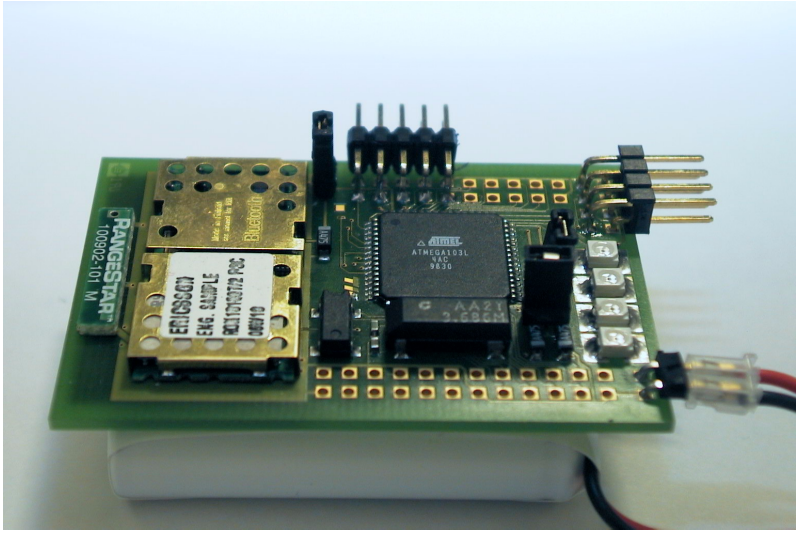
The EventCollector Concept

Distributed Infrastructure for Event Generation & Dissemination in Ad Hoc Networks



Thomas Moser, Lukas Karrer

Diploma thesis in electrical engineering

attended by Prof. Friedemann Mattern, Oliver Kasten and Michael Rohs

Abstract

In mobile ad-hoc networks, many applications need to comprehend the environment made up of the participating devices.
This thesis designs and prototypes an architecture for distributed gathering and dissemination of network information like link status or node characteristics to extract neighbourship information. Knowledge is acquired by evaluating events generated upon nodes entering or leaving each others radio range. This concept is tested both in a simulation environment as well as on a small hardware platform built around a bluetooth transceiver.

Acknowledgments

We would like to express appreciation to the following persons who have helped in the course of this thesis:

Immo Noack
Albert Weiss
Jan Beutl

Contents

CHAPTER 1

Introduction

The shift into the 21'st century was accompanied by a shift of paradigms in information technology. The personal computer age which thrived in the 1990's led over into the age of ubiquitous computing. The omnipresence of information technology is revolutionizing usage of everyday-things. Appliances are built with more and more intelligence. In a next step, these devices will begin mutual communication over wireless links on an ad-hoc basis. Undreamed-of applications will revolutionize daily living in the age of pervasive computing.

Wireless ad-hoc networks differ fundamentally from their wired counterparts. Mobile wireless ad-hoc networks (MANETs) do not rely on a base station or central control. Most of the components are mobile, thus network topology changes constantly. As nodes join and leave networks frequently, predictions about the connectivity between any two nodes at a given time is a difficult task. Moreover, connections between devices are unstable, not secure and prone to errors. Communication takes place in a peer-to-peer fashion and interaction must be enabled without prior configuration. Nodes deployed in places without existing infrastructure must cooperate in a common exploration process and keep on doing so in the ever changing environment.

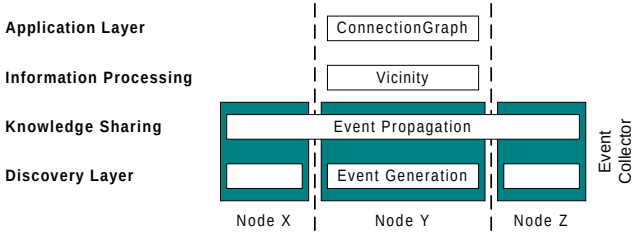
Nodes which make up ad-hoc networks may be of different kinds. Anything from supercomputers down to credit-card based low-power devices may want to interact. Since joining nodes are a priori unknown, there is little or no information about the node characteristics or capabilities of partnering devices.

The EventCollector Concept

In this thesis, we propose a distributed architecture where nodes cooperate to collect and distribute information about their environment. Situation awareness is a fundamental prerequisite for further services such as routing, tracking or navigation.

The EventCollector concept consists of four layers:

- In the “Discovery layer”, information about a node’s vicinity is collected. Various types of properties like changing link states, sensor data or information about the existence of nodes which are located in radio distance may be gathered. Generated data is encapsulated into events which are passed up one layer for storage and propagation.
- The layer responsible for “knowledge sharing” manages local storage of events and handles propagation of event data. To share collected network information, events are exchanged between adjacent network nodes and thus disseminated throughout the network. These two layers form an entity called EventCollector.
- The “Information Processing” layer processes collected events provided by the EventCollector and acts as server to entities in the “Application Layer”.
- Applications may exploit this knowledge to their benefit. For example, the history of connections can be used in multi-hop routing algorithms to determine a route between network nodes. The same information may be used to compute approximate location of mobile nodes, if the network comprises a number of stationary nodes with known location (e.g. printers, set-top boxes, or specialized “responder beacons” nodes).



Such an EventCollector architecture was implemented for this thesis. Events are generated and disseminated by EventCollectors in either a simulated or a live environment. In the live environment, the mobile nodes use Bluetooth[1] as communication medium. Gateways make event information accessible to applications running on PC's, where further processing is done. As an example, a topology snapshot and measures for mobility and average connection time is extracted out of the network information. This information is provided to an application for graphical display.

Documentation Structure

In chapter 2 of this documentation, background information to the applied technologies will be given. Topics such as ad-hoc networks, different wireless technologies and embedded systems will be covered.

Chapter 3 starts with an overview of concepts to describe mobile ad-hoc networks. Different possibilities for representation of the events and the derived information is explored. Finally, a detailed description of the EventCollector concept is given.

Chapter 4 covers the implementation of the EventCollector architecture. Detailed explanation of the software structure of both the Java simulation program and the software residing on the embedded hardware is given. Section 4 also covers software design issues for all mentioned components.

Chapter 5 describes the embedded micro controller board used as mobile EventCollector unit. Design considerations as well as operating guidelines and reference material is given here.

In chapter 6, an experimental setup and the resulting conclusions are covered. Further, problems encountered in the real world deployment of our infrastructure and the Bluetooth technology itself are discussed and possible solutions and workarounds are pointed out.

Chapter 7 lists research related to this thesis.

Finally, chapter 8 summarizes this thesis and gives an outlook over possible enhancements.

CHAPTER 2

Technical Background

This thesis bases on several key technologies such as mobile ad-hoc networks, Bluetooth and embedded design and programming. Chapter two gives an overview over these technologies and pinpoints possible pitfall and difficulties. Section 2.1 covers mobile ad-hoc networks in general, whereas section 2.2 goes into details of four wireless communication standards. Finally, section 2.3 discusses embedded designs and programming.

2.1 Mobile Ad-Hoc Networks

Compared to a fixed infrastructure network like a wired Ethernet an ad-hoc network has several particularities. There is no system administrator and no central authority. Entities involved are not known at the outset, they rather join and leave the network whenever they like. Depending on the type of network this results in a more or less changing environments. At your workplace your PDA and your PC typically make up a rather static network, while people passing each other in the hallway form a very dynamic one. Types of entities involved are not known either; the environment is heterogeneous. Cellular phones, PDAs, PCs and ear phones are among the common ones, but also cars, access control systems, cigarette vending machines and other types of electronic devices are possible. Some of these devices have sufficient electrical power because they are statically connected to the power supply sys-

tem or have some other source of energy, but small mobile devices rely solely on battery power and therefore must economize power to the extreme.

Since there is no central authority all nodes must participate in the configuration process of their network and keep on doing so constantly. One of the first question is: who is around anyway? In order to pick up communication some sort of ID or address of the communication counterpart is needed. When a connection is established, it might be erroneous and I must be prepared for it to be torn down at any moment without notice, maybe because the owner of the PDA I am communicating with walks out of range. Not just the communication link is unreliable, but also the partner itself. It might run out of power for example.

It is hard to foresee in what kind of environment an ad-hoc network will be operating. It may be a factory floor with electrical equipment that produces interference. The wireless communication technology used must be as robust as possible while keeping transmitting power at a minimum.

There is no global view of the network, every node has its own perception. Everyone knows what kind of misunderstandings can happen, when people talk at cross purposes simply because they are not on the same standard of knowledge. Nodes in an ad-hoc network must be able to cope with that difficulty.

Since transmitting power generally is kept as low as possible, the radio range is quite small. Therefore, nodes must assist each other in passing on messages to other entities. But routing is a problem, specially since the physical position is hard to determine.

2.2 Wireless Communication Technologies

Several wireless technologies are beginning to establish themselves in the mobile ad-hoc network sector. Several wireless industry groups proposed competing standards in similar and sometimes overlapping fields of application. This section compares the vision and goals compared to those of Bluetooth. The listed benefits and downsides of each protocol have contributed to the choice to use Bluetooth.

2.2.1 Bluetooth

Bluetooth [1] is an radio interface which operates in the Industrial-Scientific-Medical (ISM) Band at 2.4GHz. To replace a diverse set of non interoperable standards

for wireless communication, Bluetooth provides a universal framework for seamless communication between different devices.

In 1994, Ericsson started research for a low-power, low cost radio interface between cellular phones and accessories. 1998, Ericsson, Nokia, IBM, Toshiba, Intel joined forces in the Bluetooth Special Interest Group (SIG) to define a common standard.

Bluetooth enables up to 7 devices to communicate together spontaneously by forming a piconet. Connections support both voice and data traffic. Transceivers have been designed to be of small size and operate at low power, to be incorporated in mobile phones and PDAs working on batteries.

Careful attention has been made for worldwide usage. To circumvent any regulatory problems, the globally available ISM band, unbound by any regulatory strictures, is used as Radio Frequency carrier. As the ISM band contains many RF radiators, interference by devices such as cordless phones, microwave ovens is common. Bluetooth uses a technology called frequency hopping to cope with possible interference. The available frequency spectrum is divided into 79 channels which are switched 1600 times per second. Each channel is divided into 625µs slots to be used for data transfers.

Several requirements influenced the Bluetooth standard:

- support for both voice (high quality real time data) and data communication
- devices communicate on ad-hoc basis without user interaction.
- multiple connections are possible
- similar protection as when two devices communicate over cable is aspired. Authentication with challenge response and stream cipher encryption provide privacy.
- very small size for integration in various devices
- low power consumption

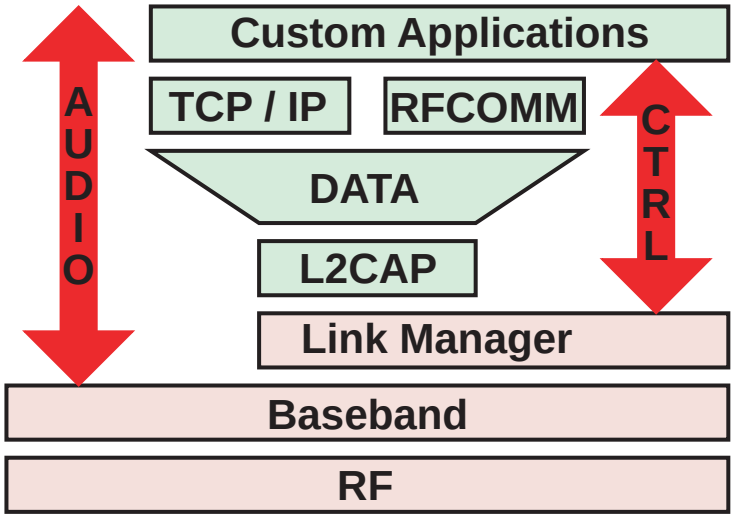


FIGURE 1. Bluetooth Protocol Stack

The Bluetooth protocol stack defines several layers:

The Radio Frequency (RF) layer sends and receives modulated bitstreams.
Baseband (BB) defines timing, framing and packet flow control on the link. Baseband provides transmission channels for both voice (SCO, Synchronous connection oriented with reserved timeslots) as well as data (ACL Asynchronous connection less point-to-point or point -to-multipoint) communication.
Link Manager assumes responsibility of managing connections, power management and enforcing fairness among slaves. It further handles link setup, security and device discovery.
The L2CAP (Logical Link Control and Adaption Protocol) layer handles multiplexing of higher level protocols, segmentation and reassembly. It provides services to upper layer protocols by transmitting data packets over L2CAP channels. Upon establishment of a connection over a channel, L2CAP negotiates several parameters such as MTU, QOS, time-outs etc.
RFCOMM provides serial cable emulation, which is used by legacy applications to communicate with other parties.
TCP / IP is defined as second major communication protocol.

In Bluetooth networks all units are peer units, distinguishable only by a unique 48Bit address. At the start of communication, the initializing unit becomes master and the other slave. The slave device synchronizes its clock with the master upon connection establishment. Master devices handle which channels the slaves shall send on and which slave unit is allowed to send. Connection establishment needs typically about .6 to 1.2 seconds. When not in use, units can sleep in a stand-by state which is beneficial for battery operation. Every 1.28 or 2.56 seconds (dependant on configuration) a unit will wake up and listen for incoming requests.

A connection is made by a PAGE message sent out by the initiator, if the receiver's address is know or by an INQUIRY message followed by a PAGE message if the address is unknown. The INQUIRY message is typically used for finding unknown devices which provide public services such as printers, gateways etc.

Units communicate with 721kBits/second with up to 15m distant devices. To save power and minimize radio interference problems, an RSSI (Remote Signal Strength Indicator) measure is used to adapt RF signal strength.

The Bluetooth SIG is promoting new usage models which create additional benefits for users of portable telephony. The two-in-one phone is a bluetooth enabled handset which acts as a portable phone at home using a Bluetooth basestation or a conventional GSM phone when used outdoors. Another frequently mentioned model is the briefcase trick. PDAs or laptops connect to the internet or company network via cellular phone which is stored in your briefcase. Automatic synchronization is another proposed benefit. As soon as one enters the office with a PDA, address list and calender are updated automatically.

Bluetooth features some negative aspects for application in mobile ad-hoc environments. The standard aims to replace point to point serial communication, thus building up on a master slave architecture. A truly peer to peer infrastructure with equal entities is not intended. Further, high energy consumption has emerged as a problem in mobile equipment relying on battery power.

2.2.1 HOME RF

The HomeRF [2] working group is developing an open specification targeting wireless communication in home environment. Both voice and data communication are defined. Just like Bluetooth, HomeRF closely integrates TCP/IP with peak data rates up to 1.6Mb/s. Ad-hoc communication between asynchronous devices is possible, a control point is only needed for audio data transfers.

Generally, HomeRF is very similar compared to Bluetooth. The only notable difference from the users perspective is an enlarged range up to 50m (which yields in higher power consumption) The higher range may be more suitable for covering an entire home whereas Bluetooth targets the Personal Area Network (PAN). The higher power consumption makes it difficult to deploy HomeRF in truly mobile devices in and ubiquitous environment. The most notable downside of HomeRF is its prevalence which is limited to mainly the United States.

2.2.2 IrDA

In 1993, the Infrared Data Association [4] set up hardware and software standards for infrared communication links. The IrDA protocol stack supports similar usage models as those of Bluetooth. Legacy applications which rely on serial ports are supported via serial cable emulation. IrDA is state of the art in printers, handheld computer and camera equipment. The advantage of using Infrared over Radio Frequency include reduced cost, lower power consumption and less regulative restrictions for usage. The most significant disadvantage of using IR as carrier is the line-of-sight restriction and a limited range. Further, IrDA supports only asynchronous point-to-point communication between 2 devices.

2.2.3 Wireless LAN IEEE 802.11b

The 802.11b [5] standard issued by IEEE defines an RF physical Layer and Medium Access Control for wireless LAN connectivity. The goal of IEEE 802.11 is providing LAN based applications in a large radio coverage with bandwidth up to 11MBits. Unlike Bluetooth's paradigm, Wireless LAN relies on central infrastructure and does not focus on ad-hoc peer-to-peer communication. Just like Bluetooth, 802.11b uses the ISM Band for communication deploying. Direct Spread Spectrum Sequencing (DSSS) is used to handle RF interference.

At the moment, hardware and power requirements do not encourage the deployment of Wireless LAN technology in mobile embedded systems. Furthermore, it is heavily influenced on data communication and thus does not provide any synchronous communication capabilities. 802.11 is widely used. Wireless LAN is enormously popular nowadays.

2.2.4 Proprietary RF Communication

Another possibility for communication is using proprietary radio links over RF technology. Numerous commercial transceivers are available, some specially

designed for low power systems. With bandwidth up to 100kbps these transceivers would fit well into the EventCollector architecture. While consuming very little power, one major disadvantage must be considered: Transceivers do not include baseband and link management specifications like that of Bluetooth. Implementation of these protocols is not only complex but also prone to errors. Disadvantages outnumber possible benefits using a proprietary protocol in the intended setup of this thesis.

2.3 Embedded Technology

Paradigms of embedded System differ greatly from their counterpart in large scale designs. Limited resources like memory or computing power may pose pitfalls which are not encountered normally.

The most limiting factor in embedded designs is power consumption. Often, mobile embedded platforms run on battery power. To achieve long battery cycles and low weight, designs must be as efficient as possible regarding power consumption. Several techniques help conserve energy.

- Highly integrated circuits: Designs with large integration need less energy.
- Low power components: Operating voltage is proportional to power consumption, thus energy saving components running on as low as 1.5V are chosen.
- Low clock frequency: CMOS components' power consumption is proportional to the clock frequency.
- Stand-by-Mode: Energy-thirsty devices are disabled or put into sleep mode when not in use

Table 1 on page 17 gives an overview over the power consumption of different devices.

TABLE 1. Energy consumption of different devices

Device	Power consumption	Normal Battery Cycle
800MHz Pentium III	60 mW	
1 LED 5V	50 mW	
4MHz Atmel Mega 103L	20 mW	

TABLE 1. Energy consumption of different devices

Device	Power consumption	Normal Battery Cycle
Mobile Phone StandBy	8 mW	250 hours
Digital Watch	0.005 mW	4 years

Another restraint is processing power. Many embedded platforms use 8 Bit CPUs at low clock frequencies. This yields in low power consumption but also restrict processing power.

A wide spread characteristic for embedded designs is small memory footprints. Frequently used embedded microprocessors such as MicroChips PIC have less than 1kByte RAM. As mentioned above, memory should be incorporated into the micro controller unit. But, static RAM is expensive and relatively large compared to other parts of the MCU such as ROM, Flash or logic.

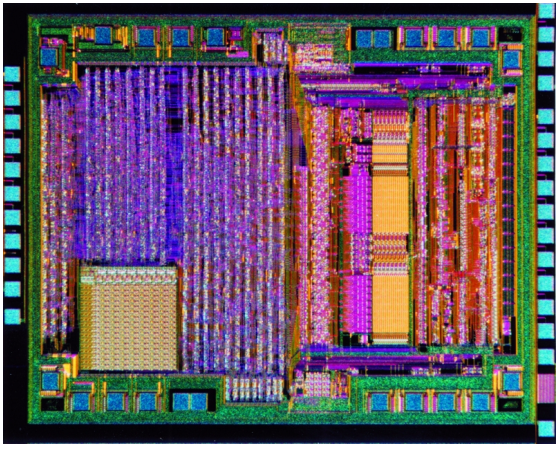


FIGURE 2. Die of AT90LS2XXX Series MCU

Figure 2 on page 18 shows a die image of an AVR AT90LS2XXX Series MCU. This micro controller contains 2kByte Flash memory and 128Byte RAM and EEprom Memory. Memory is usually recognized as regularly structured areas on the die. In this case, Flash Memory, which is located on the lower left side of the die, takes up about the same amount of space as the RAM and EEPROM combined, (situated in the center) but stores up to eight time as much information. Manufac-

turers try to keep the size of chip dies as small as possible, since space on silicon wafers costs money.

To overcome memory and processing power restrictions, embedded platforms are often programmed in assembler. Frequently, deployment of an operating systems is abandoned in favor of speed and low memory requirements.

Programming in general is less comfortable in embedded systems. As embedded devices often have only little or no input / output capabilities, programming is done on a host platform using a cross compiler. This renders debugging quite difficult. Further, requirements on code quality is extremely high. Embedded systems are often used in a zero-configuration, security-relevant areas where failures are not tolerated.

Moore's Law also applies to embedded systems. In normal designs, performance is doubled every two years. This is achieved using faster designs with more transistor and advances in technologies used. Higher speed and increased number of transistor augment power consumption, which is not desirable in embedded components. Hence, new designs will primarily rely on superior technology to gain performance at a much slower pace compared to normal designs.

Another restraint is the price of embedded systems. In an environment where computing is ubiquitous, devices must be very cheap. This again imposes requirements towards manufacturing and design of such systems.

CHAPTER 3

Concepts, Solutions & Design Considerations

The goal of this thesis is to define a distributed infrastructure for an ad-hoc network where all participating nodes contribute their share to the exploration of the network. Depending on its capabilities nodes may choose to put more or less effort into producing, storing and distributing network data. With the collected information, an application running on a node may compute properties of the network that seem useful. However, such an infrastructure poses some problems. First there is the question of how to describe and represent a mobile network in general. This is important to the nodes that need network knowledge for their task, for example, a node that runs a “Vicinity” application. Next, some possible approaches to gather neighbourhood information from a zero - configuration network are presented together with our design choices for an implementation. This basically is the exploration for the Event Collector concept. Next this concept is described in detail. Three possible information extraction algorithms are then presented and finally such an information extraction is exemplary demonstrated.

3.1 How to Describe a Mobile Network

First of all a remark on the term “network”. A network usually refers to entities that are connected by communication lines. If one of these entities looses connection, strictly spoken it doesn’t belong to the network anymore. Since a connection in a

mobile ad-hoc network may be rather unstable and not clearly up or down we do not use this term very strictly, or use “vicinity” instead. Imagine an area with some nodes which do not see each other at all. Using the strict definition, this is not a network since no connections exist. By moving one node about and letting it make contact with others, nodes learn something about their vicinity.

So, we want to explore a mobile ad-hoc network. But how is such a network described? A wired Ethernet infrastructure is in many ways a simpler case. Generally it is set up by a network administrator and is quite stable and static. Not so an ad-hoc network. Neighbours change, connections are unstable, an entity may suddenly stop responding and come up again. It is difficult to find a global view on the network. Below a representation form for ad-hoc networks is presented.

3.1.1 Graph

A network topology typically is described by a graph. Nodes of the graph represent entities while edges represent a bi-directional communication link. But for an ad-hoc network there are more important properties than just “is connected” or “is not connected”, because it is not implicitly stable. Therefore a graph representation is extended to describe not just a topology but all the properties of the network. So, a node still corresponds to a communication device but an edge now describes the whole relation from one node to the other. Such a relation basically is a record of properties. We have specified three exemplary properties (i.e. topology, connection weight and relative mobility) as described later in this chapter. As with the above graph, such a network of N nodes can be stored nicely in a N*N matrix. Every node has a relation to every other one, so the graph is fully connected. If two nodes have not had contact with each other, it still is a relation, i.e. contains information. Note that the graph represented by this matrix is directed! This is more flexible and contains more information. For Example imagine a node that sees another one, but is not detected by the latter. This scenario is realistic, since the detection of nodes is not necessarily dependent on an rendezvous or connection establishment.

In Figure 3 on page 23 a sample graph is depicted. This fully connected graph can be described by the matrix shown in Table 2 on page 23. The relation from 1 to 2 (a_{12}) and the relation from 2 to 1 (a_{12}) are different, since the graph is directed. Table 3 on page 23 is an example of such a relation record with the properties “Topology”, “Connection” and “Mobility”.

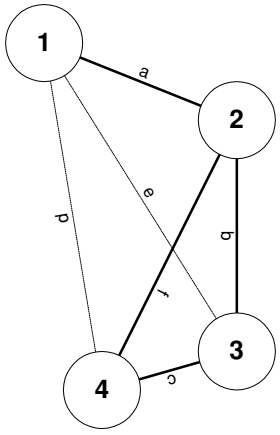


TABLE 2. Sample Network Matrix

	1	2	3	4
1		a_{12}	e_{13}	d_{14}
2	a_{21}		b_{23}	f_{24}
3	e_{31}	b_{32}		c_{34}
4	d_{41}	f_{42}	c_{43}	

TABLE 3. Sample Relation Record

Topology	Connection	Mobility
true	0.435	22

FIGURE 3. Sample Network Graph

3.1.2 Snapshot Yields Topology

A topology of the network may be obtained by taking a snapshot of the connection state of the network. It is important to stress that a topology is associated with a point in time, since the topology might change fast. Because there is no global, external view of the vicinity the information about nodes and links is generated by the nodes itselves. Then it is propagated throughout the network. In a practical scenario, a topology is always based on historical data. A current and accurate topology can never be obtained due to propagation delays. The time spread however, in which it can be expected to be more or less accurate depends on how dynamic the network changes. The determined topologies of two entities may differ from each other because both might not have the same view of their surroundings.

3.1.3 Statistical Description

As described in the previous chapter, a current topology cannot be known, instead, a snapshot of a past instant is seen. And since the intentions of nodes (physical path, uptime etc.) is not known, no accurate prediction of the topology in the future can be made. However, under the assumption, that the general habits of a node will be similar over time, it should be possible, to make a statistical prognostication. For example a printer normally stays in the same place while a PDA will be carried

around by its owner, which indeed does have particular habits. Maybe the latter gets himself a coffee six times a day and therefore walks from his office to the machine and back. The coffee machine itself probably will be turned on during the day and shut itself off at night, and hence disappear from the vicinity. Depending on the application one can define measures in the relation record to reflect such statistical values, for example average uptime. Note that all this information is collected purely by observing the network, no other knowledge is involved! Possible relation measures will be discussed later in this chapter.

3.1.4 Further Information

Zero configuration networks must configure themselves. Nodes must gather all necessary information. Such information could be a link state as seen above, but also information about the nodes itself. For example, if a node would be known to be a coffee machine, one could figure that it is rather static and immobile. But isn't this a contradiction to the previous statement of zero configuration? We believe otherwise, if these properties are restricted to a sort of "factory setting". Naturally, a coffee machine will stay a coffee machine as long as it exists. It's also clear that a coffee machine does not walk around, i.e. is spacially rather static. So why shouldn't it know about it's identity and inform the other nodes? Neither the user nor the service-man has anything to do with configuration. This is basically the idea of SDPs (Service Discovery Protocols). On the lower layers, Bluetooth incorporates an informal parameter (device class field) that can be set at production time and that other Bluetooth devices can read out. There are no standards however. We did not occupy us with this subject but it could be an extension to the EventCollector architecture and certainly bring in additional usable information.

3.1.5 No Common Perception

It must be kept in mind that the perception of the network may differ from node to node. One big issue is time. There is no global time service by definition and there may be unknown latencies on network links. It is troublesome for the nodes to try to agree on a common time base. The perception may also differ depending on the information received and algorithms used. Even if we assume that everybody uses the same algorithms to calculate relation weights between two nodes, the results are likely to differ. Not all information is received at the same time on all nodes, if it is received at all. Information may get lost or may not be propagated any further because it reached the limit of hops (TTL). Also, a node may have discarded information because of limited resources. What ever applications use the generated topology data must be able to cope with its errors and imperfections.

3.2 Possible Approaches

So we want to explore a mobile network from scratch, without any prior knowledge, find the involved entities and calculate relations between them in a truly distributed fashion. Different approaches are discussed in this chapter and our choices will be presented. An important point that influenced our decisions are the limited resources on embedded platforms, especially storage space. Previously the representation form of an ad-hoc network was described. Not every node may want to or is capable of really building such a representation. But every node must provide basic functionality to contribute to the dynamic network infrastructure configuration process.

3.2.1 Zero Knowledge Exploration

What can a node find out about it's surroundings? The first step, but not necessarily the easiest, is to discover possible neighbours. With a fixed infrastructure available, there are simple solutions to this problem like ARP on an Ethernet network. Lacking such a fixed infrastructure things get more complicated. Luckily most wireless technologies feature some sort of discovery algorithm. Bluetooth for example incorporates an inquiry algorithm that finds other BT devices within its reach. Since BT uses frequency multiplexing together with frequency hopping this is a nontrivial task and takes some time. It gets even more complicated if some devices go into power down mode or are actively transmitting themselves. This subject will be discussed in detail in chapter 6. Another way could be to just listen to the network traffic and remember the addresses of the nodes involved. While this works well on Ethernet or other multiple access technologies with only one frequency, again, this is not easily feasible on the more complex wireless communication systems. Important for now is, that there are technology specific methods to find neighbours.

In a second step the connections can be analyzed. Is it possible to open a data link to the neighbours? Are there any other properties that can be obtained, like link quality, throughput, latency, error rate or physical distance?

Bluetooth specifies some functionality to measure the quality of a link and the strength of the signal received. Interpreting these measurements as physical distance however is not straight forward, for example the transmitting power would have to be known. The Bluetooth specification is not clear in these points.

The next thing to get to even more information about the vicinity is to share the collected data with the neighbours. My neighbours might see other nodes than I and

thus extend my range of perception. Also these entities might be online longer than I am and therefore I can learn something about the time before my appearance. By sharing data with others the range of my perception increases while the difference of perception of the environment between the nodes decreases.

3.2.2 Propagation of Information Through the Vicinity

In what form will the identity of nodes be propagated? It could be done explicitly in some sort of identity packet, together with associated properties. Alternatively, sending the state of a neighbourship relation between two nodes, which contains the identities of the two neighbours, spreads the fact of their existence implicitly throughout the network. We chose to use the second, implicit form of propagation. It is much simpler and we don't have any node properties to share anyway at this point (as described above). The later introduced protocol between the nodes however can be extended to support an explicit distribution of identities and other additional information if desired.

Propagation of connection state information can be done in several possible ways:

- a client server architecture: A node asks a neighbour to reveal some information (information pull). Through this polling policy local data is the most up-to-date possible because fresh data is received whenever it is needed. On the other hand, if nothing changes, unnecessary transmissions are made.
- mandatory propagation: Every node must accept new data and propagate it, maybe even store it (information push). This results in a flooding algorithm. The data will propagate fast, transmissions will only be made when the topology changes. But nodes get data they might not want, possibly even large amounts of data.
- combination of push and pull: One could try to combine the advantages of the above two methods.

We chose to implement an information push algorithm through flooding. Mainly because of the following reasoning: A node might not want to receive some data, but maybe some further down the line does. As individual entities cannot judge whether data is relevant or not they have to propagate it all. (Later in this chapter we will see that a node can influence it's own relevance by setting the TTL value). As flooding doesn't require routing and can be stateless, it is easier to be implemented on embedded platforms.

Now, what exactly should be propagated? “Node A has seen node B”? This is an event and can be recorded and distributed without significant problems. But what about a connection property? “Node A has seen node B and the reception is 4 by 5”? What if the connection between these two nodes now slowly gets worse? Here it is hard to define the moment to generate an additional event to propagate changed link quality. This is no problem however with an information pull. The client gets the most recent measurement whenever he needs it. In an information push however a threshold must define how much a property may change until it should be retransmitted. Or maybe a change-rate threshold would be more adequate? Another way would be to send the measured values periodically, but at what interval? What if a property changes very often? In addition it is imperative, that all node employ the same mechanisms or at least publish them, so everybody knows when events are generated, and when not. (Lack of information is information as well, “no news, good news”).

We chose to implement an event based reactive infrastructure despite the problems mentioned above. This way, nodes can be a lot dumber, both in behavior and in memory. Also reaction times are smaller.

3.2.3 Event Structure

Basically an event is made up of the event source, the counterpart, a time stamp and type. In the manner: “I(55) found him(34) at 12:34:02” or “I(12) lost him(67) at 23:45:32”. A collection of such events can then be processed, for example to compute the total connect time by adding up the time spans between a each pair of found event and the corresponding lost event. But what is the corresponding lost event? By sorting them chronologically it should be immediately the following one. But if a lost - event was thrown away during propagation, there are two consecutive found events. Maybe a lost event together with the next found event get dropped somewhere. That way a connection time that is possibly much too long will be calculated without even noticing the error. Numbering the events would help to detect missing events but would also increase the event size and introduce new problems, for example when devices reboot. This solution proved to be much too unstable on unreliable networks and quite often totally wrong conclusions were made based on the collected events.

In a second revision we chose to pair up the lost and found events into one single event. It is made up of the event source, the counterpart, the found time, the lost time and some flags. When two nodes find each other, each generates such an event with the both found and lost time set to the actual time. This event testifies that these two nodes have seen each other. It doesn’t say anything about the duration of

the connection, it is assumed to be zero (lost and found times are equal). It looks somewhat like this: “I(55) found him(34) at 12:34:02 and lost him again at 12:34:02”.

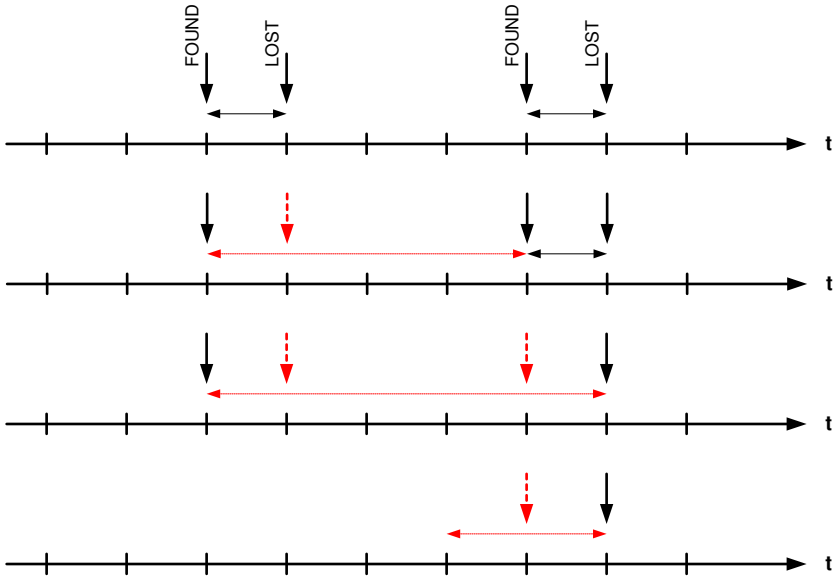


FIGURE 4. Possible Failures

When the two nodes loose sight of each other, both update the previous event with the correct loss time: “I(55) found him(34) at 12:34:02 and lost him again at 13:51:29”. This event is propagated just like before. It is easily matched to the previous, corresponding found event by the first three fields of the event, these values have not changed (source identity, counterpart identity and found time). All other nodes update existing local copies and keep on propagating.

To keep the system current during a long connections the nodes should produce spontaneous updates at regular intervals. To distinguish them from a final event a “FINAL” flag exists. These updates will be sent with the “FINAL” flag cleared, to mark the update as not final, meaning both still see each other. This is some sort of compromise to help with information extraction. It makes the algorithms more stable, since a lost event that gets lost could wrongly indicate two entities to be neighbours. For example, if no update was received from two neighbours for a rather

long time, they probably lost sight of each other but couldn't generate a final lost event or maybe they could but it wasn't propagated all the way to me.

The described algorithm is a pessimistic one. It never produces longer connect times than actually happened. Another advantage is the reduction of stored data. Updates naturally do not generate more data on storage, which is a major bottleneck of embedded devices.

Using the mentioned algorithm, let us look at an example. Assume, that a node receives the events listed in Table 4 on page 29.

TABLE 4. Example Using Events I

me	him	foundTime	lostTime	final
1	2	12:43	12:43	false
2	1	12:43	12:43	false
1	3	12:45	12:46	true

This information can be illustrated on a time line shown in Figure 5 on page 29. The first two events are displayed as a single point in the time line. The third event specifies an connection between node 1 and 3 during 1 minute starting at 12:45.

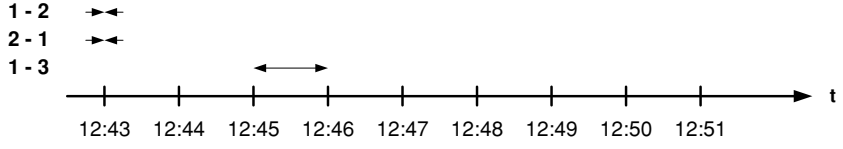


FIGURE 5. Time Line I

TABLE 5. Example Using Events II

me	him	foundTime	lostTime	final
1	2	12:43	12:48	false

Next, an update event as listed in Table 5 on page 29 is received. By looking at "me", "him" and "foundTime" we see that it is not a new event, but an update to the

first event in the previous list. We can update this entry in the event list by replacing the “lostTime” with the new one. The “final” flag is false, meaning that the neighbourship between “1” and “2” still goes on. The new time line is illustrated in Figure 6 on page 30. Now we know for sure that “1” has seen “2” between 12:34 and 12:48, maybe longer. But what about “2”? If we know that the relation between the two is truly symmetric, we can safely assume that “2” has seen “1” as well. In this case the information is redundant. On the other hand, the fact that “1” has seen “3” but not the other way around may tell us, that this relation is asymmetric. For the Bluetooth node network this could mean than node 3 does not know about the EventCollector infrastructure and does not generate events.

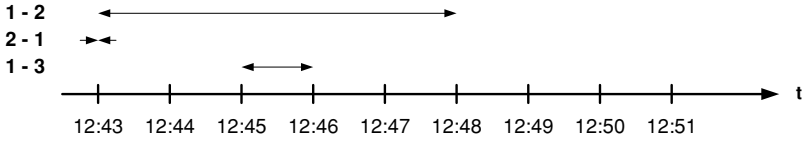


FIGURE 6. Time Line II

TABLE 6. Example Using Events II

me	him	foundTime	lostTime	final
2	1	12:43	12:50	true

A last event finalizes the connection between 1 and 2. Note that the FINAL flag is now set. Node “2” obviously realized at 12:50 that it has lost sight of “1”. After updating the second event in the event list with the correct “lostTime” and changing the “final” flag to true the new time line looks as depicted in Figure 7 on page 30.

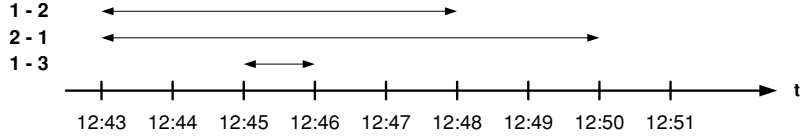


FIGURE 7. Time Line III

3.2.4 Storage of Collected Information

It is obvious that information in a distributed system is stored in a distributed way. This results in redundancy, which cannot be avoided, respectively even may be desired. It is also clear, that not every node can store the same amount of data since there is a factor of some 100,000 between the memory available on a micro controller and a desktop PC. So every node should be able to decide on its own, how much data it wants to store, if at all.

To get a lot of historic data to extract some habits of nodes it is convenient to have some device around with large of storage space, like a PC. But of course, it is not necessary.

3.3 Architecture of the EventCollector Infrastructure

This subchapter describes the EventCollector architecture. As was explored previously, it consists of nodes that interact with one another. They collect events and share it among them. This distributed collection of events together with the propagation makes up the EventCollector infrastructure. Every node may process the received data on its own and compute the topology or other properties of the network.

There are different nodes possible: mobile embedded Bluetooth devices, laptops with Bluetooth capabilities, PC with Bluetooth capabilities, or just abstract entities in the simulation environment. Despite the big difference in resources, bandwidth and mobility, no assumption is made on the capabilities of the node. In other words, all nodes are equal, and have the same functionality. Properties of nodes and connections are extracted solely from the collected events.

3.3.1 The concept of the EventCollector

The most important entity in the EventCollector architecture is the EventCollector. Simply speaking, it generates, collects and propagates events to other EventCollectors. We have such EventCollectors everywhere: embedded on the BT - devices, on desktop computers and in the BTSim simulation environment. The communication runs over a Bluetooth connection between the Bluetooth capable devices and over

TCP/IP between the Java EventCollectors, with gateways in between the two worlds.

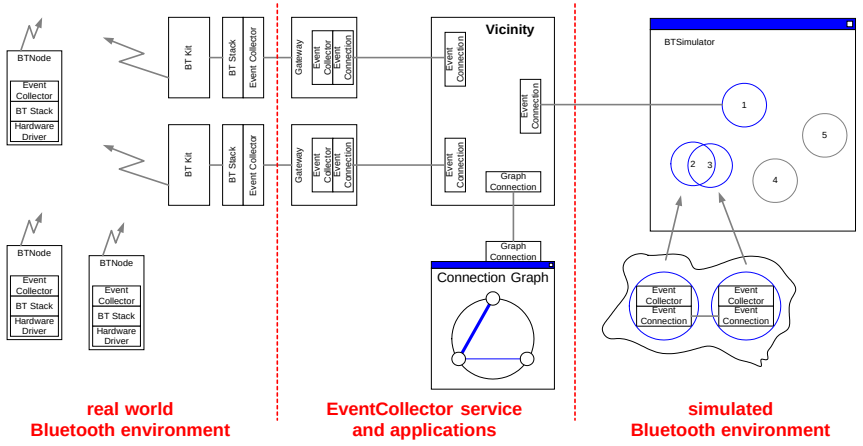


FIGURE 8. General Architecture Overview

3.3.2 Tasks of an EventCollector

An EventCollector has three main tasks.

1. collect events: Each event received is considered to be inserted into the collection. Whether it actually will be inserted or not depends solely on the EventCollector. Depending on the resources available it will keep a long list or only the most important events, according to its own priority function (filter). For example an EventCollector with very limited resources may choose to keep only the newest 10 events.
2. generate events: If an EventCollector sees another node it generates events at regular intervals as long as the other node is in sight. This is described in the previous chapter. If both involved entities are EventCollectors, they both symmetrically generate events. If only one of them is an EventCollector and the other one does not know anything about this infrastructure, naturally only the former generates an event (asymmetrically). When an EventCollector looses sight of one of its neighbours, it generates an update event with the final flag set, to indicate the end of the neighbouring relation to this particular node.

3. propagate events: Upon connection establishment between two EventCollectors, a certain amount of events is exchanged. Each Collector sends its top ranking events. It decides on its own, which newly received events it wants to keep. This propagation results in a flooding of the vicinity with events, the gathered information is efficiently spread.

Java EventCollectors can be cascaded and are therefore used in different places in our setup: Every node in the BTSim simulation environment has such an EventCollector, used in the same way as its counterpart on the embedded platform to generate events and propagate them. A Java gateway has one as well, but it doesn't generate events, it only stores events generated by the Bluetooth nodes and distribute them further into the "Java World". A Java Vicinity uses one as well, to collect events either from the EventCollector inside a Java gateway or inside a node in BTSim. This architecture is depicted in Figure 8 on page 32. An EventCollector as a stand alone entity can be used as a reservoir of events to serve other entities with large amounts of historical data, if desired. By extending an EventCollector and overwriting the addEvent() function events may be filtered before they are inserted into the list. This way it can be used to implement some sort of data mining on the events that flow through the "Java World".

3.3.3 Event Exchange Protocol

EventCollectors run on very different platforms, sometimes with very limited resources regarding memory and processing power. The protocol used to communicate between the EventCollectors must take that into account. There must be some sort of flow control, to avoid that a small device gets overrun with data. But a powerful entity should still be able to receive a large number of events. Bandwidth is not a big issue at the moment, because Bluetooth delivers a relatively high throughput, compared to the amount of data a node can store. Finally the protocol should be extensible to fix possible shortcomings of the first version or extend its functionality.

Since events are used to spread information through the network, a packet based protocol is used. It is held very simple. There are two types of packets: flow and event packets. The flow packet defines the protocol version and passes commands or parameters to the opposite side. The event packet contains one single event, together with some protocol parameters. Flow - and event packets are exactly of the same length. This way no delimiter or stuffing is needed and buffer allocation on

embedded devices is greatly simplified. (There is no dynamic memory allocation or operating system available on small embedded devices.)

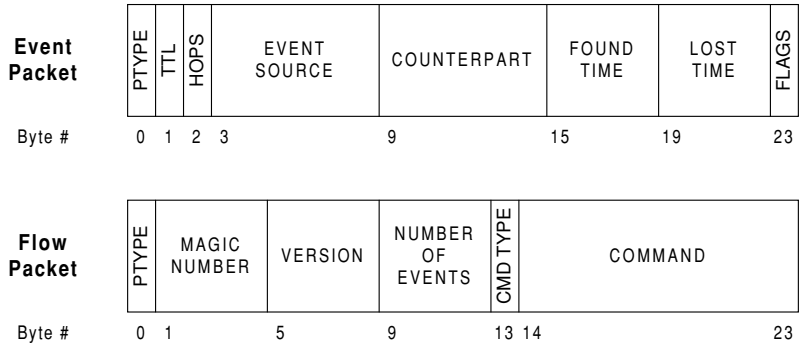


FIGURE 9. Flow and Event Packet

A Flow packet starts with a packet type field (0x2). Next there is a magic number to give additional security upon connection establishment. A newly discovered neighbour might not speak our protocol or know anything about the event infrastructure. But it might still accept a connection request or even connect to us. To minimize the chance of falsely accepting some trashy data as a flow packet the magic number (0x42744E72) is checked. Next the version field defines the protocol version. This is currently version 1. The next four bytes contain the number of events to be sent. The command type and command field can be used to define user commands or to exchange data. Currently these fields are not used.

3.3.4 Flow Control

Upon connection establishment an EventCollector sends a flow packet to its counterpart, with the upper limit N of events it wants to receive. The counterpart then sends its top N ranking events. This way a small device doesn't get overrun with large amounts of data if it connects to an EventCollector with a large history. After the initial exchange of events, all new events (either just generated ones or received ones) will be propagated to the counterpart. One exception exists: if a device initially set the upper limit of events to receive to zero, it will not receive any events at all, neither upon connection establishment nor during the connection.

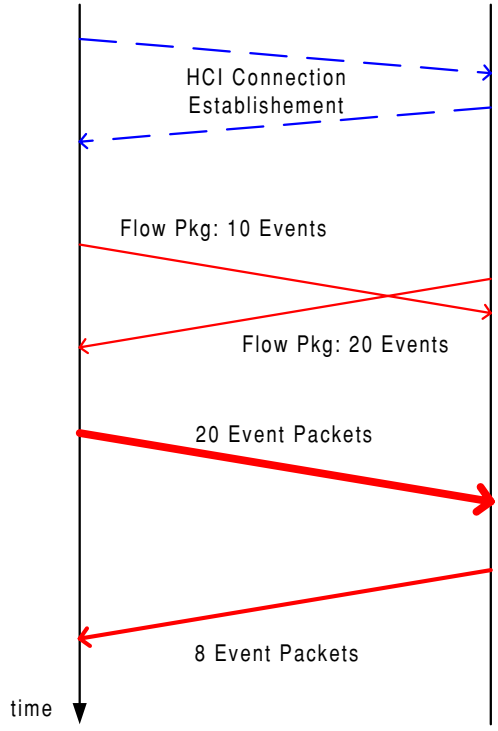


FIGURE 10. Protocol Overview

3.3.5 Flooding Algorithm

Precautions must be taken with the above described flooding algorithm to limit the amount of data spread and to limit loops and unnecessary transmissions. There are several approaches possible. The most efficient way would be if each node would remember the packets it already transmitted. The problem, as seen before, is storage space on embedded devices. An EventCollector normally contains a list of collected events, so why not use it for this purpose? That's why received events will not be propagated if the event is already in the list. All neighbours either have received this event already (regularly or at connection establishment) or they chose

not to receive it by requesting a too small amount of events at connection establishment. Updates to existing events naturally will be propagated. If the possible receiver of the event packet is identical with the source of the event, it will not be propagated, because this is obviously the source of the event. These mechanism help reduce unnecessary transmissions, but cannot guarantee the avoidance of infinite loops since the nodes are not obligated to store all the events, if any at all.

To definitely avoid infinite loops a TTL (time to live) field is used, as know from the internet protocol IP. It limits the amount of hops an event will take on its way through the system, thus avoiding infinite loops. But it also limits the reach of an event. By initially setting the TTL to a specific value a node itself can limit the range of propagation. A mobile node like a coffee cup does not need to be known in the whole world. On the other hand a printer might want to distribute its identity rather widely to offer its services and also because it is always online and therefore a good reference in an otherwise dynamic network.

The nodes itself set the initial TTL value. But the other nodes don't know at which value it was set. Therefor the actual number of hops taken during the propagation cannot be calculated by the TTL alone. That's why another counter "countUp" is introduced. Initially it is set to zero and is increased by one each time it is propagated. The number of hops can be used in different ways: For example an event collector could use it to decide which events to throw away when no more storage space is available. Or it can serve as a rough distance estimation over several nodes.

When an event collector receives an event it checks whether it is already known by comparing the "me", "him", "foundTime" and "lostTime" fields with those stored in the event list. If they actually match, then it further compares the TTLs. If the newly received version of the event has a larger TTL it obviously took a shorter propagation path. In this case the TTL and countUp of the stored event are updated and the event is further propagated. This ensures, that only the shortest propagation path is reflected in the TTL value.

If the received event is an update packet however (i.e. different lostTime), always the new TTL is stored, whether it is larger or not. This avoids a possibly very long propagation path. If this algorithm really works well in practice should be investigated further. Maybe it would be better not to update the TTL at all. In other words: make the fastest propagation path relevant, not the shortest one.

3.3.6 Error Correction

The protocol implements no error correction. Instead already error corrected transport protocols (HCI and TCP/IP) are used. This way no error correction needs to be implemented

3.3.7 Shortfalls of the Protocol

There is a potential problem with the chosen flow control mechanism. Consider the following scenario: Node A connects to node B and the latter can only receive a dozen events, which will be delivered by A. Now another EventCollector with lots of stored data connects to node A and sends many of its events to it. Node A recognizes them as new events and sends them to node B, as specified by the protocol. Here node B is overrun with data. This problem could be fixed by extending the flow packets. Node B could repeatedly send the current amount of events it is ready to receive. Node A would have to keep track of how many events it already has sent to B, which would introduce more states to the connection and complicate things. Another possibility would be to implement some sort of “Stop - and - go” protocol. But since the Bluetooth receiver has relatively large buffers compared to our micro controller, this wouldn’t really help.

Since the above described situation is unlikely to happen in our setup and since it doesn’t matter in the simulation, we chose not to fix this shortcoming. For a general application however it should be done.

3.3.8 Extensions to the Protocol

There are different provisions made for an extension of this protocol: A flow packet includes a one byte command type field and a 10 byte command or payload. Through this mechanism, new commands can be defined and be sent back and forth in flow packets. Flow packets will never be propagated, they are solely for communication between two neighbours. If propagation is desired, other packet types can be introduced. For example one could define a “environmental data packet” to flood the vicinity with sensor information. Such a packet would have the fields “PTYPE”, “TTL”, “countUp” and “me” just as an event packet. The remaining 15 bytes then could hold sensor data and sensor type, including a 4 byte timestamp of the mesurment. The normal flooding algorithm then could be used to propagate this information, so only minor changes would need to be made to the event collector algorithm. For backwards compatibility, there is a protocol version field. This way the two

communication partners then can agree upon a common protocol version. Or more precisely, the higher version protocol should be able to recognize and run the lower one. For example, it doesn't make sense to send an "environmental data packet" to a node running protocol version 1 (the one we are running), because it wouldn't recognize it and throw it away.

3.4 Information Extraction

After having collected all these events one wants to process information to learn something about the network or the vicinity. But what kind of information can be expected? How exactly can it be gained? How reliable and up-to-date is it? In this chapter these questions are discussed and three exemplary information extraction algorithms are presented. These sample algorithms are used by the Vicinity application to extract network information from its event collection.

3.4.1 Topology

As discussed at the beginning of this chapter a snapshot of the neighbourship relations of the entire network yields the topology. A "topology relation" is either true or false (i.e. "Neighbour or not", "see each other or not"). But how exactly can such a snapshot be extracted from a event collection? One must run through the event collection and look for events that have a foundTime prior and a lostTime after the given moment. This yields only the active neigbourhoods that have generated an update packet after the moment of interest. But what about two nodes that still see each other, but haven't sent an update recently? Depending on the preferences the final - flag can be taken into account. If the emphasis is set on accuracy, an event with a lost time prior to the moment of the snapshot and the final flag NOT set is nevertheless interpreted as a final lost event. If actuality is more important and possible errors are acceptable, a lost event with the final flag NOT set may not be seen as a final lost event. Instead the two entities may be regarded as neighbours with an update soon to come. Here it would be nice to know at what intervals an entity generates such updates to judge whether an update is still to come or whether the final lost event has been lost. We haven't defined such an interval because depending on the application the compromise between actuality and unnecessary transmissions may look vastly different.

In addition some redundancy can also be used to improve the estimation of a symmetrical neighbourhood relation: if two neighbours are eventCollectors, both have

generated an event (~duplex, bilateral, symmetrical), which is redundant information. Example: to calculate the connection up time, the time spread is the union of both time spreads.

Also it can be decided whether both neighbours are EventCollectors or whether one of them is just a “dumb” Bluetooth device. This can be done by observing the above described relations. If it is symmetric (or duplex) both are EventCollectors that run the proposed protocol and generate events. If the relation is just unilateral the counterpart is likely to be a “dumb” device like a Bluetooth enabled phone, that doesn’t know anything about the EventCollector infrastructure. This is almost certainly true if the TTL of the event packet isn’t close to zero (the difference between the TTLs of two neigbours is unlikely to be bigger than one). Gathering information about “dumb” devices is possible, because events are not generated upon a rendez vous of two entities, but already, when one sees the other. (Actually, it still is a rendez vous, but on a very low level, not accessible by higher Bluetooth layers. E.g. an application doesn’t realize that it get’s inquired, even though it answers the inquiry actively)

Looking at the algorithm one can see, that there is no such thing as an current and accurate topology. Depending on the latency of the propagation and the length of the periodic update interval the necessary events are only available with a certain delay.

3.4.2 Connection

A topology is a snapshot of the neighbourhood relation of the entire vicinity, as seen previously. Taking a time spread (or observation interval) in place of a single moment a “statistical topology” is received. We called this type of relation “connection weight”. It is a value between 0 and 1, contrary to the “topology relation” above, which is just true or false. This connection weight can be interpreted as “percentage of uptime”. It means for example, if two nodes A and B were 90% of the chosen time spread visible to each other, the corresponding connection weight between A and B is 0.9. The weight between B and A is 0.9 as well, since this relation is symmetrical. By enlarging the observation interval a more and more static view of the vicinity is revealed. This is a description of who usually is a neigbour of whom and therefore taking the habits of the involved nodes into account.

In addition to the connection weights (i.e. average up time) it could be interesting to know the distribution of the connected periods (up time slots). This could be done explicitly by running through the event list an calculating an additional property

value or implicitly: Instead of just adding up the connected times within the interval one could try to more weight the more recent values. For example decreasing the importance of older events exponentially. The received values should then be normalized to be in the range between 0 and 1.

While for a routing algorithm in a dynamic network a topology with actual connections is important, a location service in a building is more interested in a general overview of the vicinity. Seen under this aspect a topology is a special case of the connection weight (observation instant vs. observation interval).

3.4.3 Mobility

Two entities that see and loose each other very often are mobile relative to each other. (Of course it also could be a door, that is opening and closing and therefore interrupting the connection). It is imperative that this mobility is relative to the involved entities and not absolute to the rest of the world. For example let's consider some nodes attached to an elevator and one at each floor. When the elevator moves up and down, events are generated. Just by looking at these events, it cannot be decided, which nodes are in the elevator (mobile) and which one are static at each floor. Instead, the nodes in the elevator are static relative to each other, the same as the ones outside. But these two groups are mobile relative to each other. However, if only one globally static device is known, everybody else that's staying a while within reach will likely be static as well.

A simple way to get some measure for relative mobility is to count the lost and found events. The resulting value is somewhat equivalent to a frequency: events per time (1/s). This mobility value could be used for example for multihop routing. If it is known, that node A sees node B 200 time a day, why not pass A a message for node B? Maybe node A is sitting in the above mentioned elevator and node B is at some floor high above, out of my own range. Again, if radio contact is very weak and therefore flickering, a high mobility is falsely assumed. In combination with such a frequency, the distribution of the event over time could be interesting: over what period of time had these contacts taken place? Maybe node B has seen node A very often during a short period of time, just because the owner of B had been drinking a coffee near the elevator and then went back into his office. With the frequency and the variation one also can estimate the latency for multihop routing.

In sparsely connected network where most nodes don't have lots of contact with neighbours, a topology or connection based description might not contain a lot of useful information. The only way to collect additional data is to move around. Here

such a mobility value might describe the vicinity better, because there is no “network” as such.

CHAPTER 4

Architecture and Realization

Chapter 4 covers the software portion of this thesis. It is divided into 3 main parts. Section 4.1 covers the general architecture and realization of the infrastructure for event dissemination and collection. Section 4.2 covers the java software used to simulate the concept and evaluate data. Finally, section 4.3 describes the software written for the embedded hardware platform.

4.1 General Architecture Overview

The architecture of the EventCollector concept may be depicted in a layered structure. Figure 11 on page 44, shows the service layers of the concept. There are four Layers.

In the Discovery layer, information about a node's vicinity is gathered. Various types of information is thinkable, e.g. link states, sensor data or information about nodes which are located within radio distance. Generated data is encapsulated into events which are passed up one layer for storage and propagation. For the scope of this thesis, only one property is considered. The only information generated is about nodes entering or leaving radio distance.

Layer 2 is responsible for knowledge sharing. This layer manages local storage of events and handles propagation of event data through flooding. Issues such as flow control and preventing loops are handled here. Dependant on capabilities, more or less data is stored.

Layer one and two together form the so called EventCollector. The functionality of this EventCollector is the same on a small embedded device with 4kBytes memory as on a multi-megabyte workstation or server.

In a next layer, information processing or data mining is done. The exemplary Vicinity application uses an EventCollector as data source. Stored data is processed into information beneficial to an application. This information may be used by entities in the Application Layer to perform some task. Naturally, applications in this layer will favorably use EventCollectors containing large amounts of data.

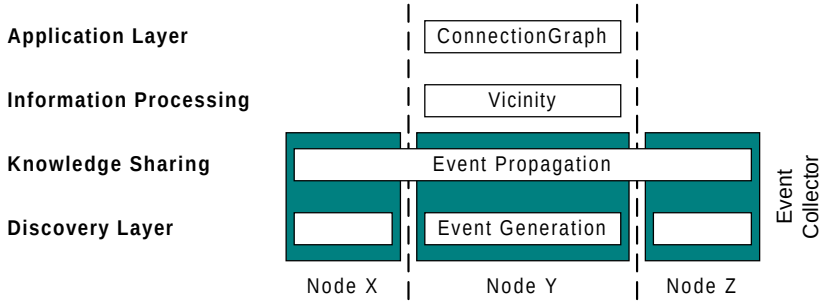


FIGURE 11. Service Layers of the EventCollector Concept

For this thesis, the Vicinity application exemplary processes data to extract information about frequency and duration of contacts which two nodes have with each other. As an example entity in the Application Layer, a graphical representation of these values has been implemented.

In Figure 12 on page 45, the general layout of participating entities is depicted. The layout is subdivided into three regions. On the left side reside entities which belong to the embedded Bluetooth environment. These nodes represent mobile platforms built for this thesis, all running a Bluetooth stack and an EventCollector. Events are exchanged over Bluetooth using the protocol described in chapter 3.

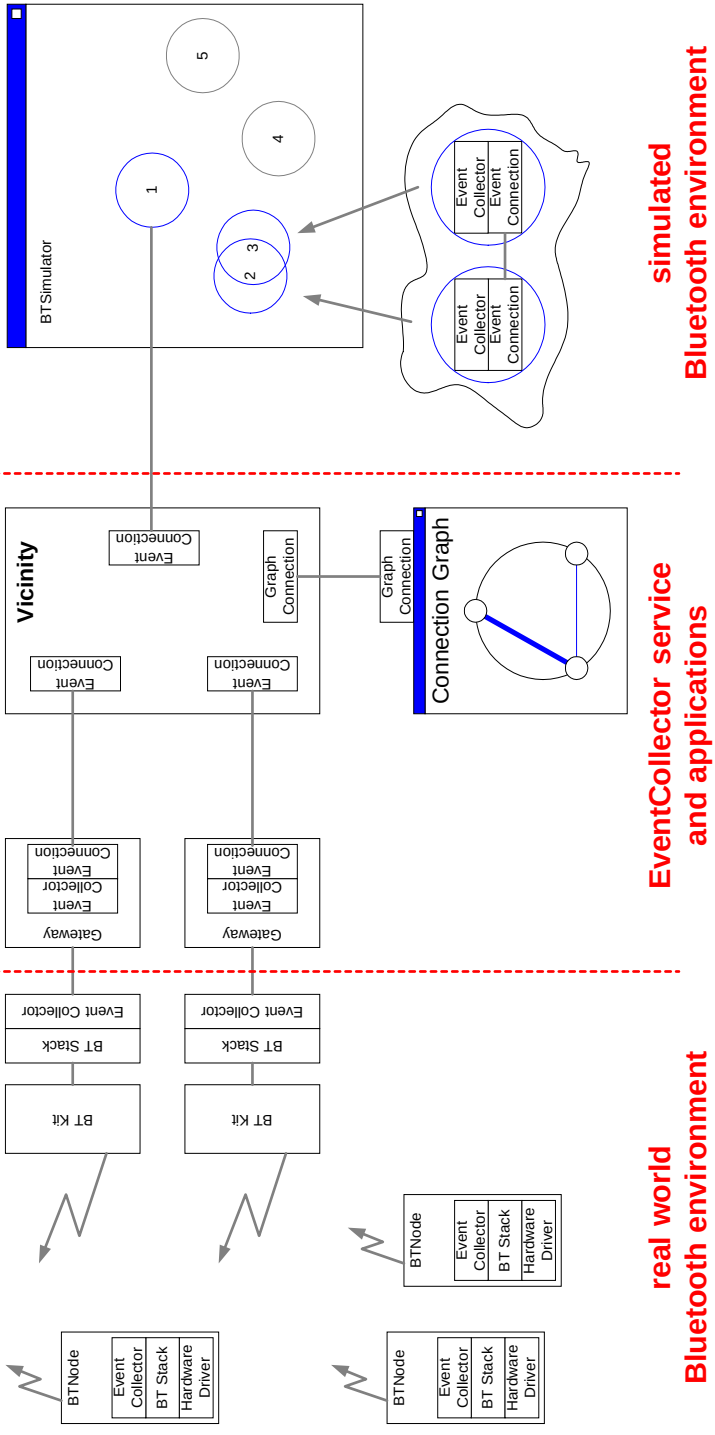


FIGURE 12. General Architecture Overview

The entities in the center make up layer three and four of the Service Layers. The right side represents the simulation environment BTSim.

There are gateways between embedded units communicating over Bluetooth and applications performing data processing running in a JVM. They are made up of an Ericssons Bluetooth Tool Kit connected over a serial interface to a Bluetooth stack and EventCollector running on Linux. To simplify software development, this application is built upon the same code base as the software running on the embedded devices. In addition the EventCollector on Linux maintains a TCP/IP connection to a Gateway application written in Java. Over this connection, raw events are passed from the Bluetooth world into the Java world.

Entities programmed in Java are built around three main classes.

1. BT_Event: This java object represents an event as described in chapter 3. They are passed among the involved EventCollectors and stored there.
2. EventCollector: The EventCollector class implements the functionality of an EventCollector in Java. This EventCollector has the same functionality as the counterpart programmed in C.
   Normally, the EventCollector is instantiated by an application. However, it is possible to use the EventCollector as a stand-alone application and cascade it with other EventCollectors.
3. EventConnection: Connections between entities written in Java are set up using EventConnections. Two instances of this class exchange serialized BT_Events over TCP/IP connections. EventConnections exchange data using the protocol outlined in chapter 3. This is the same protocol as being used over the wireless links.

The Gateway application interfaces the EventCollector running on Linux. Raw Events are received over the TCP connection and are parsed. As a result, a new BT_Event object is created and fed into the EventCollector. The Gateway application provides no additional functionality. Events are further processed by other entities connecting to the EventCollector and receiving the stored events.

Evaluation of the stored events is done using the Vicinity application. Vicinity receives events from different EventCollectors, stores them, processes them and thus is a representation of the vicinity. Data mining is performed on the stored events to extract information valuable to the user. In the current implementation, Vicinity extracts three values: a measure for topology, mobility and connection weight. This information is served to the exemplarily ConnectionGraph applica-

tion. Information received by ConnectionGraph is displayed without any other calculation.

The right side of Figure 12 on page 45, represents the simulation environment designed for this thesis. The BTSim application is used to simulate the protocol and EventCollector algorithm outlined in chapter 3 and serves as a source of events. Nodes represented by circles can be dragged around the canvas just as real-world mobile units roam around in their environment. As illustrated in the explosion view, every node in the simulation contains one EventCollector and an EventConnection to every neighbour within sight.

Outside entities like the Vicinity application can connect to a node in the simulation using an EventConnection. Thus, every simulated node listens on a predefined TCP port for incoming connections. Using this setup, events generated both from real-world and simulated node could hypothetically be mixed and disseminated throughout the system. There is no difference from an event created in the simulation to events generated upon two Bluetooth devices meeting in the real world.

4.2 Java Software

This chapters covers classes programed for the Java environment. Entities are listed according to relevance and grouped their functionality

4.2.1 Basic Classes for Framework

BT_Event

BT_Event represents an event that is generated when one Bluetooth node sees another device. A BT_Event object encapsulates the following information:

- Timestamp when the event occurred
- Source the event (BD_Addr of the node generating it)
- Counterpart (BD_Addr of the discovered device)
- The Type of Event (FOUND, UPDATE, LOST)

BT_Eventlist

An EventList stores BT_Events. The class is used to operate on this event collection. It extends TreeSet and adds the method getSame().

BT_FlowPacket

A flow packet is used to communicate between two directly connected neighbors. It is never propagated. Through such flow packets flow control can be implemented or commands and data may be passed. To extend the existing protocol with new commands or packet types other than event and flow packets, a protocol version field exists.

BT_NodeList

The BT_NodeList contains a list of BT_Nodes.

BT_Relation

A BT_Relation describes the relation between two nodes. Namely between ‘myself’ and the node ’counterpart’. Such a relation is described by different attributes as explained in the BT_Node documentation, chapter 3. Here three possible relation values are implemented: Topology, Connection_Weight and Mobility_Weight.

EventCallBack

Interface for call back between parent and children for asynchronous propagation of events. Events can be passed back and forth through addEvent(). InsertIntoConnection() and removeFromConnection() is used solely between BTSim and EventCollectors to detect connections and (more importantly) the remote teardown of a connection.

EventCollector

An EventCollector is a entity that gathers events from different sources and redistributes them. It also generates events upon a connection establishment. An EventC-

ollector is a client to different event sources: It connects actively to Gateways, FileEventServers or other EventCollectors. The collected events are made accessible to other entities through the same mechanism, so each EventCollector is a server (i.e. an event source) as well. One can specify different sources (host:port) in a urlList - Vector and the port on which other clients can connect to.

EventConnection

An EventConnection opens a bi-directional ObjectStream to another entity over TCP/IP. Then it runs a simple protocol to exchange events. It is the same protocol used on the Bluetooth nodes. In order to handle blocking read() calls, the EventConnection implements Runnable and starts a thread for reading.

EventConnections are mainly (but not solely) used by EventCollectors. In order to communicate between the parent (e.g. an EventCollector) and the EventConnection, both must implement EventCallBack. Upon connection establishment the parent's insertIntoConnectionsList() must be called. A newly arrived event can be passed to the other side by calling addEvent(). If a connection is closed, the EventConnection must call removeFromConnectionsList() to notify the parent.

A reference to a BT_EventList is passed to the constructor as a collection of events known to the parent. Since it is a reference, changes by the parent are noticeable to the EventConnection. Short summary of the protocol: When the Object Streams have been opened, a BT_FlowPacket is exchanged in both directions to indicate how many events one wants / is able to receive. Then the requested amount is being exchanged. As long as the connection stays open, new events will be propagated to the other side.

EventConnectionServerThread

The EventConnectionServerThread acts as an EventServer. It opens a server socket and waits until a connection request comes in. Then it creates a new EventConnection and starts its thread to deliver the events and potentially accept events as well. The EventConnectionServerThread has access to the connections Vector of its parent, and can therefore insert new connections into it. The management of connections (dead ones, propagation etc.) is handled by the parent (normally an EventCollector)

4.2.2 Classes for Evaluation of Data

BT_Node

A BT_Node represents a node in the vicinity. It has a BT_Relation to every other node known.

Vicinity

A Vicinity represents the vicinity of all known BT_Nodes, including the relations between the individual nodes. The knowledge is served on the specified port to applications. At the beginning we only have a collection of events, and we do not know anything about the nodes (position, mobility, uptime, connection among each other...). In order to achieve a global view of all nodes in the system, we process all known events in the eventList, extracts the node names (BD_Addr) and try to calculate some properties of the relation between each pair of nodes. At the moment two property values are implemented: Proximity_Weight and Connectivity_Weight. The Proximity_Weight is some sort of frequency of connections, i.e. how many connections per time. The Connectivity_Weight is a measure for the connection time, in percent of connect time in the specified interval. (e.g. node A is 94% of the time connected to node B) Examples: If we have a high Connectivity_Weight and a high Proximity_Weight, then the two nodes are spatial relatively close and immobile, but the connection is rather unstable. In case of a smaller Connectivity_Weight and high Proximity_Weight, the devices are likely to be more mobile, but see each other frequently. (relevant e.g. for multihop routing). A Connectivity_Weight weight of zero means, these two nodes don’t see each other and never have.

4.2.3 Classes for Graphical Representation of Data

Connection Graph

ConnectionGraph is an application which connects to a Vicinity server to retrieve a matrix representing the known environment and displays the retrieved data in graphical form. The user may choose, which evaluation from Vicinity he wants to have displayed by choosing the appropriate entry from the View menu.

4.2.4 Classes used in Simulation

BTSim

BTSim is an application to simulate the generation and dissemination of events in a Bluetooth environment. The simulator was programmed to analyze and verify the EventCollector concept.

Left-Click places a new Bluetooth node. A node is illustrated by a circle which represents the vicinity and a Bluetooth hardware address in its center. Next to the center, a list of other nodes which the specified node is currently connected to is displayed in parentheses.

Right-Click on the address of a node opens a dialog window which displays the node's configuration and all events collected so far.

Every node listens on a TCP socket for incoming connections. Nodes are moved by dragging the center on the screen. Whenever to nodes come within each others vicinity, a connection is opened between these nodes. When connected, nodes exchange event data according to the underlying protocol.

An EventList containing all stored events can be saved to disk using the according command in the File menu.

Technically, BTSim is only used as a wrapper applications around the class BTSimGraph. A more detailed explanation can be found in BTSimGraph.

BTSimGraph

BTSimGraph is a class used to simulate the generation and dissemination of events in a Bluetooth infrastructure. The simulator was programmed to analyze and verify the EventCollector concept. Normally, BTSimGraph is instantiated by BTSim, which provides a GUI.

BTSimGraph pane is divided into to parts. In the upper part, a panel is displayed which holds the graphical representation of the simulation. In the lower part, occurring events appear in a scrollable text area.

Left-click in the upper part of the pane places a new Bluetooth node. A node is illustrated by a circle which represents the vicinity and a Bluetooth hardware address in its center. Next to the center, a list of other nodes which the specified node is currently connected to is displayed in parentheses.

Right-click on the address of a node opens a dialog window which displays the node's configuration and all events collected so far.

Every node listens on a TCP socket for incoming connections. The listening port is calculated by adding 10'000 to the BD_Addr. For example, node 3 can be connected to on port 10'003. Nodes are moved by dragging the center on the screen. Whenever to nodes come within each others vicinity, a connection is opened between these nodes. When connected, nodes exchange event data according to the underlying protocol.

Every node instantiates an EventCollector upon its creation. The EventCollector collects events and handles any connections to other nodes or to the outside of the simulation. BTSimGraph takes care of the graphical representation of nodes and actions such as clicking and dragging thereof. Upon two nodes entering each others vicinity, BTSimGraph signals the EventCollector to open a new connection.

BTSimNode

BTSimNode is the class used to represent a Bluetooth device for simulation. A BTSimNode object encapsulates following Information:

- Data for graphical representation in simulation
- A reference to an EventCollector
- Vector of BTSimNode to store all currently connected nodes

BTSimNode is the simulation counterpart to a BT_Node which is used to represent a real-world node. Just as real nodes, BTSim Nodes use EventCollectors to handle connections and event dissemination to other nodes.

Upon creation, every node initiates an EventCollector which starts to listen on a TCP Socket. The port number defaults to 10000 + BT Address. This Socket is contacted by EventCollectors belonging to other nodes when establishing connections to exchange events. This Socket can be contacted even from the outside of the simulation. E.g events from real-world Bluetooth nodes can enter the simulation through such a connection.

BTSimNodeDialog

Display a Dialog box containg the node's preferences and all events currently stored.

4.2.5 Gateway

Gateway is an application which connects the Java world to the bluetooth world. The BTNode application which runs on Linux contains the Bluetooth stack and an EventCollector written in C. Events received over Bluetooth connections are sent over a TCP stream as raw data to the this application. Gateway parses this incoming stream and feeds the data into another EventCollector which can be connected to from the Java world.

4.3 Embedded Software

Chapter 4.3 covers software written for the embedded platform. It is subdivided into sections covering the drivers, the application itself and Bluetooth stack running on the device. First, general notes regarding embedded software are given. Second, drivers implemented for the hardware platform are described. Next, the port of Axis's Bluetooth stack to the AVR platform is illustrated. Section 4.3.4 describes the approach to implement a simple scheduler as operation system substitute. Last, section 4.3.5 specifies the EventCollector running on the embedded hardware.

4.3.1 General Notes on Embedded Programming

Embedded programming on the AVR CPU put up several stumbling blocks. Being restricted in memory a number of compromises had to be found. Some of the notes presented here apply generally to embedded programming as others only apply to the AVR MCU.

One major limitation is the lack of any operating system on the embedded platform. Memory allocation using malloc() is impossible. Some effort has to be put into porting of applications which use dynamic memory allocation. Further, multi threading or pseudo concurrency of tasks has to be implemented first.

The C-Library which is provided for the AVR Architecture is simplified. Functions like printf() sprintf() or scanf() are lacking. Embedded platforms usually do not have a 'standard output' method as a console which is found on PC's. Output of

debugging information has to be implemented before programming of an application is started.

Development of new software is done using cross compilation on a different host platform. Even though the GNU C Compiler may be used for different target platform, several difficulties arise. A ‘Long’ defined on a 32 Bit Operating System may not be of same size on the embedded platform. This imposes the usage of direct specification of the variable size such as u32 for an unsigned 32 Bit variable. Another problem is byte alignement. On advanced platforms such as SPARC or X86, byte alignement is used to speed up memory access. On an 8Bit architecture such as AVR, this does not only waste data memory but may also render malfunctioning programs due to different size of structures on host and target platform. Using the special attribute “__attribute__ ((packed))” on critical sections or members of structures eases this restriction.

As the GNU C Compiler is not intended to be a compiler adapted to embedded platforms, optimization is usually done in terms of execution speed or code size. Although memory is often the largest restriction, there is no optimization for memory usage as would be in a compiler developed especially for embedded applications. Tests have shown that using optimization flags “-O6” helps in reducing memory as a side effect to improving execution speed. Never the less, some improvement could be achieved using a specially designed compiler.

Code often has to be refined for embedded use. Next to reducing static memory size of variables, stack usage is a major concern. Code such as “for (int i= 0, i<4, i++) { ... } “ is perfectly right for 32Bit platforms such as Linux, since memory access is done in words of 32Bits anyway. On 8Bit architectures, using an unsigned char as counting variable is does not speed up memory access by at least a factor of 4 but also economizes 3 Bytes of stack used. Another stumbling block is heavy nesting of code. Using encapsulation renders high-quality code but wastes large sums of memory, since stack grows for every nested call of a function. Another issue are return values of functions. Using integers as return value for {-1,0,1} stresses memory requirements of the stack enormously.

Another interesting issue is the Harvard Architecture concept of the AVR Microprocessor which uses separate buses for program and data memory access. Next to several advantages, the harvard architecture has a drawback. Strings used for output must be loaded in data memory (RAM) since functions can not access data stored in program memory (Flash). This is done by an initialization routine called automatically upon power-up of the CPU. Using explicit debug messages, memory requirements after the first compilation on the target platform were stunningly high.

Few other things must be kept in mind. Due to the simple design of the core, the MCU does not support traps, software interrupts, floating point operations or a kernel mode.

4.3.2 Drivers

This chapter covers the design and implementation of software drivers for the embedded hardware platform.

Hardware Setup [Avr.h, Avr.c]

Avr.c contains functions to set up the hardware platform. AVR_init() is used to enable output on debugging ports and enable interrupts.

Additionally, three special code fragments are defined in Avr.c. As stack memory usage was quite an issue, “REPORT_SP” reports on the stack pointers value. “REPORT_SP” prints a debug statement out on the serial interface indicating the value of the definition “INIT_SP_CHECK” initializes this reporting. Every call to “CHECK_SP” compares the current value of the stack pointer register to the content of this variable and decrements the variable if the stack pointer indicates a lower memory address

Debugging Functions [Avr_Debug.h, Avr_Debug.c]

Avr_Debug.c provides several functions to print out debug information in string, char array or hexadecimal format.

LED Driver [Avr_LED.h, Avr_LED.c]

LED’s on both hardware and the STK are interfaced using the LED driver in Avr_LED.c. After calling LED_init() which enables the appropriate output port, LED’s are turned on using LED_set() and turned off using LED_clear(). Note that the LED’s on the development board STK 300 have inverse logic. These are turned on when issuing LED_clear() and vice-versa.

Pseudo Real Time Clock [Avr_Time.c, Avr_Time.h]

The events generated for the topology service need an exact timestamp. (See chapter 3). The microprocessor's Timer/Counter 0 can be clocked asynchronously from an external clock. This enables us to use this timer as a counter which is incremented by an even divisor of one second and thus 'count' time on the platform. The difference between two counted values provides an exact time spread between two events.

Two boundary condition had to be considered when designing the pseudo real time clock:

- granularity of time
- number of interrupts

The number of timer interrupts should be kept as small as possible, since interrupt nesting is not enabled on the platform. Even though the execution of the interrupt handler for SIG_OVERFLOW0 takes max 50 instructions, this delay is not negligible for other interrupts which have harder timing constraints. Further, the timer continues to run in PowerSave mode. Also in terms of power consumption, the time the CPU is actually computing should be kept minimal and thus the number of interrupts be small.

Implementation of the Pseudo Real Time Clock

Timer/Counter 0 is clocked externally by a 32.768kHz crystal, a frequency which is an even multiple of 1 second. (The 32.768kHz crystal is widely in watches etc. to provide a precise 1s signal)

A long value (32 Bit) is used to count time. The variable mseconds represents the counted milliseconds since power up. Timer/Counter 0's external clock of 32.768kHz is prescaled by 32. This results in a counter which is incremented every 1/1024 seconds. Every 256th increment, Timer/Counter 0 overflows and generates an interrupt. The interrupt handler increments the variable mseconds by 250. This yields a exact time in milliseconds with a granularity of 1/4 second.

Time_get() returns the exact time in milliseconds since power up. This is done by adding the converted value of Timer/Counter 0 to the variable mseconds. The value of Timer/Counter 0 has to be converted, as this Timer counts 1/1024 seconds and not true 1/1000 seconds due to the external clock.

Power Management [Avr_Power.c, Avr_Power.h]

In portable devices, power consumption is one of the most crucial issue. While applications in normal environments busy-wait for the next event, we would like to sleep in a state without using energy until the next event occurs.

ATMega103 provides several types of sleep mode.

- Idle Mode stops the CPU but for the interrupt system which continues to operate. Calling POWER_idle() puts the system to sleep until the next interrupt occurs. This may be an interrupt resulting from activity on an external interrupt port or a timer which overflows.
- Power Down and Power Save mode stop the CPU but for the external interrupt system. The CPU will not wake up on interrupts resulting from internal sources such as timer overflows etc. In these modes, power consumption drops almost to zero, as the MCU Master Clock is shut down.

This mode is perfect for long-term shutdowns of the device. Since Timer/Counter 0, the pseudo Real time Clock, is clocked asynchronously, interrupts generated from comparison or overflow are counted as external interrupts. This would ensure that time ‘does not get lost’. Whenever external activity occurs, the device would ‘wake up’.

When sending out data via Software UART, we need the interrupts on Timer2. If Power Down mode is used, we would not awake on this interrupt and data sent out would get garbled. In this case, we would have to be much more careful when and where POWER_idle() routines may be called without any harm.

We decided only to implement Idle Mode, as this mode posed the least problems. Moreover, the bluetooth module consumes much more energy than the MCU. Implementation of power shutdown in the bluetooth module yields much better results.

Due to time limitations, power management could not be processed to the end. There are a few additional ideas which could be implemented in a next release:

- Use of Power Idle or Power Save mode
- Reduction of CPU clock in idle state
- Shutdown of bluetooth module in idle state
- Shutdown of external components in idle state

Implementation of Power Management

POWER_idle() makes the MCU enter the Idle Mode, stopping the CPU but allowing UART, Timer/Counters, Watchdog and the interrupt system to continue operating.

This enables the MCU to wake up from external triggered interrupts as well as internal ones like the Timer Overflow and UART Receive Complete interrupts. The energy consumption of the MCU will drop to half of the value of normal operation. See “Electrical Characteristics”, chapter 6.

Serial Port over Hardware UART [Avr_UART.c, Avr_UART.h]

The ATMEGA MCU provides one Hardware Universal Asynchronous Receiver Transmitter (UART). Using the built-in hardware UART the CPU transfers complete bytes to the UART. All timing-related tasks, such as sampling incoming and sending of bits is done without any intervention of the CPU. Furthermore, full duplex transfers are possible.

The driver of the Hardware UART must be programmed in such a way, that receiving and sending of data is handled in the background without any intervention of the user application. Data exchange between user application and hardware driver is done using a shared buffer.

To simplify porting of the bluetooth stack which is written for Linux, unix system call semantics are used as interface to the driver.

Special care should be taken towards synchronization. The driver must be implemented in such a way, that data may be written into the buffer without interference with the sending interrupts, which retrieves data from the buffer. Further, the buffer must be allocated statically.

Implementation of the Hardware UART

The Hardware UART is initialized by calling UART_init(). UART_init() initializes the buffers for incoming and outgoing data, enables the interrupts needed for communication and sets up the UART for communication with 57600Baud.

User applications transmit data using the UART_write() function call. This functions stores data in the circular buffer and enables the UartDataRegisterEmpty UDRE interrupt. The MCU interrupts as soon as the UDR is empty and starts to send out the first byte. The UART_write() function returns while the content of the buffer is transmitted byte by byte without any further user intervention.

Receiving of data is handled in a similar way. UART_init() enables the ReceivedCharacterInterruptEnable RXCIE. For every byte received, the MCU interrupts and stores the incoming data in a second buffer, without any user intervention. This second buffer is interfaced via UART_read function call.

Software Serial Port [Avr_SUart.c, Avr_SUart.h]

Our hardware design needs a second serial port. Since the ATMega 103 MCU contains only one UART, a second UART has to be built in software. In terms of timing, transmission of data over an asynchronous serial line is a delicate issue.

At 57600Baud, the bit-time is 17.3usec, which equals to 64 CPU instructions at 3.6864 MHz. Sampling of data must be within 1/4 of the bit-time (16 instructions). This latency is only achievable, if the CPU would busy-wait during the transmission of one complete byte, including start- and stopbit. This again would result in a maximum interrupt latency of 10 * 17.3 usec (startbit, 8 databits, stopbit), as nested interrupts are disabled on the CPU. An interrupt latency of 173usec is too much for time critical applications. Implementing a 57600Baud software UART with given constraints is not feasible. Not to think of implementing full-duplex communication.

A resource found on the Internet confirms this assumptions. In [7] a full duplex software UART running at 38400 Baud is programmed in assembler for a MCU clocked with 11Mhz. This code needs more than 30% CPU performance when sending and/or receiving data.

Under these circumstances, we decided to implement a software UART communicating at 9600 baud half-duplex with the bluetooth module. Running at lower speed has several advantages:

- interrupt timing is less critical
- buffers may be kept small

- at high speeds, Ericssons's bluetooth module may run out of synchronization.
- onboard wiring is less prone to errors / interference
- implementation possible in high-level language

Ericssons's bluetooth module requires full-duplex communication. Even at a speed of 9600 baud, this is quite tricky. Bit-time at 9600 baud is 0.104msec which equals 384 CPU instructions. Assuming that sampling of data must be within 1/4 of the bit-time (96 instructions), follows that interrupt routines to sample or send out data must be quite short. Any other interrupts excepted, these timing constraints could be accomplished by using a second timer. As there are only 3 Timers in the MCU of which one is used as real time clock, we decided to reserve one counter for future use. Thus, communication over the software UART is handled as half-duplex. While sending data. the Bluetooth module is forced into half-duplex operation by using flow-control to pretend that no buffer space is available for incoming data.

Flow-control can be either implemented in software (Xon-Xoff), where flow packets are exchanged between sender and receiver or hardware (RTS-CTS) where flow is stopped via wired connection. Both techniques are further explained in [8] [9].

Off the shelf, Ericssons bluetooth module assumes hardware handshaking as in RTS and CTS signals. RTS (Request To Send) signal is set by the receiver on low buffer space. The sender checks CTS (Clear to Send), which is wired to the sender's RTS signal, before sending. Figure 13 on page 61 shows a detailed signal analysis.

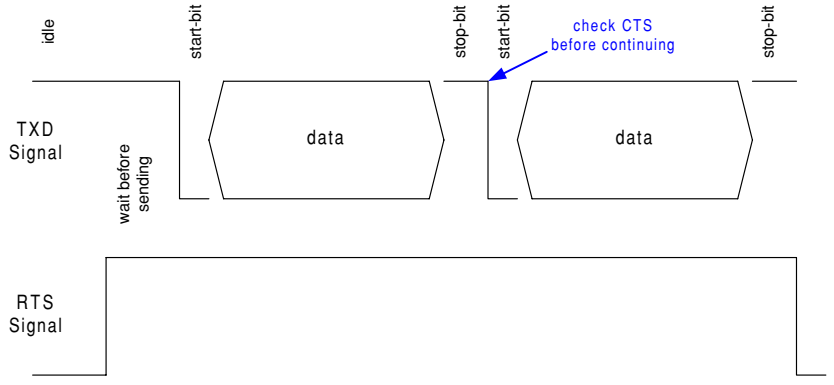


FIGURE 13. Overview of RS232 Signalling

Flow control signals RTS and CTS are used to force the bluetooth module into half duplex. Whenever data is to be sent out, the MCU indicates to the bluetooth module by unsetting CTS, that no data is to be sent. After waiting a short period, the MCU may start sending without being interrupted by incoming data.

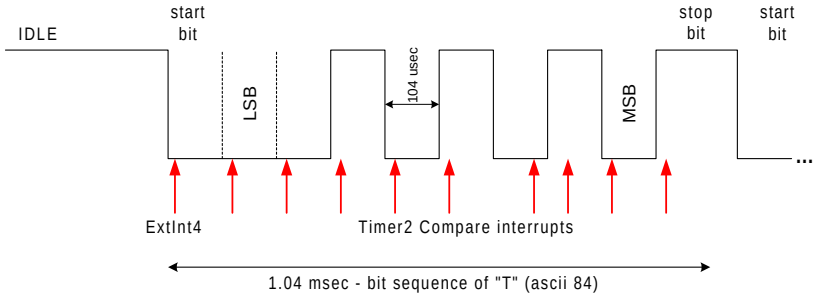


FIGURE 14. Bit Signalling of RS232 Communication

Figure 14 on page 61 specifies the bit sequence of incoming data. As seen above, timing is crucial. RXD, the receiver signal is wired to an external interrupt port. A falling edge at the external interrupt, designates an incoming start-bit. The interrupt handler starts a counter which generates TimerCompareInterrupts after every 104usec, which amounts exactly to one bit-time of 9600 Baud.

In a first implementation, the start-bit was sampled by setting the timer to 1/2 bit-time. Sampling of the startbit is needed to filter out spikes on the RXD wire, which generated interrupts. Signal analysis showed, that under certain circumstances this led to erroneous behavior due to other interrupt handlers which delayed the processing of the timer interrupt. Recall, that 1/2 bit-time will only take 180 CPU instructions. In a second revision, the start-bit was sampled right after the occurrence of the external interrupt.

Under normal conditions, the first bit should be sampled 1.5 bit-time after the occurrence of ExtInt4 and 1 bit-time thereafter. Once again, signal analysis showed too much delay if other interrupt handlers were serviced between the sampling of start-bit and the first bit of data. Even though the timer was not stopped and restarted between sampling of data-bits, under heavy load of the embedded plat-

form, delay accumulated and led to false data. A solution to this problem would be the nesting of interrupts. On the other hand, this would have led to much more complicated interrupt servicing routines, since interrupts work on the same data structures. Another issue in nested interrupts is stack size. As memory is very tight, it seemed advantageous to omit nesting.

In a third revision, of the interrupt handler, sampling of data was started exactly 1 bit-time after verifying the startbit at the beginning of ExtInt 4. The delay caused by setting up the interrupt handler and saving the registers is enough to start sampling after one bit-time. This results in TimerCompareInterrupts being serviced after about 1/3 of the bit-time of every data bit. This leaves enough room for other interrupts being serviced, as shown in Figure 14 on page 61 between bit 5 and 6.

Implementation of the Software UART

The Software UART is initialized by calling SUART_init(). SUART_init() initializes the buffers for incoming and outgoing data, enables the interrupts needed for communication and sets up the UART for communication with 9600Baud. Hardware flow-control (RTS/CTS) is enabled by default.

User applications transmit data using the SUART_write() function call. This functions stores data in a circular buffer and returns. The outgoing buffer is flushed by calling SUART_flush(). SUART_flush enables a timer which will interrupt a few cycles later. The timer is used to start transmission asynchronously in the background without busy-waiting until data has been sent out. As soon as the timer overflows, the according interrupt handler is called. The interrupt handler is programmed as a state machine, since incoming and outgoing bytes use the same timer and thus the same interrupt handler. The state machine handles flow-control and the timer settings to sample or send out bits.

Receiving of data is handled in a similar way. SUART_init() enables the External Interrupt 4, which interrupts on a falling edge on the software UARTS RXD signal. A timer is started to handle the sampling of incoming bits using the state machine of the interrupt handler.

Buffer size for sending an receiving data is defined to 32 Bytes. This value is not very crucial, since the software UART supports Hardware flow-control. Attention should be made to the two watermarks
SUART_RX_BUFFER_HIGH_WATERMARK
SUART_RX_BUFFER_LOW_WATERMARK defined in Avr_SUart.h. These two

watermarks define a hysteresis for flow-control. Since sending of data will not stop immediately after the RTS signal has been unset, flow-control must always check for buffer space in advance.

Special attention must be paid to timing. As discussed an the last section, bit-time of a 9600 Baud signal is 1.04ms which amounts to 384 CPU instructions. Nested interrupt routines are not enabled by default. Assuming that sampling of data must be within 1/2 of the bit-time (192 instructions), no interrupt handler may be larger than 192 instructions. This leaves enough time for software UART interrupt handler to complete without delaying the next sample too much.

Debugging the interrupt handler is problematic! For the same reason mentioned above, we are not able to output debug messages while being in the interrupt routine. Debugging of the software UART was handled by setting / unsetting LED's according to error conditions.

Analog Digital Converter [Avr_ADC.c, Avr_ADC.h]

The MCU's Analog to Digital Converter is accessed straight forward. After calling ADC_init(), every channel of the ADC can be accessed using ADC_read(). ADC_read() starts a conversion on the selected channel and busy-waits until the conversion is done. As the sampling rate (set in ADC_init()) amounts to 100kHz, one sample takes up to 10 microseconds.

Random Number Generator [Avr_Random.c, Avr_Random.h]

For certain applications, random numbers are needed. Avr_Random.c implements a simple pseudo random number generator. A call to Avr_srand() seeds the number generator by sampling 500 consecutive values at an unconnected port of the analog to digital converter. These 500 samples are added to an 8 Bit variable. Random noise combined with wrap-around of the 8 Bit variable yields numbers distributed in the range 0 to 255. As the sampling of 500 measures takes quite long, the next number in the random sequence is generated by adding 187 to the initial seed. 187 suits quite well, as it is a prime number and about 2/3 of the maximum range of an 8Bit value.

4.3.4 Scheduler

In a first outline of the software structure, the system was built around a state machine which reacts to activity either on the software UART connected to the Bluetooth module or the hardware UART which may be connected to an outside control. As the Bluetooth stack is reactive, this structure would have served the purpose. Simple as the solution seems, several issues can not be dealt with. Handling of erroneous behavior of the Bluetooth modules or time-outs are very difficult. Further, the system's functionality would not have been very extendible, as there would have been no way to interrupt a blocking wait for more input data to execute some additional user code.

Implementing (or porting) an operation system to our platform seemed one possibility. To simplify matters, we decided to port a simple scheduler which solves the most significant problems without implicating a large overhead of a complete OS. Applications as well as the Bluetooth stack may register callback functions to the scheduler which are executed upon occurrence of defined events such as activity on either UART or time-outs. The mentioned scheduler already existed in a piece of software written by one of the Ph.D. students of the group. Porting to the AVR platform was accomplished by our tutor. Further reference of the scheduler may be found in documented C source files.

4.3.4 Bluetooth Stack

Axis Communications corporation maintains a Bluetooth stack[6] released under the GNU General Public License. This stack is used in Axis products which are based on Linux. Being the only open source software stack at this time, it was used as basis to the port for the AVR architecture.

Axis' stack provides both kernel module and a user space application for the Linux operating system. For the port, only the user space portion of the code was used. Major difficulty posed the stack's widespread use of dynamic memory allocation and Linux system calls. Furthermore, the stack was not laid out for minimal memory footprint.

Data structures which relied on dynamic allocation of memory were reprogrammed to use static structures. Additionally, these structures were reduced to minimal memory usage. Linux system calls such as select() had do be emulated by the scheduler described in the previous subchapter.

User applications interface the stack using L2CAP layer functions. Even though HCI would have been easier to implement and less stressing on memory requirements, availability of the HCI interface is not a requirement for Bluetooth compliancy and thus not available on every Bluetooth implementation.

Due to the limited time of this thesis, the Bluetooth stack was ported to the hardware platform by our tutor.

4.3.5 EventCollector on the Embedded Device

The application that runs on top of the scheduler is basically an EventCollector with the same functionality and tasks as in Java. However, there are quite a few particularities. There are limited resources, some problems with Bluetooth in general and the ROK from Ericsson in specific. In Chapter 6.3 these problems are described in more detail.

The Bluetooth module must not be transmitting in order to be seen by an inquiry by an other module. In addition to that, the ROK must not have an ACL connection open. At the bottom line, we need to be IDLE most of the time by keeping transmissions and inquiries at a minimum. That is what the state machine depicted in Figure 15 on page 66 was designed for.

The main loop consists of three states: IDLE, INQUIRING and PUSH_LOOP. Upon startup the micro controller and the Bluetooth module are initialized. Then the IDLE state is entered. Here we are ready to respond to inquiries (unnoticed by the application) and to accept incoming connection requests (through event_collector_connect_ind()). After a random idle period between about 12 to 25 seconds a time-out occurs (timeout_cb()), an inquiry is initiated and the state machine becomes INQUIRING.

Each inquiry result is passed to the application through a inq_result_cb() and is processed immediately within that function. When the inquiry process is finished, the application is notified by the inq_complete_cp() call back function and makes a transition to PUSH_LOOP.

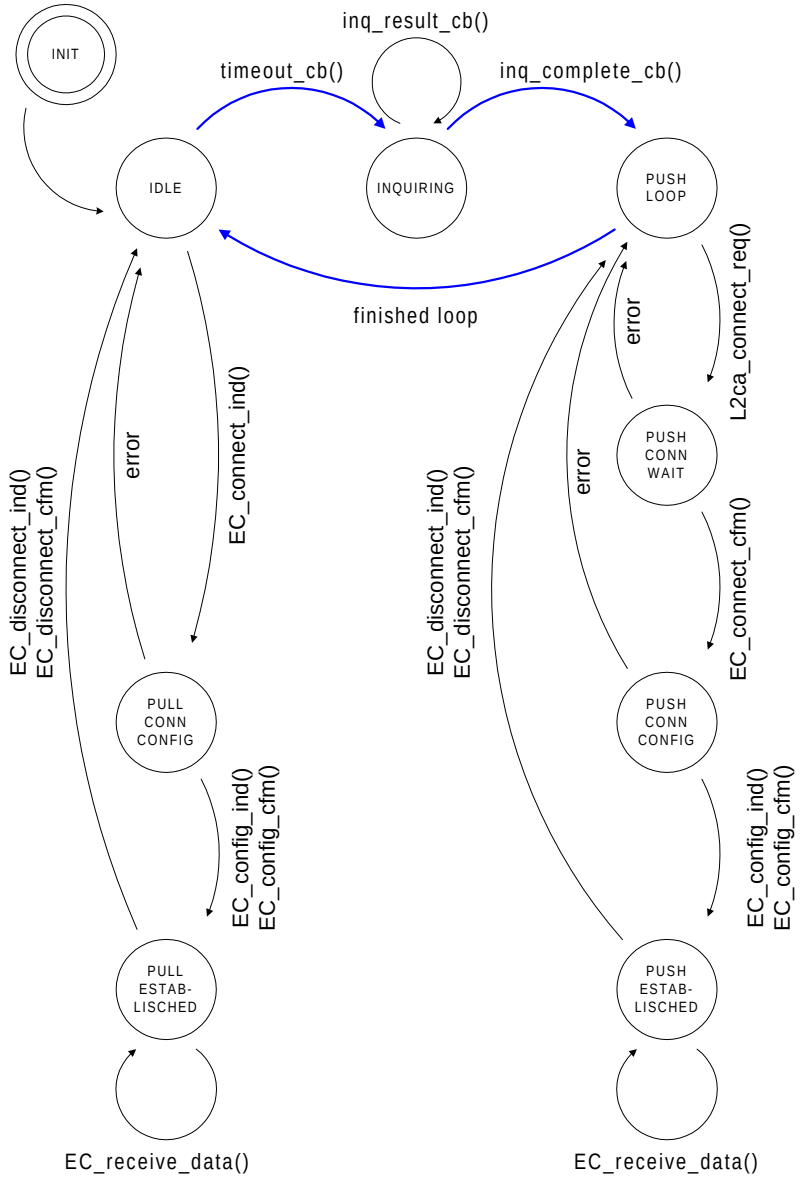


FIGURE 15. EventCollector State Diagram

Here we loop through all the neighbors in the now updated neighbors_list and try to connect to them. This of course only works, if the neighbor is within radio range and in IDLE state. If the connection establishment is successful, events are exchanged through the previously described protocol. Immediately thereafter, the connection is closed and the next neighbor will be processed until the whole list is done. Then the IDLE state is entered to wait for the next time-out to trigger the same chain again.

While in IDLE a remote connect request is indicated through event_collector_connect_ind() and the PULL_CONN_STATE is entered. Now the connection parameters are being negotiated (the L2CAP connection is in the internal state CONFIG, as described in the L2CAP specification). When one event_collector_config_ind() and one event_collector_config_cfm() callback are received both without error (status = 0) then the connection is ready, reflected by the state PULL_ESTABLISHED. All local events are sent and then remote events are received. The buffer space in the ROK are definitely large enough for this, but there is a potential deadlock if event data gets much bigger. If a termination packet is received the connection will be closed through l2ca_disconnect_req(). Also a event_collector_disconnect_ind() indicates a remote disconnect request, and it will be returned to IDLE.

The PUSH_LOOP state is the counterpart of the above described PULL states. The connection establishment and teardown is very similar. The solution with the termination packet has been chosen because (unlike TCP on the Java EventCollector) a disconnect request closes the whole connection, not just the one simplex half. All pending data is discarded. This doesn’t matter for the data that the terminating side just has sent, because the termination request will reach the opposite side only after the data has arrived there (chronologically). But the data sent by the other side will be lost.

The time-out value is chosen in a way that ensures that the state machine is IDLE for about 1/2 to 2/3 of the time.

In the BTSim simulation environment the system time is used as a global time. On the embedded platform no such global time is available since there is no real time clock, time server or any other way to get an accurate time. This problem was solved rather pragmatically. All time stamps are stored in local time, i.e. in milliseconds since startup. Before transmission of an event to some other BTNode the lostTime and foundTime are converted into ages of the events (i.e. they are subtracted from local time). The receiving node then transform the ages into his own local time. Under the assumption that data transmissions only take a short time, the inev-

itable error is small. When comparing foundTimes and lostTimes from different events it must be taken into account that these times are not absolutely accurate. In our case the tolerance was chosen to be 1 second. So, if two events have the same “me” and “him” BD_Addrs and foundTimes that differ by less than 1 second, it must be the same event.

CHAPTER 5

Technical Realization of the Embedded BTNode Hardware

To test the EventCollector concept in a life environment a hardware containing an Bluetooth module and a microprocessor was built. The board is used as a mobile demonstration unit. The battery-powered device may be carried around to test real-world setups.

As the focus of this thesis lies on the infrastructure and not the hardware itself, we searched for a partner who would layout the board according to our needs and take over responsibility for manufacturing. This partner was found in a PhD student at the Computer Engineering and Networks Laboratory TIK at the department of electrical engineering of ETH.

Chapter 5 is subdivided into several sections. Section 5.1 covers design considerations for both hardware components and design. In section 5.2, first steps with the chosen hardware are explained. An exhaustive reference to the built hardware is found in section 5.3. Next, section 5.4 lists partners and external contacts which helped in building and manufacturing the embedded hardware.

5.1 Design Considerations

Several considerations had to be made while designing the hardware for the project. Section 5.1.1 lists some general issues in the design process. Next, the selection of the microprocessor platform is covered. Section 5.1.3 discusses other criteria which were considered for the remaining hardware components.

5.1.1 General Issues

The following criteria, listed in alphabetical order, influenced the hardware design:

- short term availability

  The diploma thesis is limited to 16 weeks. Even though hardware considerations started before the official start, the selected hardware must have been available quickly.

- current drain

  The unit will be used in a mobile environment. As current drain is a crucial issue for all battery powered devices, components with low power consumption were chosen.

- ease of use

  The hardware should be as simple as possible. Since the hardware design was only a means to an end, components with minimal external circuitry were preferred.

- microcontroller features

  The unit should be applicable in miscellaneous settings. Having the Smart-Its [10] project in mind, a microprocessor which provides universal IO Ports and / or analog input / output was preferred.

- internal memory in microprocessor

  One main issue was finding a Microprocessor with internal RAM and program memory. Timing is very crucial to designs with external memory components. This complication was avoided by choosing a CPU with internal memory. Most micro controllers have internal RAM, but only very few. Components with more than 1kByte RAM are scarce. First examinations of AXIS's Bluetooth stack implementation yielded memory requirements of at least 2kBytes.

- in-circuit programmability

  To accommodate the universal usage, the device must be (re-)programmable in the final circuit.

- price

  Having the SmartIT project in mind, these devices could be used in large quantities. As for our demonstration units, price is not a main issue, but for larger scale production, low priced devices are chosen.

- speed

  The Bluetooth stack needs some processing power. As it showed, processing speed is no limitation for microprocessors meeting the requirements stated above.

- 3.3V Supply

  Since Ericsson's Bluetooth kit runs on 3.3V the entire circuit is based on 3.3V technology. This reduced complexity by maintaining only one power plane and voltage stabilization circuitry.

- availability of third-party software / hardware, application notes, compilers

  Development environments, compilers and debugger etc. are normally quite expensive. Since the software and Bluetooth stack is written in a high level language (C), a ANSI C compiler is needed. The GNU Project's C Compiler gcc is the preferred compiler for further development.

- Number of hardware UARTs

  For external communication of the device, serial RS232 is used. Serial communication is state of the art for embedded devices. Further, the Bluetooth module provides RS232, USB and I2C as interface. Since RS 232 can be debugged quite easily, this interface was chosen for communication with the bluetooth module. Altogether, the microprocessor should ideally have 2 hardware UARTs.

5.1.2 Selection of the Microprocessor Platform

Browsing the webpages of several know processor manufacturers, it turned out that the design criteria are hard to meet. Four MCU's which met most of the criteria stated above, were taken into the short list:

TABLE 7.

	[11] Triscend E5	**[12] Mitsubishi M16C**	**[13] Microchip Pic 17C756**	**[14] Atmel ATMega103L**
Size (pins)	128	100	64	64
RAM / Flash	8k / 256k	20k / 256k	1k / 32k	4k / 128k
Power Consumption	35mA @ 10MHz, 3.3V	10mA @ 10MHz, 3.3V	5mA @ 4MHz, 3.3V	5mA @ 4MHz, 3.3V
I/O Ports	10	10	5	4
AD Converter	0	16 * 12Bit	4 * 12Bit	8 * 10 Bit
Hardware UART	1	3	2	1
Price p. U.	$60	$52	$35	$17

Triscend t5 is a full-grown 16Bit microprocessor with an incorporated FPGA. The programmable logic in the processor is very appealing - unfortunately the component drains way too much current for mobile environments. Unfortunately, this MCU incorporates only 1 Hardware UART

Similarly, Mitsubishi's M16C met most requirements but current drain. Further, both mentioned parts are packages with 100+ pins, occupying a large amount of space. It seems that both parts are quite new using a modern architecture. The downside is, that no third-party development tools or applications were found on the web. Another negative aspect is the price of both components and the complexity of these 16Bit MCU's.

Microchips PIC 17C756 is the flagship of a whole range of 8 Bit microprocessors. This component met all requirements mentioned above but memory size. The included 902 Bytes of RAM is not enough for our project. Microchips harvard architecture is well suited for small embedded projects with very low RAM requirements. Although external RAM is possible, it cannot be used in memory mapped mode - external memory access need special memory fetching instructions. This would have posed large problems on software design and the used compiler.

The PIC microprocessor is used in numerous applications. Together with an existing GNU C compiler, this microprocessor would have been the ideal platform, not taking the memory problem into consideration.

The device which was finally chosen, is ATMEL's ATmega103L [14]. Most requirements stated above are met. The largest downside is, that this device has only one hardware UART. The second UART which is needed, must be programmed as a software UART

The chosen platform includes:

- 121 Powerful RISC Instructions
- Up to 4 MIPS Throughput at 4 MHz
- 128K Bytes of In-System Programmable Flash Memory
- 4K Bytes Internal SRAM
- 4K Bytes of In-System Programmable EEPROM
- SPI Interface for In-System Programming
- On-chip Analog Comparator
- Programmable Serial UART
- Real Time Counter (RTC) with Separate Oscillator
- Three Timer/Counters with Separate Prescaler and PWM
- 8-channel, 10-bit ADC
- Low-power Idle, Power Save and Power-down Modes
- Software Selectable Clock Frequency
- External and Internal Interrupt Sources
- Power Consumption of 5.5mA (active) and 1.6mA (idle) at 4 MHz
- 32 Programmable I/O Lines
- Operating Voltages of 2.7 - 3.6V

5.1.3 Hardware Design Considerations for the BTNode Platform

The intended use of the BTNode Hardware extends beyond the scope of this diploma thesis. The platform should be deployable for other projects, specially since Bluetooth kits are very rare at the moment. In a first step, a catalog of requirements for our intended usage was created. Second, other features intended for future use were listed and if possible, included into the hardware design. As a starting point for our own design, we used the schema of the STK300, the evaluation kit sold by ATMEL.

The following enumeration lists these considerations in sequence of their importance.

1. To keep current drain as small as possible, the platform is built using 3.3V parts. Both Ericsson's Bluetooth module and the low power version of ATMEL's microprocessor, ATMega103L run on 3.3V. A low-dropout voltage regulator attached to an industrial 3.6V Lithion-Ion battery provides constant power for the board.
2. The low power version of the microprocessor can be clocked up to 4 MHz. Using a 3.6864MHz crystal, most commonly used baudrates can be generated by the UART without any divergence. See page 64 of [14].
3. The generate an accurate timing signal, the possibility to externally clock Timer/Counter 0 is used. Thus, a second crystal of 32.768kHz is attached to the MCU. The 32.768kHz crystal is widely used in watches etc. to provide a precise 1s signal.
4. The platform must be in-system programmable. That is, the content of the Flash memory which holds the executable program, may be reprogrammed at any time. Atmel ships an In-System Programmer ISP together with evaluation kit STK300. The hardware platform uses the same pinout on the programmers interface as the evaluation kit, hence this ISP can be used to program the system.

   Employing this feature restricts the usage of some pins. Pins which are shared between Ports that are externally connected and used by the ISP cannot be used at the same time. The Pins used for the serial RS 232 connection, RXD and TXT, conflict with the ISP. Thus the UART and the ISP cannot be used simultaneously and either of them must be disconnected while the other one is in use. Having to disconnect the UART while programming the system prevents the employment for communication with the bluetooth module.

   The supplied programmer runs on 5V VCC. Fortunately, the programmer is based on a Atmel MCU which also runs on 3.3V. Thus the STK's programmer is run only with 3.3V, even though the standard supply voltage is higher.
5. The platform's UART may be connected to a standard serial port of a PC. The pinout of the on board interface has been chosen, that a straight-through cable can be used. A null-modem cable is not necessary. The on board RS232 transceiver provides the standard +- 15V needed for serial communication.

   Dependant on the application running on the board, hardware flow-control (RTS/CTS) may need to be enabled. As the MCU's Port E is partially used for RXD and TXD, RTS and CTS are also connected to Port E.
6. The Bluetooth module is connected to standard I/O pins. Communication is done over a UART implemented in software, since the hardware UART cannot

be connected permanently. The TXD connection from the Bluetooth module needs to be wired to an edge triggered interrupt. The interrupt is used to start the reception of an incoming byte in the MCU. Port E pin 4 has been chosen as edge triggered interrupt with 4th highest Interrupt priority. Timing is very crucial for the software UART. Never the less, future applications may need higher prioritized interrupts. The remaining connections of the software UART have been wired to PORT A.

7. As Port B and Port D provide some useful features for future use, they are led out to external connectors. These pins can either be used as general I/O. Some pins have extra functionality. Timer/Counter 1, which is not used in our software designed, can be clocked externally over pin 7 port D (PD7), or used as Pulse Width Modulator Source on PB5 and PB6. PD0 - PD3 can be used as external, level triggered interrupts. The pinout used on the hardware platform is the same as being used on the evaluation board. This facilitates future development, as external add-ons may be connected either to the STK300 or the BTNode.
8. The remaining pins of port A are connected to small Light Emitting Diodes (LED's) to be used for optical feedback and / or debugging purpose. The resistors R3 to R6 are needed to reduce current through the LED's. The value of these resistor can be increased up to 500 Ohms to reduce brightness in favor of power consumption.
9. The remaining pins of port E are wired to external connectors to be used as general purpose I/O
10. The MCU features a 10 Bit AD-Converter. The ADC is connected to an 8-channel Analog Multi-plexer which allows each pin of Port F to be used as an input for the ADC. Port F is led to an external connector intended for future use.

    The ADC has a separate analog supply voltage and analog reference pin. To keep things simple, the both are connected to VCC. This wiring is sufficient for general use. For high-precision measurements, at least the reference must be redesigned.
11. Bluetooth communicates at 2.4 GHz. At this speed, special care must be taken into antenna design. To evade HF problems, an industrial 2.4GHz antenna was used. Using this antenna required some special layouting, as described in [23].
12. The power supply of the MCU and Bluetooth module may be disconnected with according Jumpers. These jumpers must be set for normal use, but may be removed for debugging purposes. In addition, current consumption of may be measured over these jumpers.

5.2 First Steps with the chosen Microprocessor

To get to know the hardware infrastructure, a development kit was bought from local the distributor. See “5.4 Contacts” on page 82. This chapter covers the first steps with this evaluation board. The STK300 [15] development board is available for very decent pricing, costing about $100.

The swiss distributor of ATMEL products recommends IAR’s Embedded Workbench as C Compiler and ICE-300 as emulator. These products, only available for the Microsoft Windows platform, are quite expensive, costing about $2000.- each.

Since development of software here is preferably done on the Linux platform, we decided to start with a demo version of the ImageCraft C Compiler [16] [17] on Microsoft Windows and try to get GNU-cc running to continue development in Linux. Using the commercial products on windows enabled us a trouble-free start with the hardware. A small blinking-LED program was compiled on the PC and downloaded to the board via the in-circuit programmer attached to the PC’s parallel port. [18] Several resources on the web [19] pointed out that support for the ATMEL AVR is worked on the GNU-cc projects. Several patches for gcc, binutils and stripped down glibc “floated” around several sites, mainly in Russia and Poland. [21]

It seems, that support for ATMEL’s AVR Processor is now in the latest GCC CVS snapshots and will be final in GCC 3.0 release. For the meantime, we have compiled CVS snapshots of gcc, binutils and glibc for AVR platform to be used for the project [20]. Making the first steps with a demo-version of ImageCraft’s C Compiler helped resolving some small problems with gcc. (default register assignments etc.)

The mandatory “Hello World” program was implemented quite fast. The STK300 board was connected to the hardware programmer and the PC’s serial port as described in the STK’s manual[18].

5.3 Hardware Reference

This chapter covers the custom built hardware. Section 5.3.1 gives an overview of jumpers and connectors which are available. The next section describes the steps needed for setting up operation. In section 5.3.3 a hardware errata is given. Section

5.3.4 list electrical characteristics and section 5.3.5 lists some additional notes on the hardware platform.

5.3.1 On board Connectors and Jumpers

Figure 16 on page 77 shows the main devices and connectors on the embedded hardware. The usage of all Jumpers / Connectors is defined in Table 8, “Definition of Connectors,” on page 78.

Some principal parts are outlined on Figure 16 on page 77. T1 depicts the Bluetooth antenna. Ericsson’s Bluetooth module is represented as U2. U1 represents the Micro Controller Unit. Connectors and Jumpers are illustrated as J1 to J10. Last, the board’s four LED’s are outlined as D1 to D4.

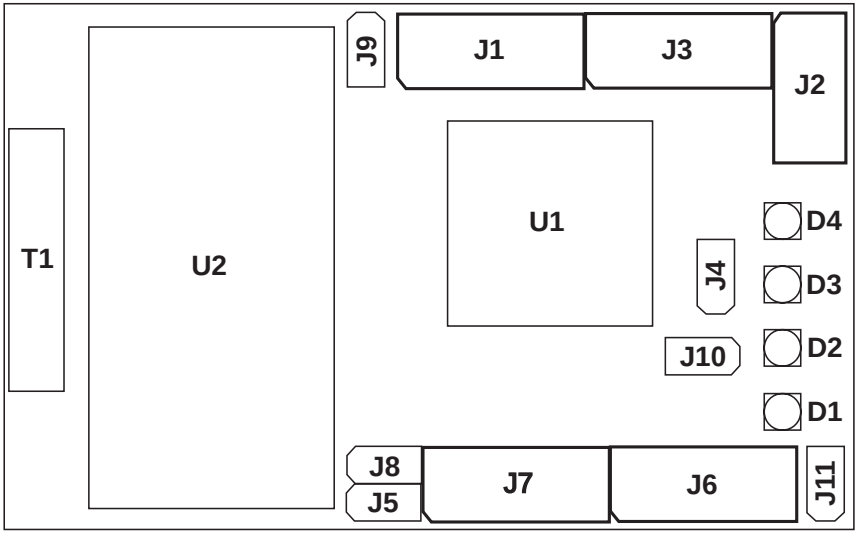


FIGURE 16. BTNode Platform

Table 8, “Definition of Connectors,” on page 78 lists all connectors and jumpers and specifies their functionality.

Note: Some jumpers are only used fur current measurements such as J4 or J9. These jumpers are set during normal operation.

Connector J6 and J7 connect to standard I/O pins on the embedded platform. Con-

TABLE 8. Definition of Connectors

Part	Description
J1	Programmer Interface
J2	Serial Interface
J3	Port F
J6	Port D
J7	Port B
J4	CPU Current Access
J5	Interrupt 1
J8	Interrupt 2
J9	Bluetooth Current Access
J10	Power On / Off
J11	Power Connector

nector J3 is wired to the AD converter. These pins may also be used as simple input pins. In addition to the port pins, each header has a connection for ground and vcc to supply external circuits.

LED's D1 - D4 are wired to the lower half of Port A. These outputs may be used for visual signalization or debugging purposes.

Next to the normal ports which connect to the MCU's standard I/O pins, there are several connectors for special purposes. Connector J2 is used as serial interface to the board. The pinout is wired in such a way that no crossover cable is needed. Connector J2 may be wired straight-through to the serial port of the PC. For proper operation, hardware handshaking (RTS / CTS) needs to be enabled on the PC Table 9, "Pinout of Serial Interface J2," on page 78 specifies the exact setting.

TABLE 9. Pinout of Serial Interface J2

Pin	Function
1	not connected
2	TXD

TABLE 9. Pinout of Serial Interface J2

Pin	Function
3	RXD
4	DSR
5	GND
6	DTR
7	CTS
8	RTS

Connector J1 is wired to contact the standard InCircuit Programmer which ships with the STK300.[15] Due to the fact, that some wires are shared between programmer and serial port, both circuits may not be connected at the same time. Programming the device will fail if the serial port is connected at the same time.

5.3.2 Setting Up Operation

To set up operation, several jumpers need to be installed. Jumper J4 and J9 are used for current access of the Bluetooth kit and MCU. In normal operation, these Jumpers are set.

Power is connected to connector J11. The board must be supplied with 3.6V to 3.8V DC with ground connected to Pin 1 of jumper J11. To turn on power on the board, set jumper J10. Now, the CPU starts running the previously stored program which resides in non-volatile FLASH memory. If the MCU was not programmed, new software must be downloaded as described below.

Polarity is crucial for all connector. Figure 17 on page 80 specifies the exact header layout.

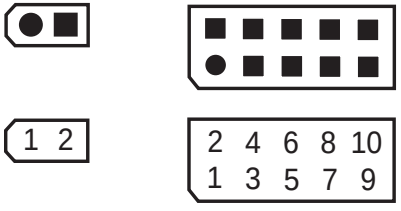


FIGURE 17. Header Layout of Connectors

Next, Software must be compiled as described in [20].

Now, the board can be programmed with the Parallel Port Programmer delivered with the STK300 [15] development board. Programming is done using uisp[22], a Linux programming software distributed under GPL.

In order to communicate with the embedded platform, a PC can be connected to the board's serial port, using the supplied cable. This cable connects every pin from the PC's DB-9 connector straight through to the embedded hardware as listed in Table 10, "PC-BTNode Serial Adapter Cable," on page 80.

TABLE 10. PC-BTNode Serial Adapter Cable

Pin in DB-9	Description	Pin on
1	Not Used	1
2	RXD	2
3	TXD	3
4	Not Used	4
5	Ground	5
6	Not Used	6
7	RTS	7
8	CTS	8
9	Not Used	9

If everything is set up correctly, the board starts doing whatever it is to do.

5.3.3 Hardware Errata

Unfortunately, the hardware design as it is implement at this time, contains one large compromise. The MCU supports a second configuration for the SRAM memory. Port A and port C may be used as Data / Address bus to accessing the external SRAM memory. If the internal 4k Bytes of memory is not sufficient, up to 64k may be addressed in external components. 4k Ram leaves very little space for future applications. Having the possibility to add external memory to the platform via daughter-board, would have opened enormous possibilities for future applications.

For no apparent reason, our partner who did the layouting did not want to wire these two ports to external connectors.

A second problem is the Power_ON connection of the Bluetooth module. Power_ON is used to manually disable the bluetooth module to save energy. We wished, that the layout makes this pin available, even though it is only connected to VCC in this first version. In the current layout, this pin is connected to VCC under the bluetooth module, not available to external components.

The schematic and layout has one small error. Pull-Up resistor R10, which is used to provide a logical 1 to the Reset pin of the MCU is connected to ground. This resistor MUST be connected to VCC, otherwise the board will NOT run at all.

5.3.4 Electrical Characteristics

TABLE 11. Electrical Characteristics

Symbol	Parameter	Typ. Rating
Vcc	Input Voltage	3.4V - 3.8V
Icc	Power Supply Current @4MHz, 3.6 V Vcc	
	Power Down, Bluetooth detached	3 mA
	Running, Bluetooth detached	8 mA
	Running, 1 LED active, BT detached	12 mA
	Running, Bluetooth in PageScanEnable mode	28 mA
	Running, Bluetooth inquiring	56 mA
Iccmax	Max current	100 mA

5.3.5 Notes on Manufacturing

This chapter covers different notes of the manufacturing of the board. The PCB itself is routed with quite small distances between wires. Therefore, wo chose to have the PCB manufactured professionally. The manufacturing costs about 25$.

Next difficult task is to equip the board with components. Ericsson’s Bluetooth Module is built on a Ball Grid Array package. Soldering and placing of this component manually is impossible. Therefore, we had an external company place and solder the top side of the board. Initial cost for this work is quite high, as a solder mask

and program for the placement machine have to be made. The initial cost amounted to about $250. For every board which was placed and soldered, another $15 were due. Unfortunately, only 4 Bluetooth Modules were at hand. As the solder masks are preserved for about a year, manufacturing another series is much cheaper for the next lot.

The same initial cost would have arisen for the bottom layer of the board. This sum was quite large for doing only four boards. Therefore, we decided to solder the bottom layer by hand.

5.4 Contacts

For manufacturing, several other parties were participating. Table 12, “Third-Party Contacts,” on page 82 lists all involved.

TABLE 12. Third-Party Contacts

Swiss Distributor of ATMEL products	**ANATEC AG** Sumpfstrasse 7 6300 ZUG, Switzerland www.anatec.ch Tel. 041/748 32 32
Distributor for Bluetooth Antenna	**Scantec Gmbh** Industriestr. 17 D-82110 Germering www.scantec.de Tel. +49 89/899 14 30
Distributor for SMD Crystall	**Eurodis Schweiz AG** Bahnstr. 58 8105 Regensdorf www.eurodis.ch Tel. 01/843 32 32
Distributor for other electronic components	**Farnell AG** Brandschenkestr. 178 8027 Zürich www.farnell.com Tel. 01/204 64 64

TABLE 12. Third-Party Contacts

Distributor for battery and charger	**CONTREL AG** Boesch 35 6331 Huenenberg www.contrel.ch Tel. 041/781 17 17
PCB Manufacturer	**Walter Schoch AG** Dorfstr. 84 8912 Obfelden Tel. 01/762 41 41
Placing of Components	**elfab AG** Stetterstr. 25 5507 Mellingen www.elfab.ch Tel. 056/481 80 20

CHAPTER 6

Experiments & Results

Chapter 6 covers experiments and results obtained in simulation. Section 6.1 describes a simulation performed with our architecture and discusses results extracted thereof. Section 6.3 lists our experiences gained using the Bluetooth technology

6.1 Simulation of Event Propagation and Evaluation of Data

This chapter describes a single simulation run in the BTSim simulation environment. A realistic scenario was designed and played through. Events are generated and disseminated throughout the network. The obtained data is evaluated using the vicinity application. Interpretation of the results rounds off this chapter.

6.1.1 Simulation Run

Let us assume a setup as depicted in Figure 18 on page 86. A hypothetical office with three rooms is home to several people using Bluetooth equipped devices. At the top, Jim's office is drawn. Jim is using a Bluetooth equipped laptop and PDA. In the center, we see Mary's office. Mary has a Bluetooth enabled mobile phone. In

the room at the far right, Anne is working an a Desktop computer which has Bluetooth built-in. At the left side of the map, a Bluetooth enabled copier machine is located.

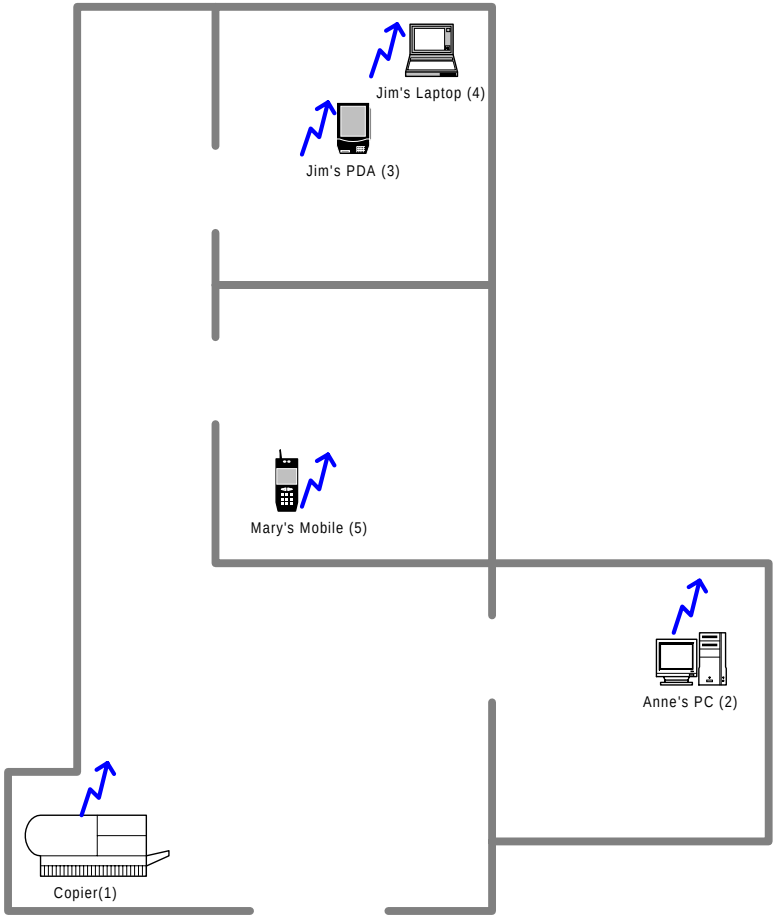


FIGURE 18. Example Scenario

The next paragraphs describe events generated in a real-world scenario of people moving about in our hypothetical office. We will simulate the scenario alongside

the description of a normal working day. To start the simulation, we run the BTSim application by issuing `java BTSim` on the command line.

First, two static nodes, the copier machine on the left and Anne's PC are placed by clicking the mouse into the graphic area of BTSim. (Figure 19 on page 87)

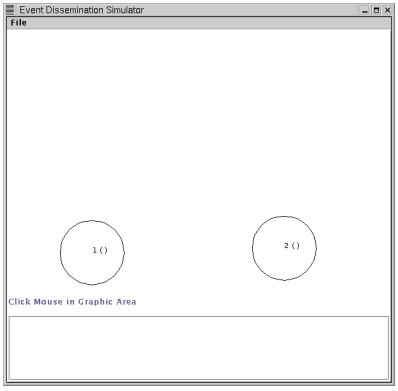


FIGURE 19. Example Scenario Step 1

As the two nodes haven't seen each other yet, no events have been generated so far. A little later, Jim shows up at work, enters his office and turns on his laptop. This is simulated by clicking into the BTSim window and thus "generating" Jim as node 3. Then dragging him passed node 2, the copier machine. Jim's PDA picks up the copier machine's Bluetooth signal and opens a connection to this machine. At this point, the first events in the setup are generated. Copier machine and PDA will each generate and store a found event. Then the two events are exchanged. For simplicity, we assume that neither the PDA nor the copier machine has any previous events stored in memory. Walking past the copier machine, into his office, the devices leave radio distance and the connection is terminated. (Figure 20 on page 88)

The events stored in every node can be displayed by right-clicking into the center of the desired node. Node 3 has the following events stored.

```
Node:3 him:1 foundTime:10.40.11.449 lost-
Time:10.40.13.559 countUp:0 TTL:5 finalFlag:true

Node:1 him:3 foundTime:10.40.11.439 lost-
Time:10.40.13.559 countUp:1 TTL:4 finalFlag:false
```

The first event displayed originates from node 3 and indicates that node 1 has been seen at 8:44:48 and lost again at 8:44:54. The finalFlag is set to true, since node 3 knows that the connection was disrupted.

The second event stored in node 3 originates from node 1 indicating that node 3 has been seen. This event was then passed to node three over the bluetooth connection. This can be seen by examining the countUp value which is 1 opposed to 0 in the first event stored, the TTL is decreased by one.

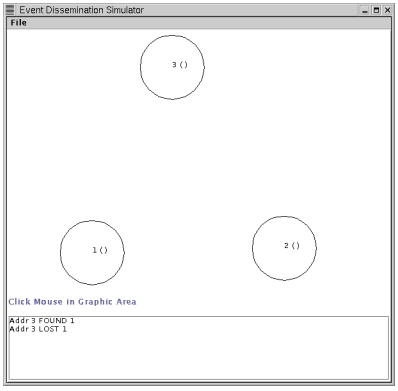


FIGURE 20. Example Scenario Step 2

In this second event record, the finalFlag is false, since the copier couldn't notify Jim's PDA of the disrupted connection, because the connection, well, was disruppted. The lostTime indicates the time of the last update event sent from node 1 to node 3.

Some time later, Jim switches on his laptop. Laptop and PDA are in the same room and thus in radio range. We simulate this by placing a fourth node on the simulation which overlaps node 3. The overlapping nodes 3 and 4 are displayed in blue color indication a connection. (Figure 21 on page 89)

Right-click on node 4 displays four events stored in this node. First, the records generated upon node 3 seeing node 4 (laptop <-> PDA) are listed. Further, events generated by node 1 and 3 (Jim's PDA 'meeting' the copier machine) some time ago have been transmitted into node 4. Thus, node 4 'learned' something about the past behavior of node 3 by receiving foreign events. Note the countUp value of two which confirms, that these events have been passed along twice.

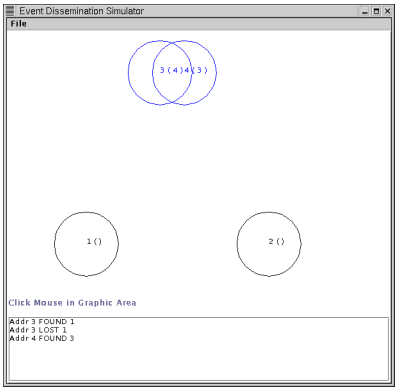


FIGURE 21. Example Scenario Step 3

We now assume that Jim wants to know about the wireless network around him. He starts up the Vicinity application on his laptop which serves topological data to other applications as depicted in Figure 11, “Service Layers of the EventCollector Concept,” on page 44. The Vicinity connects to the local EventCollector, receives the stored events and evaluates them. The program is started as follows:

```
java Vicinity -u http://localhost:10004 -p 3001
```

The “`-u`” parameter indicates, that Vicinity should connect to the EventCollector at localhost:10004. On this TCP port, the EventCollector of node 4 is listening for connection to exchange events. `Vicinity` itself listens on TCP port 3001 (“`-p`” parameter) for incoming connections to serve other applications like `ConnectionGraph`. Vicinity evaluates all events received from the connected node (in this case node 4) and provides processed data on the specified port.

The application for visual representation of the processed data, `ConnectionGraph`, is started by issuing

```
java ConnectionGraph -u http://localhost:3001
```

on the command line. Again, the “`-u`” parameter indicates the server port to connect to. The ConnectionGraph displays a representation of the mobility value upon starting. All nodes reported by the Vicinity are placed on a circle. Lines between

nodes represent a mobility value. Mobility indicates how often the two corresponding node have seen each other. It basically is a frequency. The more encounters two nodes have had, the higher the mobility value, the thicker the line between the two nodes. In our example (Figure 22 on page 90), Jim's PDA has seen the two other nodes exactly once, which yields thin lines between the nodes.

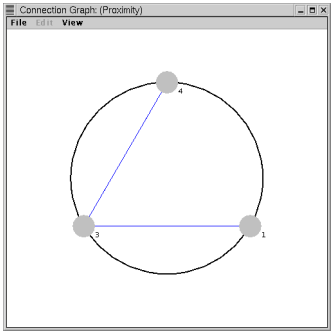


FIGURE 22. Example Scenario Step 3 - Visualization of Proximity

Quantity of connections is one aspect, connection time another. In the ConnectionGraph application, the representation is switched using the `view` menu. The connection weight (Figure 23 on page 90) presents a different view. The line connecting node 3 and 4 is much thicker than the line connecting node 1 and 3. The physical connection between 1 and 3 was much shorter than the connection between 3 and 4 which still lasts.

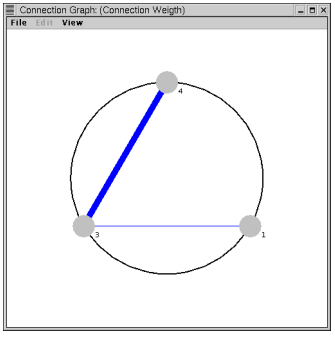


FIGURE 23. Example Scenario Step 3 -Visualization of Connection Weight

A further representation is the snapshot of the current topology as is known to node 4. Using the `view` menu in the ConnectionGraph application, the representation is switched to the topology. (Figure 24 on page 91)

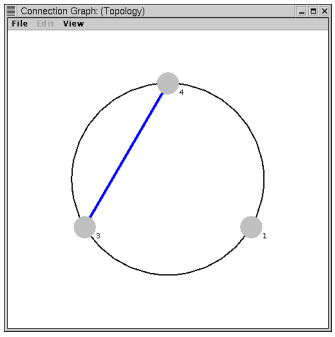


FIGURE 24. Example Scenario Step 3 -Visualization of Topology

At this snapshot's time, only node PDA and laptop maintain a connection which is indication by the line connecting node 3 and 4 in the graphic.

Note, even though node 2 (Anne's PC) exists since the beginning of the simulation, none of the evaluations above showed any sign of life of this node. As mentioned in chapter three, we are only operating on the point of view of one specified node. As none of the nodes have ever entered radio distance of node 2, existence of Anne's PC is not known.

Let us continue with the simulation. During the morning, Jim leaves his room a few times, each time interrupting the connection with his laptop. Once, he even leaves the office completely, passing by the copier machine. We simulate Jim's behavior by dragging node 3 (Jim's PDA) around the screen.

The Mobility representation in the ConnectionGraph application changes. (Figure 25 on page 92) The connecting line between Jim's laptop and PDA (node 3 and 4) is thicker than the line connecting node 3 and 1 since every reconnection between laptop and PDA augmented the proximity value between these nodes.

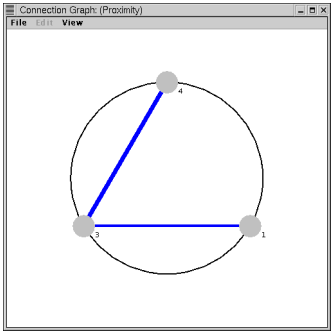


FIGURE 25. Example Scenario Step 4 - Visualization of Proximity

When Jim passes by the copier machine for the second time, the copier machine receives events generated by connections between Jim's PDA and laptop. At this point, the copier machine receives information about this other node (Jim's laptop) and its interaction with Jim's PDA. However, the copier machine's view of the network differs slightly from that of the PDA presented in Figure 25 on page 92 because it is based on partially different events.

We continue our simulation: Some time later, Mary and Anne walk into the office. Mary accompanies Anne into her room, where Mary's PDA picks up the signal from Anne's PC. Next, Mary walks by the copier machine into her own room. Some time later, she carries some records to Anne's room and returns again to her room, passing by the copier machine both times. The final layout is depicted in Figure 26 on page 93.

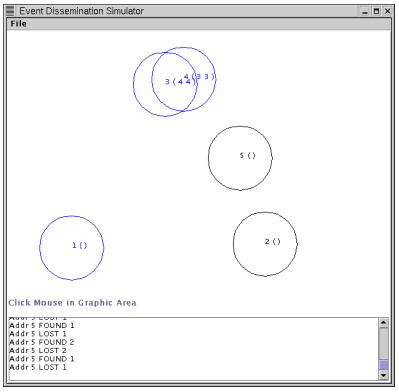


FIGURE 26. Example Scenario Step 5

Now, we would like to give a simple example on how to use the collected data. Let us assume, that Anne started a large job on the copier machine. Having finished the job, the copier machine would like to send Anne a message to inform her of the finished job. As a first possibility, the copier machine tries to send the message directly to Anne's PC. A possible route to node 2 could go either over a direct connection or over multiple hops, if there existed a path from node 1 to node 2. Consulting the local topology snapshot (Figure 27 on page 93), the copier machine derives that in it's perspective, there is no current connection possibility to Anne's PC (node 2).

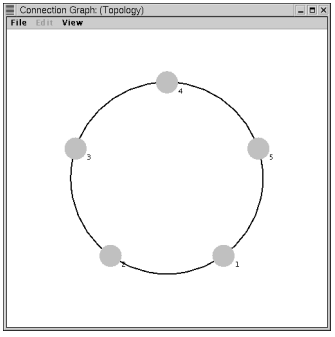


FIGURE 27. Example Scenario Step 5 - Visualization of Topology

As no connection exists, multi hop routing is needed. Consulting the mobility representation (Figure 28 on page 94) gives a clue how to reach node 2.

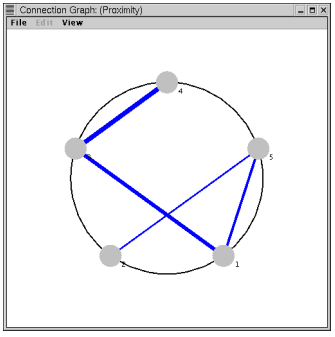


FIGURE 28. Example Scenario Step 5 - Visualization of Topology

TABLE 13. Event List of the Copier (Node 1)

Node	him	foundTime	lostTime	countUp	TTL	finalFlag
1	5	10.42.17.358	10.42.21.082	0	5	true
5	1	10.42.17.352	10.42.17.610	1	4	false
2	5	10.42.08.810	10.42.09.750	2	3	false
5	2	10.42.08.804	10.42.14.552	1	4	true
1	5	10.42.03.387	10.42.06.048	0	5	true
5	1	10.42.03.382	10.42.06.048	1	4	true
1	5	10.41.35.663	10.41.38.935	0	5	true
5	1	10.41.35.661	10.41.38.935	1	4	true
1	3	10.41.00.189	10.41.02.416	0	5	true
3	1	10.41.00.184	10.41.00.184	1	4	false
3	4	10.40.36.509	10.40.38.558	1	4	true
4	3	10.40.36.513	10.40.36.513	2	3	false
3	4	10.40.29.980	10.40.32.822	1	4	true
4	3	10.40.29.984	10.40.32.822	2	3	true
3	4	10.40.23.552	10.40.26.330	1	4	true
4	3	10.40.23.555	10.40.26.330	2	3	true
3	4	10.40.18.030	10.40.20.178	1	4	true
4	3	10.40.18.033	10.40.20.178	2	3	true
1	3	10.40.11.439	10.40.13.559	0	5	true
3	1	10.40.11.449	10.40.13.559	1	4	true

From the mobility information, it can be derived, that node 5 (Mary's PDA) had a connection to Anne's PC some time in history. Thus, next time Mary passes the copier machine, there is a possibility that she is heading to Anne and therefore be used as a piggy-back carrier for the message. The message will probably not be given to node 3 (Jim's PDA) since there has not been a connection to node 2 so far. The behavior of Jim doesn't make him a good messenger for messages to Anne. If the mobility evaluation yields two different paths, a preferred route could be chosen on the basis of the frequency of possible connections. Nodes which connect frequently to node 2, will connect again with much higher probability than a node which has had only a single encounter.

To examine the given example in detail a complete list of the event known to the copier is given in Table 13 on page 94.

6.1.2 Discussion of the Simulation

This short example already yielded a considerable number of events. If more nodes are involved and if they are more mobile this number rises quickly. For small embedded devices this means that the point where events must be discarded is reached quickly. It therefore helps if more powerful entities are in the area to store large amounts of events, like Anne's PC, for example.

In the EventCollector concept one event describes the whole connection from the start to the end. Whether the connection ends or continues, the existing event only gets updated with the new lostTime or the finalFlag. This results in a considerable compression. Updates only generate network traffic but do no new events and thus do not require more storage space on the nodes. The number of events could be cut in half by generating events upon rendezvous instead of a asymmetric discovery. This however is not easily feasible with Bluetooth and also wouldn't allow to discover "dumb devices".

By sending updates more frequently applications can be informed quicker of network changes. For example to obtain a current topology short update intervals are required while the generation of a map of a building requires hardly any updates at all. There is a compromise between speed on one hand and bandwidth and power on the other.

6.2 Using a Real World Setup

The intention was to play through a similar scenario as in the previous subchapter with the BTNodes in real life. The purpose was to show that the result was comparable to the simulation. Since the porting of the Bluetooth protocol stack could not be finished in time, real world trials were not carried out. Preliminary testing with available parts of the stack showed no obstacles for a successful deployment of the EventCollector concept.

6.3 Experiences with the Bluetooth Technology

Bluetooth is a very feature rich communication basis. It is very robust, features error corrected communication channels, high throughput and even some sort of real time communication service. As Bluetooth becomes more and more common, it will be a cheap and powerful platform. Yet there are several problems for applications as seen in our work. This subchapter is not a Bluetooth overview, instead it summarizes our experiences with the Bluetooth technology. Most aspects deal with Bluetooth in general, but some also with the Ericsson ROK 101 007 modules in particular.

Bluetooth was designed to replace point-to-point communication using cables. The architecture of Bluetooth is inherently client-server based. Also a Bluetooth network is quite static, it reacts rather slow to network changes. These circumstances resulted in some workarounds to implement the EventCollector on the embedded platform, as described in chapter 4.

The architecture of Bluetooth is inherently client-server based. There is a client that makes an inquiry and a server that responds. This nature is also reflected by the fact, that there is a master and some slaves. Nodes are not equal. This paradigm makes sense for a computer with some peripheral devices or for a cellular phone and its head set. However, it poses some problems in a truly symmetric peer-to-peer network.

When two nodes meet, one of them becomes the master, the other one slave. A third node will be slave as well. But there is no direct communication possible between slaves. Two slaves can see each other through inquiries, but cannot contact each other directly. To communicate, either of them may request a role change to become master, or the two nodes may form a new piconet, again with one of them

as master and the other one as slave. But what about the third one, the old master? A node can be a member of more than one piconet at the same time, but it can only follow the frequency hopping sequence of one of them. It must switch back and forth between the piconets and is only available for the members of the momentarily active piconet. There is no true peer-to-peer or inter-piconet communication.

There are some problems finding other BT nodes within the surrounding. Frequency hopping is an excellent way to cope with interference. It assumes however that all communication partners hop synchronically, e.g. change to the same frequencies at the same time. When a new communication partner appears, it must synchronize itself with the others. This is a nontrivial task, since it has no idea on which frequency they are and what the next hop will be. The new node has to try different frequencies in a specific order until it finds some or all of its neighbours. This process involves a lot of transmissions and thus costs time, power and bandwidth. The compromise chosen in the Bluetooth standard favors bandwidth and power over time. This means, it may take a rather long time to find other entities in the vicinity. An inquiry takes roughly 1.3 seconds, but only after 4 to 5 inquiries chances are high to have found all neighbours. Reference to fast connection establishment may be found in [34]. This value is a rough estimation and in only fairly right if the RF stage of the other nodes are not actively transmitting (data or inquiry) at that time and are not in power down mode. The latter restriction is caused by the fact, that at RF level communication actually is simplex. The module can either send or receive, but not both. Duplex communication channels are emulated in higher protocol layers in software. Also during inquiries the whole data traffic is interrupted. Again, this is not a problem in a fairly static client - server situation where the whole piconet is synchronized through the master.

A desirable feature for applications such as ours would be a kind of packet sniffer to passively discover other nodes around. This would be less power intense than the active inquiry process. But again, frequency hopping complicates things here. Another feature could be to report to the upper Bluetooth stack layers when the module gets inquired.

The Ericsson ROK 101 007 modules in particular have additional restrictions. They are not multi point capable, only one connection can be open at once. As seen above, a Bluetooth module cannot respond to inquiries while actively transmitting. The ROK in particular cannot respond to inquiries at all, if there is a data connection, even if it is idle. Another problem is the high power consumption of these modules. But newer modules use considerably less power than the ROK.

There are several workarounds for the above problems. We have seen, that we need to be idle in order to be discovered by others and we have only one connection at a time. We want to inquire as often as possible and exchange enough data with the neighbours while still be idle often enough to be seen by the others. Therefore we must find a compromise between being active through inquiry and data communication and being idle. We try to be idle for about 2/3 of the time by restricting the inquiries and by closing a data connection immediately after the data has been exchanged. For example: 5 seconds inquiry, 5 seconds data exchanging with neighbours and 20 seconds idle. As seen here, such a period takes about 30 seconds, which is rather long.

The overall period must be randomly distributed. Otherwise it would be possible that two nodes would never see each other. If they coincidentally would inquire at the same time they were in phase and always would be.

Since we do not have a guarantee to see a neighbour within our reach at every inquiry, we only declare a neighbour as lost if we haven't seen him for 3 consecutive inquiry periods. So in the worst case it takes about 90 seconds to realize that a certain node has left. If a node connects to us this is a life sign, much as an inquiry. This helps reduce wrongly lost neighbours.

By applying the above algorithms, it is possible to avoid most of the problems imposed by Bluetooth and the ROK. Piconets in this scenario exists only very shortly. They consist of two nodes while they are exchanging data. Then the piconet is torn down immediately. The downside of this solution is the slow reaction to changes in the vicinity. This reaction time could be reduced with some additional workarounds. For example one could try to make less inquiries while accepting longer times to find a node. To compensate for this one could ping the already known Bluetooth addresses to find out if they are still there. This way a leaving node is detected much faster, especially with fully Bluetooth compliant modules that can reply to pings and inquiries while other connections are open.

As for now reaction times are rather long. One cannot just walk passed another BT node and be sure to have made contact with it! This is not very nice, but it suffices to demonstrate our infrastructure.

CHAPTER 7

Related Research

This chapter covers related research carried out for this thesis. It is subdivided into 2 sections. Section 7.1 treats several papers focusing on the conceptual part of this thesis whereas section 7.2 handles technical related parts such as Bluetooth technolgy.

7.1 Research Relating to Conceptual Aspects

One of the early implementations of event gathering to obtain location information was the Active Badge Location System [24]. So called BAT's, small portable infrared beacons, were used to track people at an installation of the AT&T Laboratories in Cambridge.

The Cricket Location Support System [25] is a current approach to the problems encountered in the Active Badge Location System. Cricket is the result of several design goals, including user privacy, decentralized administration, network heterogeneity, and low cost using HF responder beacons.

[27] covers algorithms for position and data recovery in wireless sensor networks. The mathematical aspects of network connectivity to reconstruct node positions via linear or semi definite programming is explored.

In [28], a family of adaptive protocols that efficiently disseminates information among sensors in energy constrained wireless networks is presented. Several other methods are compared to the suggested SPIN (Sensor Protocols for Information via Negotiation) protocol.

In [26], a new paradigm for local communication between devices in Ubiquitous Computing environments is proposed. Local communication in the RAUM system is established using spatial criteria.

7.2 Research Relating to Technical Aspects

Smart Dust [29] is a research project of the University of California, Berkeley. It covers autonomous sensing and communication in devices measuring less than one cubic millimeter. Special attention has been paid towards miniaturization of sensor and communication technology.

In [30], a smart dust implementation using commercial-off-the-shelf components was conducted as a diploma thesis. The author used similar hardware devices as we did for our platform. For communication, a self implemented protocol relying on commercial HF transceivers is used.

[31] illustrates the Vision, Goals and Architecture of the Bluetooth standard. Several other competing standards such as IEEE 802.11, HomeRF and IrDA are touched and compared to Bluetooth.

A report on Bluetooth Basics for Internet Appliance Design [32] gives an easy to understand introduction into different aspects of the Bluetooth standard.

One of the chairmen of the Bluetooth air protocol group has written a very detailed report on the Bluetooth Radio System [33]. Topics such as modulation, medium access, connection establishment and error connection are covered in great technical details.

In [34], delay bottlenecks in connection establishment are pinpointed and a technique for fast establishment of ad-hoc connectivity is introduced.

CHAPTER 8

Summary & Outlook

This thesis proposes a distributed infrastructure for ad-hoc networks which allows participating nodes to improve situation awareness. Nodes with vastly different capabilities cooperate in collecting information about the environment. This infrastructure is event based. Its abilities was shown through simulation. The live demonstration with our Bluetooth nodes has yet to be completed.

The resulting EventCollector concept has proven to be a robust and flexible platform to explore an ad-hoc network on a rather abstract level. It is based on events that are created and distributed by the nodes themselves. Such events are generated upon two nodes meeting or loosing sight of each other. In addition a node affirms an ongoing neighbourship by sending update events on a regular basis. This keeps the system current and makes it less susceptible to transmission errors. These found, lost and update events are flooded throughout the network. Each node may collect as many events as it likes and thus serves as a reservoir for other, newly appearing nodes. Through analysis of its event collection every node may extract the relevant properties of the network and build its own perception of the environment. This infrastructure is extensible to distribute other information than just “have seen” and “have lost”.

The three exemplary measures for neighbourhood have been shown to provide useful information to higher level applications:

- The topology for example could be used for routing as it shows who is currently connected to whom.
- If no direct connection path exists, the mobility value helps to find a piggy back node for multi hop routing. It takes past behavior into account in the form “node A frequently has contact with node B”.
- The connection weight measures the average connect time of a link. It therefore is a rough measure of physical proximity within the chosen time interval.

To set up a live demonstrator a small mobile node with a Bluetooth module was built. Besides communication it features peripheral capabilities, e.g. for sensors. The Bluetooth technology proved to be a very robust and comfortable platform for our purpose, allowing us to concentrate on the main issues. However, there are some drawbacks like high power consumption and long reaction times to network changes. While power consumption certainly will decrease in the near future, the delays are determined by the simplex HF stage and the frequency hopping used.

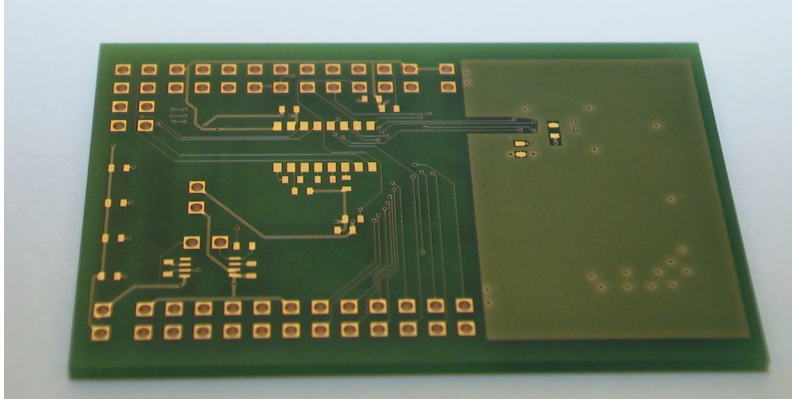
For future work the vicinity data may be enriched with additional information:

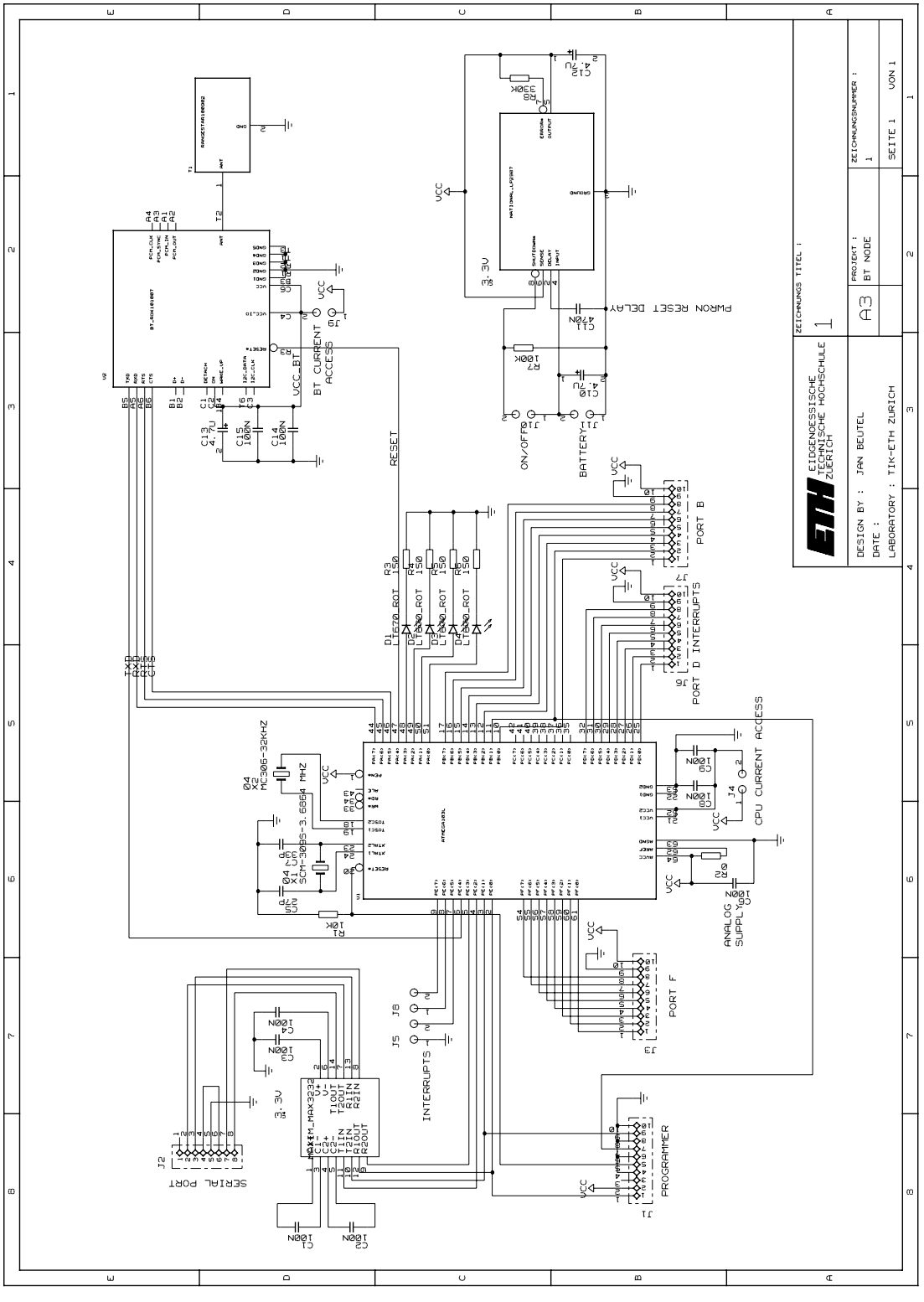
- Link status with bit error rate, throughput or latency may help to find better routing routes.
- By distributing distance estimations between nodes, the physical arrangement may be deduced more accurately.
- Nodes, which provide descriptive data about themselves, help improve the interpretation of the extracted information. For example if some nodes might be found static one could start to map the topological information to a chart and thus provide navigation and tracking services.

With additional sensor data like temperature, noise, light or motion even more can be learned about the vicinity. As a result, a node’s perception can be refined.

Appendix A

Circuit Diagram




ZEICHNUNGS TITEL :
EIDGENOESSISCHE TECHNISCHE HOCHSCHULE ZUERICH
DESIGN BY : JAN BEUTEL
DATE :
LABORATORY : TIK-ETH ZURICH
A3
PROJEKT : BT NODE
ZEICHNUNGSNUMMER : 1
SEITE 1 VON 1
BT_ROK101007
MAX3232
ATMEGA128L
MC306-32KHZ
SCM-30B5-3.6864 MHZ
BT CURRENT ACCESS
VCC_BT
RESET
PWRON RESET DELAY
ON/OFF
BATTERY
PORT B
PORT D INTERRUPTS
PORT F
CPU CURRENT ACCESS
ANALOG SUPPLY
INTERRUPTS
PROGRAMMER
SERIAL PORT
TXD
RXD
CTS
3.3V
VCC

Appendix B

Bill of Material

TABLE 13. Bill of Material

Part No.	Device Type	Description	Distributor
C1, C2, C3, C4, C6, C8, C9, C14, C15	SMD CAP-100N,SAVX0603	100nF Capacitor	Farnell
C5	SMD CAP-27P,SAVX0603	27pF Capacitor	Farnell
C7	SMD CAP-33P,SAVX0603	33pF Capacitor	Farnell
C10, C12, C13	SMD AP_POL-4.7U,STANTAL_BSIZE	4.7µF Capacitor	Farnell
C11	SMD CAP-470N,SAVX0603	470nF Capacitor	Farnell
R1	SMD RES-10K,SAVX0603	10kΩ Resistor	Farnell
R2	SMD RES-0,SAVX0603	0Ω Resistor	Farnell
R3, R4, R5, R6	SMD RES-150,SAVX0603	150Ω Resistor	Farnell
R7	SMD RES-100K,SAVX0603	100kΩ Resistor	Farnell
R8	SMD RES-330K,SAVX0603	330KΩ Resistor	Farnell
D1, D2, D3, D4	DIO_LED-LT670_ROT,SLED_LST670_A	Red Light Emitting Diode	Farnell
X1	QUARZ_04-SCM-309S-3.6864 MHZ	3.6864MHz Oscillator	Eurodis
X2	QUARZ_04-MC306-32KHZ,SMC306	32.768kHz Oscillator	Eurodis
IC U1	ATMEGA103L_STQFP64A-STQFP64A,SA 4MHz	ATMEL ATMEGA 103L	Anatec
IC U2	BT_ROK101007_SROK101007-SROK10A	Ericsson Bluetooth Kit BT_ROK101007	Ericsson
IC U3	SMD NATIONAL_LP2987_SS08-SS08		Farnell

TABLE 13. Bill of Material

Part No.	Device Type	Description	Distributor
IC U4	MAXIM_MAX3232_SSOIC16-SSOIC16	RS232 Driver	Farnell
T1	RANGESTAR100902_SRS100902-SRS1A	Antenna	Scantec
J1, J3, J6, J7	CON-010_10-PIN-T_JUM2X5	2*5 Jumper	Farnell
J2	CON-008_8-PIN-T_JUM2X4	2*4 Jumper	Farnell
J4, J5, J8, J9, J10, J11	JUM-002_2-PIN-T_JUM1X2	1*2 Jumper	Farnell
B1	NICD Tel Pack 3.6V, 600mAh/ VEG620*3	NiCd Accu	Contrel

Appendix C

Application Notes

This chapter covers additional applications which are possible using the hardware platform built for this thesis.

11.1 Uart2Suart

The Uart2Suart application implements a pass-through mode. Data received on the hardware UART is transparently passed on through to the software UART and vice versa. Upon power up, the Bluetooth module is initialized, PAGE_SCAN_MODE enabled such that the module can be inquired and the baudrate is set according to the software UART's capabilities. Thereafter, the hardware platform can be used just like Ericssons' Bluetooth Tool Kit.

LED 0 flashes every second for 250msecs (as specified in AVR_Time.c)
LED 1 is set whenever the CPU is working and unset upon entering sleep mode. This time spread where the CPU is actually processing is too short to be recognized by human eyes. Therefore, the CPU seems always asleep.
LED 2 is set upon errors on the software UART (as specified in AVR_SUart.c)
The initialization of the bluetooth module requires some time. LED 3 signals that initialization is complete and that pass-through mode is enabled.

The hardware platform can be connected via straight-through serial cable to the PC's serial port set to 57.6kBaud. Hardware handshaking (not Xon/Xoff!) must be enabled on the PC's serial port!

11.2 AVRnd

AVRnd is an application to test the Random Sequence Generator implemented using the analog to digital converter. A sequence of 10 numbers are printed out on the serial port.

LED 0 flashes every second for 250msecs (as specified in AVR_Time.c).

The hardware platform can be connected via straight-through serial cable to the PC's serial port set to 57.6kBaud. Hardware handshaking (not Xon/Xoff!) must be enabled on the PC's serial port!

11.3 ADCTest

ADCTest is an application which reads sensor data connected to the analog to digital converter input.

Every 2 seconds, sampled values of port 0 and port 1 of the ADC are printed out over the serial interface.

LED 0 flashes every second for 250msecs (as specified in AVR_Time.c)

Figure 29 on page 111 shows the schematic of the circuit used to extend the board with a light and temperature sensor. The devices, a photoresistor and NTC resistor, are standard components with no special characteristics. Our devices have a mean resistance of about 10kOhms. Together with a normal 10kOhm resistor, a voltage divider is formed. The resulting voltage is measured with the analog to digital converter.

The hardware platform can be connected via straight-through serial cable to the PC's serial port set to 57.6kBaud. Hardware handshaking (not Xon/Xoff!) must be enabled on the PC's serial port!

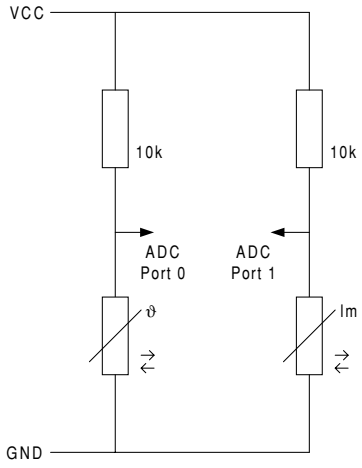


FIGURE 29. Schematic for the ADC Extension

11.4 Using a Software UART on the STK Board

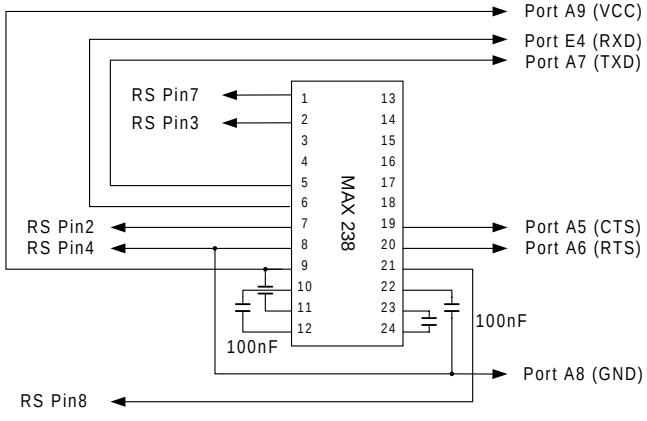


FIGURE 30. External Wiring for Software UART on STK

Figure 30 on page 111 show the external wiring needed to connect a second serial device to the STK 300. Ports used on the STK side are defined in AVR_SUart.c.

Note: A null-modem cable is needed to connect the extension board to the serial port of a PC!

Appendix D

Bibliography

[1] Bluetooth Special Interest Group, Bluetooth, http://www.bluetooth.com/, February 2001

[2] HomeRF Working Group, HomeRF, http://www.homerf.org/, February 2001

[3] Jack P.F. Glas, Spread Spectrum, http://cas.et.tudelft.nl/~glas/thesis/

[4] Infrared Data Association, IrDA, http://www.irda.org/, February 2001

[5] IEEE, The 802.11b standard, http://www.ieee.org/, February 2001

[6] Axis Communications corporation, The Bluetooth on Linux Homepage, http://developper.axis.com/software/bluetooth/, February 2001

[7] Peter Danegger, Full duplex software UART, http://www.specs.de/users/danni/avr/soft/uart/index.htmhttp://www.specs.de/users/danni/avr/soft/uart/index.htm, February 2001

[8] Winfield Stanton and Thomas Spencer, Primer on Asynchronous Modem Communication, http://www.microsoft.com/technet/hw/async232.asp, February 2001

[9] Linux HOWTO, Text-Terminal-HOWTO, http://stone.trew.it/doc/howto/html/Text-Terminal-HOWTO-10.html, February 2001

[10] Olliver Kasten, SmartIts Project, http://www.inf.ethz.ch/~kasten/research/smart-its/, February 2001

[11] Triscend Corp., Triscend E5 family, http://www.triscend.com/products/dse5csoc.pdf, February 2001

[12] Mitsubishi Corp., MC16 family, http://www.mitsubishichips.com/data/datasheets/mcus/mcupdf/um/62eum.pdf, February 2001

[13] Microchip Inc., PIC 17c75x family, http://www.microchip.com/download/lit/pline/picmicro/families/17c75x/datasheet/30264a.pdf, February 2001

[14] ATMEL, ATMega 8Bit AVR Series, http://www.atmel.com/atmel/acrobat/doc0945.pdf, February 2001

[15] ATMEL, STK300 Development Board, http://www.atmel.com/atmel/acrobat/doc1161.pdf, February 2001

[16] ImageCraft, Development tools for AVR, http://www.imagecraft.com/software/adevtools.html, February 2001

[17] ATMEL, First Steps with ImageCraft C Compiler, http://www.atmel.com/atmel/acrobat/doc1630.pdf, February 2001

[18] ATMEL, STK300 Starter Kit User Guide, http://www.eu.atmel.com/atmel/acrobat/doc1149.pdf, February 2001

[19] AVR Forum, The AVR Forum, http://www.avr-forum.com/, February 2001

[20] Lukas Karrer, Notes on AVR Platform and GNU Tools, http://www.lka.ch/projects/avr/, February 2001

[21] Denis Chertykov, GNU tools for the ATMEL AVR micro controllers, http://home.overta.ru/users/denisc/, February 2001

[22] Incircuit Serial Programmer, http://www1.itnet.pl/amelektr/avr/uisp/, February 2001

[23] RangeStar, Overview over Ultima Series Antennas, http://www.rangestar.com/pdf/ultima_technical_overview.pdf, February 2001

[24] A. Hopper, V. Falcao, J Gibbons, The Active Badge Location System. ACM Transactions on Information Systems 10, 1 (January 1992)

[25] N. Priyantha, A Chakraborty, H. Balakrishnan, The Cricket Location-Support System, Proc. of the Sixth Annual ACM International Conference on Mobile Computing and Networking (MOBICOM), August 2000.

[26] M. Beigl: Spatially aware local communication in the RAUM system. Proceedings of the IDMS, Enschede, Niederlande, October 17-20, 2000,

[27] Lance Doherty, Algorithms for Position and Data Recovery in Wireless Sensor Networks, http://basics.eecs.berkeley.edu/sensorwebs/publications/lance_thesis.pdf, February 2001

[28] Joanna Kulik, Wendi Rabiner, Hari Balakrishnan, Adaptive Protocols for Information Dissemination in Wireless Sensor Networks, Proc. 5th ACM/IEEE Mobicom Conference

[29] University of California, Berkeley, Autonomous sensing and communication in a cubic millimeter, http://www-networking.eecs.berkeley.edu/~pister/SmartDust/, February 2001

[30] Seth Edward-Austin Hollar, Smart dust implementation using commercial-off-the-shelf components, http://www-bsac.EECS.Berkeley.EDU/~shollar/shollar_thesis.pdf, February 2001

[31] J. Haartsen, M. Naghshineh, J. Inouye, O. Joeressen, W. Allen, Bluetooth: Vision, Goals and Architecture, ACM Mobile Computing and Communications Review, Volume 2, Number 4, October 1998

[32] Rebecca Spaker, Bluetooth Basics, http://www.embedded.com/internet/0007/0007ia1.htm

[33] Jaap Haartsen, The Bluetooth Radio System, IEEE Personal Communications, February 2000

[34] T. Salonidis, P. Bhagwat, L Tassiulas, Proximity Awareness and Fast Connection establishment in Bluetooth, Mobile and Ad Hoc Networking and Computing, 2000. MobiHOC.2000.