.3.3 Reason: Anonymous not allowed

Login reject reason ”Anonymous not allowed” is represented by value ”2”. The server responds with this reason if no username was specified in Login response, but server does not allow anonymous login.

3.3.3.4 Reason: Already logged in

Login reject reason ”Already logged in” is represented by value ”3”. The server responds with this reason if the username specified in Login response is already logged in the system.

3.3.3.5 Reason: Unknown error

Login reject reason ”Unknown error” is represented by value ”4” or above. The server responds with this reason if it encountered an unspecified error during login.

3.3.4 Login success

Login reject is sent by the server in response to Login response if login succeeds.

login-success-packet =	packet-len login-success-cmd message
login-success-cmd =	”LOGNOK”
message =	*CHAR

Example data: ”00013LOGNOKWelcome”

3.3.5 Message

Message can be sent either by server or by client to the other participant. It represents single textual message.

message-packet =	packet-len message-cmd message
message-cmd =	”MESSAG”
message =	*CHAR

Example data: ”00011MESSAGHello”

3.3.6 Question select

Question select is sent by the server. Question select represents question that has number of alternatives to select from. Question id MUST be unique to each question, regardless of the type (select, order, input), during single session. Valid response SHOULD consist of selection of at least min-alternatives but no more than max-alternatives alternatives from total of alternative-count alternatives, where 0 <= min-alternatives <= max-alternatives <= alternative-count <= 10 and max-alternatives > 0.

Instant question type represents regular question that MUST be responded with a single answer packet. Continuous question type represents question that can be answered multiple times, but not after question end packet has been received with question id identical to this question.

question-select-packet =	packet-len question-select-cmd min-alternatives max-alternatives question-len question-text question-type response-type question-id question-time alternative-count alternatives
question-select-cmd =	”QUESTS”
min-alternatives =	2DIGIT
max-alternatives =	2DIGIT
question-len =	3DIGIT
question-text =	*CHAR
question-type =	”0” | ”1”
response-type =	”0” | ”1”
question-id =	3DIGIT
question-time =	3DIGIT
alternative-count =	2DIGIT
alternatives =	alternative | alternative alternatives
alternative =	alternative-len alternative-text
alternative-len =	2DIGIT
alternative-text =	*CHAR

Number of characters in question-text token is equal to parsed value of question-len token. In question-type token, value ”0” represents instant type and ”1” represents continuous type. In response-type token, value ”0” represents user send and ”1” represents auto send. Number of characters in alternative-text token is equal to parsed value of alternative-len token.

Response type SHOULD NOT be set to auto send unless both min-alternatives and max-alternatives are equal to 1.

Example data: ”00037QUESTS0102005What?000030600203Yes02No”

3.3.7 Question order

Question order is sent by the server. Question select represents question that has number of alternatives to select and order from. Question id MUST be unique to each question, regardless of the type (select, order, input), during single session. Valid response SHOULD consist of selection and correct ordering of at least min-alternatives but no more than max-alternatives alternatives from total of alternative-count alternatives, where 0 <= min-alternatives <= max-alternatives <= alternative-count <= 10 and max-alternatives > 0.

Question time represents time in seconds during witch the question is valid and must be answered to. Client MAY send answer to a particular question when question times out, if user has not already done so. Time equal to zero indicates that question is valid until either disconnect packet is sent or received or question end is received.

Instant question type represents regular question that MUST be responded with a single answer packet. Continuous question type represents question that can be answered multiple times, but not after question end packet has been received with question id identical to this question.

question-order-packet =	packet-len
	question-order-cmd
	min-alternatives
	min-alternatives
	question-len
	question-text
	question-type
	question-id
	question-time
	alternative-count
	alternatives
question-order-cmd =	”QUESTO”
min-alternatives =	2DIGIT
min-alternatives =	2DIGIT
question-len =	3DIGIT
question-text =	*CHAR
question-type =	”0” | ”1”
question-id =	3DIGIT
question-time =	3DIGIT
alternative-count =	2DIGIT
alternatives =	alternative | alternative alternatives
alternative =	alternative-len alternative-text
alternative-len =	2DIGIT
alternative-text =	*CHAR

Number of characters in question-text token is equal to parsed value of question-len token. In question-type token, value ”0” represents instant type and ”1” represents continuous type. Number of characters in alternative-text token is equal to parsed value of alternative-len token.

Example data: ”00036QUESTO0102005What?00030600203Yes02No”

3.3.8 Question input

Question input packet is sent by server and represents request to type a textual answer to a question. Answer consists of text, numbers or both.

Question time represents time in seconds during witch the question is valid and must be answered to. Question id MUST be unique to each question, regardless of the type (select, order, input), during single session. Client MAY send answer to a

particular question when question times out, if user has not already done so. Time equal to zero indicates that question is valid until either disconnect packet is sent or received or question end is received.

question-input-packet =	packet-len question-input-cmd max-answer-len answer-type question-id question-time question-text
question-input-cmd =	”QUESTI”
max-answer-len =	3DIGIT
answer-type =	”0” | ”1” | ”2”
question-id =	3DIGIT
question-time =	3DIGIT
question-text =	*CHAR

Answer type token contains values ”0”, which represents text input, ”1”, which represents numeric input and ”2”, which represents both text and numeric input.

Example data: ”00021QUEST0501003060What?”

3.3.9 Answer

Answer packet is sent by client as an answer to question that has been asked. Question can be any type and can have any options.

Answer packet MUST contain question id that represents question that has been asked but yet not have been terminated by either receiving question end packet of identical id, or by timeout indicated by question time field of the question packet. Client MAY, however, send answer to question which has just timed out, if user has not already done so.

answer-packet =	packet-len answer-cmd question-id answer
answer-cmd =	”ANSWER”
question-id =	3DIGIT
answer =	*CHAR

Example data: ”00012ANSWER010394”

3.3.9.1 Answer token for question select packet

Answers to question select packet are represented by numbers of alternatives selected. The numbering is zero-based, thus if first and third alternatives are selected, the answer equals to ”02”. The answer token MUST NOT contain same character more than once.

Example data: ”00012ANSWER013349”

3.3.9.2 Answer token for question order packet

Answers to question order packet are represented by numbers of alternatives in the order user selected them. The numbering is zero-based, thus if third and first alternatives are selected in that particular order, the answer equals to ”20”. The answer token MUST NOT contain same character more than once.

Example data: ”00012ANSWER014394”

3.3.9.3 Answer token for question input packet

Answers to question input packet are represented by the text typed by user.

Example data: ”00013ANSWER0151941”

3.3.10 End question

End question packet is sent by the server and is a request to terminate particular question identified by question id supplied within the packet.

Client SHOULD NOT send answer to question that has been terminated by question end right after the question has been ended, and MUST not send answer to question that has been ended at some point earlier.

end-question-packet =	packet-len end-question-cmd question-id message
end-question-cmd =	”ENDQUE”
question-id =	3DIGIT
message =	*CHAR

Example data: ”00020ENDQUE015Lunch break”

3.3.11 Disconnect

Disconnect packet is either send by the server or the client. Server MAY send packet when it notifies clients that session has been ended, and client MAY send disconnect

packet when it is about to disconnect from the session.

disconnect-packet =	packet-len disconnect-cmd message
disconnect-cmd =	”DISCON”
message =	*CHAR

Example data: ”00018DISCONLecture ends”

3.3.12 Keep alive

Keep alive packet is sent by the server to ensure that clients are still connected. Client MAY determine that connection has been lost if no packet has arrived in certain time. Default timeout value is 5.0 seconds and keep alive interval 2.0 seconds.

keep-alive-packet =	packet-len keep-alive-cmd
keep-alive-cmd =	”KEEPAL”

Example data: ”00006KEEPAL”

3.4 Communication restrictions

Server SHOULD send login request packet only once for each client after connection establishment. Login request SHOULD be first packet sent to client after connection establishment. Server SHOULD NOT send question packets (question select, question order, question input) to client prior login success packet.

3.5 Examples

3.5.1 Login

Server:	Login Request ”00007LGNREQ0”
Client:	Login response ”00026LGNRSP080123456704999900 01”
Server:	Login Success ”00013LOGNOKWelcome”

3.5.2 Question

Server:	Question select ”00037QUESTS0102005What?000030600203Yes02No”
Client:	Answer ”00010003ANSWER1”

4 Acknowledgments

Sections 1.2 and 2 have been edited after a number of Network Working Group RFCs noted in chapters in question. Specification development was greatly assisted by Mikko Tyrväinen and Vesa Lappalainen.

D Architecture Notation and Views

Notation used in figures in this thesis is presented here. Figure D.1 present the notation used to express logical elements of the system, Figure D.2 presents notation in process graphs, Figure D.3 presents notation used in figures portraying elements during development time, and Figure D.4 presents notation for figures rendering physical layout of the system.

The figures are adapted from *Kruchten's four views*[1] presented (among others) by Bass et. al. in [3, 41] and represent logical view, process view, development view, and physical view, respectively. Some parts of the notation are loosely based on UML, but since there exists no widely accepted standard for software architecture documentation, a tailored representation is formalized and presented here for clarity.

Logical views (or module views in [3]) portray key abstractions in the architectural design.

Process views (component-and-connector in [3]) present concurrency and distribution of functionality.

Development view (allocation view mapping software to development environment in [3]) shows the organization of software modules, libraries, and subsystems.

Physical view (deployment or allocation view in [3]) maps distribution of elements onto processing and communication nodes.

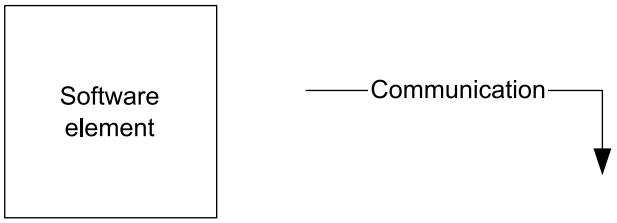

Figure D.1: Logical notation.

Notes

[1]Also known as "Four Plus One" approach.

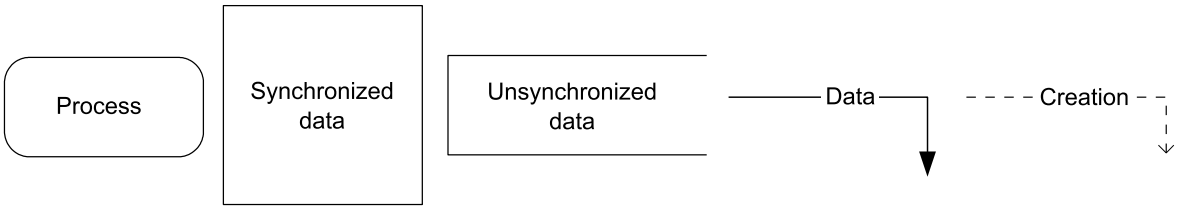

Figure D.2: Process notation.

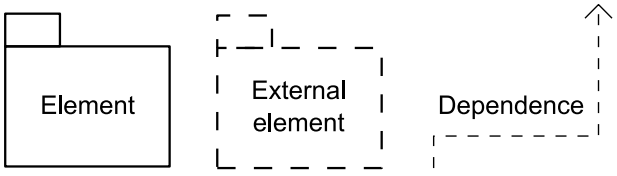

Figure D.3: Development notation.

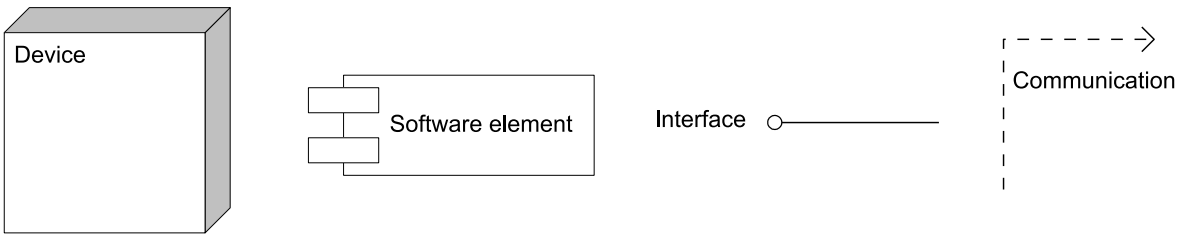

Figure D.4: Physical notation.

E Relevant Parts of Source Code of Router

At its current state, the InSitu system and its tools consist of over 25 KLOC, spread over more than half-dozen programs. Listing all of the source code on paper is naturally impractical in any form or for any purpose. What is presented here, is a glimpse of the part of core of router, which, in essence, handles the connection establishment and tasks related to performance measurement. This chapter thus provides insight to details of some of the connection schemes.

Due to emulation of object-oriented approach used, a `const` pointer to "object" *this* performing the operation is provided as first parameter and somewhat unusual naming convention (while using C as programming language) is used: pointers to type, e.g., `router_t*` are redefined as plain `router`.

E.1 router.c

This section presents parts from the core module of router. Its responsibilities include orchestrating the operation of all other parts of the system, handling IO during normal operation, multiplexing the messages from server and clients and storing and managing the clients' data.

E.1.1 router_create_connection

This method prepares device data and initiates connection to given address. The number of hello point and `R00` timestamp are passed for tracing the connection establishment.

```
void router_create_connection( const router this,
                               const int hp,
                               const char* address,
                               const char* r00 )
{
  device device = NULL;

  // Lock the device data to safely manipulate their state
  int mutex_result = pthread_mutex_lock( &this->device_mutex );
  if( mutex_result != SUCCESS )
  {
    sysutil_log( LOG_WARN, 0, "Mutex lock failed: %s\n", strerror( errno ) );
```

```
  }

  int device_num;
  for ( device_num = 0; device_num < this->max_device_count; device_num++ )
  {
    // Find first device slot that is disconnected
    if ( device == NULL && this->bt_devices[device_num].state == STATE_DISCONNECTED )
    {
      device = &this->bt_devices[device_num];
    }
    // Check whether there exists another device with same address
    if ( strncmp( address, this->bt_devices[device_num].address, BT_ADDR_LEN ) == 0 )
    {
      sysutil_log( LOG_WARN, 0, "Device with adress %s is already connected\n",
                   address );
      device = NULL;
      goto end;
    }
  }
  if ( device != NULL )
  {
    device->state = STATE_CONNECTING;
  }
  else
  {
    sysutil_log( LOG_WARN, 0, "No space for device %s\n", address );
    goto end;
  }

  // Store the address, R00 timestamp and hello point to device data
  strncpy( device->address, address, BT_ADDR_LEN + 1 );
  strncpy( device->R00, r00, TSLEN );

  device->hp = hp;

  // Create and start thread to perform connection establishment

  pthread_attr_t pthread_attr;
  memset( &pthread_attr, 0, sizeof( pthread_attr_t ) );

  conn_args_t* args = malloc( sizeof( conn_args_t ) );

  args->device = device;
  args->router = this;

  if( pthread_create( &(device->thread), NULL, (void*)router_connect_device, (void*)args )
              != SUCCESS )
  {
    sysutil_log( LOG_WARN, 0, "Could not start thread for device at %d\n", device->id );
  }

  // Setting up data for connection establishment is done
  // Device mutex can be released now
end:
  mutex_result = pthread_mutex_unlock( &this->device_mutex );
  if( mutex_result != SUCCESS )
```

```
  {
    sysutil_log( LOG_WARN, 0, "Mutex unlock failed: %s\n", strerror( errno ) );
  }
}
```

E.1.2 router_connect_device

This method is run in a separate thread for each device. The method delegates the connection establishment to `btanager` by invoking `btmanager_connect` (see Section E.2.2) and then finishes the connection establishment by updating device and program states accordingly.

```
void router_connect_device( conn_args_t* args )
{
  const device device = args->device;
  const router this = args->router;

  sysutil_log( LOG_EVENT, 0, "Connecting to %s\n", device->address );

  // Perform some timing
  watch_t w;
  watch_init( &w );
  watch_start( &w );

  // Let the Bluetooth manager perform the connection establishment
  if ( btmanager_connect( this->btmanager, device ) == SUCCESS )
  {
    sysutil_log( LOG_EVENT, 0, "Connected to %s\n", device->address );

    struct timeval timev;
    gettimeofday( &timev, NULL );
    double end = (double)timev.tv_sec + (1.e-6) * timev.tv_usec;
    device->time = end - w.start;

    watch_print( &w );
    watch_destroy( &w );

    // Update the program state and device data
    sysutil_set_nonblock( device->socket );

    int mutex_result = pthread_mutex_lock( &this->device_mutex );
    if( mutex_result != SUCCESS )
    {
      sysutil_log( LOG_WARN, 0, "Mutex lock failed: %s\n", strerror( errno ) );
    }

    device->state = STATE_CONNECTED_PENDING;
    this->connected_devices++;

    mutex_result = pthread_mutex_unlock( &this->device_mutex );
    if( mutex_result != SUCCESS )
```

```
    {
      sysutil_log( LOG_WARN, 0, "Mutex unlock failed: %s\n", strerror( errno ) );
    }
  }
  else
  {
    // Failed to connect, log the event and clean up the device data

    char* msg = "Could not connect to %s: %s\n";
    sysutil_log( LOG_EVENT, 0, msg, device->address, strerror( errno ) );
    sysutil_log( LOG_WARN, 0, msg, device->address, strerror( errno ) );

    watch_print( &w );
    watch_destroy( &w );

    int mutex_result = pthread_mutex_lock( &this->device_mutex );
    if( mutex_result != SUCCESS )
    {
      sysutil_log( LOG_WARN, 0, "Mutex lock failed: %s\n", strerror( errno ) );
    }

    device->state = STATE_DISCONNECTED;

    device_destroy( device );
    device_init( device );

    mutex_result = pthread_mutex_unlock( &this->device_mutex );
    if( mutex_result != SUCCESS )
    {
      sysutil_log( LOG_WARN, 0, "Mutex unlock failed: %s\n", strerror( errno ) );
    }
  }

  free( args );
  args = NULL;

  // This thread may now terminate itself. Further IO with devices is performed
  // by main thread via multiplexing.
  pthread_detach( pthread_self() );
}
```

E.1.3 router_send_timestamp

This method sends timestamp with given value and name and masks its sender to be the device with given id. The message, therefore, appears to be sent by the client.

```
void router_send_timestamp( const router this, char* from, char* name, int id )
{
  stream_t s;
  stream_init( &s, 35 );

  strncpy( &s.data[0], "00030MESSAGTIMESTAMP", 20 );
```

```
  strncpy( &s.data[20], from, 12 );
  strncpy( &s.data[32], name, 3 );

  packet_t p;

  // Put the data to received from packet to mask the sender appropriately
  packet_init_recv_from( &p, &s, id );

  // This method copies the packet data to be scheduled for sending
  router_send_to_server( this, &p );

  packet_destroy( &p );
  stream_destroy( &s );
}
```

E.1.4 router_send_modules

This method sends module IDs stored to given device. The message appears to be sent by the client in question.

```
void router_send_modules( const router this, const device device )
{
  stream_t s;
  stream_init( &s, 24 );

  strncpy( &s.data[0], "00019MESSAGMODULES", 18 );
  sprintf( &s.data[18], "%03d", device->hp );
  sprintf( &s.data[21], "%03d", device->dongle );

  packet_t p;

  // Put the data to received from packet to mask the sender appropriately
  packet_init_recv_from( &p, &s, device->id );

  // This method copies the packet data to be scheduled for sending
  router_send_to_server( this, &p );

  packet_destroy( &p );
  stream_destroy( &s );
}
```

E.1.5 router_handle_sendto

This method typically handles the *Send to* packet (see Section 4.4.1). As mentioned in Chapter 5, however, the debugging subprotocol utilizes *Message* packet from IMPv2. Therefore, whenever contents of a *Send to* packet is about to be forwarded, we need to check whether there is need to react to the subprotocol. Namely, the router also stores timestamps, which need to be sent.

The debugging functionality is not intended to be left to final program, at least not in this form, but is instead portrayed here for illustrating the mechanics used in performance measures.

```
void router_handle_sendto( const router this, const stream stream )
{
  stream->pos = PACKET_LENGTH_LENGTH + PACKET_COMMAND_LENGTH;

  // Exctract the ids of the devices we need to send the contents of this packet.
  int device_count = 0;
  stream_read_int( stream, 3, &device_count );

  int* ids = (int*)malloc( device_count * sizeof( int ) );

  int i;
  int connected = 0;
  for( i = 0; i < device_count; i++ )
  {
    int id = 0;
    stream_read_int( stream, 3, &id );

    // Skip the devices that are not connected
    if( this->bt_devices[id].state != STATE_CONNECTED )
    {
      sysutil_log( LOG_WARN, 0, "Attempt to send data to disconnected device at %d\n",
                   id );
      continue;
    }

    ids[connected++] = id;
  }

  int len = stream->len - stream->pos;

  char data[len+1];
  int b;for(b=0;b<len+1;b++)(data[b])='\0';

  stream_read_char( stream, len, data );

  /// BEGIN MEASUREMENT CODE

  // Here we violate the line between IMPv2 and IRP for debuggin purposes and check if
  // we have TIMES command of the subprotocol.

  bool sendTimes = FALSE;

  if( strncmp( data, "00011MESSAGTIMES", 16 ) == 0 )
  {
    sendTimes = TRUE;
  }

  /// END MEASUREMENT CODE

  // Send to appropriate devices
```

```
  for( i = 0; i < connected; i++ )
  {
    device device = &this->bt_devices[ids[i]];

    /// BEGIN MEASUREMENT CODE

    // Send the necessary performance data back to server
    if( sendTimes )
    {
      router_send_modules( this, device );
      router_send_timestamp( this, device->R00, "R00", device->id );
      router_send_timestamp( this, device->R10, "R10", device->id );
      router_send_timestamp( this, device->R15, "R15", device->id );
      router_send_timestamp( this, device->R20, "R20", device->id );
    }

    /// END MEASUREMENT CODE

    sysutil_log( LOG_DATA, 0, "Sending '%s' to device %d\n", data, device->id );

    if ( device->out_buffer.end + len > device->out_buffer.base->len )
    {
      sysutil_log( LOG_ERROR, 0, "Out buffer full\n" );
    }
    else
    {
      // Schedule the message to be sent by copying it to device's out buffer
      strncpy( &device->out_buffer.base->data[device->out_buffer.end], data, len );
      device->out_buffer.end += len;
      this->pending = TRUE;
    }
  }

  free( ids );
}
```

E.2 btmanager.c

This section presents parts from the Bluetooth manager module. It responsibilities include listening of hello points and initiating and running connection establishment to Bluetooth devices.

E.2.1 btmanager_listen_hp

This method is run for each notification module in a separate thread. The method essentially loops infinitely, accepts incoming connections as they are detected and invokes `router_create_connection` (see Section E.1.1) for each connection.

```
void btmanager_listen_hp( hp_args_t* args )
{
  const int device_id = args->id;
  const router this = args->router;
  free( args );

  // Create and initialize socket
  struct sockaddr_rc loc_addr = { 0 }, rem_addr = { 0 };
  int hp_socket, client;

  hp_socket = socket( AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM );

  int reuse_addr_flag = 1;
  setsockopt( hp_socket, SOL_SOCKET, SO_REUSEADDR, &reuse_addr_flag,
              sizeof( reuse_addr_flag ) );

  if ( hp_socket == ERROR )
  {
    sysutil_log( LOG_WARN, 0, "Could not create hello point: %s\n", strerror( errno ) );
    pthread_detach( pthread_self() );
    return;
  }

  bdaddr_t local_bt_addr;
  memset( &local_bt_addr, 0, sizeof( local_bt_addr ) );
  hci_devba( device_id, &local_bt_addr );

  char bt_addr[18];
  bt_addr[17] = '\0';
  ba2str( &local_bt_addr, bt_addr );
  sysutil_log( LOG_DEBUG, 0, "Using device '%s' as hello point\n", bt_addr );

  loc_addr.rc_family = AF_BLUETOOTH;
  loc_addr.rc_bdaddr = local_bt_addr;
  loc_addr.rc_channel = (uint8_t)30;

  if ( bind( hp_socket, (struct sockaddr *)&loc_addr, sizeof(loc_addr) ) != SUCCESS )
  {
    sysutil_log( LOG_WARN, 0, "Could not bind hello point: %s\n", strerror( errno ) );
    close( hp_socket );
    pthread_detach( pthread_self() );
    return;
  }

  // close socket when thread cancels (at shutdown)
  pthread_cleanup_push( btmanager_close_socket, (void*)hp_socket );

  int backlog = 1;

  sysutil_log( LOG_DEBUG, 0, "Setting backlog to %d\n", backlog );

  if ( listen( hp_socket, backlog ) != SUCCESS )
  {
    sysutil_log( LOG_WARN, 0, "Listen returned error: %s\n", strerror( errno ) );
  }
```

```
  char addr[BT_ADDR_LEN+1];

  unsigned int opt = sizeof( rem_addr );

  // Enter the loop, we exit by thread interruption
  while ( TRUE )
  {
    if( !btmanager_is_device_at( device_id, NULL ) )
    {
      sysutil_log( LOG_ERROR, 0, "Hello point at %d disconnected\n", device_id );
      return;
    }

    // Accept the connection
    sysutil_log( LOG_DEBUG, 0, "hp %d ready to accept\n", device_id );
    client = accept( hp_socket, (struct sockaddr *)&rem_addr, &opt );
    sysutil_log( LOG_DEBUG, 0, "hp %d done accepting\n", device_id );

    if( client == -1 )
    {
      sysutil_log( LOG_WARN, 0, "Accept returned error: %s\n", strerror( errno ) );
      continue;
    }

    // store the R00 timestamp here
    char r00[TSLEN];
    router_print_time( r00 );

    // Close the connection
    close( client );

    ba2str( &rem_addr.rc_bdaddr, addr );
    sysutil_log( LOG_EVENT, 0, "Accepted connection to hello point %d from %s\n",
                 device_id, addr );

    // Begin connection establishment procedure
    router_create_connection( this, addr, r00 );
  }

  // We never get here, but the push macro has opening bracket, so close it
  pthread_cleanup_pop( TRUE );
}
```

E.2.2 btmanager_connect

This method performs the actual connection establishment. First, the appropriate module is selected, then a SDP connection is made to remote device, which is followed by service discovery. Finally, a connection to service is established and a cleanup and program state update is performed.

```
int btmanager_connect( const btmanager this, const device device )
{
  int ret_val;
  int l_port = -1;

  int shortest_queue = 99;
  int module_number = -1;
  int i = 0;

  // Lock the Bluetooth manager and select the Bluetooth module

  pthread_cleanup_push( btmanager_release_mutex, (void*)&this->bt_mutex );
  int mutex_result = pthread_mutex_lock( &this->bt_mutex );
  if( mutex_result != SUCCESS )
  {
    sysutil_log( LOG_WARN, 0, "Mutex lock failed: %s\n", strerror( errno ) );
  }

  for ( i = 0; i < this->module_count; i++ )
  {
    if ( this->free_slot_counts[i] - this->queue_lengths[i] <= 0 ) // no free slots
    {
      continue;
    }
    if ( this->queue_lengths[i] < shortest_queue )
    {
      shortest_queue = this->queue_lengths[i];
      module_number = i;
    }
  }

  if ( module_number != -1 )
  {
    this->queue_lengths[module_number]++;

    int port_candidate;

    // Assign a different port for devices in same module
    for( port_candidate = MIN_LOCAL_PORT;
         port_candidate <= MAX_LOCAL_PORT;
         port_candidate++ )
    {
      if( btmanager_is_reserved( this, module_number, port_candidate ) )
      {
        continue;
      }

      btmanager_reserve( this, module_number, port_candidate );
      break;
    }
    l_port = port_candidate;
  }

  // Done selecting the module, release mutex
  pthread_cleanup_pop( TRUE );
```

```
  if ( module_number == -1 )
  {
    sysutil_log( LOG_WARN, 0, "No space for device %d\n", device->id );
    errno = EBUSY;
    ret_val = ERROR;
    goto err;
  }

  // Begin creating the SDP connection

  device->dongle = module_number;
  device->l_port = l_port;

  // Determine local bt address
  bdaddr_t local_bt_addr;
  memset( &local_bt_addr, 0, sizeof( local_bt_addr ) );
  hci_devba( module_number, &local_bt_addr );

  // determine remote bt address
  bdaddr_t remote_bt_addr;
  str2ba( device->address, &remote_bt_addr );

  sysutil_log( LOG_EVENT,
               0,
               "Creating sdp connection from local device %d to remote device %d...\n",
               module_number,
               device->id );

  // Create session between local bt address and remote bt address
  sdp_session_t *session = NULL;

  // Lock the mutex of the Bluetooth module in question to avoid collision at hardware level
  pthread_cleanup_push( btmanager_release_mutex,
                        (void*)&this->dongle_mutexes[module_number] );
  int mutex_result = pthread_mutex_lock( &this->dongle_mutexes[module_number] );
  if( mutex_result != SUCCESS )
  {
    sysutil_log( LOG_WARN, 0, "Mutex lock failed: %s\n", strerror( errno ) );
  }

  // Store the R10 timestamp here
  router_print_time( device->R10 );

  // Try to connect
  int flags = 0;
  int attempts = 3;
  do
  {
start:
    session = sdp_connect( &local_bt_addr, &remote_bt_addr, flags );

    if( session )
    {
      sysutil_log( LOG_DEBUG,
                   0,
                   "Tried to connect from %d to %d: 0\n",
```

```
                module_number,
                device->id );
    break;
  }
  else
  {
    sysutil_log( LOG_DEBUG,
                0,
                "Tried to connect from %d to %d: %d\n",
                module_number,
                device->id,
                errno );
    sleep( 1 );
  }

  // Try again if we did not succeed and there is still attempts left
  if( session == NULL &&
    ( errno == ECONNRESET ||
      errno == EHOSTDOWN ||
      errno == ETIMEDOUT ||
      errno == ECONNABORTED
    ) && --attempts > 0 )
  {
    sysutil_log( LOG_DEBUG,
                0,
                "Retry (%s) from local device %d to remote device %d...\n",
                strerror( errno ),
                module_number,
                device->id );
    goto start;
  }
}
// We don't accept EBUSY, EMLINK or EAGAIN as valid failure, continue unconditionally
while ( session == NULL && ( errno == EBUSY || errno == EMLINK || errno == EAGAIN ) );

// Done connecting, release the module mutex
pthread_cleanup_pop( TRUE );

if( session == NULL )
{
  ret_val = ERROR;
  goto err;
}

// Begin searching the service. The service id is hard-coded and predefined

sdp_list_t *attrid, *search, *seq;
uint32_t range = 0x0000ffff;

attrid = sdp_list_append( 0, &range );
search = sdp_list_append( 0, &this->remote_uuid );

// Get a linked list of services
int attempts = 3;
int result;
do
```

```
{
  sysutil_log( LOG_EVENT, 0, "Listing services from device %d...\n", device->id );

  result = sdp_service_search_attr_req( session,
                                        search,
                                        SDP_ATTR_REQ_RANGE,
                                        attrid,
                                        &seq );
}
while( result != SUCCESS && --attempts > 0 );

if( result != SUCCESS )
{
  char* msg = "No service found from device %d\n";
  sysutil_log( LOG_EVENT, 0, msg, device->id );
  sysutil_log( LOG_WARN, 0, msg, device->id );
  sdp_close( session );
  ret_val = ERROR;
  goto err;
}

sdp_list_free( attrid, 0 );
sdp_list_free( search, 0 );

sysutil_log( LOG_EVENT, 0, "Searching services from device %d...\n", device->id );

int r_port = -1;

// Loop through the list of services
for( ; seq; seq = seq->next )
{
  sdp_record_t *rec = (sdp_record_t*)seq->data;
  sdp_list_t *access = NULL;

  // get the RFCOMM channel
  sdp_get_access_protos( rec, &access );

  if( access )
  {
    r_port = sdp_get_proto_port( access, RFCOMM_UUID );
  }
}

free( seq );
sdp_close( session );

if ( r_port == -1 )
{
  errno = EHOSTUNREACH;
  ret_val = ERROR;
  goto err;
}

// Begin opening the connection to service

int connected = 0;
```

```
  int bt_socket = -1;
  attempts = 3;
  bool first = TRUE;
  do
  {
retry:
    if ( bt_socket != -1 )
    {
      close( bt_socket );
    }

    if( ( bt_socket = socket( PF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM ) ) == ERROR )
    {
      sysutil_log( LOG_WARN, 0, "Failed to create socket for device %d\n", device->id );
      errno = EHOSTUNREACH;
      ret_val = ERROR;
      goto err;
    }

    int opt;

    opt = 0;
    if( setsockopt(bt_socket, SOL_RFCOMM, RFCOMM_LM, &opt, sizeof(opt)) != SUCCESS )
    {
      sysutil_log( LOG_WARN, 0, "Failed to set socket options: %s\n", strerror( errno ) );
      goto err;
    }
    opt = 0;
    if ( setsockopt(bt_socket, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt) ) != SUCCESS )
    {
      sysutil_log( LOG_WARN, 0, "Failed to set socket options: %s\n", strerror( errno ) );
      goto err;
    }

    struct sockaddr_rc rem_addr, loc_addr;

    memset( &loc_addr, 0, sizeof( loc_addr ) );
    loc_addr.rc_family = AF_BLUETOOTH;

    btmanager_reversebt( &local_bt_addr );

    baswap( &loc_addr.rc_bdaddr, &local_bt_addr );

    loc_addr.rc_channel = l_port;

    if( bind( bt_socket, (struct sockaddr *) &loc_addr, sizeof( loc_addr ) ) != SUCCESS )
    {
      sysutil_log( LOG_WARN, 0, "Failed to bind socket for device %d\n", device->id );
      ret_val = ERROR;
      close( bt_socket );
      goto err;
    }

    memset( &rem_addr, 0, sizeof( rem_addr ) );
    rem_addr.rc_family = AF_BLUETOOTH;
    baswap( &rem_addr.rc_bdaddr, strtoba( device->address ) );
```

```
  rem_addr.rc_channel = r_port;

  if( first )
  {
    sysutil_log( LOG_EVENT, 0, "Connecting to service at device %d...\n", device->id );
    first = FALSE;
  }
  else
  {
    sysutil_log( LOG_DEBUG,
                 0,
                 "Retry (%s) to service at device %d...\n",
                 strerror( errno ),
                 device->id );
  }

  // Connect to service
  connected = connect( bt_socket, (struct sockaddr *)&rem_addr, sizeof(rem_addr) );

  if( connected != SUCCESS && ( errno == EBUSY || errno == EAGAIN ) )
  {
    goto retry;
  }
}
while ( connected != SUCCESS && --attempts > 0 );

if( connected != SUCCESS )
{
  sysutil_log( LOG_DEBUG,
               0,
               "connected=%d attempts=%d id=%d\n",
               connected,
               attempts,
               device->id );

  close( bt_socket );
  ret_val = ERROR;
  goto err;
}

// Store the R20 timestamp here
router_print_time( device->R20 );

// Lock the Bluetooth manager mutex and update program state
pthread_cleanup_push( btmanager_release_mutex, (void*)&this->bt_mutex );
int mutex_result = pthread_mutex_lock( &this->bt_mutex );
if( mutex_result != SUCCESS )
{
  sysutil_log( LOG_WARN, 0, "Mutex lock failed: %s\n", strerror( errno ) );
}

this->free_slot_counts[module_number]--;

// Updated, release the mutex
pthread_cleanup_pop( TRUE );
```

```
  device->socket = bt_socket;
  device->r_port = r_port;

  ret_val = SUCCESS;

err:

  if ( module_number != -1 )
  {
    // Lock for cleanup update
    pthread_cleanup_push( btmanager_release_mutex, (void*)&this->bt_mutex );

    int mutex_result = pthread_mutex_lock( &this->bt_mutex );
    if( mutex_result != SUCCESS )
    {
      sysutil_log( LOG_WARN, 0, "Mutex lock failed: %s\n", strerror( errno ) );
    }

    this->queue_lengths[module_number]--;
    if( ret_val == ERROR )
    {
      btmanager_unreserve( this, module_number, l_port );
    }

    // Cleanup done, release
    pthread_cleanup_pop( TRUE );
  }

  return ret_val;
}
```