Topology and Position Estimation in Bluethooth Ad-hoc Networks

Diploma Thesis
Urs Frey

Supervisors
Jan Beutel, Matthias Dyer

Winter Semester 2002/2003, DA-2003-09

TIK Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Position awareness can be valuable extension for mobile devices. GPS is not always suitable for that purpose. Distributed algorithms that use range estimates between nodes can provide another solution to the positioning problem. The goal of this thesis was to study positioning algorithms and implement such a system for a Bluetooth ad-hoc network.

The BTnodes, a Bluetooth enabled small-scale prototyping platform were used as network nodes. As a part of the project, service routines for Bluetooth ad-hoc networks were developed. This included a Bootloader that allows selective network flooding for reprogramming remote devices. xHop, a multihop protocol, was implemented to make remote configuration and debugging possible. Experiments have been carried out to analyse xHop performance in a Bluetooth network. The results showed a large improvement over earlier implementations.

Finally, the distributed positioning algorithm Hop-TERRAIN has been implemented for the BTnodes. It showed that a good scheduling of Bluetooth connection establishment plays a crucial role for the *Refinement* phase of Hop-TERRAIN. A way to achieve such a scheduling is proposed.

Contents

1 Introduction **1**

2 Triangulation **3**
2.1 Angles 4
2.2 Distances 4
2.3 Proposals of Positioning Algorithms 6
2.3.1 Grid Based 6
2.3.2 Convex Position Estimation 6
2.3.3 Local Coordinate Systems 7
2.3.4 Other Related Work 7
2.4 Hop-TERRAIN 8
2.4.1 Start-Up Algorithm 8
2.4.2 Refinement Algorithm 9

3 Platform: BTnode **13**
3.1 Overview 13
3.2 ATmega128 14
3.2.1 IO interfaces 14
3.2.2 Memory Sections 14

4 Bluetooth **18**
4.1 Overview 18
4.2 Specific Bluetooth Terminology 19
4.3 Protocol Layers 20
4.4 Distance Measurement 23
4.5 Bluetooth Ad-Hoc Network 23

5 Software Implementation **26**
5.1 BTnode System Software Stack 26
5.2 Application as a Command Line Interface 27
5.3 Application Specific Protocol Layers 27
5.4 Connection Manager 28
5.4.1 Inquiry Scheduler 29
5.4.2 Packet Forwarding and Connection Management 29

5.5 RDSR Routing in Ad-Hoc Networks 31
5.5.1 xHop Packet Format and Commands 32
5.5.2 Results 34
5.6 Bootloader Application 34
5.6.1 Intel Hex File 36
5.6.2 Selective Network Flooding 37
5.6.3 Reprogramming the Flash 39
5.6.4 Results 40
5.7 Positioning with Hop-TERRAIN 40
5.7.1 Start-up Phase 40
5.7.2 Refinement Phase 41
5.7.3 Results 43
5.7.4 BTerrain 44

6 Conclusion **50**

A Source Files **52**

B Glossary **53**

References 54

List of Figures

2.1 Node types . . . 3
2.2 Triangulation with angles . . . 4
2.3 Triangulation with distances . . . 4
2.4 Grid based Positioning . . . 7
2.5 Bounding boxes . . . 8
2.6 Building up local coordinate systems . . . 9
2.7 *Start-up* phase, broadcasting anchor positions . . . 10
2.8 *Start-up* phase, triangulation . . . 11
2.9 *Refinement* phase . . . 12
2.10 Sound nodes . . . 12

3.1 BTnode hardware . . . 13
3.2 UART ports . . . 14
3.3 Memory sections . . . 15
3.4 Section placement . . . 17

4.1 Hardware partitioning . . . 21
4.2 Upper protocol layers . . . 22
4.3 Connection types . . . 24
4.4 Network consisting of dumbbell connections . . . 25

5.1 BTnode System Software . . . 27
5.2 Custom layers on top of L2CAP . . . 28
5.3 Connection manager . . . 30
5.4 Pending packets structure . . . 31
5.5 xHop example . . . 32
5.6 xHop packet format . . . 32
5.7 Experiment setup . . . 34
5.8 xHop per-hop-delay . . . 35
5.9 Bootloader data in SRAM . . . 37
5.10 First Bootloader hop . . . 38
5.11 Bootloader init command . . . 38
5.12 Bootloader data command . . . 38
5.13 `CMD_ANCHOR` . . . 41
5.14 `CMD_HT_ANCHOR` . . . 41

5.15 CMD_REF_REQ 41
5.16 CMD_REF_MEASURE 42
5.17 Answer for CMD_ANSWER_RSSI 42
5.18 Answer for CMD_ANSWER_POSI 43
5.19 *Refinement* experiment setup 44
5.20 Slot allocation arrays 45
5.21 Range - connectivity relationship 46
5.22 Allocation failures for 1.5, 2.0 and 2.5 seconds 46

Preface

I would like to thank Prof. Dr. L. Thiele that I was able to write this thesis in his research group at TIK/ITET, ETH Zurich.

I would also like to thank my supervisor Jan Beutel who introduced me to the MICS-NCCR project. He let me find may own way through the thesis and he could always help out when I could not see a solution to a problem. I would like to thank him that I was able to go with him to EPF Lausanne once for a BTnode demonstration. In February, an MICS-NCCR meeting was hold in Zurich were I could attend. Apart from several interesting talks, I got an insight into the organisation structure of such a national research project.

At the institute of Prof. Dr. F. Mattern at the Computer Science Department, I would like to thank Oliver Kasten for his introduction to the BTnode system software and for many useful software engineering tips. I would also like to thank Matthias Ringwald; we had several interesting discussions about the BTnodes that let me see the problems from a different perspective and were good inspiration for new ideas.

I would also like to thank Philipp Scherler who assisted me during a memory debugging session and when there were some hardware fixes to do. Thanks also to the Service group at TIK for the good organisation of the thesis and their administrative support.

Finally, I would like to thank my parents for their love and support during my time as a student at ETH Zurich.

Urs Frey *Zurich, 14th March 2003*

Chapter 1

Introduction

A large demand for wireless communication anywhere and anytime arose in the last few years, as small, battery-powered devices got available. One mean to support such communication will be through mobile ad-hoc networks. These networks are systems of autonomous mobile nodes, connected by wireless links that are free to move randomly and to organize themselves arbitrarily.

It will often be important for the devices to know about their position and the topology of their surrounding neighbours. Many applications can make use of such information. In sensor networks, this information can be used to tag the measured data with the corresponding position. For routers in such ad-hoc fashioned networks, topology and position information can be used to forward incoming packets. On mobile phones and other handheld devices, this information could be provided to the user or applications running on them. This is just to mention a few.

One possibility to obtain position information is to use the satellite based GPS [1] or mobile phone GSM network. However, GSM may not provide accurate enough position estimates and GPS cannot easily be used inside buildings. Another reason that may prevent the use of GPS is the additional cost occurring for receivers.

In wireless ad-hoc networks, there is another possibility to achieve position awareness. The communication links between two devices provide some geometrical information that can be used to triangulate a position estimate.

The goal of this thesis was to implement such a distributed positioning algorithm for a Bluetooth ad-hoc network. The BTnodes, a small-scale Bluetooth prototyping platform, equipped with an AVR microcontroller, were used as mobile network nodes.

The main part of the work included implementing service functions to maintain large BTnode networks. A connection manager was written to support the chosen Bluetooth centred topology of the network. To allow configuring remote nodes that are out of range from the main development node, xHop has been implemented. xHop allows multihopping along a predefined route and it provides a mean to execute commands on remote nodes. With xHop, it got possible to configure remote nodes without attaching them to a computer and it made reading out status information feasible too.

Developing code and testing it within a BTnode network was rather cumbersome

at the beginning as prior to executing the code on several nodes, every single node had to be collected and attached to the programming interface to download the new code. Therefore, a Bootloader was implemented that allows to remotely reprogramming a BTnode network with one single command at the developing station.

With support of the these service functions, Hop-TERRAIN, a distributed positioning algorithm, was then implemented for the BTnodes. Some experimentation revealed that a good scheduling is required for fast position updates in a Bluetooth network.

Chapter 2 introduces the concept of positioning in wireless networks. It describes a few positioning possibilities, before Hop-TERRAIN is explained in detail. Chapter 3 then presents the BTnode platform with focus on the memory sections supplied by the AVR and the node. The wireless technology Bluetooth employed for the ad-hoc network is introduced in Chapter 4. Finally, Chapter 5 explains the implemented service and positioning functions before Chapter 6 concludes the report.

Chapter 2

Triangulation

Triangulation is a term coined by land survey and geodesy. If we can measure the angles of a triangle, then we can use simple trigonometric laws and calculate the length of the vertices and the coordinates of the edges. The term is so widely used for any positioning method, that it will be used throughout this report to describe any kind of positioning, even if it is not based on angles in a triangle.

Any location discovery approach is based on two phases: a measurement or data acquisition phase followed by a phase that combines the measured data [2].

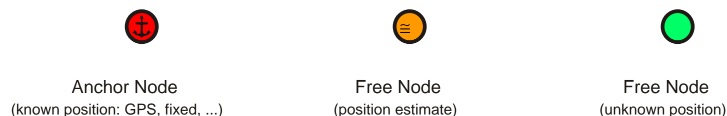


Figure 2.1: Node types

The following positioning figures differentiate between three types of nodes, see Figure 2.1. Anchor (or beacon) nodes have a known location, which they obtain by different means then through the ad-hoc network, e.g. by using GPS or a fixed position. Free nodes start a priori without a known location and can only obtain a position through one of the triangulation method discussed below. The free nodes, shaded orange, already calculated their positions and can therefore be used in following iterations to derive triangulation solutions. Typically, these nodes store a value that states how accurate their position estimate is. The green ones symbol free nodes without any position estimate and are therefore called unknown nodes.

Section 2.1 and 2.2 describe triangulation methods for the two categories of geometric constraints; angles and distances. Following, several positioning algorithms are presented, before the one chosen for implementation on BTnodes is described in Section 2.4.

2.1 Angles

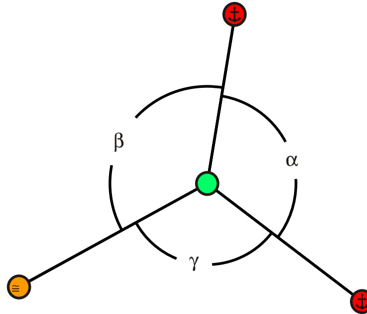


Figure 2.2: Triangulation with angles

Angles are used by Angle of Arrival (AoA) based systems and can for example be measured with antenna arrays. However, such systems are usually considerably more expensive then the ones described below. When using angles, the second phase typically employs simple laws of trigonometry to calculate position estimates.

2.2 Distances

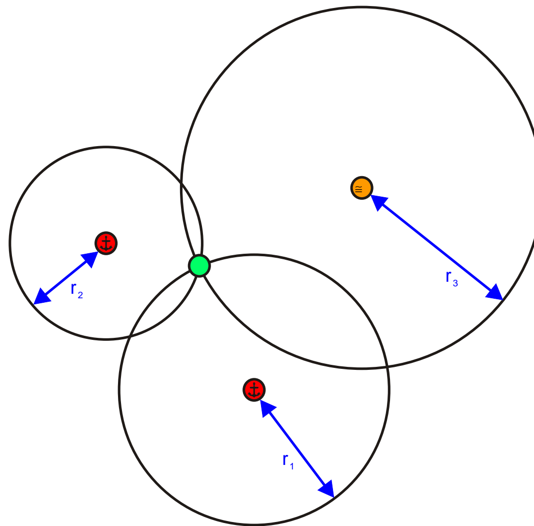


Figure 2.3: Triangulation with distances

Distance estimations can be retrieved in several ways. Measuring the Received

Signal Strength Indicator (RSSI) provides a rough estimation of the distance between a sender and a receiver node. Based on a known transmit power and propagation loss a distance can be calculated. This method is mainly used for RF signals. Time based methods like Time of Arrival (ToA) and Time Difference of Arrival (TDoA) use the known propagation speed of the signal to calculate a distance. These methods can be applied to many different signals, such as RF, acoustic, infrared and ultrasound ones, but they require fast and precise signal processing.

Once the data acqusition phase is over, distances can be combined to a position estimate with one of the following approaches. Hyperbolic triangulation is the intuitive method of intersecting circles (2D) or spheres (3D) to obtain a position. In two dimensions, the minimum of distance measurements required is three, in three dimensions it is four. If there are more measurements available then the minimum required, a Maximum Likelihood estimation is typically applied. The position is calculated such that the differences between the measured distances and the distances from the estimated position to the known nodes are minimised. As shown in [3] this can be done by using a traditional least squares algorithm.

The equation system to be solved for n reference points in 3D looks like:

$$\begin{bmatrix} (x_1 - u_x)^2 + (y_1 - u_y)^2 + (z_1 - u_z)^2 \\ (x_2 - u_x)^2 + (y_2 - u_y)^2 + (z_2 - u_z)^2 \\ \vdots \\ (x_n - u_x)^2 + (y_n - u_y)^2 + (z_n - u_z)^2 \end{bmatrix} = \begin{bmatrix} r_1^2 \\ r_2^2 \\ \vdots \\ r_n^2 \end{bmatrix} \tag{2.1}$$

With $(x_i,\, y_i,\, z_i)$ representing the three dimensional coordinates of the ith reference point and $(u_x,\, u_y,\, u_z)$ those of the unknown node, for which the triangulation is performed. r_i are the distances between the unknown node and the corresponding reference points and e_i the error which is minimized. The system can be linearised by subtracting the last row. The following relations results after some arithmetic shuffling:

$$Ax = b + e \tag{2.2}$$

with

$$A = -2 \begin{bmatrix} (x_1 - x_n) & (y_1 - y_n) & (z_1 - z_n) \\ (x_2 - x_n) & (y_2 - y_n) & (z_2 - z_n) \\ & \vdots & \\ (x_{n-1} - x_n) & (y_{n-1} - y_n) & (z_{n-1} - z_n) \end{bmatrix} \tag{2.3}$$

$$u = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} \tag{2.4}$$

$$b = \begin{bmatrix} r_1^2 - r_n^2 - x_1^2 + x_n^2 - y_1^2 + y_n^2 - z_1^2 + z_n^2 \\ r_2^2 - r_n^2 - x_2^2 + x_n^2 - y_2^2 + y_n^2 - z_2^2 + z_n^2 \\ \vdots \\ r_{n-1}^2 - r_n^2 - x_{n-1}^2 + x_n^2 - y_{n-1}^2 + y_n^2 - z_{n-1}^2 + z_n^2 \end{bmatrix} \tag{2.5}$$

and the solution:

$$\widehat{x} = \left(A^T A\right)^{-1} A^T b \tag{2.6}$$

This is a system with three unknowns and therefore at least three rows are required for a unique solution. We have $n-1$ rows for n measurements; consequently, at least four measurements are required. Often it will be of use to assign a weight to each measurement. For uncorrelated measurements, this will result in a diagonal matrix C with the measurements' weights in the corresponding rows. As the last row was subtracted for linearisation, its weight is implicitly set to 1. The solution of the weighted system is:

$$\widehat{x} = \left(A^T C A\right)^{-1} A^T C b \tag{2.7}$$

which typically involves a QR decomposition using Gram-Schmidt or the Cholesky algorithm [4].

2.3 Proposals of Positioning Algorithms

This section will present some of the extisting positioning concepts. The algorithm that was finally chosen to be implemented on the BTnodes is called Hop-TERRAIN and is explained in more detail in Section 2.4.

2.3.1 Grid Based

The GPS-less system [5] is an RF-based positioning method that requires a number of nodes to be placed at known positions that form a regular mesh and transmit periodic beacon signals containing their respective positions. This algorithm does not need any accurate distance measurements; it only uses the information if a node is in range or out of range of a certain beacon node, see Figure 2.4. It has the advantage of simplicity, scalability and low power consumption. However, it is coarse grained and needs a considerable amount of infrastructure.

2.3.2 Convex Position Estimation

The authors of [6] propose a method based exclusively on connectivity-induced constraints. Know peer-to-peer communication in the network is modelled as a set of geometric constraints on the node positions.

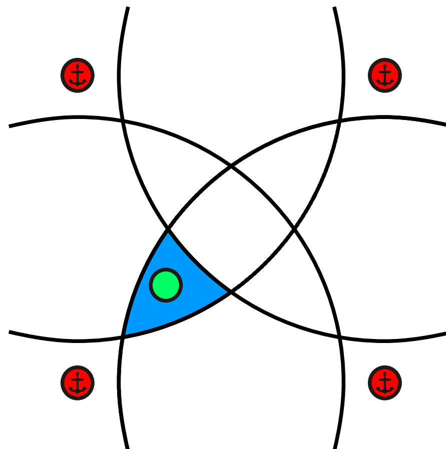


Figure 2.4: Grid based Positioning

The geometric constraints can be of an angular (node in/out of sight for e.g. optical transmission) or radial nature (in/out of range for RF, ultrasound or similar systems). The problem is solved with a Linear Program (LP), evaluating every single geometric constraint. Additionally, a method for placing a rectangular bound around the possible positions is presented. Using LP to solve the problem has the advantage of getting an exact solution, or an exact bounding box, see Figure 2.5.

2.3.3 Local Coordinate Systems

The algorithm presented in [7, 8] uses distance measurements to estimate positions, see Figure 2.6. As long as there are no anchor nodes in view, unknown nodes build up their own local coordinate systems. The first unknown node just assumes to be on position (0, 0, 0), a second will be placed on the positive x axis at position (r, 0, 0). When a first anchor node comes in view of the locally built coordinate system, a translation of the local system will be required. For a second anchor a rotation and eventually a flip for the third (in 3D a second rotation will follow before an eventual flip) is required.

2.3.4 Other Related Work

The RADAR system [9] is an RF based system for indoor localisation, where a few base stations will measure signal strength and a centralised server localises the nodes based on signal strength maps. The Cricket location support system [10] uses ultrasound instead of RF signals. Fixed beacons inside buildings distribute geographic information. Another ultrasound system is BAT [11], which uses fixed receivers to pick up signals from mobile nodes. AHLoS [2] is system using ultrasound and RF and does not rely on an infrastructure setting.

A positioning approach for ad-hoc networks is taken in the PicoRadio project at

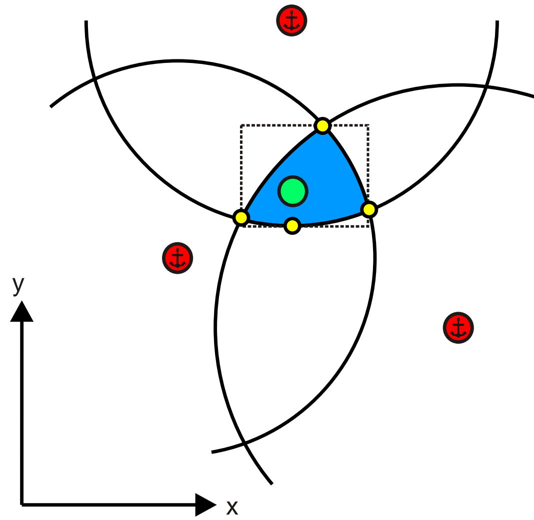


Figure 2.5: Bounding boxes

UC Berkley [12]. It provides a geolocation scheme for an indoor environment, based on RF received signal strength measurements and pre-calculated signal strength maps. Another algorithm within the PicoRadio project is Hop-TERRAIN, which is explained in the following section.

2.4 Hop-TERRAIN

Hop-TERRAIN [13] is a distributed ad-hoc algorithm proposed at the Berkeley Wireless Research Center. The decision to implement Hop-TERRAIN on the BTnodes (see Chapter 3) was based on the algorithm's simplicity, the absence of any infrastructure and its distributed nature. This section explains the algorithm, implementation details and results can be found in Section 5.7.

Hop-TERRAIN is a two-stage algorithm. The first phase is called *Start-up* phase and addresses the problem of sparse anchor nodes. The second, called *Refinement* phase, will then improve the inaccurate initial position estimates.

2.4.1 Start-Up Algorithm

The purpose of the *Start-up* phase it to solve the sparse anchor problem, which comes from the need for at least four reference points with known locations in a three-dimensional space in order to uniquely determine the location of an unknown object. Too few reference points result in ambiguities that lead to underdetermined systems of equations. Usually, most nodes will start without known locations and only a few randomly distributed anchors will exist. It is therefore highly unlikely that any randomly

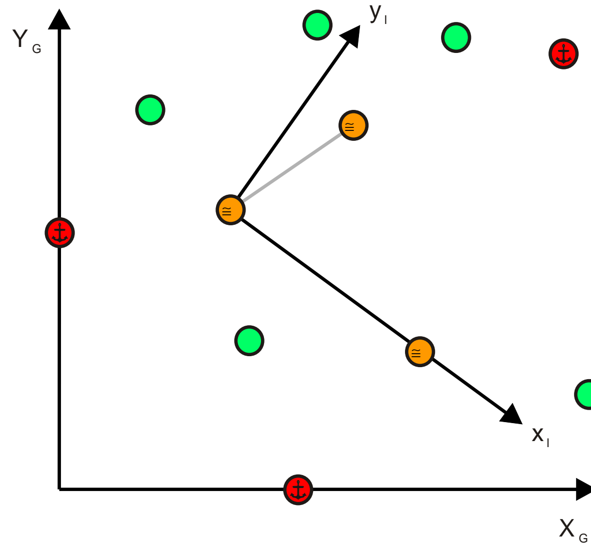


Figure 2.6: Building up local coordinate systems

selected node in the network will be in direct range with a sufficient number of anchor nodes to derive its own position. Hop-TERRAIN solves this problem by trading off accuracy for consistency. The *Start-up* phase will provide rough guesses of the nodes' initial positions. It is shown in [13] that this is enough as an input to the second phase for refining the position estimates.

The algorithm works as follows: At large time intervals, each of the anchor nodes launches the Hop-TERRAIN algorithm by initiating a broadcast containing its known location and a hop count of 0. All of the one-hop neighbours surrounding the anchor will record the anchor's position and a hop count of 1. Then they perform another broadcast containing the anchor's position and a hop count of 1. This process continues until each anchor's position and an associated hop count value have been spread to every node in the network, see Figure 2.7. It is important that nodes receiving these broadcast packets only store and rebroadcast a certain anchor's position if they have not received such a packet with the same or smaller hop count before.

Once a node has received data regarding at least four (3D) anchor nodes, it is able to perform a triangulation to estimate its location, see Figure 2.8. This will of course only be a very rough estimation of the actual positions.

2.4.2 Refinement Algorithm

With the initial position estimates of Hop-TERRAIN in the *Start-up* phase, the objective of the *Refinement* phase is to obtain more accurate positions, using the estimated ranges between nodes, see Figure 2.9.

Refinement is an iterative algorithm in which the nodes update their positions in

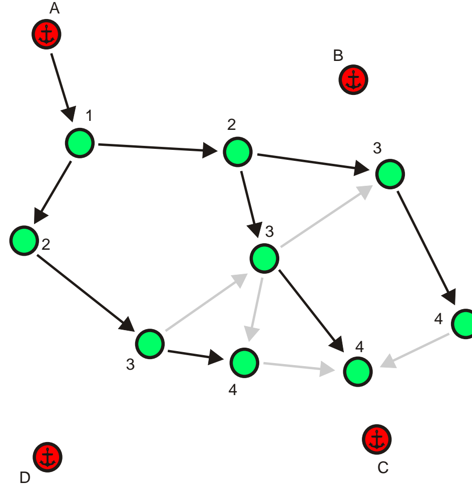


Figure 2.7: *Start-up* phase, broadcasting anchor positions

a number of steps. At the beginning of each step, a node broadcasts its position estimate, receives the positions and corresponding range estimates from its neighbours and computes a least squares triangulation solution to determine its new position. Often the constraints imposed by the measured distances will force the new positions towards the true location of the node. *Refinement* stops and reports the final result once updates become small.

Without any prevention, the large errors induced by RSSI measurements will propagate fast throughout the network. Therefore, a confidence metric was included in the *Refinement* algorithm. Instead of solving the unweighted least squares (equation 2.6) the weighted version is solved (equation 2.7). Each node assigns a confidence weight between 0 and 1 to its position estimate. Anchors immediately start with a confidence value of 1. Unknown nodes start with a low value (0.1) and may raise their confidence after subsequent *Refinement* iterations. Whenever a node performs a successful triangulation, it sets its confidence level to the average of its neighbours levels. In general, this will raise the confidence level. It is shown in [13] that including confidence levels improved the *Refinement* phase considerably.

A second improvement to the *Refinement* phase was necessary in order to detect ill-connected groups of nodes.

Detecting that a single node is ill-connected is easy: if the number of neighbours is less then four in 3D (three in 2D) then the node is ill-connected. However, detecting that a group of nodes is ill-connected is more complicated, since some global perspective is necessary. A heuristic is employed that operates in an ad-hoc fashion, yet is able to detect most ill-connected nodes. The underlying premise for the heuristic is that a sound node has independent references to at least four in 3D (three in 2D) anchors. That is, the multi-hop routes to the anchors have no link in common. For example,

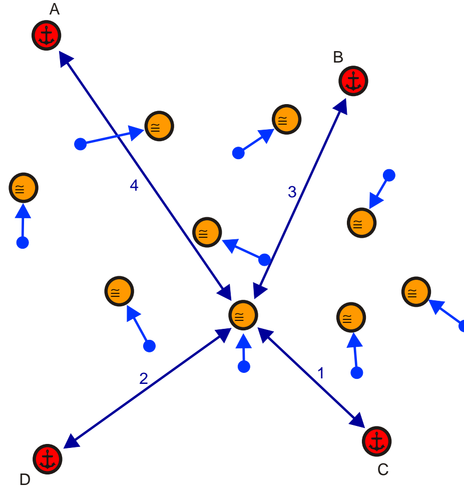


Figure 2.8: *Start-up* phase, triangulation

node 1 in Figure 2.10 meets these criteria and is considered sound (2D). However, node 2 does not.

In the *Start-up* phase, the Hop-TERRAIN algorithm floods the anchors positions through the network and nodes record the hop count of the shortest path to each anchor. Hop-TERRAIN also records the neighbour IDs on the shortest path. These IDs are collected in a set of potentially sound neighbours. When the size of this set reaches four in 3D (three in 2D) a node declares itself sound and may enter the *Refinement* phase. The neighbours of the sound node add its ID to their sets and may in turn become sound, etc.

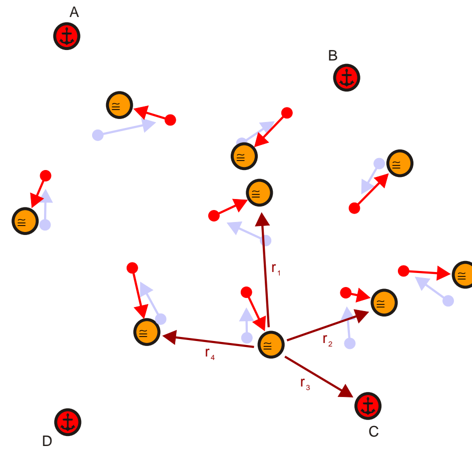


Figure 2.9: *Refinement* phase

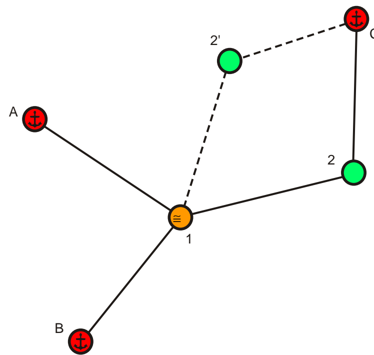


Figure 2.10: Sound nodes

Chapter 3

Platform: BTnode

3.1 Overview

The BTnodes [14] are a mobile Bluetooth platform developed within the NCCR-MICS [15] project with the intention to prove several aspects of mobile ad-hoc computing in practice.

Figure 3.1: BTnode hardware

The platform is equipped with an Ericsson ROK Bluetooth module [16], an ATmega128 [17] as a host controller, an external SRAM memory and a few components for power management, see Figure 3.1. The choice of Bluetooth as a radio interface makes the BTnodes available for a wide range of applications, as it is an industrial standard. On the local side, the ATmega offers several general-purpose interfaces to connect the node to sensors, displays and other hardware. Otherwise, the node is intentionally kept simple.

There are two main advantages of using such a platform to prove mobile ad-hoc concepts. Firstly, compared with a laptop computer equipped with a radio interface, such a platform is *really* mobile. Battery powered, it can operate standalone for several days. As a proof of concept, it can be attached to everyday items, like clothes, door locks, teacups [18], egg boxes [19] and so one. Secondly, it can be deployed in large quantities, due to a substantial lower cost compared with laptop or handheld devices.

3.2 ATmega128

The ATmega128 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. By executing instructions in a single clock cycle, the ATmega128 achieves throughputs approaching 1 MIPS per MHz.

The ATmega128 features several different memories introduced in Section 3.2.2, a instruction set of 133 instructions, 32 8-bit general purpose registers, a 2-cycle multiplier, several timers and counters, a real time counter (RTC) with separate oscillator, two 8-bit PWM channels, six sleep modes and it supports JTAG. Among the peripheral interface are some standard IO interfaces explained in Section 3.2.1. The chip operates at 2.7V to 5.5V and at a speed up to 8MHz.

3.2.1 IO interfaces

The ATmega has several IO interfaces apart from other general-purpose IO pins. The chip has a JTAG interface, a byte oriented two-wire serial interface, two programmable serial UARTs, a master/slave SPI interface and several ADC inputs.

For the BTnode the two serial UARTs are of great importance. One is used to communicate with the Ericsson Bluetooth module and serves as a programming interface during Flash reprogramming. The other UART is not used by the node itself and can be used for applications. The positioning application uses this port as a debug and control interface, see Figure 3.2. Status and debug information are written to the serial port and can e.g. be displayed with `minicom` under Linux. In the other direction, commands can be sent to the nodes through this port. The application is therefore implemented as a command line interface.

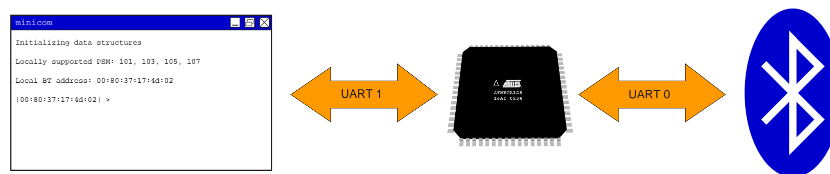


Figure 3.2: UART ports

3.2.2 Memory Sections

The AVR architecture has two main memory spaces, the data memory and the program memory. In addition, the ATmega128 features a 4kB EEPROM memory for non-volatile data storage. On the BTnode, the memory structure is slightly extended with an external 256kB SRAM. Figure 3.3 shows the basic structure, Figure 3.4 shows the section placement and Table 3.1 lists the main features of the different memory sections.

The Flash memory is exclusively used to store program data, like instructions and constants. The Flash can be self reprogrammed, which is used for the Bootloader, see

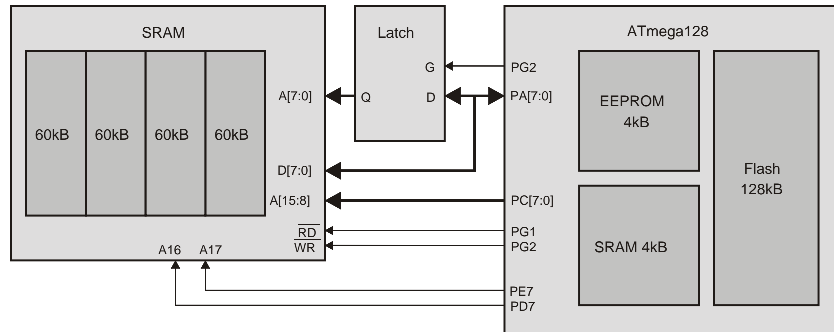


Figure 3.3: Memory sections

Section 5.6. The EEPROM can be used to store features associated with a certain BTnode. The EEPROM is a non-volatile memory; therefore, the data will still be available after power has been switched off and on again. The advantage of having an EEPROM on board is the possibility to store a feature, which may differ from node to node individually and load an identical program to all nodes. The appropriate code sections are then executed depending on the features stored in the EEPROM. For the positioning application this is used to store whether a node is an anchor or not and if it is a fixed anchor its position will be stored in the EEPROM as well.

The SRAM deserves some special attention. The ATmega128 comes with an internal SRAM of 4kB, whereof the first 256 addresses are assigned special functions and mapped to registers. In addition, the architecture features an external memory interface, which drives the ports A, C and G when enabled. The processor will automatically access the external SRAM when instructions access data with addresses greater or equal 0x1100. Addresses are 16 bit wide; therefore, the maximal supported external RAM is 64kB, whereof the first 4kB will never be accessed as those addresses are mapped to the internal RAM. However, most BTnodes are equipped with a 256kB external SRAM. To give access to the additional 180kB, not addressable via the standard interface, two general purpose I/Os have been used. It can be used with an extended memcopy routine `xcopy`, partly written in assembler, which controls the two additional I/O lines. Addresses are 32 bit wide and span over a range from zero to about 180kB.

To make efficient use of the memory on bank 0, the linker can be told where to place which section [20]. There are four different sections, which are important for memory placement. The first one is the stack, that stores function calls, local variables and return addresses. It is important to keep this section within the internal memory, as it is used frequently and a fast access is important. The *.data* section is where initialised variables are placed. *.bss* contains uninitialised global or static variables. These variables will however be set to zero on start up if they are not explicitly declared as *.noinit*. The last section is the heap, which is only important when using `malloc()` and `free()`. As using those functions is tricky on microcontrollers they are not used in this project.

Memory	Size	Reset	Power Down	Access Time	Write Time	Write Cycles
Registers	32b	overwritten	volatile	1 cycle	1 cycle	unlimited
SRAM intern (Stack)	4kB	initialised	volatile	2 cycles	2 cycles	unlimited
SRAM extern (Data section)	60kB	initialised	volatile	3 cycles	3 cycles	unlimited
SRAM extern (additional banks)	180kB	retain data	volatile	xcopy (~20)	xcopy (~20)	unlimited
Flash	128kB	retain data	non-volatile	instruction 1 cycle, data 3 cycles	SPM (per page ~4ms)	100 cycles
EEPROM	4kB	retain data	non-volatile	CPU halted for 4 cycles	CPU halted for 2 cycles (write time 8.5ms)	100000 cycles

Table 3.1: Memory sections

It is important to know what code is executed after a reset, but prior to the jump to the `main()` routine. Without modifications, the stack is initialised first, then data is copied from the flash to the *.data* section and the *.bss* is zeroed out, before execution of `main()` starts. When using an external RAM, the access interface must be enabled before data is initialised, which means it should be done in *.init0*. For a 256kB RAM as on the BTnodes, care must also be taken to set the additional control lines early enough.

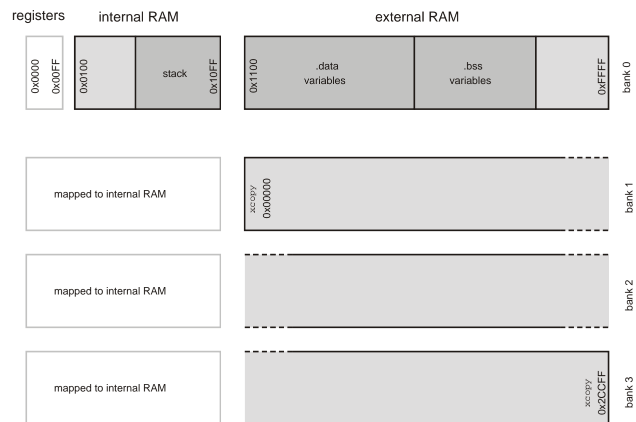


Figure 3.4: Section placement

Chapter 4

Bluetooth

4.1 Overview

New technologies and application are now emerging that use wireless communication. The IEEE 802.11 standard as the accepted choice for the networking community supports handovers and high data rates. However, this technology is expensive and therefore not compatible with price-conscious consumer products. Therefore, Bluetooth [21] was designed to provide a mean to create wireless, low power, cost-effective and ad-hoc connectivity between all kinds of devices.

Bluetooth provides a standard radio interface for short distance communication. It can be used for numerous applications, like PC peripherals, headsets, handhelds, door locks, just to name a few. Bluetooth operates in the 2.4GHz band, known as the Industrial Scientific and Medical (ISM) band. It is free for unlicensed use in most parts of the world, however this freedom has a price, many other technologies use this band as well: 802.11b, Home RF, some DECT variants and others [22]. Additionally, Bluetooth is subject to interference from a variety of other sources: microwave ovens, sodium lights, thunderstorms, overhead cables, communication in other bands (GSM, CDMA) and poorly suppressed engines. To achieve a degree of robustness to interference Bluetooth utilises a frequency-hopping scheme: Frequency Hopping Spread Spectrum (FHSS). During a connection, devices hop 1600 times a second, which ensures that packets subject to interference can be retransmitted on a different frequency. Though providing robustness, this hopping results in longer times to build up a connection, as devices have to meet each other on the same frequency and synchronise prior to establishing a connection.

The process of discovering other devices is called inquiry. The inquiring device will then hop with double speed and therefore meet the surrounding devices after a while. When establishing a connection the devices will synchronise to the pseudo random hopping scheme of the master device, which is based on the master's Bluetooth address.

4.2 Specific Bluetooth Terminology

- **Inquiry and Inquiry Scanning**
  To discover other nearby devices, a Bluetooth device conducts an *inquiry.* It is possible to issue single inquiry commands or to tell the module to conduct a periodic inquiry.
  The listening mode for inquiry is called *inquiry scan.* Only devices in inquiry scan will respond to inquiries. It is possible to hide from other devices by not enabling inquiry scan. Inquires or inquiry scans can be carried out as an unconnected device, a master or a slave. However, a slave's responsibility to regularly listen for master transmissions means it will not be able to devote as much of its time to the procedure.
  In the current BTnodes Bluetooth stack operation mode and with the ROK modules, slaves do not perform inquiry scan and are therefore not reported to inquiring devices. A device that is performing an inquiry will not simultaneously perform an inquiry scan and is therefore not discoverable either.

- **Paging and Page Scanning**
  To create a connection between Bluetooth devices one device *pages* another device, which must be in *page scan* to respond. A successful page (connection creation) results in an ACL connection between the paging device, which, by default, becomes the master and the paged device, the slave. A connection handle is returned to the upper stack layers.
  As devices need to synchronise, creating a connection requires some time. If the remote device's clock offset is known, it can be used by the paging device and connection time can be as little as 4 ms. However, when not in a link, device offsets drift. The longer the elapsed time since the last connection between two devices, the less accurate the offset information. It will take longer to connect next time. If one device has been powered off and on between connections, the offset information is useless, no better than a random guess. The theoretical worst-case duration for a page is just over five seconds. Interference or the presence of SCO links may extend this time. The default timeout time is 5.12 seconds.

- **ACL** - Asynchronous Connection Less
  This is a low-level Bluetooth data connection.

- **SCO** - Synchronous Connection Oriented
  This is a low-level Bluetooth duplex voice connection. To set up a SCO connection, an ACL connection must be established first.

- **Hold Mode**
  A device in hold mode is temporarily inactive until a hold timer expires. A master might use hold mode to allow slaves to save power if it knows it will not communicate with them for a while, e.g. when it is connecting to a new slave.

- **Park Mode**

A device in park mode has given up the active member address that identifies it as a part of a piconet, see Section 4.5. It is inactive expect for occasional beacon slots when it wakes up to listen for unpark messages that can be used to reactivate it. Parked devices are allocated special access window slots in which they can request the master to reactivate them by unparking.

- **Sniff Mode**
  A low-power mode where a device only wakes up to listen for data in periodic sniff slots.

4.3 Protocol Layers

BTnodes use a hosted software stack implementation, which means that the lower stack resides on the ROK Bluetooth module and the upper stack layers are implemented in software that runs on a PC or on the AVR microcontroller. Implementing the upper layers in software offers the widest flexibility. The lower layers that format the over-air transmissions, handle error detection and re-transmission and manage the links between devices, reside in the ROK module. There is no access to those parts of the Bluetooth layers.

Figure 4.1 shows the chosen model. Section 5.1 explains the system software implementing the upper layers in more detail.

The Host Controller Interface (HCI) sits between the upper layers and the lower layers of the stack. The two most common physical transports are UART and USB, whereas the BTnodes use the UART mode.

Figure 4.2, which is based on [22], shows the upper Bluetooth protocol layers in more detail.

The layers shaded in light blue are described in the Bluetooth Specifications and the Bluetooth Profiles [21]. The boxes in dark blue symbolise layers that are common to most applications, however are not specified in the Specifications. The application part is shown in orange.

Following a short description of the most commonly used Bluetooth protocols and layers:

- **L2CAP**
  Logical Link Control and Adaptation Protocol multiplexes upper layer data onto the single Asynchronous Connectionless (ACL) connection between two devices and, in the case of a master device, directs data to the appropriate slave. It also segments and reassembles the data into chunks that fit into the maximum HCI payload. Locally, each L2CAP logical channel has a unique Channel Identifier (CID), although this does not necessarily match the CID used by the remote device. CIDs 0x0000 to 0x003F are reserved with 0x0000 being unused, 0x0001 carrying signalling information and 0x0002 identifying received broadcast data.
  The stack layers that sit above L2CAP can be identified by a Protocol Service Multiplexer (PSM) value. Remote devices request a connection to a particular

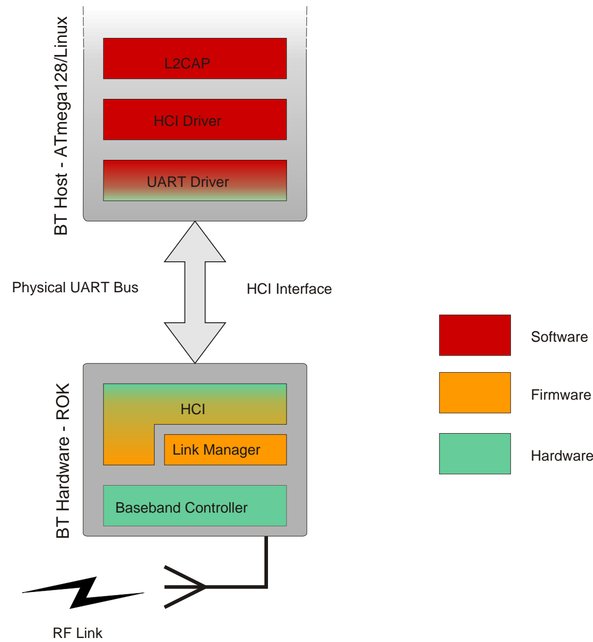


Figure 4.1: Hardware partitioning

PSM, and L2CAP allocates a CID. There may be several open channels carrying the same PSM. Each Bluetooth defined layer above L2CAP has its own PSM:

- SDP - 0x0001
- RFCOMM - 0x0003
- TCS-BIN - 0x0005
- TCS-BIN-CORDLESS - 0x0007

- **RFCOMM**
  RFCOMM emulates full 9-pin RS232 serial communication over an L2CAP channel. It is based on the TS 07.10 standard for a software emulation of the RS232 hardware interface. Version 1.1 of the Bluetooth Specification has added to the capabilities of the standard TS 07.10 specification by providing flow control capabilities.

- **OBEX**
  The Object Exchange standard (OBEX) was developed by the Infrared Data Association (IrDA) to facilitate operations common to IR-enabled devices.

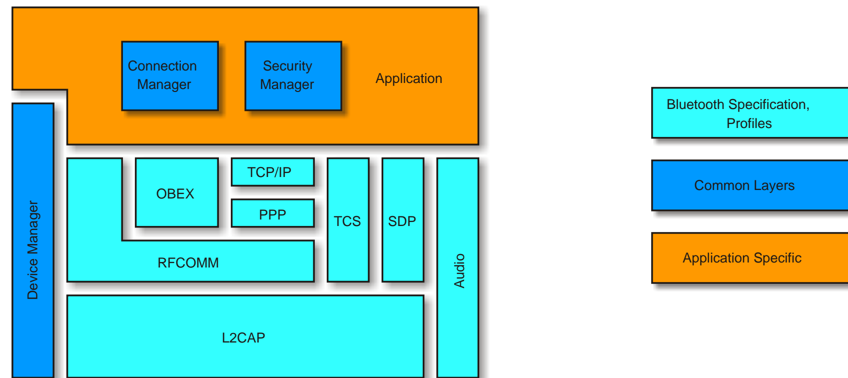


Figure 4.2: Upper protocol layers

OBEX allows users to put and get data objects, create and delete folders and objects, and specify the working directory at the remote end of the link.

- **PPP**
  The Point-to-Point Protocol (PPP) is the existing method used when transferring TCP/IP data over modem connections. The Bluetooth Specification reuses this protocol in the LAN Access Profile to route network data over an RFCOMM port.
  Work is already underway on a TCP/IP layer that will sit directly above L2CAP, bypassing and removing the overhead of PPP and RFCOMM.

- **TCS**
  Telephony Control Protocol Specification (TCS), is based on the International Telecommunication Union-Telecommunication Standardization Sector (ITU-T) Q.931 standard for telephony call control. It includes a range of signalling commands from group management to incoming call notification, as well as audio connection establishment and termination. It is used in both the Cordless Telephony and Intercom profiles.

- **SDP**
  The Service Discovery Protocol differs from all other layers above L2CAP in that it is Bluetooth-centred. It is not designed to interface to an existing higher layer protocol, but instead addresses a specific requirement of Bluetooth operation: finding out what services are available on a connected device. The SDP layer acts like a service database. The local application is responsible for registering available services on the database and keeping records up to date. Remote devices may then query the database to find out what services are available and how to connect to them.

- **Management Entities**

Device, Security and Connection Managers are not protocol layers so much as function blocks. The Device Manager handles the lower level operation of the Bluetooth device. The Connection Manager is responsible for coordinating the requirements of different applications using Bluetooth channels and sometimes automating common procedures. The Security Manager checks that users of the Bluetooth services have sufficient security privileges.

4.4 Distance Measurement

HCI Specification [21] offers a command to read the Received Signal Strength Indicator (RSSI) from a Bluetooth module; `HCI_Read_RSSI`. The command will read the difference between the measured RSSI and the limits of the Golden Receive Power Range for a connection handle to another Bluetooth device. Any positive RSSI value returned by the Host Controller indicates how many dB the RSSI is above the upper limit, any negative value indicates how many dB the RSSI is below the lower limit. Zero indicates the RSSI is inside the Golden Receive Power Range.

How accurate the dB values will be depends on the Bluetooth hardware, the only requirement is that the hardware is able to tell whether the RSSI is above, in or below the Golden Receive Power Range. It shows that with the used Ericsson ROK modules only very rough distance estimates can be made solely based on the RSSI value [23].

There exists another HCI command to get the link quality of a Bluetooth connection; `HCI_Get_Link_Quality`. However, the Ericsson module does not support it. The ROK modules come with an Ericsson specific command to read error rates for a certain link; `ERICSSON_BER`. Currently, no effort has been made to use this command for distance estimates.

4.5 Bluetooth Ad-Hoc Network

Networks are structures consisting of devices that are in some way connected to each other. Ad-hoc networks have the special feature that they can build up and change the topology of the network autonomously. In a Bluetooth ad-hoc network the links between the individual devices are RF connections. A Bluetooth link can be regarded as a radio wire. Even though wireless, connections have to be established, resembling the process of plugging in cables. As in a cable network, interconnecting the devices takes a considerable amount of time compared with the data rate that can be used once a connection is established. This is because Bluetooth uses FHSS, hopping around 79 bands of each 1 MHz. Devices have to synchronise to the same hopping scheme before being able to transmit data.

When two Bluetooth devices are connected, one of the devices acts as a master and the other devices acts as a slave. There is no direct master-master or slave-slave communication. On the other hand, a device can perform the role of a master and a slave simultaneously.

Figure 4.3 shows the three different possibilities when building up a Bluetooth network.

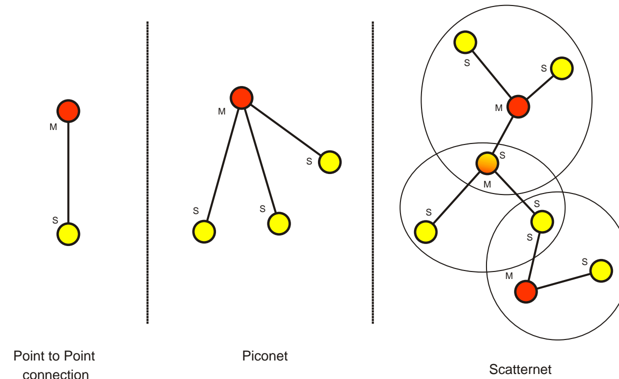


Figure 4.3: Connection types

The simplest situation is a single connection between a master and a slave. When a master is connected to several slaves, we speak of a *piconet*. One piconet consists of one master and up to seven slaves. When connecting several piconets together we get a *scatternet*. This can be achieved by devices that are shared in two piconets. A device can be a slave in several piconets but be a master in only one piconet. A device can be shared either when taking part in two piconets as a slave, or when being master in one piconet and slave in the other. These Bluetooth constraints impose challenges for Bluetooth ad-hoc networking. In [24] a Bluetooth scatternet formation algorithm is presented. It is a distributed, ad-hoc algorithm that involves electing leader devices and performs *merge* and *migrate* procedures. A *merge* will merge two piconets together to one. A *migrate* moves slaves from one piconet to another one.

Building up a scatternet for the proposed positioning algorithm was not implemented because of the following reason. It is important for positioning that the nodes are as highly interconnected as possible. The more connections we can build up, the more geometric constraints we can use to calculate a position. In a scatternet topology, this high connectivity cannot be provided. Hardware limitations prohibit devices to take part in an unlimited amount of piconets. Therefore, many devices that would be in range of each other will not be able to establish a connection.

Another reason that prevented the scatternet approach is that it is currently not clear to what degree the used Ericsson ROK modules support scatternets. Several manufacturers now support the limited form of scatternet required for a master/slave role switch while master of an existing piconet, but maintaining the scatternet for any length of time is still problematic. The Bluetooth Specification gives no way for a slave to demand hold, sniff or park modes (see Section 4.2) from a master; they must always be requested. The master is entitled to refuse such requests, so it is impossible to guarantee that a slave in one piconet will be granted the time required to participate in

another piconet as a master or a slave. Even if devices choose to simply switch between piconets as they see fit, ignoring the normal request procedures, there are still problems with how to time these switches in order to maintain multiple connections [22].

The solution that was therefore chosen to build up an ad-hoc Bluetooth network was that of many single dumbbell like connections, see Figure 4.4.

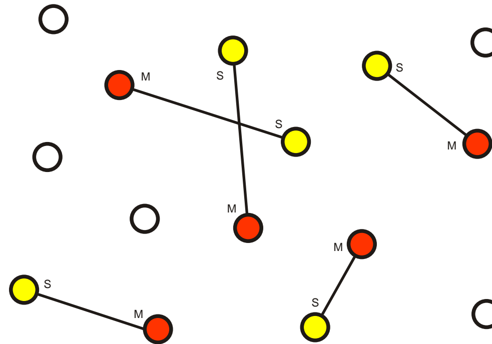


Figure 4.4: Network consisting of dumbbell connections

In an idle state, there are no open connections in the network. Connections will only be opened if there is data that has to be sent from one to another node. After successfully connecting, the data will be sent. If there is no pending data for an open connection it is closed immediately again. Obviously, this approach does not allow fast multihop data transmission as paging is a lengthy process (see Section 4.2). On the other hand, it allows us to open all connections that are possible among devices, which are in range of each other. This is an absolute necessity for positioning algorithms as errors induced by RSSI measurements can be averaged out this way. Section 5.7.4 present a proposal how a dumbbell like ad-hoc network could be scheduled.

Chapter 5

Software Implementation

The software for the BTnodes is written in standard C. It consists of two parts, the BTnode system software described in the next section and the application code described in Sections 5.3 - 5.7. The system software comes as a library of functions that provide access to the BTnode resources. The software can be compiled with the avr-libc [20] to run on the ATmega128 or with a standard C library to emulate a BTnode on any PC like system. The two main hardware interfaces used on the BTnodes are the two serial ports (see Figure 3.2), which can be easily emulated on a PC. One of the ports interfaces to the Bluetooth hardware, therefore a Bluetooth module must be connected via a serial port to the computer for emulation. The other port is mainly used for printing status information or as a command line interface. When emulating on a PC, this port is simply mapped to the standard IO. The possibility to compile the software for a PC is a great advantage, as it remarkably speeds up the development time. The software can be immediately executed after compilation without need for reprogramming the Flash. Debugging is far easier under Linux than on an embedded system as well.

5.1 BTnode System Software Stack

The BTnode system software is an event-driven, lightweight operation system made up of three different parts [25, 26]. Figure 5.1 shows a simplified block diagram of the system.

The boxes shown in light green are low-level hardware drivers that interface directly to the BTnode hardware. These drivers currently consist of a real time clock (RTC) that is driven by a separate oscillator, allowing for long timeouts consuming the lowest possible power. Another driver interfaces to the two serial ports, one of those is used for the HCI interface and the other can be used by the application. There is support of the standard `printf` function. I^2C is supported by yet another driver, e.g. to attach a sensor board to the BTnodes. Other low-level drivers include a LED driver, a power mode driver and an AD converter driver.

The second part of the system software is the dispatcher shown in yellow. It im-

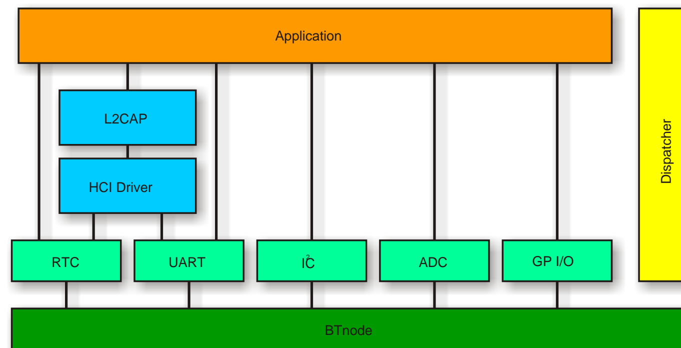


Figure 5.1: BTnode System Software

plements cooperative multitasking. Only one task (event handler) can be active at any time. Events are processed in the order they appear. Therefore, every event handler depends on the previous event handler to terminate in time. Low-level drivers or other software components can generate an event to notify other system components to take some action. The dispatcher mainly consists of a FIFO queue that stores the events that occurred and calls the corresponding event handlers to process them.

The third and largest system software part is shown in light blue. It implements a custom-made, reduced Bluetooth protocol stack [25]. At the moment, there is support for most HCI commands and the L2CAP layer is implemented. RFCOMM and SDP are currently under development.

5.2 Application as a Command Line Interface

As explained above, the application has been implemented as a command line interface. Commands can be sent to the BTnodes over the second serial port of the AVR micro-controller, see Figure 3.2. Several commands have been added to the Bluetooth testing commands that come with the BTnode system software. Some of those commands are only used for testing and debugging. Other commands need to read from files and can therefore only be issued on a PC emulation of the software, marked with (*) in Table 5.1, that gives an overview of all commands.

5.3 Application Specific Protocol Layers

All the applications implemented during this thesis build on top of the L2CAP layer, as shown in Figure 5.2. The part in light blue is provided by the BTnode system library. The green parts are part of the application, which is shown in orange, but may be included in the system software in a future version.

The connection manager handles all communications with the Bluetooth specific L2CAP layer and is described in following section. Currently, only very basic forms of

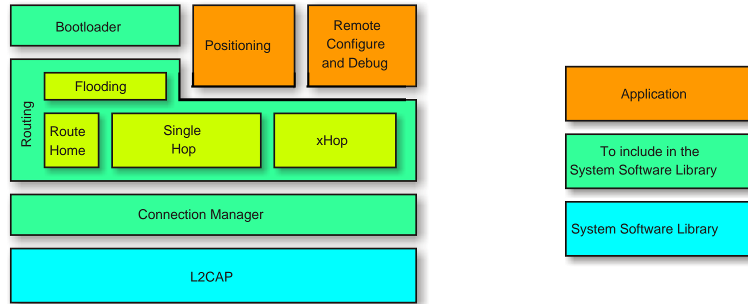


Figure 5.2: Custom layers on top of L2CAP

routing have been implemented. xHop is part of a dynamic source routing protocol and supports multihopping along a predefined route. It is described in Section 5.5. Another routing scheme is to forward packets home to a main development node. This method will not scale to large ad-hoc networks; however, it is useful for small networks with one single PC node that is used for code development. The protocol is used to signal the development node that a BTnode has been successfully reprogrammed and rebooted after network flooding with new code, see Section 5.6. At the moment, there is also the possibility to send data over one hop without an xHop header. With support for larger L2CAP packets, this could be dropped as then adding an xHop header for one hop, which is 15 bytes long, will not be of significance.

Flooding, that is used for the Bootloader, makes use of both single hop and route home packets. Positioning uses single hop packets for the *Start-up* and *Refinement* phase of the algorithm. However, anchor configuration and position estimate queries can be made both by single hop or xHop packets. The same is true for other remote queries and configuration commands.

5.4 Connection Manager

To facilitate data transmission between devices a connection manager has been implemented that sits immediately above the L2CAP layer and is responsible for all data transmission in the network. The chosen ad-hoc network topology throughout this thesis is the one of a dumbbell like structure as described in Section 4.5. In such a network, sending data is closely coupled with opening and closing connections.

The connection manager consists of two parts. The first part is the inquiry scheduler that periodically performs inquires and maintains a list of known devices that can be queried by the upper layers. The other part is the actual connection manager that accepts data packets from upper layers and is responsible for opening and closing connections and delivering the packets.

5.4.1 Inquiry Scheduler

The inquiry scheduler is implemented in the files "known_dev.c/h". It performs inquiries in a periodic interval and manages a list with the found devices. The list contains the known device's Bluetooth address, their clock offset, a time stamp of the last time the node was reported and a status byte.

The routine also checks the remote's CoD (Class of Device), which is used to decide whether the remote device is a BTnode or not. A special CoD was chosen to represent the BTnodes and which is set when booting the nodes. However, this approach does not conform to the Bluetooth Specifications as the unused CoDs are reserved for future applications/profiles and may not be used for custom purposes [21]. It was still decided to do so as SDP is currently not implemented and as the CoD can be read during an inquiry without the need to open a connection to the remote device. Another simple possibility to identify BTnodes would be to set a specific local name which could then be read from a remote device. However, this requires opening a connection.

The function `btaddr_get_index` returns an index into the list of known devices for a given Bluetooth address. Devices that are once found during an inquiry are removed from the list if they did not respond to inquiries for `DEVICE_LOST_TIMEOUT` ms.

5.4.2 Packet Forwarding and Connection Management

The basic scheme for sending data in the dumbbell like network structure works the following way. The application registers a data packet that has to be sent to a given Bluetooth address with a function call to `register_packet`. The application tags the packet with a timeout constant and a maximal number of retries in case of a forwarding failure. The data is copied to buffers inside the connection manager and the function returns immediately. The connection manager repeatedly calls the function `forward_pending_packets` that checks for open connections and for pending packets. If there is an open connection to a remote device and there is data to send to that device, then it will send it. Otherwise, it chooses a packet among the pending ones and tries to open a connection to the destination device. Currently, the process of choosing a pending packet to send next is random. However, adding a priority to pending packets and choose packets according to their priority would be an easy and useful extension.

The connection manager is also responsible to close a connection if there are no pending packets for the device on the other end of the connection. As the command for closing a connection immediately disconnects without regarding ongoing data transmissions, care had to be taken when disconnects can be issued. The device that receives data will process it immediately and if it has an answer to the sending device, it will send it on the still open connection. Once there are no answers for data that was received a device will disconnect from the remote device. This ensures that a connection is closed after all data was sent. Bluetooth is a wireless technology and there are many possibilities for failures that may occur. The device to which data was sent may fail to disconnect. As it is not the sender's responsibility to close connections it may be left open. A slave in an open connection is no longer able to respond to connection requests from other masters and is therefore blocked for further data transmissions.

Care had to be taken that even on disconnection failures, open, unused connections will still be closed as soon as possible. Figure 5.3 shows a simplified block diagram for the connection manager. Pending data packets are stored within the connection manager as depicted in Figure 5.4.

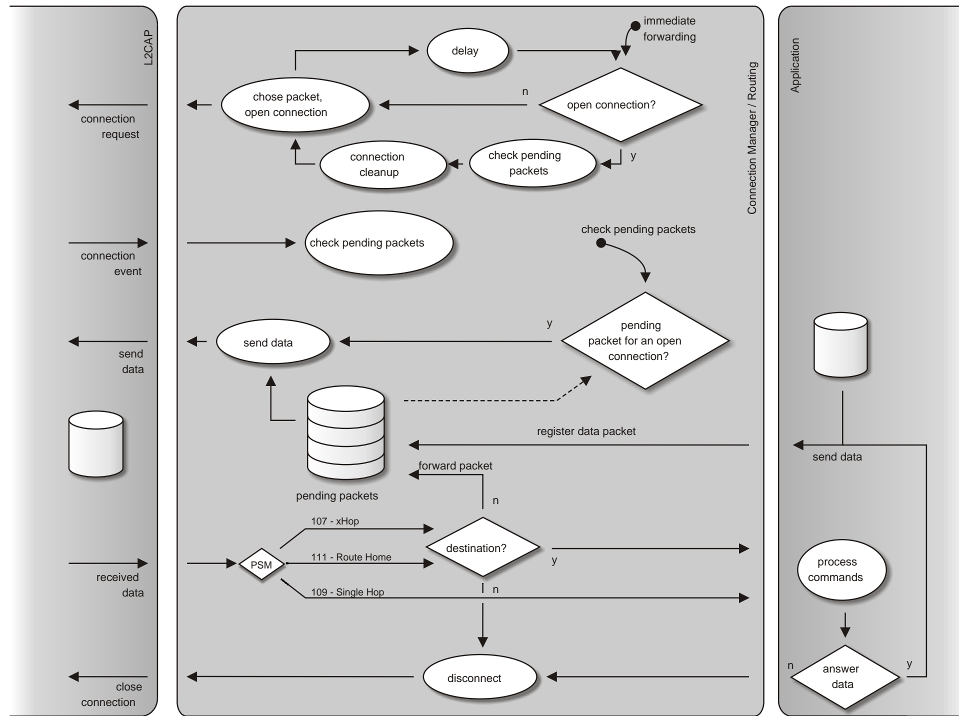


Figure 5.3: Connection manager

Several different packet types are possible for pending packets. The `rssi` and `multipoint` types are used for positioning and explained in Section 5.7.

The two figures 5.3 and 5.4 are explained with an example of a remote position query over two hops. On a PC emulation of the software, named node A, the command for a position query on node C, is issued with the xHop route ABC that requires two hops. The generated packet is registered by the application layer within the connection manger of node A. As currently no connection is open, the packet cannot be sent immediately. Rather, it is delayed until the connection manager chooses it among the other pending packets. It will then initiate a request to open a connection to node B. Once the Bluetooth module sends the event that a connection could be established, the connection manager loops through all the pending packet and will find the one that caused the connection request. This packet will then be sent to the remote device. On node B, the incoming packet arrives at the routing layer carrying the PSM for xHop packets. As it is a 2-hop packet, it has not yet arrived at the destination node. The packet is modified for the second hop and registered as a pending packet. Simultane-

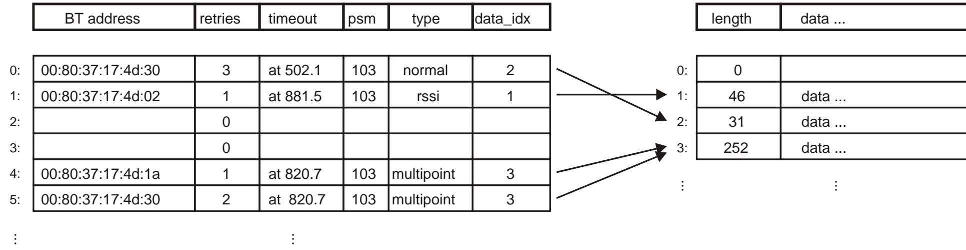


Figure 5.4: Pending packets structure

ously, the connection to the PC emulation node A is closed. As the packet should not be delayed for too long, immediate packet forwarding is requested from the connection manager which causes a connection request to the destination node C of the packet. Once, the connection is established, the packet is sent over the last hop. On the routing layer of the destination node C, the packet is forwarded to the application layer, which will start to process the commands. As the command sequence requests to send the answer back to the node that issued it, the connection is not yet closed. Rather, an answer packet is registered for node B. As the connection to node B is still open, this packet can be sent immediately. Arriving on node B, this node will do the same job for the packet just in the opposite direction as it did when the packet arrived first. This involves a connection request to node A and the closure of the connection to node C. Finally, the packet carrying the answer arrives at node A that will process the answer data. As there is no answer to the received packet, the connection to node B is closed.

5.5 RDSR Routing in Ad-Hoc Networks

During the semester thesis [27], several existing routing concepts for ad-hoc networks have been compared and analysed for their suitability for Bluetooth networks. Dynamic Source Routing (DSR) was chosen as a basis of a routing concept and a reduced version RDSR was proposed. The implementation of RDSR on Linux consisted of the packet forwarding part of RDSR, which is a scheme how packets can be sent along a known route over several hops. A packet format that corresponds to the source route option of RDSR was presented and named xHop. xHop provides a mean for multihopping and a script like command language for remote command execution. During the semester thesis [28], xHop was implemented to run on the BTnodes, however without the support of the BTnode system software (see Section 5.1).

RDSR is an on-demand routing protocol, which means that routes are not updated continuously. When a node requires sending data, it first needs to initiate a route discovery process before it can use the retrieved route to send the data with xHop.

To allow remote configuration and remote data queries over several hops in a BTnode network, xHop was implemented for the positioning application. An application example of xHop is given in Figure 5.5. Node A sends an xHop packet to nodes D

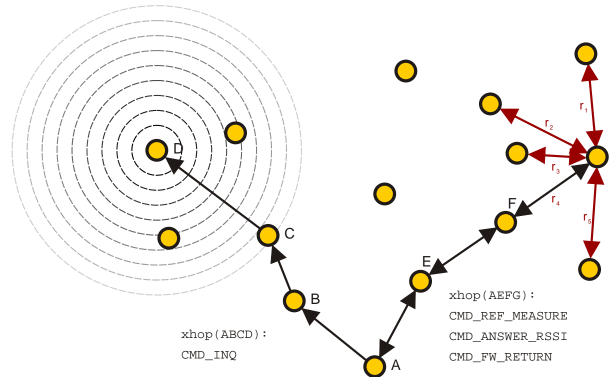


Figure 5.5: xHop example

and one to node G. It requests node A to perform an inquiry. In the other packet, it requests node G to perform a *Refinement* iteration and return the triangulated position on the reverse xHop route back to node A.

5.5.1 xHop Packet Format and Commands

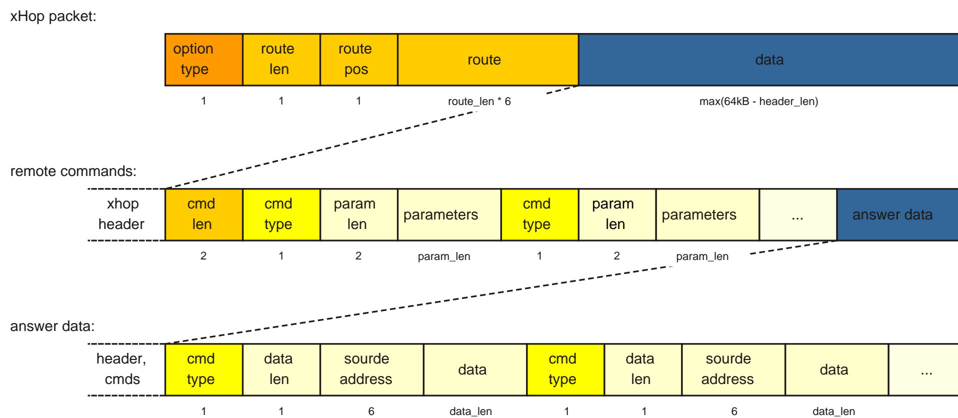


Figure 5.6: xHop packet format

The xHop packet format in the current implementation mostly conforms to the one presented in [27], see Figure 5.6. The source route is contained in the xHop header. The *route_len* field specifies the length of the route, the *route_pos* points to the current position within the route. When a node receives an xHop packet it increments the *route_pos* field and if it matches *route_len*, then the packet reached the destination

node and the commands are processed. Otherwise, the node forwards the packet to the address to which the incremented *route_pos* pointer points.

In the payload of the packet, commands are stored in a script like fashion. After a field containing the total length of all commands, several commands with their parameters can follow. The destination node starts with executing the first command. Once it is processed, it is removed from the packet and processing continues with the next command. If a command is reached that generates an answer, this answer is appended to the end of the packet. Some commands require forwarding the packet to other devices. If such a command is reached, the packet is sent without the commands that were already processed. No commands following the forwarding command will be processed until the packet reaches the node specified by the command that caused the forwarding. This mechanism allows executing commands on the route of an xHop packet and collecting data from several nodes along the route.

In the current implementation, an xHop packet can be launched with the command `xhop` on the BTnode command line (`hop` sends a command packet without xHop header over one single hop). The command can only be issued on a PC emulation of the application as it involves reading files that contain the xHop route and the xHop commands. The xHop route is read from the file "xhop.route" in the current directory. An example file could look like:

```
00:80:37:17:4d:01
00:80:37:17:4d:05
00:80:37:17:4d:30
00:80:37:17:4d:1a
00:80:37:17:4d:30
00:80:37:17:4d:2a
```

The Bluetooth address of the node that launches the xHop packet must not be supplied as it is read from the attached device and inserted automatically at the beginning of the route.

The commands can be specified in the file "cmd.bat". An example that queries a remote node for its estimated position and prints the string "pc requests position" on the remote device looks like:

```
CMD_PRINT pc requests position
CMD_ANSWER_POSI
CMD_FW_RETURN
```

Currently the commands listed in Table 5.2 are implemented.

The commands used for positioning and the Bootloader are explained in more detail in the corresponding sections, as well as their answer format. Some of the commands used for the Bootloader or for positioning cannot be specified in the "cmd.bat" file.

5.5.2 Results

To measure the performance of the xHop implementation some experiments have been carried out. Figure 5.7 shows the route over which the xHop packets have been sent. The setup involved one Linux PC (node A) and four BTnodes (nodes BCDE). Node A launched an xHop packet with the route ABCDEBCDEA that requires 9 hops to reach the destination, which is in this case again node A. Node A measures the time difference from the launching until the packet reaches the destination. As the connection manager performs periodic inquires, during which a node is not able to build up a new connection, packets can get delayed. The inquiries were scheduled to take places every 195 $\pm$ 15 seconds and require a time of 6 seconds. There was always at maximum one xHop packet in the network. Out of 57 launched xHop packets 55 reached the destination node. Two got lost due to expired timers or used up retries counters within the connection manager.

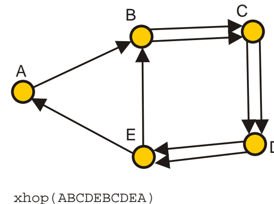


Figure 5.7: Experiment setup

The remaining 55 packets have a per-hop-delay distribution between 1 and 6 seconds as shown in Figure 5.8. The average time spent for one hop was 2.1 seconds. Even though this is still quite a long time for one hop, if compared to the speed data can be sent once a connection is open, it is remarkably faster then on previous implementations [27, 5]. This is probably mainly due to the newer Bluetooth hardware used and due to improvements in the Bluetooth system software.

5.6 Bootloader Application

To simplify maintenance and application development of BTnode ad-hoc networks a Bootloader was developed. The name Bootloader is actually misleading, as the routine is rather some kind of network program distribution method and it is not called during booting the node. The name comes from the fact that the Flash section within the ATmega where code is located that is able to write the Flash itself is called Bootloader section.

The goal of the Bootloader is to selectively flood a network with newly available program code, rewrite the node's program memory and restart the nodes.

The routine consists of three components. The first component runs only under Linux and reads an Intel HEX file containing the executable program code, the second

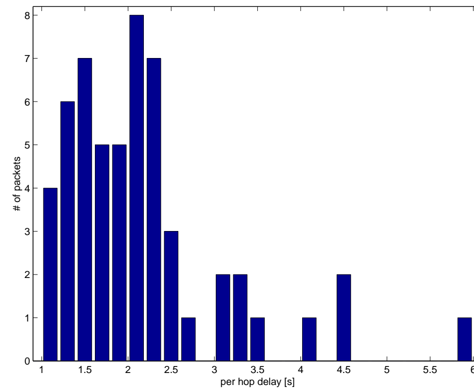


Figure 5.8: xHop per-hop-delay

component is responsible for the selective network flooding and the third part rewrites the microcontroller's Flash memory and reboots the nodes.

There are two different approaches to reprogram remote nodes. The first possibility implements a tiny Bluetooth stack inside the AVR bootloader section, which will reprogram the Flash as it receives code over the air. This has the advantage that no additional RAM is required to store program code temporarily and that code segments can be selectively replaced, only requiring to send to appropriate code segments. However, implementing a tiny Bluetooth stack is tricky, especially as the Bluetooth stack in the system software is still frequently undergoing changes. The other problem would be that a transmission failure due to one device leaving the transmission range would result in inconsistent program code. This could however be detected by the bootloader and the program would not be executed. Rather, the Bootloader, which will never be rewritten over the air, will request the code again. Until the new code could be successfully reprogrammed, the application will not be available anymore though.

The other possibility, which was implemented, makes use of the external RAM for temporary storage of program code before the Flash is reprogrammed. The advantage is that on unsuccessful transmission, the old application will not be overwritten and can still be executed. There is also no need for an additional Bluetooth stack solely for the Bootloader application. The only code that resides in the Bootloader section of the Flash is a simple RAM to Flash copy routine that is never reprogrammed over the air. Care must however be taken that modifications at the Bluetooth stack and the Bootloader code are carefully inspected before the network is flooded, as otherwise the reprogrammed nodes may not be anymore able to execute the Bootloader function and can therefore not be reprogrammed again. Nodes would then have to be collected and reprogrammed with functional code with cable programmers. As with the approach explained above, it is possible to replace code in the Flash selectively. However, compilation and linking may contain caveats for that purpose, as code not replaced must be located in the same way for the new and old program.

The Bootloader routine is started with the command `program` on the BTnode com-

mand line.

5.6.1 Intel Hex File

The starting point for the network reprogramming is the complied source code as an Intel HEX file. This format stores executable instruction code in a readable hexadecimal format. Therefore, four 8-bit ASCII characters are required to store one 16-bit instruction. In a first version of the Bootloader, only bank 0 of the external SRAM was available (see Section 3.2.2), therefore, memory was a limited resource. One memory bank is 64kB (including the internal memory) that must be shared for the Bootloader application and for the system software. The Flash may take up to 128kB of program code. Reprogramming in two steps was considered difficult, as it would have to be ensured that the first part is able to perform all Bootloader functions for retrieving the second part. However, the applications at that time were significantly smaller then 60kB. It was decided to stored program code within the SRAM in a more compressed form then it is done in an Intel HEX file. The Bootloader therefore performs some minor shuffling when reading an Intel HEX file.

Intel HEX files are composed of records of different types. Each record is made up of five fields that are arranged in the following format:

```
:llaaaatt[dd...]cc
```

Each group of letters corresponds to a different field, and each letter represents a single hexadecimal digit. Each field is composed of at least two hexadecimal digits-which make up a byte-as described below:

- `:` is the colon that starts every Intel HEX record.
- `ll` is the record-length field that represents the number of data bytes (`dd`) in the record.
- `aaaa` is the address field that represents the starting address for subsequent data in the record.
- `tt` is the field that represents the HEX record type, which may be one of the following:
  - `00` - data record
  - `01` - end-of-file record
  - `02` - extended segment address record
  - `04` - extended linear address record
- `dd` is a data field that represents one byte of data. A record may have multiple data bytes. The number of data bytes in the record must match the number specified by the `ll` field.

- `cc` is the checksum field that represents the checksum of the record.The checksum is calculated by summing the values of all hexadecimal digit pairs in the record modulo 256 and taking the two's complement.

The gnu-avr utilities make use of the data, end-of-file and extended segment address records. Extended addresses are required to place code at address greater then 64k.

L2CAP provides reliable data transmission, therefore, when reading from the HEX file the checksum field is dropped. To get a slightly more compressed format, program data is stored in the following way on the BTnode's SRAM:

```
LLLAAA[DDDD...]
```

where `LLL` is the record length in bytes, `AAA` the starting address and `D` the program code (`LLL` times). Here, letters represent one byte. Addresses and length fields are 24-bits therefore.

5.6.2 Selective Network Flooding

All nodes that should be reprogrammed can be specified in the file "bt_addrs.txt" on the PC emulation of the software. A header is then composed which is added in front of the program data section in the simulated additional SRAM banks on the PC. The data is laid out in the following format:

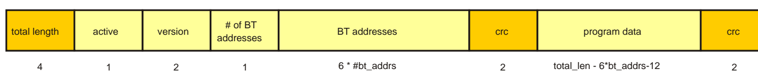


Figure 5.9: Bootloader data in SRAM

The total length field specifies the amount of program data together with the header size. The *active* field is a flag byte that is set to 1 if the data in the SRAM is still to be processed and 0 once the data has been processed. This is necessary as after all data has been received, the Bootloader is called, which will, after successful reprogramming the Flash, reset the node. On booting the node, it will check if it contains active Bootloader data to forward to other nodes. Once finished, this flag is set zero, so that after a later reset the Bootloader will not start an unintended flooding. This is also the reason for the added CRC fields. After power down and up again, the node can check if the data in the SRAM is valid Bootloader data or not. Two CRC fields are added as the program data can be rather long and usually checking the header is enough to detect corrupted data.

Currently, only nodes specified in the Bootloader header address field take part in the flooding algorithm. After reprogramming and rebooting a node, it will therefore only send program data to devices listed in its SRAM. The node will remove itself from the list once it reprogrammed itself. It would be a simple extension, to add a flag for every address that states, whether a node should be programmed or not. Nodes could

then take part in flooding the network without being reprogrammed. The version field is read from a file and incremented every time the bootloader is started. It could be modified to use a hashed format of the current time and date, to remove the need for a special version file. The version is stored in the devices EEPROM after successful reprogramming. It could be included during compilation time in the program code itself, as program version is a feature concerning the actual program.

The actual sending of the program data is done in two steps. First, the initiating node asks the one to which it wants to forward its program data if it has already loaded the current version or not. The reason for this two-step procedure is that sending program code of several 10kB, is a rather lengthy process and should be omitted if not necessary. Figure 5.10 illustrates the procedure for the first hop from an emulation on a PC to a BTnode.

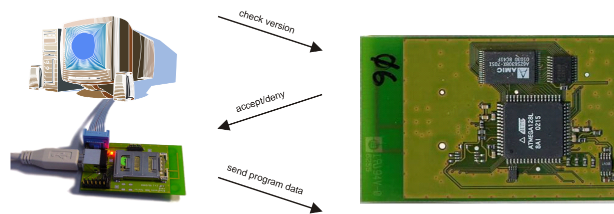
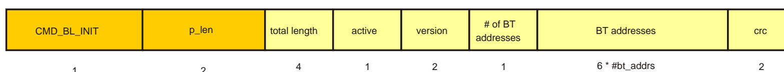


Figure 5.10: First Bootloader hop

The packet that initiates the process contains the header of the Bootloader data, as it is stored in the SRAM:

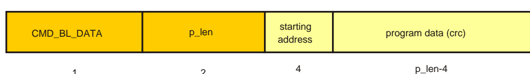


CMD_BL_INIT	p_len	total length	active	version	# of BT addresses	BT addresses	crc
1	2	4	1	2	1	6 * #bt_addrs	2

Figure 5.11: Bootloader init command

The remote device then sends an answer packet either accepting the reprogramming or denying it due to an already ongoing reprogramming or because the current version is already loaded. If the remote node denied, its address will be removed from the list of devices to be reprogrammed. Otherwise, it will be removed after sending program data has successfully finished.

Data is sent in packets like the ones depicted in Figure 5.12.



CMD_BL_DATA	p_len	starting address	program data (crc)
1	2	4	p_len-4

Figure 5.12: Bootloader data command

Data is sent in chunks that fit into the maximum allowed L2CAP packets. The maximum specified in the Bluetooth Specifications is 64kB. However, with the current implementation, smaller packets are sent. Currently 256 bytes per packet is the maximum. This is also the reason that sending Bootloader packets with xHop is not feasible. A program size of 80kB (which the application is) requires more then 300 packets, this amount will take far too long to send with xHop. Over one hop the packets can be sent quite fast though. However, sending subsequent packets on the same link required some modifications to the connection manager. Usually, the receiving device is responsible to close a connection if it has no answer to send back to the initiator, see Figure 5.3. As the initiator now wants to send many packets, the receiving device is not allowed to close the connection. This is achieved by registering a special packet among the pending packets in the connection manager. It will cause the connection to stay open; however, this special packet will not be sent to the remote device. The packet type is called `keep_open` and will be removed after it expired. Such a packet is registered when the first Bootloader data packet is received. Once the last data packet is received, the application will directly close the connection to the sender.

5.6.3 Reprogramming the Flash

The third step in reprogramming a BTnode network is to rewrite the contents of the Flash on the AVR devices. For this purpose, a routine has been written that copies the program data from the SRAM to the Flash.

Code that is able to rewrite the Flash must be located in the Bootloader section of AVR Flash memory [17]. Some flags can be set for different bootloader section sizes. They must be set to a section size of 4kB. The bootloader section then starts at the address 0x1F000. It can be executed by jumping to the section with

```
asm volatile ("jmp 0x1E000"::).
```

The flags are set in a way, that after a reset, the code in the application section is executed and not the bootloader section. The bootloader section is only executed with the above code issued in the application section. When copying finished, program execution is continued at the start of the application section, which has the same result as pressing the reset button.

The code in the bootloader section copies data to the Flash in a page like manner, using the `spm` instruction. This section is not reprogrammed over the air. It has to be downloaded once by cable. Some care has to be taken when downloading the bootloader code and application code. It can be done the following way:

```
cd linux
make                               //to compile the application for Linux
cd ../avr
make                               //to compile the application for the avr
make boot                          //to compile the Bootloader program section
uisp -dprog=stk500 -dserial=/dev/ttyS0 -dpart=ATme ga128 --erase --upload if=main.hex
                                   //download program
```

```
uisp -dprog=stk500 -dserial=/dev/ttyS0 -dpart=ATmega128 --upload if=btnode_boot.srec
                                       //download bootloader section
cd ../linux
main -u0 /dev/ttyS1 57600 fc -u1 stdio //start the system software
program                                //to initiate the bootloader on BTnode command line
```

5.6.4 Results

Here a few numbers to give a hint at the Bootloader's performance. Downloading a 80kB by uisp [29] and the STK500 [17] takes approximately 30 seconds. Sending 80kB by Bluetooth could be as faster then 1 second. In the current implementation, it takes approximately 8 seconds.

Reprogramming the Flash takes about 5 seconds. For secure network flooding several waits were included. After program has been sent to all nodes, it takes some time for the flooding to settle, as nodes will check some neighbour's program version, even though they were already reprogrammed. No large-scale experimentation has been carried out, however a few numbers in Table 5.3:

The time required for four nodes is only slightly faster then reprogramming by cable. However, using the bootloader has the advantage that nodes do not need to be collected. Reprogramming only requires to type the command `program` and then gives time to get a cup of coffee. Reprogramming by cable needs plugging in and out cables every 30 seconds.

5.7 Positioning with Hop-TERRAIN

A three dimensional version of the Hop-TERRAIN algorithm (see Section 2.4) was implemented for the BTnodes. Coordinates are the three float values (x, y, z). The algorithm has been implemented according to the specifications given in [13]. Experimentation was carried out with the *Refinement* part of the positioning method, presented in Section 2.4.2.

The procedure of the algorithm differentiates between anchor and free nodes. Anchor nodes could use a GPS based position or a fixed position. To set an anchor's fixed position remotely the command `CMD_ANCHOR` was implemented. It can be specified in the "cmd.bat" file in one of the two following ways:

```
CMD_ANCHOR posX posY posZ   //set anchor to posX, posY, posZ
CMD_ANCHOR off              //free anchor
```

It can be sent with either `xhop` or `hop`. The command in a packet has the format shown in 5.13, either for setting an anchor or for freeing it again.

5.7.1 Start-up Phase

During the *Start-up* phase of the algorithm, the anchor positions are flooded into the network and and it is determined if a node is sound or not. This is done with the

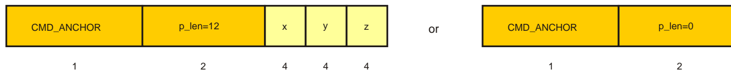


Figure 5.13: `CMD_ANCHOR`

following command.

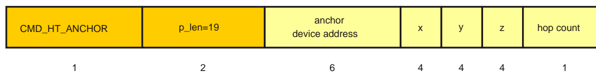


Figure 5.14: `CMD_HT_ANCHOR`

It can be sent from the anchors at long time intervals. If a node received a packet containing this command, and it has not received one from the same anchor with a smaller hop count, it will broadcast the same packet to all surrounding nodes with the hop count increased by one. To simplify broadcasting to all known devices, the connection manager was enhanced by the function `register_flooding_packet`. The function will register a pending packet of the type `multihop` for all devices listed in the inquiry scheduler (see Section 5.4.1). The data for those pending packets is however only stored once, see Figure 5.4.

5.7.2 Refinement Phase

The *Refinement* phase requires that position information is exchanged between neighbouring devices and that the RSSI value is measured for the corresponding connection handles. The request to a known device for its position is issued with the command shown in Figure 5.15.

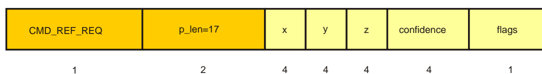


Figure 5.15: `CMD_REF_REQ`

It contains the issuing node's position, its confidence level, which states how accurate its position is and some flags. The flags specify if the node is an anchor and if not, then it states if the node is sound or not. The remote device will answer to this request with a packet containing the same information for the remote device.

Again, these packets are registered within the connection manager at a certain time interval to be sent to all surrounding devices. Another flag is set for this packets, which requests the connection manager to read the RSSI value once the desired connection has been opened.

The time interval at which *Refinement* is initiated can be specified with the two following commands on the BTnode command line:

```
ref_set interval
ref_start
```

`ref_set` sets the *Refinement* interval to the given amount of seconds. *Refinement* is then issued at this interval $\pm$ `REF_PERIODE_VAR` seconds, to ensure that not always all nodes will issue the command in the same order. Setting `ref_set` to zero will stop the *Refinement* after it is launched the next time. It can be restarted with `ref_start`. A single *Refinement* measurement to all surrounding nodes can also be issued remotely with the following command:

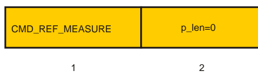


Figure 5.16: `CMD_REF_MEASURE`

or just `CMD_REF_MEASURE` in "cmd.bat".

Once enough geometric constraints have been collected to surrounding nodes, a triangulation will be performed, resulting in the node's position estimate. Currently, this can only be done on a Linux emulation of the software. GSL [30] was used to solve the least squares problem. To overcome this limitation, a command has been implemented that reads out a remote nodes RSSI measurements and returns it to a Linux PC where the calculation can be done. In the "cmd.bat" file the command sequence for this, looks like:

```
CMD_ANSWER_RSSI
CMD_FW_RETURN
```

Both of this commands do not need any arguments. The first will write the measured RSSI values into the packet's answer space. The second returns the packet to the sender, either using single hop or xHop with the reverse route.

The answer for such a request has the format shown in Figure 5.17.

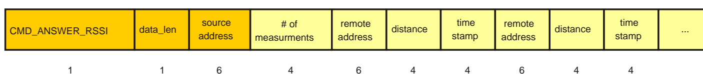


Figure 5.17: Answer for `CMD_ANSWER_RSSI`

When the answer is processed by the destination device, it will print the information like shown below:

```
<<<<< XHOP PACKET >>>>>
---- xhop route --------
```

```
|    0b:4d:17:37:80:00 |
|    00:4d:17:37:80:00 |
| -> 30:4d:17:37:80:00 |
------------------------
xhop packet reached destination, process...

process answer for CMD_ANSWER_RSSI ...
------------------------------------------
received rssi data from BTnode 0b:4d:17:37:80:00
   distance to 30:4d:17:37:80:00 0.20m, about 609s ago
   distance to 09:4d:17:37:80:00 0.79m, about 0s ago
------------------------------------------
close connection to 00:4d:17:37:80:00, no packet pending to this device
```

A last command `CMD_ANSWER_POSI` without any parameters is used to read out a remote device's position information. The answer has the following structure:

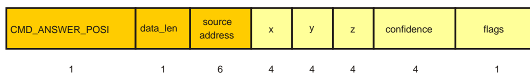


Figure 5.18: Answer for `CMD_ANSWER_POSI`

resulting in a print out like

```
<<<<< XHOP PACKET >>>>>
---- xhop route --------
|    0b:4d:17:37:80:00 |
|    00:4d:17:37:80:00 |
| -> 30:4d:17:37:80:00 |
------------------------
xhop packet reached destination, process...

process answer for CMD_ANSWER_POSI ...
------------------------------------------
received position data from BTnode 0b:4d:17:37:80:00
   position:  (X=3.50 / Y=10.02 / Z=1.73), confidence=0.81, sound
------------------------------------------
close connection to 00:4d:17:37:80:00, no packet pending to this device
```

when processed.

5.7.3 Results

Some experiments have been carried out with the *Refinement* part of the Hop-TERRAIN algorithm. The setup was like shown in Figure 5.19.

All nodes wer doing *Refinement*. Time was started when the centre node entered the setup. It was stopped when this node acquiered enough information to perform a triangulation, which means that it had to collect the RSSI information to all nodes in the setup. With the *Refinement* interval set to 60 seconds, the first triangulation could

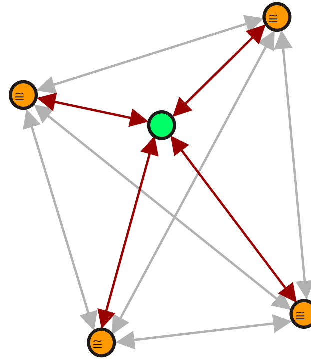


Figure 5.19: *Refinement* experiment setup

be performed after 5 minutes 30 seconds. In a second and third try the node failed to acquire enough RSSI measurements. With the interval set to 100 seconds six tries were measured, three of them failed to calculate the position. The fasted result was obtained after 1 minute and 15 seconds, the slowest after more then $4\,\frac{1}{2}$ minutes. Out of 5 experiments with the interval set to 120 seconds only one failed. The fastest time at which triangulation was performed was 1 minute 25 seconds, the slowest slightly less then 4 minutes.

Closer inspections of the not very promising results reveal that the long times for *Refinement* are mainly caused by a scheduling problem. As explained in Section 4.5, the approach of building up connections before sending data and immediately disconnecting after the data has been sent is used. However, connecting requires rather long as the devices have to synchronise (Section 4.2) before data can be sent.

When periodic inquires are done, the Bluetooth module keeps track of the surrounding devices clock offset, which greatly helps to speed up connection establishment. However, during an inquiry the devices are not able to answer to a connection request from a remote device, which means that the remote device that requested the connection has to wait until the request returns with the status indicating the connection failed.

5.7.4 BTerrain

As the main cause for the long times presented in Section 5.7.3 for a single *Refinement* iterations is a scheduling problem, a way the cope with that was studied. The algorithm presented here was however not implemented on the BTnodes and was only roughly simulated to estimate its feasibility for a Bluetooth ad-hoc network.

One way to cope with the scheduling problem is to collect some information from the surrounding devices and agree on a shared schedule to send data. Such an algorithm must run in an ad-hoc, distributed fashion. Synchronising the whole BTnode network to a global time is not feasible and not necessary. A local synchronisation of the nodes on either side of a connection is not resource consuming and can be done accurately enough. Synchronisation of connection requests and inquiries does not ask for very high

precision, a precision of some 100ms should be enough. Once a connection is open, data can be sent rather fast without long delays, therefore a synchronisation with accuracy in the ms region can be achieved by sending the local time of one BTnode to the other.

The proposed algorithm is a simple extension of the Hop-TERRAIN algorithm to make it feasible for Bluetooth networks with long connection times.

Time is divided into periods of T_p. Every node will be able to perform one *Refinement* step in one such period. Every period T_p is further divided in n_s slots of duration $\frac{T_p}{n_s}$ and every node will keep an array with n_s entries for slots allocation. When an inquiry on a device reports a new neighbouring device, it will try to open a connection to that device. Once this connection could be established, the two devices agree on a certain amount of slots n_{data} they will use for future low priority communication, like *Refinement* data packets. This in done in the following way; the node that opened the connection will send its slot allocation array to the remote device together with its time offset into the current period. The remote device will then merge its own array with the one just received. It will shift the allocation arrays according to the received time offset. It will then randomly choose the required amount of slots needed for data communication among those slots that are empty on both devices. The remote device then sends back which slots have to be reserved for future communication.

The two devices also have to agree upon which one is responsible for opening the connection. Figure 5.20 shows three nodes with their slot allocation arrays.

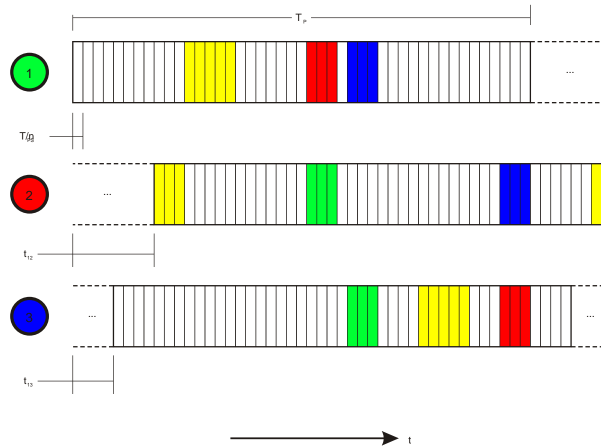


Figure 5.20: Slot allocation arrays

In Figure 5.20, the period T_p of 90 seconds is divided into 45 slots of $t_{data} = 2$ seconds. Opening a connection, sending data and disconnecting it takes approximately 2 seconds (see Section 5.5.2); therefore, three slots are allocated for a data packet. This is because the slots are only synchronised to the precision of one slot. An interval of slightly more then one slot may use up three slots. On node 1, the slots reserved for communication with node 2 are shaded in red, the ones for node 3 in green and vise versa for the other nodes. The slots shaded yellow are the places where an inquiry

could take place. A proposal is to schedule the inquiry anew for every period among the free slots on the local device. It may still collide with inquiries and communication slots on neighbouring devices though.

Obviously, for highly connected networks, the slots may be filled up quickly and allocating slots may become impossible. Some simple simulation and statistic analyses give a hint at the time that is required for one period so that all nodes are able to allocate communication slots. Simulations have been done with 200 nodes placed within an area of 100m times 100m. The communication range of the devices was varied from 10m to 16m to provide networks for different connectivity. At border regions, the network will have a smaller connectivity then at more centred regions. However, the range simply translates into the network connectivity. Figure 5.21 shows the maximum connectivity and the average connectivity for the situation described above.

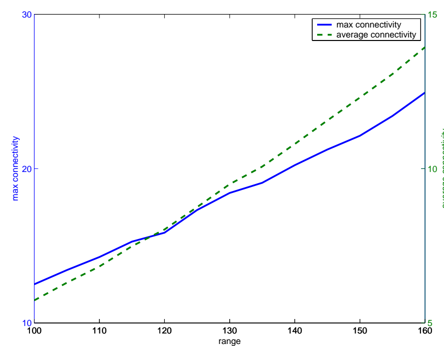


Figure 5.21: Range - connectivity relationship

The period length is now varied from 120 seconds down to 40 seconds, divided in 240 slots. It was counted how many connections between two nodes could not be allocated due to filled up slot allocation arrays. Figure 5.22 shows the result for a data packet length of t_{data} = 1.5s, 2.0s and 2.5s.

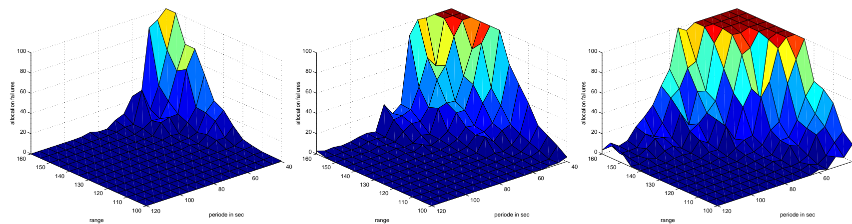


Figure 5.22: Allocation failures for 1.5, 2.0 and 2.5 seconds

A good connectivity for positioning algorithms with large errors on distance measurements is about seven or more [3]. For the simulated situation, this results in a communication range of about 11 meters. Assuming a rather pessimistic t_{data} of 2 seconds and not tolerating any allocation failures the middle graph of Figure 5.22 shows

that a period of approximately 60 seconds provides enough time for successful slot allocation. Given the total number of slots $n_s = 240$, a period of $t_p = 60$ seconds and $t_{data} = 2$ seconds, a communication will need $n_{data} = 10$ slots, requiring a synchronisation accuracy of 250ms. The average node will therefore use 70 slots for low priority *Refinement* packets, or a mere 30%. Peak nodes that are highly connected approximately use 140 slots or about 60%.

60 seconds for one *Refinement* iterations is already quite a lot faster then the results shown in Section 5.7.3, which are even based on experiments with a low connectivity of 4.

As learned during this diploma thesis, only implementing such an algorithm will prove that the BTerrain approach is feasible and what performance can be achieved.

Command	Description	Explatiation
`inq`	execute inquiry, print results	
`print`	print previously discovered devices	
`bbcon idx`	open baseband connection to device with `idx`	
`bbdisc hdl`	disconnect baseband connection	
`con idx psm`	open l2cap channel to device with `idx` on `psm`	
`disc l_cid`	disconnect l2cap channel with `l_cid`	
`send l_cid len`	send on l2cap channel a chunk of legth len	
`rname idx`	request remote name of device with `idx`	
`role hdl`	discover role for connection handle `hdl`	
`switch hdl role`	switch role for `hdl` to role {0, 1}	
`name name`	change local name to `name`	
`txpower hdl`	read transmit power to handle `hdl`	
`rssi hdl`	read rssi of connection `hdl`	page23
`ref_start`	start *Refinement* iterations	page 41
`ref_set`	set refimenet interval in sec (0 => stop)	page 41
`memcpy`	test external SRAM and with `xcopy` routine	page 14
`eeprom`	print EEPROM contents	
`(*) hop`	compose single hop packet from files and send	page 32
`(*) xhop`	compose xhop packet from files and send	page 32
`(*) program`	selective network reprogramming	page 34
`(*) now`	log time_stamp	

Table 5.1: BTnode commands

Command	Constant	in “cmd.bat”	Description
`CMD_INVALID`	0		invalid command
`CMD_PRINT`	1	`CMD_PRINT text`	print a string
`CMD_ANSWER_PRINT`	3	`CMD_ANSWER_PRINT text`	put string in answer space
`CMD_FW_RETURN`	14	`CMD_FW_RETURN`	return to sender
`CMD_BL_INIT`	20		bootloader init
`CMD_BL_DAT`	21		bootloader data
`CMD_ANCHOR`	31	`CMD_ANCHOR x y z` `CMD_ANCHOR off`	remote anchor configuration
`CMD_HT_ANCHOR`	32		Hop-TERRAIN anchor broadcasting
`CMD_REF_REQ`	40		Refinement request
`CMD_REF_MEASURE`	41	`CMD_REF_MEASURE`	measure RSSI
`CMD_ANSWER_RSSI`	50	`CMD_ANSWER_RSSI`	RSSI -> answer
`CMD_ANSWER_POSI`	51	`CMD_ANSWER_POSI`	position -> answer

Table 5.2: xHop/hop packet commands

# of nodes	approximate time	forwarding path
1	30 seconds	`Linux->BTnode`
2	60 seconds	`Linux->BTnode->BTnode`
3	110 seconds	`Linux->BTnode->BTnode` `Linux->BTnode`

4	120 seconds	`Linux->BTnode->BTnode`
		`Linux->BTnode`
		`Linux->BTnode`

Table 5.3: Bootloader result

Chapter 6

Conclusion

I would like to present some ideas of what I think should or could be done in the future concerning the BTnode project and positioning. The currently available system software showed quite good result to handle the Bluetooth modules. However, quite a lot of effort must be spent to make the software more stable and to implement additional Bluetooth layers. Layers like RFCOMM and SDP would be useful for communication to industrial Bluetooth products, which is one of the main advantages to use Bluetooth as a wireless interface. There are some bugs within the system software that should be fixed and a more consistent error handling should be implemented.Sophisticated use of the Bluetooth hold, park and sniff modes will be required for low power applications and Scatternet operations.

Concerning the application development throughout this thesis, the connection manager needs some rewriting. The connection manager is currently slightly interwoven with the routing layer. These two layers should be separated completely. There are many more commands that could be implemented for real remote configuration and debugging. Some mean of logging events should be provided by either the system software or the application itself. The application currently uses a simple EEPROM API, however meanwhile there is a more powerful EEPROM handling API available for the system software.

If possible, the Bootloader and the connection manager should be made available to the BTnode freaks community. These functions could maybe be included in the system software. However, both have some special requirements not shared among all BTnode users. The Bootloader makes use of the whole 256kB external memory, which requires patching the board with two wires. The connection manager is only useful for dumbbell like networking, and does not provide any use for scatternetting.

Real position awareness is still very challenging in Bluetooth networks. The results and some Bluetooth analyses showed that fast position updates required for mobile nodes (walking speed or faster) are not possible with Bluetooth due to its limitations. However, Bluetooth may provide positioning in quasi-static environments, were mobile nodes can be located at times they do not move. The scheduling approach presented in Section 5.7.4 may provide a solution to the scheduling problem in a dumbbell like

network. This approach could be simulated with NetSim [31] to reveal its limitations. It would maybe prove useful to implement an accurate Bluetooth node simulation in the NetSim environment, that takes into account all the special Bluetooth limitation, like master-slave relationships, connection establishment times, inquires and so on.

For fast position updates required for real mobile nodes, the BTnodes could be enhanced with a radio interface for positioning purposes. Such a radio may allow more precise RSSI measurements and faster connection times. For fast data communication and interfacing to consumer electronics, Bluetooth will still be useful on the boards.

During my diploma thesis, I could gain a good understanding of the wireless Bluetooth technology. I could learn about its many advantages, like the simple API it provides through the HCI interface, but I soon also saw the difficulties of such a technology. Wireless communication is an error-prone mean to interconnect devices, which the Bluetooth module cannot completely hide from the user. Connections may not get established and they may suddenly, unintentionally disconnect. It is therefore important to implement good error handling for wireless communication.

Positioning algorithms for ad-hoc networks are quite a challenge to be implemented on an 8-bit microcontroller Bluetooth platform like the BTnodes. Even though my application does not yet allow real position awareness, I was glad that I could calculate position estimates at all in the end, even though it requires a rather long time. On the other hand, I was surprised myself how powerful the Bootloader and xHop turned out for BTnode network maintaining.

Appendix A

Source Files

File	Flash Size	Description
src/anchor.c(h)	544	Remote anchor configuration
src/avr-boot.h		SPM Flash write routines
src/boot_defs.h		Record type definitions
src/bootloader.c(h)	4465	Application part of the Bootloader
src/bt_cmds.c(h)	8528	BTnode command line
src/btnode_boot.c	1122	Bootloader Section
src/crc.c(h)	440	CRC
src/ee_cntr.c(h)	679	EEPROM API for Linux (AVR)
src/file_log.c(h)	2	Event logging
src/hop_terrain.c(h)	953	Broadcast anchor positions
src/known_devs.c(h)	1621	Inquiry scheduler
src/main.c(h)	2087	Main
src/packet_forwarding.c(h)	6576	Routing and Connection Manager
src/positioning.c(h)	592	Coordinate definitions
src/process_packets.c(h)	3159	Process xHop commands
src/ram_banks.c(h)	836	`xcopy` routines
src/ram_bank_asm.S		Assembler part of `xcopy`
src/ram_port_def.h		IO defines for `xcopy`
src/refinement.c(h)	5320	Hop-TERRAIN *Refinement*
src/triangulate.c(h)		Triangulation (Linux only)
avr/Makefile		AVR Makefile
linux/Makefile		Linux Makefile
linux/bt_addrs.txt		Nodes to reprogram (Bootloader)
linux/cmd.bat		xHop commands
linux/eeprom.dat		Linux EEPROM emulation
linux/hop.route		`hop` destination
linux/prog_version.txt		Current program version
linux/xhop.route		`xhop` route

Appendix B

Glossary

ACL	Asynchronous ConnectionLess
ADC	Analog to Digital Converter
AoA	Angle of Arrival
BT	Bluetooth
CDMA	Code Division Multiple Access
CID	Channel Identifier
CoD	Class of Device
DECT	Digital Enhanced Cordless Telecommunications
DSR	Dynamic Source Routing
EEPROM	Electrically Erasable Programmable Read-Only Memory
FHSS	Frequency Hopping Spread Spectrum
GPS	Global Positioning System
GSM	Global System for Mobile Communication
HCI	Host Controller Interface
I2C	Intelligent Interface Controller
ISM	Industrial, Scientific and Medical (radio spectrum)
JTAG	Joint Test Action Group
L2CAP	Logical Link Control and Adaptation Protocol
LAN	Local Area Network
LP	Linear Program
MANET	Mobile Ad Hoc Network
OBEX	Analog to Digital Converter
PPP	Point-to-Point
PSM	Protocol Service Multiplexer
PWM	Pulse Width Modulation
RDSR	Reduced Dynamic Source Routing
RFCOMM	RF-oriented emulation of the serial COM
RF	Radio Frequency communication
RSSI	Received Signal Strength Indicator

RTC	Real Time Clock
SCO	Synchronous Connection Oriented
SDP	Service Discovery Protocol
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
TCP/IP	Transmission Control Protocol / Internet Protocol
TCS	Telephony Control Protocol Specification
TDoA	Time Difference of Arrival
TERRAIN	Triangulation via Extended Range and Redundant Association of Intermediate Nodes
ToA	Time of Arrival
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
XHOP	Multihopping Protocol

Bibliography

[1] Gps. http://www.colorado.edu/geography/gcraft/notes/gps/gps_f.html.

[2] Andreas Savvides, Chih-Chieh Han, and Mani B. Strivastava. Dynamic fine-grained localization in ad-hoc networks of sensors. Technical report, University of California, Networked and Embedded Systems Lab, 2001.

[3] Jan Beutel. Geolocation in a picoradio environment. Master's thesis, ITET ETH Zurich, Computer Science UC Berkeley, 1999.

[4] G. Strang and K. Borre. *Linear algebra, geodesy and GPS*. Wellesley-Cambridge Press, 1997.

[5] Bulusu N., Heidemann J., and Estrin D. Gps-less low-cost outdoor localization for very small devices. *IEEE Personal Communications*, 7(5):28–34, Oct. 2000.

[6] L. Doherty, K.S.J. Pister, and L. El-Ghaoui. Convex position estimation in wireless sensor networks. In *Proceedings IEEE INFOCOM 2001*, volume 3, pages 1655–1663, 2001.

[7] S. Capkun, M. Hamdi, and P. Hubaux-J. Gps-free positioning in mobile ad-hoc networks. In R.H. Sprague, editor, *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, 2001.

[8] Chris Savarese, Yashesh Shroff, and Greg Lawrence. Triangulation in ad-hoc networks: An energy efficient solution. Technical report, CS252, EECS Department, UC Berkeley, 2001.

[9] P. Bahl and V.N. Padmanabhan. Radar: an in-building rf-based user location and tracking system. In *Proceedings IEEE INFOCOM 2000*, volume 2, pages 775–784, 2000.

[10] N.B. Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *MobiCom 2000*, pages 32–43, 2000.

[11] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The anatomy of a context-aware application. In *MobiCom'99*, pages 59–68, 1999.

[12] I. O'Donnel etal. Picoradio working group. http://bwrc.eecs.berkeley.edu, 1999.

[13] Chris Savarese, Jan Rabaey, and Koen Langendoen. Robust positioning algorithms for distributed ad-hoc wireless sensor networks.

[14] Jan Beutel. Btnodes. http://www.tik.ee.ethz.ch/~beutel/bt_node.html.

[15] Nccr-mics: Mobile information and communication systems, terminodes. http://www.terminodes.org.

[16] Ericsson rok 101 001 bluetooth module. http://www.tik.ee.ethz.ch/~beutel/projects/datasheets/ericsson/rok10100\%7.pdf.

[17] Atmel avr microcontrollers. http://www.atmel.com.

[18] Mediacup @ teco. http://mediacup.teco.edu/, 1999.

[19] Frank Siegemund. Smart-its on the internet - integrating smart objects into the everyday communication infrastructure. Technical report, ETH Zurich, September 2002.

[20] Avr libc. http://savannah.nongnu.org/download/avr-libc/doc/avr-libc-user-manual/.

[21] BlueTooth Special Interest Group. http://www.bluetooth.com.

[22] David Kammer, Gordon McNutt, and Brian Semese. *Bluetooth Application Developer's Guide*. Syngress, 2002.

[23] Stefan Tschumi. Sa - positioning an ad-hoc networks. Master's thesis, TIK, ETH Zurich, 2002.

[24] Ching Law and Kai-Yeung Siu. A bluetooth scatternet formation algorithm. Technical report, MIT, 2001.

[25] Oliver Kasten. Btnode system software. http://www.inf.ethz.ch/vs/res/proj/smart-its/btnode.html\#sw.

[26] J. Beutel, M. Dyer, O. Kasten, M. Ringwald, F. Siegemund, and L. Thiele. Bluetooth smart nodes for mobile ad-hoc networks. submitted for publication ACM MobiSys.

[27] Lars Wernli and Riccardo Semadeni. Sa - blootooth unleashed, wireless netzwerke ohne grenzen. Master's thesis, TIK, ETH Zurich, 2001.

[28] Egon Burgener und Peter Fercher. Sa - grenzenlose piconetze mit bluetooth. Master's thesis, TIK, ETH Zurich, 2002.

[29] Avr in-system programmer - uisp. http://savannah.nongnu.org/projects/uisp/.

[30] Gsl - the gnu scientific library. http://sources.redhat.com/gsl/.

[31] Ernesto Wandeler. Sa - netsim. Master's thesis, TIK, ETH Zurich, 2002.