UNIVERSITÄT DUISBURG-ESSEN
FAKULTÄT FÜR WIRTSCHAFTSWISSENSCHAFTEN
INSTITUT FÜR INFORMATIK UND WIRTSCHAFTSINFORMATIK
LEHRSTUHL FÜR PERVASIVE COMPUTING

Bachelorarbeit

Design and Implementation of a Distributed, Low-Power, Signal Strength Measurement System

Jan Nauber
Matrikelnummer: 3003433

Networked Embedded Systems Group
Institut für Informatik und Wirtschaftsinformatik
Fakultät für Wirtschaftswissenschaften
Universität Duisburg-Essen

February 9, 2017

Erstprüfer: Prof. Dr. Pedro José Marrón
Zweitprüfer: Prof. Dr. Torben Weis
Zeitraum: 18. November 2016 - 10. Februar 2017

Contents

Abstract

Wireless Sensor Networks (WSN) are an emerging technology that enables new possibilities for localization and tracking. Radio Tomographic Imaging (RTI) is a localization technique that analyzes the signal strength between nodes inside a WSN, making it possible to localize and track a person without it needing to carry any device.

To do this, a system that measures the signal strengths of each link inside a WSN is needed. The approaches suggested by the literature however assume that all the nodes inside a WSN can hear each other. This makes it impossible to apply the same solution to a WSN that covers large areas like a whole floor or a whole building. To make this possible, a new approach is necessary that is able to measure the signal strength of each link in a WSN where not every node can hear all the other nodes.

This thesis provides an approach for a self calibrating signal strength measurement system that takes into account that not all the nodes can hear each other, qualifying it to be deployable in large areas. To measure the signal strength, each node needs to send a message. The suggested approach creates a schedule that defines a predecessor and successor for every node inside the WSN. On the basis of this schedule the nodes are able to send messages and measure the signal strength in a synchronized way that avoids message collisions which would result in invalid measurements for the computation of the RTI. To collect the measurement from the WSN paths from each node to the central point are defined in a calibration phase.

The system is then tested in a WSN with 32 nodes that covers a whole floor with a size of $531m^2$. These tests show that the suggested approach can take measurements quick enough to enable localization with RTI.

Chapter 1

Introduction

1.1 Motivation

Wireless Sensor Networks (WSN) are an emerging technology that enables a lot of different applications for environmental and habitat monitoring, analysis of structures, or localization and tracking. In the field of localizing and tracking, Radio Tomographic Imaging (RTI) was developed. It is a localization technique that analyzes the signal strengths between nodes inside a WSN, making it possible to localize and track a person without it needing to carry any device. To do this, a system that measures the signal strengths between every node inside a system is needed. Such a system already exists, but it assumes that all the nodes of the WSN can hear each other to calibrate itself and to make the collection of the necessary information possible. This limits the functionality of the system to small areas and it cannot be used to monitor for example a whole floor or a whole building, excluding the possibility to localize in such areas. To make localization with RTI in large areas possible, a new approach is needed that makes the measurement of the signal strength in a distributed WSN possible.

1.2 Problem definition

To measure the signal strength, nodes need to send messages on the basis of which the signal strength can be measured. The problem is that two messages sent at the same time can collide and distort the measured signal strength. This makes a procedure necessary that schedules all the nodes inside the WSN in such a way that only one node sends a message at a given point in time. Moreover, to get a full dataset of the signal strength measurements, the measurements from each individual node have to be collected at a central point making RTI localization possible. All this needs to work under the assumption that not every node is in range of all the other nodes.

The goal of this thesis is to develop an approach that schedules the nodes message sending in a way that no two or more nodes send at the same time, measures the signal

strength and collects the data at a central point. The system should be self calibrating and usable with any given WSN. The idea is to have a system with an initial calibration phase that defines paths to the central point from every node in the network to be able to collect the data even if a node is not directly connected to the central point. Moreover the calibration phase will define a schedule in a way that every node has a predecessor that is in its range. If the node then receives a message from its predecessor it can send its own message.

1.3 Bachelor Thesis Structure

First, the Material and Methods chapter will give an overview about Wireless Sensor Networks, Radio Tomographic Imaging, the existing signal strength system and a Wireless Sensor Network that is located at the University of Duisburg-Essen and used to test the developed approach. Then the Approach chapter explains the developed approach for a signal strength measurement system working in every WSN. Next, in the Implementation chapter it will be explained how the developed approach can be implemented. After that, in the Evaluation chapter, the developed system will be evaluated. Last, the Discussion chapter will discuss the approach.

Chapter 2

Materials and Methods

2.1 Wireless Sensor Networks

A Wireless Sensor Network (WSN) is a collection of small, low-cost, low-power sensor nodes that do not only have sensing capacities but also computing capacities. Additionally, these nodes are able to communicate with each other via a wireless medium. Such a WSN deployed in an area of interest makes it possible to monitor the area in a desired way enabling applications for environmental and habitat monitoring, analysis of structures, or localization and tracking.

The typical architecture of a WSN includes one node that functions as the sink. This means that this node is the connection to a PC or the Internet to bring the information collected by the nodes to the user. All the nodes then are spread over the area of interest. When placing the nodes it is important that each node can somehow communicate with the sink, may it be a direct connection or an indirect over other nodes. Are all the nodes in direct range of each other it is a single-hop network meaning each node can transmit a message directly to all the other nodes. Otherwise the network is called multi-hop. This means nodes can not necessarily transmit messages directly to all the other nodes but may need to transmit the message over other nodes that forward it to the destination.

The application running on the node is designed specifically for the purpose of the WSN. Since the nodes should operate independently, they are often powered by battery. This makes power one of the constrains of a WSN and applications need to be designed to have a low power consumption. Moreover the application designed for the nodes needs to take the constrains of low computation capability and low storage of the sensor nodes into account. [Somb] [LMP+]

2.2 Radio Tomographic Imaging

Radio Tomographic Imaging (RTI) is a method to localize people inside an area covered by a WSN. To do so, the WSN monitors the received signal strength (RSS) of each link

inside the network by letting each node send broadcast messages over radio. Whenever a person stays or moves inside the monitored area, it affects the RSS of some or all links. The changes can then be processed and a position of the person can be estimated. This makes it possible to localize a person without it having to carry any device. [BKP]

2.3 Multi-Spin

When monitoring the RSS of each link inside a WSN, it is important to take into account that not only changes inside the environment can affect the RSS. Also multiple messages sent at the same time interfere with each other and distort the measured change of the RSS. To counteract this, a method to schedule the messages in a way that only one node sends at a given point in time is needed. In the literature the Multi-Spin [BKP] algorithm is suggested that defines for each node a point in time when it should send its message.

In Multi-Spin, time is divided into *slots* and *cycles* where a *cycle* is the time all the nodes need to send one message each. Then a *cycle* is divided by the number of nodes inside the network resulting in one *slot* for each node, as shown in Figure 2.1. Now each node sends in one of these slots. The order in which the nodes send is defined by their ID. To make this possible, the nodes need to somehow synchronize at the beginning. Therefore each node broadcasts messages until one message is received by the nodes. Then the nodes synchronize their time with that message, define their timeslots and start sending in the order of their IDs. Whenever a node receives a message, it can calculate the time until the next *cycle*, so the nodes stay synchronized all the time. [BKP]

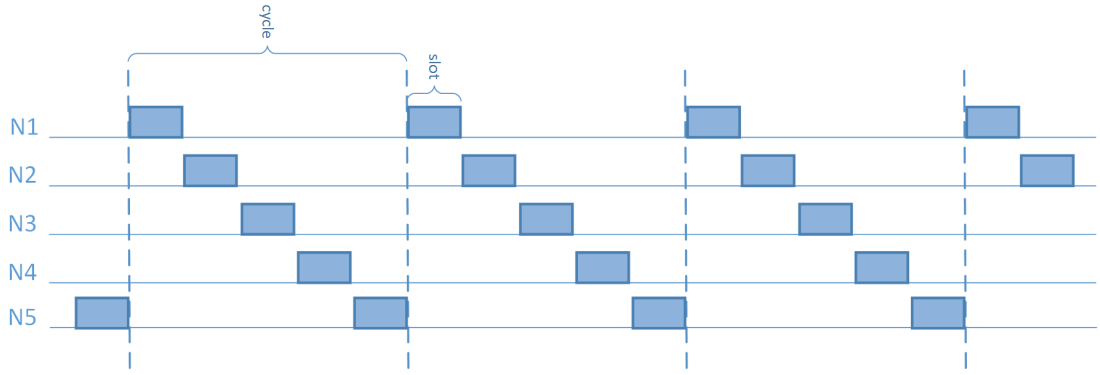


Figure 2.1: Time is devided into *slots* and *cycles*. A *cycle* is the time all the nodes need to send their message. A *cycle* is then divided by the number of nodes to create a *slot* for each node where it can send its message. [BKP]

For the collection of the RSS measurements an extra node that overhears all the messages is connected to a PC. The nodes include their ID and the last RSS measurements inside their messages. The extra node now receives the messages from all the other nodes and forwards them to the PC. [BKP]

This method is fast, efficient and stable. However it only works under the assumption that all the nodes can hear each other. When not all the nodes hear each other, the synchronization would not be that accurate, making it necessary to resynchronize for each *cycle*. Moreover the data collection does not work at all. There is no node that is able to hear all the other nodes and therefore collect the information by simply listening to the sent messages. Therefore a different method is needed for a widely spread multi-hop network.

2.4 Collection Tree Protocol

To collect data from a multi-hop WSN, multiple data collection protocols exist. One of these is the Collection Tree Protocol (CTP). The CTP creates paths in form of trees. The nodes can then send their data along this path to a collection point. To schedule the forwarding of messages, each node implements a message queue.

To create the paths, every node maintains the cost of its path to the collection point. The cost of a node's path is the cost of the link between itself and its parent added to the cost of its parent's path. Collection points have a path cost of zero. Now all the nodes send broadcast messages in a specific pattern which include the node's path costs. If a node receives a broadcast message that provides a smaller path cost, it can adjust its path to the new one.

Whenever a node now needs to send data, it can simply send it to its parent which will put the message inside its message queue and then forward it whenever it is possible. This makes it a good collection protocol for continuous data collection since nodes can send data whenever they have some, it recalibrates itself continuously and has high packet delivery rate. [GFJ+]

The problem with collecting the signal strength information for RTI is that it needs to be done at a specific point in time to not distort the measurements. This means a method is needed to start the collection and to know when it is finished.

2.5 Wireles Sonsor Network Testbed

A Wireless Sensor Network testbed is a WSN deployed in a controlled environment to test applications. This creates a realistic test environment showing not only the theoretical

functionality like a simulation but also makes it possible to examine communication loss, energy constraints and the influence of the real environment.

On the third floor of the SA building at the University of Duisburg-Essen such a WSN testbed is located. It is set up as a tool for researches on WSNs in an indoor environment. It covers half of the building including a large main corridor, two laboratories, two smaller corridors leading to three offices each, seven smaller storage rooms, an elevator and one server room. The arrangement of the rooms is laid out in Figure 2.2. All in all, the area covers 531 m^2. All the rooms are in daily use by the people working in the offices and the laboratories, keeping the area under constant change.

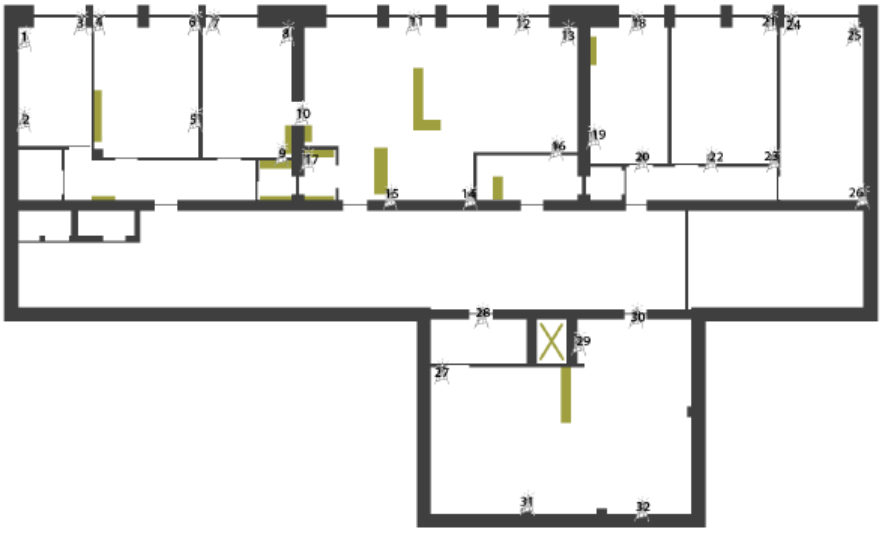


Figure 2.2: Floor plan of the area where the testbed is located. The position of the nodes is shown by the numbered antennas.

To monitor the area, 32 nodes are distributed over the rooms like shown in Figure 2.2. The nodes are only placed inside the offices and laboratories and are not always placed at the same height. To make programming of the devices easy, all the devices are connected to Raspberry Pis via USB. A script then makes it possible to copy the source code to all the Raspberry Pis where the code is compiled and then sent to each node individually. The connection to the Raspberry Pis makes it also possible to collect information directly from each node individually over serial forwarders running on the Raspberry Pis.

2.5.1 TelosB Mote

The sensor nodes used for the Testbed are Crossbow's TelosB Motes. The Crossbow TelosB Mote is an open source platform for researchers developed by the University

of California, Berkeley. It provides an 8 MHz Texas Instrument MSP430 low power microcontroller with 10kB RAM that is programmable via a USB connector. For communication it includes an IEEE 802.15.4 compliant radio frequency transceiver with an embedded antenna. This makes transmissions in a frequency band from 2.4 to 2.4835 GHz possible. Moreover the Crossbow TelosB Mote has a light sensor, an infra-red sensor, a humidity sensor and a temperature sensor installed making it possible to monitor the environment. Last, it has three LED lights installed that can be used for visual output of the mote. The USB connector that can be used to program the microcontroller can also be used to exchange data with and to power the Crossbow TelosB Mote. If it is not connected via USB it can also be powered by two AA batteries. [CT]

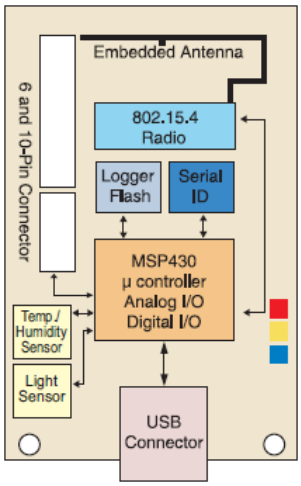


Figure 2.3: The structure of the Crossbow TelosB Mote and the included components. [CT]

2.5.2 TinyOS

TinyOS is an operating system for embedded systems, designed to manage the limited resources and power, and to provide reactive concurrency and flexibility. It is widely used by multiple research groups and companies worldwide to support their applications.

The main task of TinyOS is to schedule tasks and events to provide safe concurrent operations. Additionally it also provides a large amount of reusable system components, like timers, sender or receiver, that provide a huge variety of functionalities. When compiling an application, only the TinyOS components that are used by the application are included creating an application-specific OS.

To provide functionality, each component can have three different types of computational abstractions. First there are commands making it possible for other components to call

a functionality of the component providing the command. Then there are events which can be signaled by a component. A component receiving the event can then react to it and act accordingly. These two constructs make interaction between components possible. Last, for internal functionality of components there are tasks. These can be posted by a component and then are executed when it is their turn.

Whenever a command, event or task gets called it is pushed into the queue of the TinyOS scheduler. The scheduler then executes the tasks inside the queue using a FIFO scheduling policy. [LMP+]

2.5.2.1 Programming for TinyOS

Applications for TinyOS are written in nesC which is a C dialect and integrates the possibility to implement configurations, modules, interfaces, commands, events and tasks. An application consists of components, represented by a nesC module, that provide and use interfaces. The components implement the functionality of the application. All the commands and events a component provides are defined by an interface. Moreover an application for TinyOS also has a configuration that defines which components are used and describes their connections.

The Listings 2.1, 2.2 and 2.3 show an example application for TinyOS with two components of which one provides an interface and the configuration for the application. This example shows the connections between components, interfaces and the configuration and also shows the use of commands, events and tasks. In Listing 2.1 the two components *MainComponent* and *ExampleC* are represented. *MainComponent* uses two interfaces. The first one is the *Boot* interface. It requires the component to implement the *Boot.booted*() event. This event is the first event signaled by TinyOS after the node booted. Moreover the *MainComponent* uses the *ExampleI* interface that is defined in Listing 2.2. It requires the component to implement the *exampleDone*(*error_t error*) event. The *ExampleI* interface is provided by the *ExampleC* component and requires it to implement the *startExample*() command. Moreover the *ExampleC* component implements a task *doSomething*().

Now to connect everything, the configuration *Application* in Listing 2.3 first defines all the components needed for the application. The first component used in this application is the *MainC* component that provides the *Boot* interface and is given by TinyOS. Next, there are the two components from Listing 2.1. Now the configuration needs to connect the interfaces of the components to the components providing the corresponding interfaces. This means *MainComponent.Boot* gets connected to *MainC* and *MainComponent.ExampleI* gets connected to *ExampleC*.

When a device would get programmed with this example, the first thing that would happen is that *MainC* signals the *Boot.booted*() event resulting in executing the code

for the event inside *MainComponent*. This code calls the *ExampleI.startExample* command. Since the interface is connected to *ExampleC*, the corresponding code that posts the *doSomething*() task gets executed. Then at the end of *doSomething*(), *ExampleI.exampleDone*(...) gets signalled by *ExampleC*. This means the event gets triggered in *MainComponent* and the corresponding code gets executed. [LMP+] [Soma]

Listing 2.1: Two example components. *MainComponent* uses the example interface *ExampleI* from 2.2 and the interface *Boot* that TinyOS provides. *Boot* requires the *MainComponent* to implement the *Boot.booted*() event that is the first event that gets signaled by TinyOS after the node booted. *ExampleC* provides the example interface and therefore needs to implement the event and also implements a task *doSomething*().

```
module MainComponent {
  uses interface Boot;
  uses interface ExampleI;
} implementation {
  event void Boot.booted() {
    call ExampleI.startExample();
  }

  event void ExampleI.exampleDone(error_t error) {
    if(error == SUCCESS) {
      //some task...
    }
  }
}

module ExampleC {
  provides interface ExampleI;
} implementation {
  task void doSomething() {
    //some task...

    signal ExampleI.exampleDone(SUCCESS);
  }

  command void startExample() {
    post doSomething();
  }
}
```

Listing 2.2: Example interface providing a command *startExample*() and an event *exampleDone*().

```
interface ExampleI {
  command void startExample();
  event void exampleDone(error_t error);
}
```

Listing 2.3: The configuration *Application* links all the components.

```
configuration Application {
} implementation {
  components MainC;
  components MainComponent
  components ExampleC;

  MainComponent.Boot -> MainC;
  MainComponent.ExampleI -> ExampleC;
}
```

Whenever a task, command or event gets called it is not executed directly but gets added to the queue of the TinyOS scheduler. This means that all the tasks, commands or events called before execute first including the currently running one.

Chapter 3

Approach

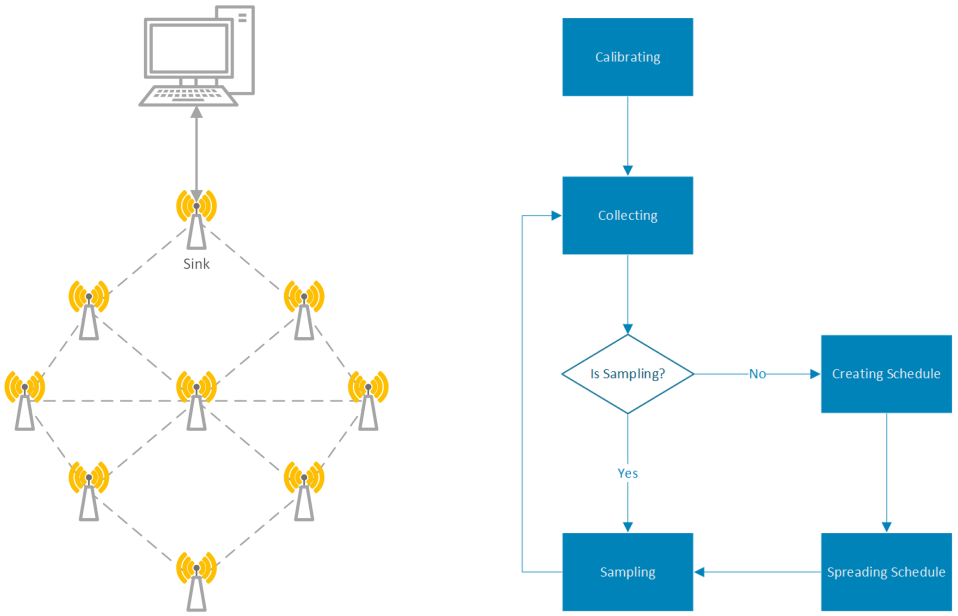
This chapter explains the general structure and functionality of a system that measures the received signal strength (RSS) of every link in a multi-hop Wireless Sensor Network (WSN). First, the chapter gives an overview of the general structure of the system. Then each individual task of the system is explained in detail.

3.1 General Structure

To measure the received signal strength (RSS) of each link, all the nodes need to send messages. Since two nodes sending a message at the same time distorts the RSS measurements, we need to make sure that only one node sends a message at a given point in time. One approach to achieve this is the multi-spin method that defines a time slot for each node according to its ID, so the nodes know when to send their message. However this method does not work inside a multi-hop WSN since it relies on the fact that every node can hear all the other nodes. This makes a different approach necessary that schedules the nodes and collects the data from the WSN. A similar approach would come to mind that also uses defined time slots but includes the collection and synchronization needed for it to work inside a multi-hop WSN. However, a problem with such a time slot based system is that when using a platform like TinyOS, where no full control over the execution of events and tasks is given, time slots cannot be defined accurately. This means, the time slots need to include an error that takes into account that a receive event does not necessarily gets triggered when the message was received.

To eliminate the delay time slots bring along, a different approach is suggested in this thesis which is based on predefined predecessors for each node. Based on this, each node is able to send a message directly after receiving a message from its predecessor still ensuring that only one node sends at a given point in time. Moreover each measurement *cycle* is divided into a sampling and a collection phase making it possible for the approach to be used inside a multi-hop WSN.

The approach requires a WSN in which one node functions as the sink. The sink is connected to a base station with high processing power and a lot of storage. The



(a) The physical architecture of the system.

(b) The general processes and the order in which they are executed.

Figure 3.1

structure of the WSN is represented in Figure 3.1a. Since the requirement for the approach is that it works inside a multi-hop WSN, it is a challenge to collect data from the network and to create a fitting schedule that defines predecessors for all the nodes inside the network. To be able to do so, a first calibration phase is needed to create paths from every node to the sink and collect information about the connections between nodes. Then the information about the connections needs to be collected at the sink and sent to the base station in a collection phase. When the base station has received all the information, it can create the schedule. After that the base station has to send the schedule to the sink that starts spreading it inside the network.

Is the schedule spread and all the nodes know when to send their messages, the sampling of the RSS can start. Therefore messages are sent according to the predecessors defined in the schedule. All the nodes receiving the messages are now able to measure the received signal strength to the sending node and store this information. When the schedule is completed, another collection of the data is needed to gather the measured RSS at the base station for further processing. When the collection is done, the system can start sampling again and then again collect the data until the system is stopped. This process is shown in Figure 3.1b.

3.2 Calibration

The calibration has two tasks. It needs to figure out for each node individually which nodes a node has in its range and it needs to create paths to the sink. These paths are used later to send data to the sink and to spread information inside the network. To achieve these tasks, each node broadcasts multiple messages without any specific pattern, meaning they do not need to take into account that messages can collide.

Each node needs to be able to keep track of the nodes in range. Therefore each node needs to have its own neighbour table. When a node receives a message, it can put the node it received the message from inside that neighbour table. All the neighbour tables will later be the basis to create the schedule.

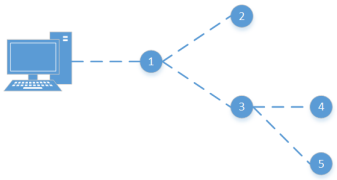
To later collect data from each individual node at the sink, each node needs to know to which node it needs to send its data to so it will reach the sink. This node is the parent of the node. Moreover to spread data inside the network, each node also needs to know which nodes send to itself in direction of the sink. These nodes are the children of a node. When each node has found its parent and its children, all the paths together form a tree structure with the sink as the root.

To find the parents for the nodes, each node needs to save one extra information and also include this information inside the messages it sends. This information is the quality of the current path a node has to the sink. At the beginning all the nodes except the sink initialize their path quality with the worst possible value. The sink initializes its path quality with the best possible value. Now when a node receives a message, it adds to the received path quality of the sending node the quality of the link between itself and the sending node. The result is the path quality to the sink for the node, when it chooses the sending node as its parent. The quality of a link is represented by the received signal strength. When the new path quality is calculated, the node compares the calculated value with its current path quality. If the calculated value is better, the node sets its parent to the sending node and saves the new path quality.

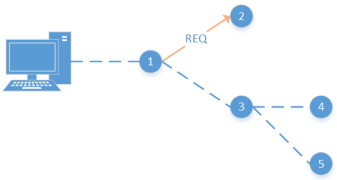
Now, to not only know the direction to the sink but also the dissemination from the sink to the whole network, the nodes need to find their children. Therefore each node simply includes their own parent inside the message it sends. A receiving node checks if it is the parent of the sending node. If that is the case, the receiving node can save the sending node as a child.

3.3 Collection

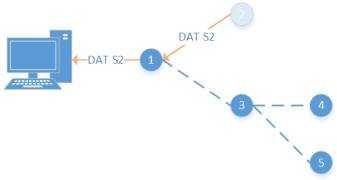
To collect the data from the network, we make use of the created paths and their tree structure. The process is shown in Figure 3.2. The Figure shows a wireless sensor



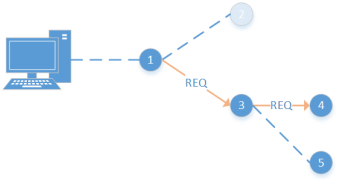
(a) At the beginning all the nodes are unmarked and still have data to send.



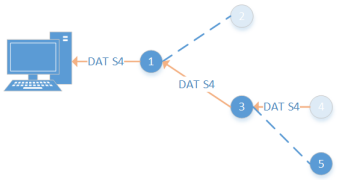
(b) Node 1 sends the first request to node 2.



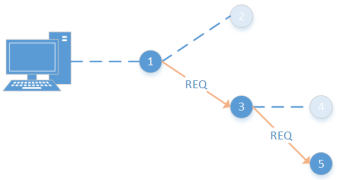
(c) Node 2 does not have any children, so it sends its data and gets marked.



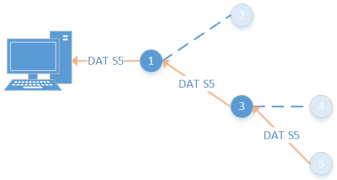
(d) Now node 2 is marked, so node 1 sends a request to node 3, which forwards it to node 4.



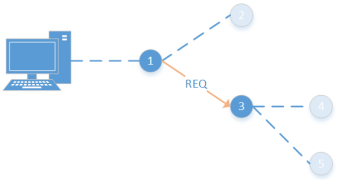
(e) Node 4 does not have any children, so it sends its data and gets marked.



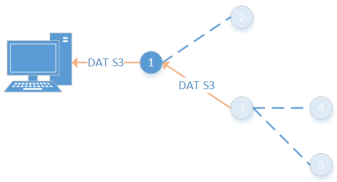
(f) Node 3 is still not marked, so node 1 sends a request to it. Node 4 is marked so node 3 forwards the request to node 5.



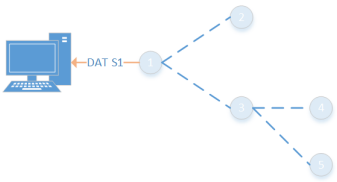
(g) Node 5 does not have any children, so it sends its data and gets marked.



(h) Node 3 is still not marked so node 1 sends a request to it



(i) All the children of node 3 are marked so it starts sending its data and gets marked itself



(j) Now all the children of node 1 are marked and it can send its own data to the base station and gets marked

Figure 3.2: This is an example for the collection. REQ means request and DAT Sx stands for data from source x

network represented by the nodes and the paths to the sink created in the calibration phase. Note that the nodes could also have other connections between each other.

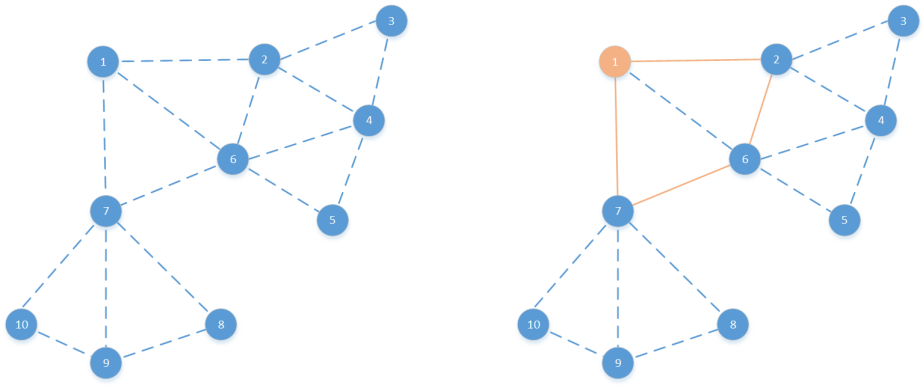
To start the collection the sink sends a request to one of its children. The child receiving that request checks if it has any children himself and if that is the case, it forwards the request to one of them. When the request reaches a node without any children, the node sends its data to its parent which forwards the data to its parent, until the data reaches the sink, which forwards it to the base station. Every node that receives a data message locally marks the source node of that data as done. When the sink has sent the received data to the base station, it sends a new request to one of its children that has not been marked as done. That child again forwards that request to a child that has not been marked as done. If the request reaches a node that has no children or every child has been marked as done, it sends its own data. This process will be repeated until the sink does not have any more children left that are not marked as done. Then the sink can send its own data to the base station and finish the collection. All nodes now need to unmark all the other nodes, so another collection is possible.

Since a node only sends its data when it receives a request created by the sink the sink is able to control when the collection starts and when it ends. This makes it possible to split the collection and the sampling, preventing distortion of the signal strength measurements. This would not be possible with a protocol like CTP that does not provide any control over the collection procedure.

3.4 Creating the Schedule

A fitting schedule is a path through the graph structure of the WSN that visits every node at least once and starts and ends at the same node. The schedule needs to form a circle because the sink needs to collect the data after each round. This means the sink needs to know when every node has send its message which is the case when it is the last node in the schedule. Since the sink also knows when the collection is done it makes sense that it starts the next round and therefore is also the first node of the schedule.

The optimal schedule would be a Hamilton-Circle, which is a circle inside a graph that visits each node exactly once. To figure out if a Hamilton-Circle exists, it is only possible to bruteforce all the possible combinations. This is highly inefficient and possibly we do not even get a result at the end. Therefore this thesis suggests a simple method that makes sure every node gets visited at least once and the path starts and ends at the same node. This method however is not able to create an optimal schedule.



(a) The example network with all its connections between nodes

(b) Example for a rooted circle with node 1 as the root

Figure 3.3: A rooted circle inside a example network

3.4.1 Rooted Circles

The suggested method is based on smaller circles inside the graph. These circles have a root node that is the start and the end point of the circle. All the nodes inside this circle are in range of the root of the circle. These circles will be called rooted circles. In Figure 3.3b such a rooted circle is represented inside the network depicted in Figure 3.3a. To create a rooted circle, one node needs to be chosen as the root. Then the base station takes one node from the roots neighbour table and choose it as the second node in the circle. Then the base station looks into the second nodes neighbour table and chooses a node that has the root node inside its neighbour table as the next node in the circle. The base station then does the same for the chosen node and this process is repeated until one node does not have a neighbour that has also the root as a neighbour. Then the circle is closed and the path goes to the root. When we look at the example in Figure 3.3b this means node 1 was chosen as the root. Then node 2 was picked as the second node in the circle. Node 2 has multiple neighbours but only one of them, node 6, has the root node 1 as a neighbour. This means node 6 is chosen as the next node in the circle. Node 6 now has node 2 and node 7 as neighbours that also have the root as a neighbour, however node 2 is already inside the circle so node 7 is chosen as the next node. Node 7 now has no more neighbours that have the root as a neighbour and the circle is closed.

3.4.2 Creating a Full Schedule

To create a full schedule the whole graph needs to be filled with rooted circles. Therefore the base station chooses the sink as the first root and creates a rooted circle around it.

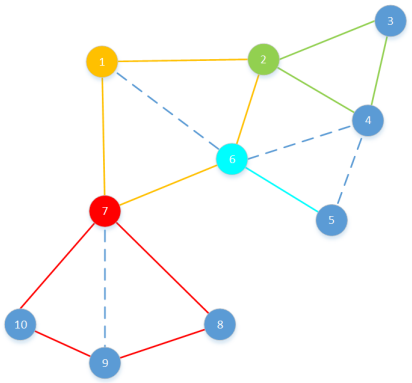


Figure 3.4: Full schedule for the example network of Figure 3.3a. The order in which the nodes send their messages would be: 1 2 3 4 2 6 5 6 7 8 9 10 7 1.

When the circle is done, the base station goes through all the nodes of the circle and, if possible, creates rooted circles around them as well. Then we do the same for all the new circles until every node was suggested as a root once. When creating new circles it is not possible to choose a node already inside another circle to be part of the new circle. In Figure 3.4 the network from Figure 3.3a is fully covered by rooted circles. The first circle created was the one that has node 1 as a root. Then all the nodes inside that circle were chosen as new roots to create new rooted circles. The first thing one could notice is that the circle with the root 6 only has one other node and does not really form a circle. This happens if the circle is closed directly after the second node following the root is chosen because there are no more neighbours left that also have the root as a neighbour. When running the schedule, this means a message would be sent from node 6 to node 5 and then from node 5 back to node 6. Also note that in theory there is a bigger rooted circle possible with the root 6 when node 4 would be included, but the rooted circle with the root 2 was created first and included node 4, blocking it for rooted circles created at a later point in time. Listing 3.1 provides the pseudocode of an algorithm to cover a whole graph with rooted circles.

Listing 3.1: Pseudocode that covers a graph with rooted circles

```
RootedCircle createRootedCircle(Node root) {
  Node node = getNextForCircle(root, root)

  If(node != null) {
    RootedCircle rootedCircle = new rootedCircle(root)
    rootedCircle.addNode(node)
    node.setPartOfCircle(true)

    while((node = getNextForCircle(root, node, rootedCircle)) != null) {
      rootedCircle.add(node)
      node.setPartOfCircle(true)
    }
```

```
    return rootedCircle
  } else {
    return null
  }
}

Node getNextForCircle(Node root, Node neighbour) {
  for each (Node node in root.getNeighbourList)
    if(node.isNeighbourOf(neighbour) and not node.isPartOfACircle())
      return node
  return null
}

List<RootedCircle> coverGraphWithRootedCircles(Node firstRoot) {
  List<RootedCircle> circleList = new List<RootedCircle()>

  circleList.add(createRootedCircle(firstRoot))

  for each (RootedCircle circle in circleList) {
    for each (Node node in circle) {
      RootedCircle newCircle = createRootedCircle(node)
      if(newCircle != null)
        circleList.add(newCircle);
    }
  }
}
```

3.5 Spreading the Schedule

To spread the schedule we are again making use of the tree structure of the created path. When the sink received the schedule from the base station it forwards it to its first child. The child forwards it to one of his children and so on. When a node that received the schedule has no more children, it sends the schedule back to its parent. The parent receiving the schedule now forwards the schedule to its next child. If a parent received the schedule back from all its children, it forwards it to its own parent. In Figure 3.5 an example for this process is given. The Figure shows a wireless sensor network represented by the nodes and the paths to the sink created in the calibration phase. Note that the nodes could also have other connections between each other.

When looking at the example, one can see that already in Figure 3.5g the schedule is received by all the nodes but still there are messages sent that could seem useless at this point. However at this point in time it is not known if the sink has any more children that did not receive the schedule yet. Therefore the schedule needs to travel all the way back to the sink. Only at the moment the schedule reaches the sink and the sink does

not have any more children left that need to receive the schedule, it is sure that the schedule is fully spread.

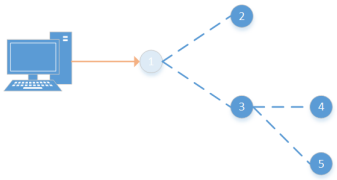
3.5.1 Sampling

When all nodes received the schedule it is possible to sample the received signal strength by sending messages according to it. Therefore the sink starts broadcasting a message. When its successor receives the message it can broadcast its own message and so on until the last message was send and the sampled data can be collected. However we need to take into account that we do not have a perfect schedule where every node only appears once, meaning a node could have multiple predecessors and successors. Therefore we need to include the successor of the sending node inside the message so a receiving node can see if he is the correct successor at that moment and evaluate the correct successor for itself.

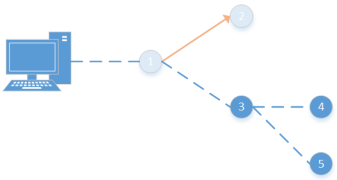
3.5.2 Message Drops

A problem of the proposed method are message drops. If a successor does not receive the message of its predecessor the whole system would stop. Here, a similar technique a timeslots based system like Multi-Spin uses comes in handy. For this method every node needs to know the whole schedule and not only its own predecessors and successors. Then whenever a node receives a message it can look up how many nodes need to send between the node that just send and itself. Than the amount of sending nodes is multiplied by the maximal time a node needs to send a message. The result is the time after the node can send its own message, without receiving a message from its predecessor. This procedure is shown in an example in Figure3.6. The shows a simple schedule. Node 3 sends a message but the successor node 4 does not receive it. However node 6 and 5 receive the message and can therefore calculate the times they need to wait until they can send their messages. However since node 4 has not received a message jet it is not able to send a message and the same applies to node 2.

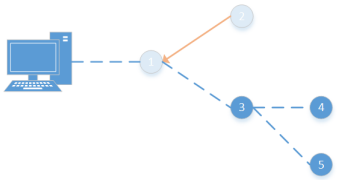
Again we need to take into account that a node can appear multiple times in the schedule. Therefore after a node has sent a message it needs to calculate the maximal time until it can send its next message.



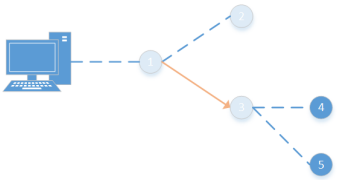
(a) The base station sends the schedule to node 1.



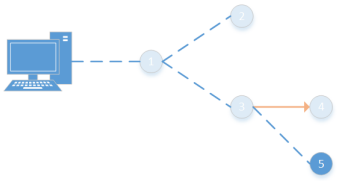
(b) Node 1 forwards the schedule to node 2



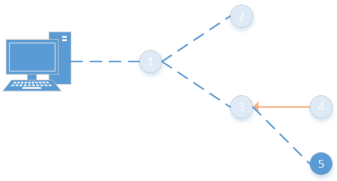
(c) Node 2 does not have any children and sends the schedule to its parent node 1



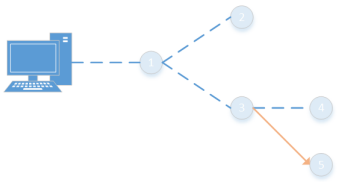
(d) Node 2 is now marked so node 1 forwards the schedule to node 3



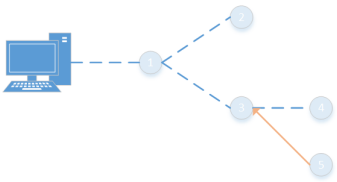
(e) Node 3 forwards the schedule to node 4



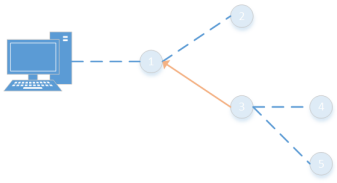
(f) Node 4 does not have any children and sends the schedule to its parent node 3



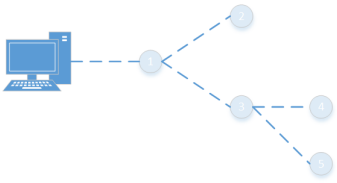
(g) Node 3 has still node 5 as an unmarked child and forwards the schedule to it



(h) Node 5 does not have any children and sends the schedule to its parent node 3



(i) Node 3 has no more unmarked children and sends the schedule to its parent node 1



(j) Since node 1 does not have any more unmarked children the schedule is now fully spread inside the network

Figure 3.5: This is an example for a path a schedule message takes to be spread inside the network

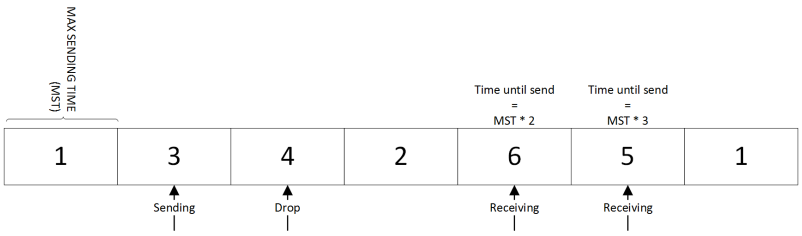


Figure 3.6: Representation of a schedule. Node 3 sends a message. Node 4 drops the message but node 6 and 5 receive it so that they can calculate the time when they should send

Chapter 4

Implementation

This chapter will explain one way to implement a system like the one described in Chapter 3 based on TinyOS. First it will give a general overview of the system's structure. Then the individual parts of the system are explained in more detail.

4.1 General Structure

The system is divided into two main components. The base station, which stores and processes the data, and the nodes, which collect the data. Both parts of the system need their own hardware and software to complete their tasks.

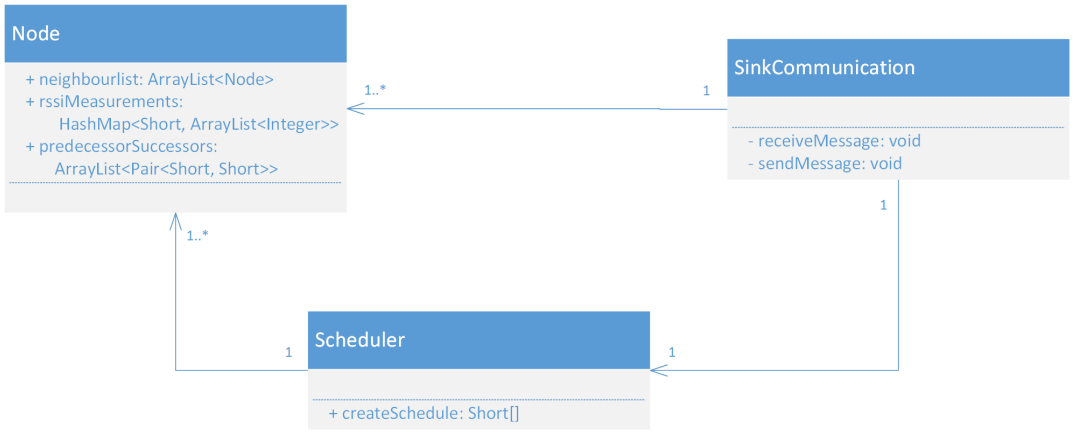


Figure 4.1: General structure and functionality of the base station

The base station can be a computer running a Java application connected to the sink via a USB-Cable. In Figure 4.1 you can see the general structure of the Java application with its classes and their basic functionality. First, there is the *SinkCommunication* class that handles the communication with the sink. It is responsible for receiving and processing messages and for sending messages to the sink. Then there is the *Node* class

which stores all the information about one node. It stores the neighbours of the node, the measurements a node made and its predecessors and successors in the schedule. The last class is the Scheduler that is responsible of creating a schedule based on the nodes data.

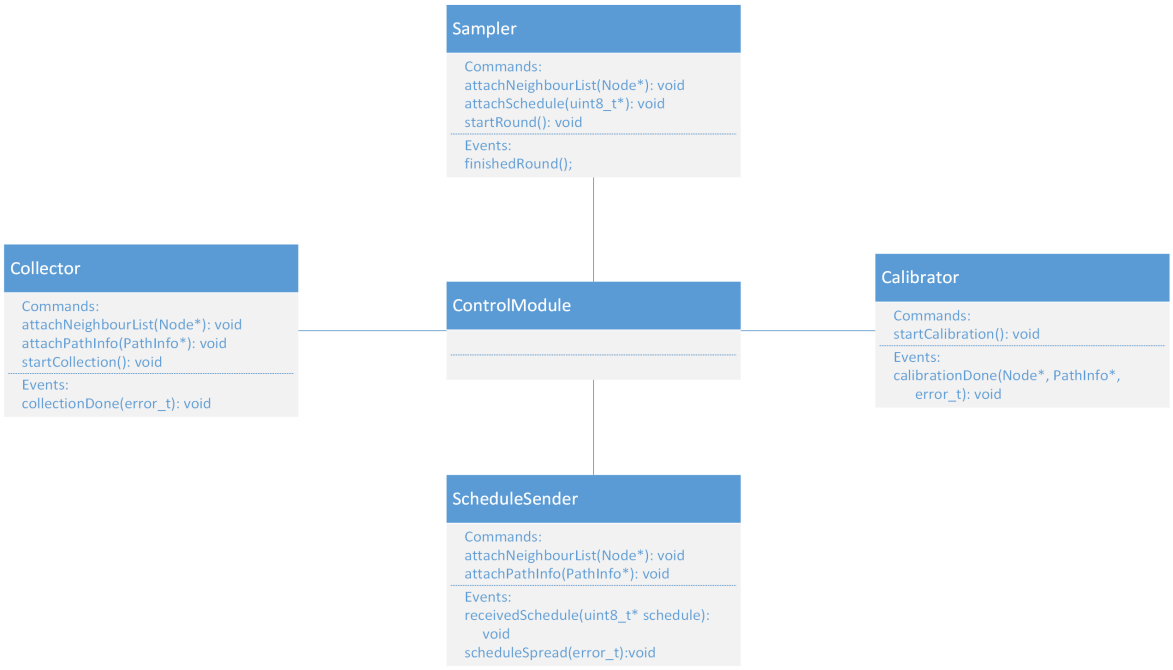


Figure 4.2: The interfaces each component provides. The *ControllModule* does only use all the other interfaces and does not provide one itself.

As nodes the TelosB motes described in Chapter 2.5.1 can be used. The application running on the nodes can then be written in NESC for TinyOS. Since the application running on the motes has multiple independent tasks it makes sense to create one component for each task. In Figure 4.2 the suggested structure is displayed, containing the components *Control*, *Calibrator*, *Sampler*, *ScheduleSender* and *Collector* represented by their provided interfaces.

The *Control* component connects all the other components and makes sure the informations the other components provide get delivered correctly to the component needing that information. Therefore it does not need an interface to provide functionality. The *Calibrator* components is responsible for the calibration of the network. It has the *startCalibration* command to start the calibration and the *calibrationDone* event that gets signalled when the calibration is done and provides the gathered informations. The *Collector* component covers the data collection. This component needs to know the neighbours and the information about the node's parent and children. Therefore

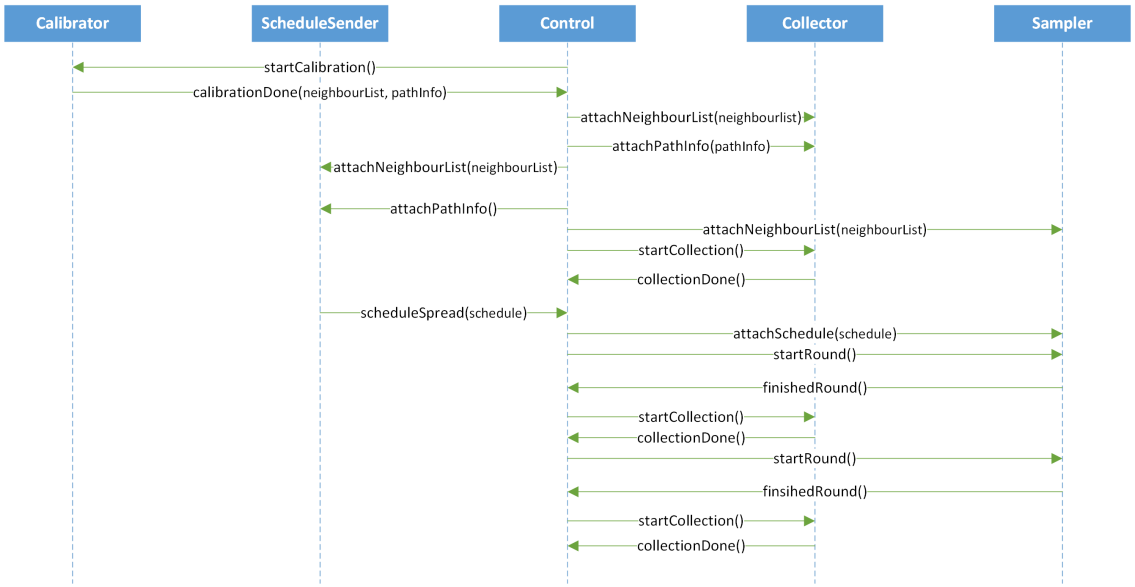


Figure 4.3: The order of commands and events of the sink from the start of the application until the sampling.

the interface of the components provides the two commands *attachNeighbourList* and *attachPathInfo* to attach this information to the component. The *Control* component can then simply attach the information to the *Collector* when the *calibrationDone* event triggers. Moreover, the *Collector* component needs the event *collectionDone* to signal when the collection is done. To spread the schedule inside the network the application provides the *ScheduleSender* component. It also needs the information about the neighbours and the parent and children of the node and therefore has the same commands as the *Collector* to attach these informations to it. It does not necessarily need a command to start the spreading since it can directly react when the node receives a schedule message from the base station. Lastly the component has two events *receivedSchedule* and *scheduleSpread*. The component signals the *receivedSchedule* event when the node received the schedule and provides the schedule. When the schedule is spread, the sink signals the *scheduleSpread* event. The last component is the *Sampler*. It needs the neighbour list and the schedule and therefore has the commands *attchNodeList* and *attachSchedule* to provide these to the component. Moreover it has the *startRound* command to start one round of the schedule and the *finishedRound* event that the component signals when this round is finished. Figure 4.3 represents a possible sequence of the commands and events.

ToDo: scheduleReceived event fehlt

4.2 Receiving and Storing Information at the Base Station

To receive and send messages from the sink it is possible to make use of the TinyOS Java libraries. These provide the necessary tools to connect the application with the sink via a serial forwarder. To synchronize the message types sent by the motes and the Java application, TinyOS provides the possibility to generate message classes directly from the nesC message structures. The Java application needs to know two different messages. One that contains the information collected by a node and one that contains the information about the schedule. The application now needs to implement message listener for both types of messages so it is able to receive and react to each message individually.

To store information about all the nodes the Java application needs to implement a hash map containing one *Node* object for every node inside the network with its ID as the key. This makes it easy to access the information of each individual *Node* object. To store the information about a node's neighbours, the *Node* class implements a list containing references to *Node* objects representing the neighbours of the node. To store the RSSI measurements of a node the *Node* class implements a hash map containing lists. Again the key values of the hash map are node IDs. The lists inside the hash map contain all the measurements from the node corresponding to the key.

Now whenever the Java application receives a message containing measurements of a node it gets the corresponding node object from the hash map containing all the nodes. Then it updates the neighbour list of the node and saves the measurements inside the measurement hash map of the node object.

4.3 Calibration

To calibrate the network by using the procedure explained in Chapter 3.2, the Calibration component needs to send messages without a specific pattern. These messages need to contain information about the path quality and the parent of the node. A possible message structure that fulfils these requirements is shown in Listing 4.1.

Listing 4.1: Message structure for the messages containing information about the parent and the path quality of the sending node.

```
nx_struct CalibrationMsg {
  uint8_t parent,
  uint8_t path_quality
}
```

To make sure that all the nodes send their messages roughly in the same time interval, a way to indicate when the calibration starts is needed. This can be achieved by one node

starting its calibration phase and the sending of messages. Then when another node receives a $CalibrationMsg$ and has not started its calibration it starts its calibration. Another problem is that if all the nodes send their messages directly one after another there would be a lot of messages in the network at the same time. To reduce this each node has a small delay between the messages it sends. To make this even more efficient each node could have a a different delay between its messages. This delay can be calculated as

$$\Delta p = T + \mathrm{ID}^2 \ \% \ N$$

where T is the minimal time between sends, ID is the node's ID, and N the number of nodes in the whole network. Now whenever a node sends a message it can start a timer that fires after Δp and then it can send the next message when the timer fires. Whenever a node sends a message it includes its stored parent and path quality.

To collect data from other nodes each node needs a neighbour list where it can store the nodes in range and the measured values. This neighbour list can be implemented by an array of the $Node$ struct shown in Listing 4.2 with the size N. The structure can store the last received signal strength, if the node is in range and the parent of the node. The other fields are used for the other phases.

Whenever a node receives a message it first sets the *inRange* field for that node to *true*, store the measured received signal strength in the *last_rssi* field and set the *parent* field to the *parent* field inside the message.

Listing 4.2: Structure to store information about nodes, containing the last RSSI measured, if the node is in range, the parent of the node, if the node received the data from node, if the node received the schedule from the node and the messages received from the node

```
typedef struct Node {
    int8_t last_rssi;
    int8_t inRange;
    uint8_t parent;
    int8_t receivedData;
    int8_t receivedSchedule;
    uint16_t msg_counter;
}
```

The next thing for the node is to evaluate if it should use the sending node as a parent. This is done like explained in Chapter 3.3. However, the link quality is not directly described by the RSS. Instead the RSS is mapped to a value where a high RSS represents a low value and a low RSS a high value. An example for the mapping of the RSS to the path quality is given in Table 4.1. The mapped value then represents the link quality, where a lower value indicates a better path quality.

This mapping is determined by the links inside the used WSN and needs to defined for each WSN individually. Meaning if there are only links with very low RSS values it makes no sense to start the mapping at a much higher RSS.

Table 4.1: Example mapping of RSS to a value representing the path quality

RSS	Quality
-50	1
-70	2
-80	7
-90	14

When the link quality is mapped it is added to the *path_quality* received in the message. Then the calculated value is compared to the own stored path quality of the receiving node. If the calculated value is smaller than the stored path quality the sending node is a more suitable parent and the stored parent and path quality is adjusted accordingly.

A node concludes its calibration phase and signals the *calibrationDone* event after sending a defined number of messages. This number however is also determined by the network and its complexity.

To make sure the sink does not start to collect data from the network before every other node finished its calibration it needs to wait after sending all its messages. When it finished waiting it can signal the *calibrationDone* event. The time the sink needs to wait depends on the number of nodes, the amount of messages send in the calibration and the calculation of the break between messages.

4.4 Collection

Before starting the collection, the Collection component needs the information gathered by the Calibration component. When the neighbour list and path info is attached to the component the sink can start the collection by sending a request message like the one in Listing 4.3 to one of its children. To find a child a node can go through its neighbour list and simply check if the *parent* field in the *Node* struct is set to the ID of the node. When a node receives a request message it searches for a child of itself and forwards the request like described in Chapter 3.3.

Listing 4.3: Structures representing a request message.

```
typedef nx_struct RequestMsg {
  nx_uint8_t id;
  nx_uint8_t round;
}
```

When the request reaches a node without any children the node fills its measurements into the *measurements* field of a data message like the one represented in Listing 4.4. Since it is possible that a node has more measurements than the maximal amount of data fitting into one message, we need the field that indicates how much data is left. If a node has more measurements it directly fills multiple messages and sends them one after another to its parent. The receiving node buffers all the data messages until it received the message where the $dataLeft$ field is set to zero. Then it forwards all the received messages one after another. Whenever a node received the whole dataset of a node it marks that node as done sending by simply setting the *receivedData* field for the node to true. Note that the receiving node does not necessarily mark the node it directly received the dataset from but the source of the dataset indicated by the $from$ field inside the data messages.

Listing 4.4: Structures to send the data from a node to the sink. The NodeMsg contains the measurements from one neighbour including the id of the neighbour and the measured RSSI. Inside the DataMsg the source of the measurements, the answered request id and how much data is left is stored.

```
typedef nx_struct Measurement {
  nx_uint8_t id;
  nx_int8_t rss;
}

typedef nx_struct DataMsg {
  nx_uint8_t from;
  nx_uint8_t request_id;
  nx_uint8_t dataLeft;
  Measurement measurements[MAX_MSG_PAYLOAD - 3];
}
```

A problem that needs to be addressed are message drops. Without detecting a message drop and resending the message afterwards, the collection simply stops if one message does not get received. To detect a message drop each node receiving a message directly sends an acknowledgement to the sending node. This is required for both the request and the data message and is done before any processing of the message. If the sending node did not get an acknowledgement for a message, it resends that message. Since not only the actual message can get lost but also the acknowledgement, it can happen that a message gets retransmitted one or multiple times although the message was already received and processed at the receiving node. The processing of the same message multiple times however can create unintended behaviour, resulting in the system to stop working and/or messages still being sent after the collection is already finished and the sampling already started thus distorting the RSSI measurements.

To prevent this, each request gets an ID from the sink in ascending order and the round of the collection. Whenever a node sent its data, it increases its saved round by one and set the saved request ID to zero. When a node receives a request, it checks if the round

of the request equals the saved round. If yes it checks if the id of the request is larger than its saved last request ID. If that is that the case it saves the received request ID and processes the request, otherwise it discards the requests. To catch already received data messages we include the request id the data was requested with inside the data message. When a node receives a data message it can then check if the data is corresponding to the request the node sent. If that is the case it also checks if the $dataLeft$ field in the message is smaller than the one in the data message previously received. If that is not the case, the data inside that message were already received and the message gets thrown away.

4.5 Creating the Schedule

The creation of the schedule happens at the base station when it received the data about the connections between nodes. The method presented in Listing 3.1 could be directly translated into the fitting Java code. To do so another class for the rooted circle needs to be added to the general structure of the application. When implementing the method it is important to take into account that links are not always bidirectional meaning when a node 1 is in the neighbour list of another node 2 this only means that node 2 can receive data from node 1, not necessarily that node 1 also receives messages from node 2. This means the neighbour list only contains the nodes a node can receive messages from, resulting in the fact that when we choose any a node from a nodes neighbour list the chosen node can never be a successor but only a predecessor of a node. This results in the schedule being created backwards.

Moreover it is a good idea to not simply pick a random node as a predecessor from the neighbour list of a node, but the one with the highest measured RSS. This reduces message losses while running the schedule, resulting in less time needed for one round.

To bring the schedule into a form that is readable for all the nodes, an array can be created that contains all the node IDs in the order they are expected to send messages. For the example in Figure 3.4 the array looks like this:

$$Short[]\ schedule = \{1, 2, 3, 4, 2, 6, 5, 6, 7, 8, 9, 10, 7, 1\}$$

4.6 Spreading the Schedule

Before the spreading of the schedule can start, the *ScheduleSender* component needs the information collected in the Calibration. If the information is attached to the ScheduleSender the spreading of the schedule automatically starts when the base station sends a schedule message like the one in Listing 4.5 to the sink. The message contains the schedule array created by the base station, then the information about how much of the

schedule is left and an ID to detect already received messages. When the sink receives the schedule message it forwards it like described in Chapter 3.5.

Listing 4.5: Schedule message structure.

```
typedef nx_struct ScheduleMsg {
  nx_uint8_t msgid;
  nx_uint8_t dataLeft;
  nx_uint8_t schedule[MAX_MSG_PAYLOAD-2];
}
```

Whenever a node receives a message, it first checks if it already received that part of the schedule on the basis of the *dataLeft* field. If it did not receive that schedule part yet it copies it to the right place in its own schedule array. Then it marks the node it got the message from and forwards it like described in Chapter 3.5. Last, it checks if the *dataLeft* field inside the just forwarded schedule message equals to zero. When that is the case the component signals the *receivedSchedule* event and provides the full schedule array to the *Control* component. To make it possible to receive multiple schedule parts it is important that whenever a node sends the schedule to its parent, it needs to unmark all its marked nodes, so if necessary it can later send the next schedule part to them.

If the schedule message reaches the sink and the sink does not have any more unmarked children it forwards the schedule to the base station and unmarks all its marked nodes. This way the base station knows that either the schedule is spread or it can send the next schedule part that then is spread the same way as the first one. If the sink sent the last part of the schedule to the base station the sink signals the *scheduleSpread* event to let the *Control* component know that every node knows the schedule and the sampling can be started.

Again we need to make sure that no messages get lost, so in this phase as well each message is directly replied by an acknowledgement message. If the sending node did not get the acknowledgement it resends its message. To check if a message was already received we have the *msgid* field in the message. Whenever a node forwards the schedule it increases this field and save the value. Then when a node receives a message it checks if the *msgid* is larger as the saved one. If that is the case it is a new message and needs to be processed. Otherwise the message gets thrown away.

4.7 Sampling

When the schedule is spread and the sample component knows about the schedule and the neighbour list, the sampling can start. The sink starts by broadcasting a message like the one shown in Listing 4.6. The *receiver* field is set to the node ID of the successor of the sink. Then all the nodes receiving the message measure the received signal strength

and save it inside their neighbour lists. Then each node that received the message compares their own id with the id saved in the *receiver* field of the message. If they are equal the node evaluates the fitting successor, save it inside a new message and then also broadcasts it. When the sink receives a message and does not have another successor it signals the $finishedRound$ event.

To evaluate the fitting successor the approach coming in mind would be to just go through the schedule array and search for the sending node followed by the searching nodes id. The successor then would be the node after that. Since this would require a loop it is inefficient and unnecessarily stretches the timeslot to catch message drops. Therefore it is suggested to evaluate all the predecessors and successors of a node before the sampling starts by going through the whole schedule once and save them, in the order of appearance, in a structure like the one shown in Listing 4.7. Additionally each node stores the index of the next predecessor successor pair. Then when a node is the successor it can simply check if the predecessor at the saved index is correct and if that is the case send the new message storing corresponding saved successor and calculate the next index as

$$I + 1 = (I + 1) \ \% \ \text{numberOfPS}$$

It can happen that the predecessor at the saved index is corresponding to the sending node. This can happen when a message drops and the node gets skipped. In that case we have to go through all the predecessor successor pairs to find the right successor and adjust the saved index accordingly. However this is still more efficient than going through the whole schedule.

Listing 4.6: Sample message structure

```
typedef nx_struct SampleMsg {
  nx_uint8_t receiver;
}
```

Listing 4.7: Structures to save the successors and predecessors

```
typedef struct PredecessorSuccessor {
  uint8_t predecessor;
  uint8_t successor;
}

typedef struct SchedulePSCollection {
  uint8_t numberOfPS;
  PredecessorSuccessor preSuc[MAX_PREDECESSORS_SUCCESSORS];
}
```

To catch the message losses like explained in Chapter 3.5.2 every time a node receives a message and is not the successor it needs to run through the schedule array and calculate how many nodes sent between the sending node and the receiving node and calculate the time it needs for the schedule to reach the node. Then the node starts a timer that

fires after exactly that time. Moreover the node needs to save the successor in case of the loss. The timer is reset to the new time whenever the node receives a message. When the timer fires the node simply sends a message with the *receiver* set as the saved successor. Since a node can appear multiple times inside the schedule it needs to check if it needs to send another message at a later. To do so it can pretend it received its own message and calculate the time to its next send like for any other message.

Chapter 5

Evaluation

5.1 Experiment Setup

To test the suggested approach for a RSS measurement system described in Chapter 3 the approach was implemented like described in Chapter 4. The system was then deployed and tested inside the Testbed described in Chapter 2.5. Two experiments where run. The first one was at night where no one was inside the building where the Testbed is located. The second one was at daytime when there were people working inside the offices covered by the Testbed and also in the other parts of the building.

The implemented system required significant efforts in testing and debugging. Unfortunately, some bug still manifested during the execution of the collection created by message drops. This bug immediately brings the system to a stop due to discarding messages that need to be processed. Since the system directly stops when the bug occurs it does not effect previous processes making the in the experiments measured times also valid for a fully working system. However this made it necessary to start the system multiple times for each experiment to get as many values as possible. In the experiments the application was restarted after half an hour to catch the stops.

To calibrate the system 700 messages were sent with a minimum break of 20 ms. After the sink send all its messages it takes a 50 second break to make sure every node send its message. The timeslot for the drops was defined as 18 ms. This is really big but when running the system inside a simulation it could be measured that the processing and sending of a message took between 5 ms and 16 ms the extra 2 ms cover a possible error while measuring these values.

5.2 Experiment Night

The first experiment took place at night between 02:14h and 05:54h. At this time no one was present at the whole floor. In this time system was restarted 7 times. Between 03:49 and 04:51 the application actually ran the whole 30 minutes and was not stopped

by the mentioned bug but by the next restart. In all the other cases the application stopped during a collection. In Table 5.1 the times for each start of the application is shown as well as the over all average times for each phase.

Table 5.1: Measured values for each run of the system. Time = Actual timespan in which the system took measurements; CT = Calibration Times [ms]; HOP = Number of hops inside the schedule; ST = Spread time [ms]; ACT = Average collection time [ms]; CSTD = Standard deviation of collection time [ms]; ART = Average round time [ms]; RSTD = Standard deviation of round time [ms];

Time	CT	HOP	ST	ACT	CSTD	ART	RSTD	Rounds
02:14 - 02:38	65611	44	784	3592	267	540	55	338
02:45 - 02:51	64545	42	842	3637	280	519	38	73
03:17 - 03:19	64630	42	767	4211	444	502	32	13
03:49 - 04:19	64539	41	740	3609	300	501	62	420
04:21 - 04:51	64565	44	804	3581	245	544	51	419
04:53 - 04:57	64699	40	756	4287	343	429	48	35
05:24 - 05:26	64587	44	787	4150	343	496	23	9
Average	64739	42	782	3625	276	524	54	

The average calibration time was 64739 ms. The time of all the different calibrations did not really change since the exact amount of messages and the break between messages is defined. It takes quite long since a huge amount of messages are sent by each node with a break up to 32 ms. Also the break the sink takes after it send its message was not calculated perfectly making it possible to reduce it, resulting in a shorter calibration. When the calibration phase took roughly 64 seconds and the sink took a 50 second break after finishing sending its messages, it means the sink was finished sending its messages after 14 seconds. Now with the defined break for all the nodes in Chapter 4.3 we can calculate the maximum time a node needs to send all its messages. The result would be that maximal time a node needs to send all its messages is about 40 seconds. This means the break of the sink can be reduce to around to 25 seconds reducing the over all calibration time by 25 seconds.

The average schedule has 42 hops meaning that 42 messages need to be sent in one round the schedule takes. The schedule with the largest amount of hops has 44 and the smallest has 40 hops. The different schedules emerge through different measured RSS values while calibrating.

Spreading the schedule took in average 782 ms. This is fast since one schedule of 40+ hops needs two messages. This means that to spread one schedule part inside the whole network took around 370 ms.

The collection of the gathered information took averagely 3625 ms with an average standard deviation of 276 ms. Compared with the spreading of the schedule this is

really slow. However we need to take into account that the collection requires more messages and processing than the spreading. To analyse if the time of the collection changes during the curse of time and if a new calibration is needed, Figure 5.1 shows the times of all the collections between 03:49h and 04:19h. However the Figure does not show any significant change over the 30 minutes of time indicating that there would be no new calibration necessary to maintain performance of the application.

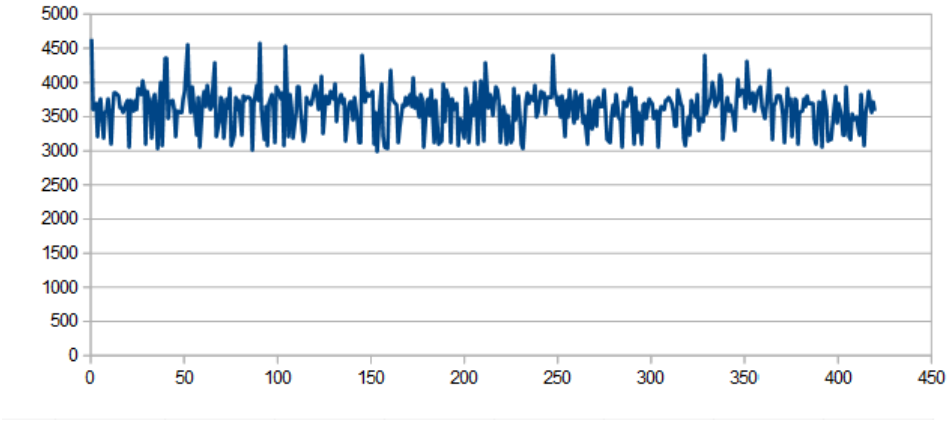


Figure 5.1: Collection times of the measurements from 03:49h to 04:19h

Sending all the messages for one round takes in average 524 ms with an average standard deviation of 54 ms. Again we want to analyse if the round time changes during the curse of time. Therefore the round times of all the rounds between 03:49h and 04:19h are shown in Figure 5.2. Again we do not see a significant change in the time one round takes. This means that message drops do not increase over the 30 minutes.

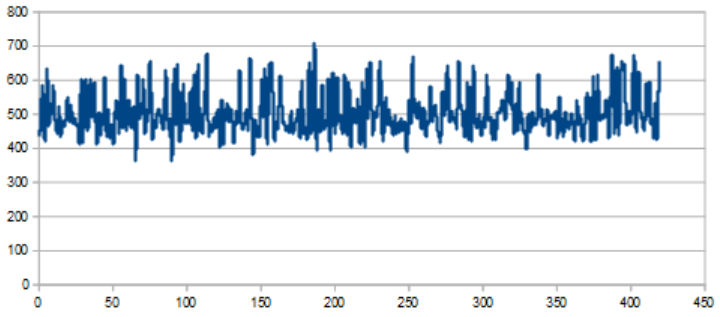


Figure 5.2: Sampling times of the measurements from 03:49h to 04:19h

5.3 Experiment Day

The second experiment took place from 09:32h to 14:17h. In this timespan the application was started and restarted 9 times. The first person arrived at 09:00h, the second at 10:00h, the third at 11:20h, the forth 13:00h and the last one at 13:30. Moreover one person arrived and left somewhere between 10:30h and 12:30h. Moreover one left at 12:00h. All these people where working in their offices. The results of the experiment are shown in Table 5.2

Table 5.2: Measured values for each run of the system. Time = Actual timespan in which the system took measurements; CT = Calibration Times [ms]; HOP = Number of hops inside the schedule; ST = Spread time [ms]; ACT = Average collection time [ms]; CSTD = Standard deviation of collection time [ms]; ART = Average round time [ms]; RSTD = Standard deviation of round time [ms];

Time	Persons	CT	HOP	ST	ACT	CSTD	ART	RSTD	Rounds
09:32 - 09:38	1	64647	43	823	4401	296	492	41	63
10:04 - 10:14	2	64558	44	796	4035	322	563	59	121
10:36 - 10:53	2	64644	41	850	3740	282	482	40	231
11:07 - 11:09	2	64577	41	849	3746	423	533	60	14
11:39 - 12:09	3-2	64607	39	797	3498	358	495	42	433
12:11 - 12:28	2	64654	41	764	3556	294	495	45	240
12:43 - 13:13	2-3	64583	44	824	4160	511	539	65	368
13:15 - 13:29	3	64544	44	797	3697	398	542	58	183
13:47 - 13:53	4	64580	41	778	3494	371	528	46	73
Average		64599	42	809	3773	372	514	50	

Since the mean values of both experiments fall in each others standard deviation it seems that typical activities in an in an indoor environment do not impact the functioning of the approach. Again we look at the individual times of the collection from 11:39h to 12:09h in Figure 5.3 to see if they change over time. In general the time for a collection does not increase over time, however at around collection 400 we can see a huge deviation. Here the collection once takes 5186 ms and once 7293 ms. After that the collection time normalizes again. This deviation could indicate that at that point in time a, for the links inside the Tesbed, huge change in the environment caused a lot of message drops creating this long collection. Most likely this change was created by the person leaving at around 12:00h. It left the floor by using the elevator which has a huge impact on the links in the testbed.

In Figure 5.4 we take a look at the individual times of the rounds between 11:39h and 12:09h. Here we can see slight fluctuations of the time. However we can not say if these are created by decalibration or higher activity by the persons inside the offices. For

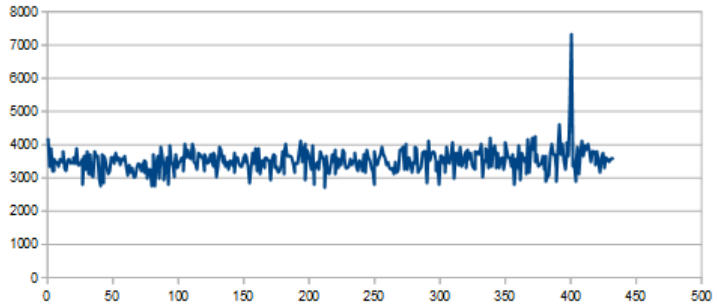


Figure 5.3: Collection times of the measurements from 11:39h to 12:09h

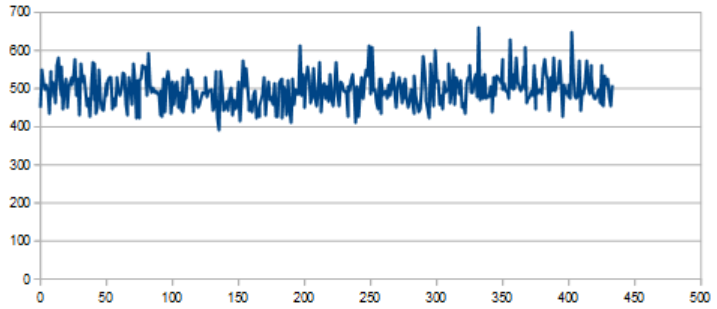


Figure 5.4: Sampling times of the measurements from 11:39h to 12:09h

example increase of time starting at round 300 can also be created by the person leaving since it started walking around.

5.4 Usability for RTI localisation

The experiments show that the system is usable in an office environment and delivers one full RSS measurement averagely every 4 seconds. The measurement are fast enough to detect a moving person. The long collection phase creates jumps of 3.5 seconds between each measurement. In that time a person can move a lot and could make the tracking of the person difficult. However it could be possible to have multiple measurement rounds before collecting the data. This could improve the tracking.

5.5 Comparison to a Timeslot Based Approach

The approach suggested by this thesis tries to improve a fully timeslot based approach like Multi-Spin by defining predecessors and successors for each node to eliminate the in a error included in a timeslot. These two approaches can be compared by calculating the time a round would need. Therefore we assume that a timeslot based system would choose the same timeslot size as the predecessor successor approach to catch message loss.

In the experiments a timeslot of 18 ms was used to catch message loss. There are 32 nodes inside the network meaning a timeslot based approach would need 576 ms for one round. The average time a round needed in the experiments was around 520 ms which is actually faster although there are in average 10 messages more that need to be send with the successor predecessor. This advantage could be improved even more by optimising the schedule, shrinking the amount of messages that need to be send in one round.

An advantage of the timeslot based approach would be that rounds have a very stable time. The average round time in the experiments is indeed faster than the time of a timeslot based system, however the round time fluctuates meaning that the times range from around 370 ms to around 640 ms. Depending on the purpose of application a stable time could be preferred over the faster average time. Moreover in a timeslot based system it is not possible that one node does not send a message while in the successor predecessor approach it can happen that a node does not send a message when it never received a message until it is its turn. However this should happen rarely.

Chapter 6

Discussion

6.1 Possible Improvements

The first improvement for the calibration was already explained. Here time can be saved by simply calculating the correct break and not having the system do nothing for 25 seconds.

The next improvement is for the collection. When using nodes with a lot memory it could be a good idea to not always send the data from one node directly to the sink but first collect all the data of a nodes children before forwarding it. This way the requests and data messages would take the same path as the schedule while spreading it. This would reduce the amount of requests and the overall collection time.

Another improvement can be done to the algorithm that creates the schedule. Instead of just covering the network with rooted circles it is possible to later connect circles in a way that reduces the amount of messages. Also to improve choosing a next node it would be a good idea to change the calibration in a way that not only the last measured RSS for each link gets saved but a average RSS for a link. A different way to maybe improve the schedule is to simply select the same path a schedule message would take while spreading it. The experiments already show that spreading the schedule is faster than the rounds with the created schedule.

6.2 Conclusion

Because the existing approach for signal strength measurement does not work inside a multi-hop wireless sensor network a new approach was developed that works for multi-hop WSN. The new approach consists of different phases. First there is a calibration phase where informations about the network are collected. Moreover paths through the WSN to collect information at a central point and to spread information inside the network are created. Then in the next phase a schedule that defines predecessors and successors for each node is created and spread inside the network. Last, the measurement

of the signal strength. Therefore each node will send messages according to the schedule, meaning one node starts sending a message. All the nodes hearing that message will measure the signal strength and stores it. When a node hears a message from its predecessor it can send its own message. This goes on until every node send a message. Then the stored measurements of the nodes are collected at the central point and a new measurement round can begin enabling the possibility to monitor the signal strength of each link over a period of time.

This approach was implemented and tested in a WSN testbed located at the University Duisburg-Essen. The experiments show that for a WSN with 32 widely spread nodes one measurement round takes about 500 ms followed by a 3,5 second collection phase. This means the system can deliver one full set of measurement every 4 seconds by taking multiple rounds of measurements one after another. This is sufficient to fulfil the requirements of measurements necessary to compute the RTI with a person walking in the environment.

Chapter 7

Acknowledgements

(This part is optional, and it could be completely excluded by deleting
\include {content/chapters/chapter7}
from the Firstname_Lastname_Diplom_Master_arbeit.tex file)

This paragraph could mention people or institutions that supported you to some extent with your work or friends and relatives that supported you during your study period.

Bibliography

[BKP] Bocca, Maurizio, Ossi Kaltiokallio and Neal Patwari: *Radio Tomographic Imaging for Ambient Assisted Living.*

[CT] Crossbow Technology, Inc.: *TelosB Mote Platform Datasheet.*

[GFJ+] Gnawali, Omprakash, Rodrigo Fonseca, Kyle Jamieson, David Moss and Philip Levis: *Collection Tree Protocol.*

[LMP+] Levis, P., S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer and D. Culler: *TinyOS: An Operating System for Sensor Networks.*

[Soma] Someone: *http://tinyos.stanford.edu/tinyos-wiki/index.php/TinyOS_Documentation_Wiki.*

[Somb] Someone: *WSN book.*

Erklärung

German

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

English

I hereby declare that I have written this Bachelor thesis independently, using no other than the specified sources and resources, and that all quotations have been indicated.

Essen, February 9, 2017
(Place, Date)

Jan Nauber