Dynamic Energy Harvest Aware Routing on Sun SPOT sensor nodes

The projekt is composed by:

stud.polyt. Kristian Hede, s062378

Supervisors:

Professor Jan Madsen

PhD. Student Mikkel Koefoed Jakobsen

Deadline: June 19, 2009 (printed June 19, 2009)

Handed in: June 19, 2009

This report and enclosed appendix is handed in as a part of a bachelor thesis at the Danish Technical University. The project represents 20 ECTS point.

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +4545882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary

A protocol called Energy Aware Routing Protocol (EARP) has been developed at DTU in an attempt to prolong the lifetime of a wireless sensor network, where the nodes of the network has the ability to collect environmental energy through a solar collector, in order to recharge the battery of the node. EARP has been tested through simulators and is now given to me to implement on a small imbedded device called a Sun SPOT (Sun Small Programmable Object Technology). The protocol is to run on a WSN consisting of a number of Sun SPOTs. A major part of this report deals with considering the way the nodes should communicate with each other and how the wireless sensor network should be initialized and maintained. The application created to run on the Sun SPOTs to do a practical implementation of EARP works at a basic level. The Energy Aware Routing Protocol works as intended, but certain technical problems with the Sun SPOTs were never completely solved in the final application to run on the Sun SPOTs. But the application does work well enough to demonstrate EARP.

Resumé

En protocol som hedder Energy Aware Routing Protocol (EARP) er blevet udviklet på DTU i et forsøg på at forlænge levetiden i et trådløst sensornetværk, hvor knudepunkterne i netværket har evnen til at indsamle energi fra sine omgivelse vha. En solfanger,med hensigt på at oplade knudepunktets batteri. EARP er blevet testet gennem simulatorer og er nu blevet givet til mig for at jeg skal implementere det i et lille indlejret stykke elektronik kaldet en Sun SPOT (Sun Small Programmable Object Technology). Protokollen skal køre på et trådløst sensornetværk bestående af et antal Sun SPOTs. En stor del af denne rapport håndterer overvejelserne der er gjort, mht. hvordan knudepunkterne skal kommunikere med hinanden og hvordan det tådløse sensornetværk skal initialiseres og vedligeholdes. Applikationen der er skabt til at køre på disse Sun SPOTs, for at implementere EARP i praksis, virker på et grundlæggende niveau. Protokollen virker som hensigten dikterer, men nogle tekniske problemer med disse Sun SPOTs nåede ikke at blive løst i den endelige version af applikationen der skal køre på disse Sun SPOTs. Men applikationen virker nok til at kunne demonstrere EARP.

Preface

This project was completed at DTU Informatics, the Technical University of Denmark from February through June 2009. The project qualifies for a degree in Bachelor of Science in Software Engineering.

The report deals with the practical implementation of a protocol called Energy Aware Routing Protocol (EARP) on a small imbedded device called a Sun SPOT (Sun Small Programmable Object Technology), and how to have these Sun SPOTs communicate with each other during and after the initialization of a wireless sensor network consisting of a number of Sun SPOTs. If nothing else is noted, all figures, pictures, tables and code in the report are developed by me. The developed software is available on the enclosed CD.

I would like to acknowledge and thank my supervisors Jan Madsen and Mikkel Koefoed Jakobsen for ongoing advice and discussions throughout the project.

Lyngby, June 2009

Kristian Hede

Contents

Summary iii

Resumé v

Preface vii

1 Introduction 1
1.1 Thesis Statement . . . 1

2 Sun SPOT 3
2.1 The Radio . . . 3
2.2 Three sensors, eight LEDs and two buttons . . . 4
2.3 The Basestation . . . 4
2.4 Other specifications . . . 5
2.5 Limitations . . . 5
2.6 Development Environment . . . 6

3 Energy Aware Routing Protocol (EARP) 7
3.1 The Basic Idea of EARP . . . 7
3.2 Why EARP . . . 8
3.3 How is EARP implemented . . . 9
3.4 Demonstration . . . 10

4 Wireless Sensor Network Setup 13
4.1 Initialization . . . 14
4.2 General Setup . . . 16
4.2.1 Aloha . . . 17
4.2.2 Ethernet . . . 18
4.2.3 TCP . . . 18
4.2.4 Port multiplexing . . . 19
4.2.5 Time division multiplexing . . . 20
4.2.6 Port multiplexing - Controlled pattern . . . 20
4.2.7 Listen before send with multiple ports (ports as a semaphore) . . . 21
4.2.8 Radiogram Protocol Point-to-Point Connection . . . 23
4.3 The Selected Setup . . . 24
4.4 Maintenance . . . 25
4.4.1 Inserting a node . . . 26
4.4.2 Removing a node . . . 26

5 Program Structure **29**
5.1 UML Diagram . . . 29
5.2 Important Methods . . . 30
5.2.1 interpretPackage() . . . 30
5.2.2 relayData() . . . 30
5.3 The Desktop Application . . . 31

6 Results **33**
6.1 Chinese Delegation Demonstration . . . 33
6.2 Initialization . . . 33
6.3 Maintenance . . . 35
6.4 User's Manual . . . 36

7 Discussion **39**
7.1 Scalability . . . 39
7.2 Maintenance . . . 40
7.3 The Process . . . 40
7.3.1 Debugging . . . 41
7.3.2 System Model as starting point . . . 41

8 Conclusion **43**
8.1 Known errors . . . 44
8.2 Further Work . . . 44

A Output from video [1] **47**

B Output from video [2] **49**

List of Tables

2.1 List of lumiance values 4
2.2 Battery Lifetime, the table is from [3, p. 27] 5

3.1 Routes of packages from video 12

6.1 Data from 3C12 35
6.2 Data from 44FC 35
6.3 Data from 3C6F 36
6.4 Data from 44FC 36
6.5 Trying to route through a removed node 36
6.6 Data from 3C6F 37
6.7 Data from 44FC 37

List of Figures

2.1 Two Sun SPOTs and some coins to show the size of the Sun SPOTs . . . 3

3.1 A circular wireless sensor network . . . 7
3.2 Node heights . . . 10
3.3 How to read the LEDs on the Sun SPOT . . . 11
3.4 A WSN. The last four characters of the Sun SPOTs IEEE address is shown. The first twelve are the same on them all (0014.4F01.0000) . . . 12

4.1 Sun SPOTs with hardcoded neighbors . . . 14
4.2 The left node A can talk to the right node B, but not the other way around 14
4.3 Pure Aloha, picture taken from [4] . . . 17
4.4 Each time slot is reserved for a node . . . 20
4.5 'Clusters' of seven nodes . . . 21
4.6 Top node C receives two messages on port 10 . . . 22

5.1 UML diagram of the Sun SPOT application . . . 29

6.1 A WSN of 3 SPOTs . . . 34

Chapter 1

Introduction

This is a bachelor thesis dealing with a wireless sensor network (WSN) and the practical implementation of a given protocol. In the WSN data has to be relayed from a node through the network to a sink and the nodes of the network do not have access to a power outlet - this causes them to depend on the limited lifetime of a battery to stay functional. If you have a WSN consisting of hundreds, or maybe thousands of nodes, you want to avoid having to go around recharging the nodes manually. It is possible to have the nodes recharge themselves, by giving them the ability to harvest energy from their surroundings. This can be achieved through various means, but it is not the method of harvesting environmental energy, which is relevant in this thesis - it is the effect of this harvesting, and how to optimize this effect for the entire network - which is another crucial part of this thesis. The effectiveness of the approach which is chosen to optimize this effect, is measured in the number of nodes which have exhausted their power supply completely, and is no longer able to relay or collect data while the WSN is still operating. Due to the fact that the radio is the greatest consumer of power - it is the routing of data throughout the WSN, which I have been given an algorithm called EARP [5] (Energy Aware Routing Protocol) to implement on a small imbedded device developed by Sun called Sun SPOT [6] (Sun Small Programmable Object Technology). The Sun SPOTs are to be used as nodes in the WSN.

1.1 Thesis Statement

For this thesis I have been provided with a number of Sun SPOTs, which are to represent the nodes of a WSN. The point of the WSN is to route packages from any source to a sink in the network. To this end I have been given an algorithm known as EARP. The EARP is in charge of deciding which route to choose when sending packages through the network. EARP has been tested in a simulated environment, and the results concluded that EARP is more efficient than simple directed diffusion [5]. The nodes in the simulated environment harvested energy through a solar collector. The simulation could generate the effect of influential factors overshadowing particular parts of the WSN, like clouds or trees. The result of this was that nodes in shadow had a greater risk of running low on power than nodes in the sun. It is my job to implement EARP on the Sun SPOT, in order to make the jump from simulation to physical reality. Knowing that making this jump almost always reveals unforeseen problems and challenges, the assignment was made susceptible to add-ons discovered during the implementation process. The basic guidelines of the assignment were the following:

- A virtual battery should be created and used instead of using the actual battery of the Sun SPOT. Using the actual battery would require me to add a solar collector of some sort to the Sun SPOT.
- A Sun SPOT contains a light sensor, which I should use to read the actual light intensity, to calculate the rate at which the virtual battery should recharge.
- The routing algorithm EARP should be implemented in the WSN where the nodes are assumed to be stationary.

The thesis could go in several different directions from here, where each direction included different additional features to be added to the program. Some of the more obvious directions were discussed, while the possibility of unknown challenges presenting themselves during the process was left as a valid option. The directions discussed were the following:

- The Sun SPOTs have some documentation [3, p. 25-27] on how much power is actually used when performing at different intensities (using the radio, lighting the LEDs etc.). It was briefly discussed as an option, to include some work on how the Sun SPOT could perform, when using the actual battery instead of the virtual one. This topic did not end up being handled in this report.
- It was stated in the basic assignment that nodes were assumed to remain stationary. But in a dynamic WSN, the final product would be a lot more applicable if it could take into account when the radio range of a node would be altered (an obstacle could get in the way). This would cause some nodes to fall out of radio range of their original neighbors. Or if a node was moved within the network, or simply removed altogether. This subject was quickly acknowledged as a high priority, and is considered one of the main challenges of this thesis.

A lot of the unforeseen challenges in this project turned out to be in connection with learning how the hardware worked, and in getting the radio communication to work properly, on an unknown amount of nodes. For testing I first had eight Sun SPOTs at my disposal, which already revealed challenges with respect to radio communication and loss of data during transmission. I later received another eight Sun SPOTs making my total testing count 16. The final WSN to test the program on is supposed to consist of somewhere between 30 and 50 nodes.

To summarize:

The project platform is ad-hoc wireless sensor nodes with energy harvesting capability, consisting of the Sun SPOT nodes. The subject is routing packages from any source to a sink node in an energy and latency aware method. The nodes are assumed to be stationary when deployed.

Chapter 2

Sun SPOT

Sun SPOT is short for Sun Small Programmable Object Technology, and it is developed by Sun Microsystems. According to Sun they were developed with several different intentions, one of which was the intention to make them easy to program - even for programmers with little or no experience with previous imbedded systems [6].

The Sun SPOT features several different kinds of hardware. In this project, the most important piece of hardware is the radio of the Sun SPOT, which allows for communication between several Sun SPOTs. Furthermore the Sun SPOT has three sensors; a light sensor, an accelerometer and a thermometer, it features eight LEDs and two switches. The Sun SPOT has more hardware which will be summarized later.

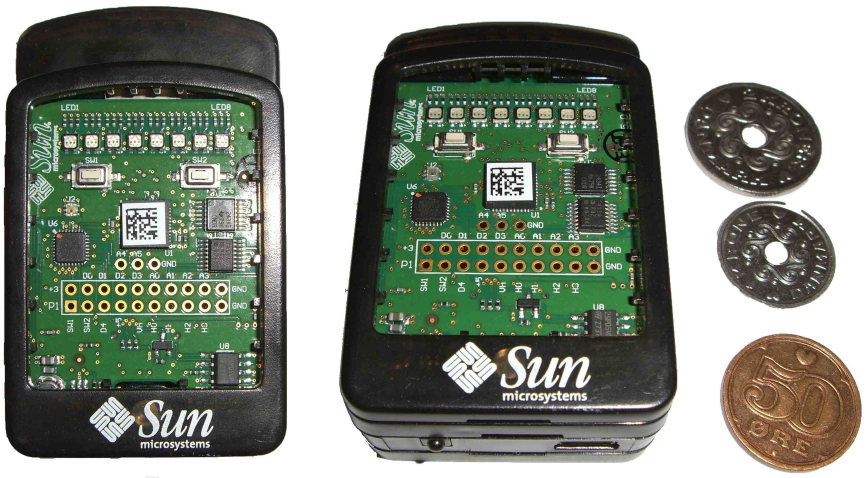

Figure 2.1: Two Sun SPOTs and some coins to show the size of the Sun SPOTs

2.1 The Radio

One of the main functions of the Sun SPOT is its ability to perform wireless communication via radio (a CC 2420 radio chip [7, p. 27]). It is built upon the IEEE 802.15.4 standard, and the driver does not run on an operation system. It runs directly on the Squawk Java Virtual Machine (VM), which uses J2ME (Java Mobile Edition). Sun also

provides a Java Sun SPOT Software Development Kit (Java Sun SPOT SDK), which has several build-in features ready to be used by the programmer. One of these features is the protocol called Radiogram [3, p. 39]. The radiogram allows the exchange of packets between two devices. This can be achieved in two different ways. One way is to open a connection to a specific Sun SPOT (using the IEEE address), which has build-in handshaking, that will retry a number of times when a packet has failed to send, or send an error if the packet continuously fails to be sent. The other means of communication using the radiogram is to broadcast. Broadcasting is a way of sending out packets to anyone that is listening on the same port. This method does not include any guarantee of deliverance, since there is no way of telling who is in range of the broadcasting Sun SPOT at the time of broadcasting.

2.2 Three sensors, eight LEDs and two buttons

The Sun SPOT has three kinds of sensors: An accelerometer [3, p. 34], which can measure how much force is used to move the Sun SPOT, and in what direction. There is a thermometer [3, p. 38], which can measure the temperature and it has a light sensor [3, p. 37], which can return a value indicating the light intensity. I will only be using the light sensor, in order to create a virtual solar collector on the Sun SPOT. The returned value of the light sensor indicates the measured luminance [lx] value, see table 2.1 from [3, p. 37]

Luminance [lx]	Returned value (int)
1000	497
100	50
10	5

Table 2.1: List of lumiance values

The Sun SPOT features eight different LEDs [3, p. 35]. The LEDs use the RGB lighting (Red/Green/Blue), which means they each contain a red element, a green element and a blue element. There are different ways of telling the Sun SPOT what color to show on the LEDs. The method I use consistently thoughout the program is setting the value of each element. The value ranges from 0 to 255. This means that the highest intensity of blue is (0, 0, 255). The LEDs show white by setting every color the same value, hereby, hereby making the brightest white (255, 255, 255).
The Sun SPOT also features two "switches" [3, p. 36]. I believe this is an unfortunate naming, because a switch indicates a device which can be stationary at an ON or an OFF state. However the so called switches are on when pressed down, but they pop back off when released, hence I believe the correct term of these devices is a button. The Sun SPOT SDK possesses the ability to tell when the buttons are pressed, when the buttons are held down and when the buttons are released. I use these buttons for multiple purposes: I use them to send debugging information to the basestation, I use them to have the LEDs show certain information and I use them to send a package through network with the intention of reaching the sink.

2.3 The Basestation

When you purchase a Sun SPOT pack, you get two Sun SPOTs and one basestation. The basestation does not contain buttons, LEDs, sensors or even a battery. The only

thing it possesses is a radio. The function of the basestation is to be attached to a terminal by USB, where a desktop application could be running, and interpreting data the basestation receives from the Sun SPOTs. It is also possible for the basestation to send out data, this could for example be to tell the Sun SPOTs in the field to take some sort of action. In this project, the basestation serves as nothing more than a receiver and transmitter for a desktop application designed to give commands to the network and interpret potential data.

2.4 Other specifications

The Sun SPOT draw power from a rechargeable 720 mAh battery whose lifetime is depicted in table 2.2

Sun SPOT State	Battery Life Estimate
Deep sleep	909 days
Shallow sleep, no radio	23 hours
Shallow sleep, radio on	15 hours
CPU busy, no radio	8.5 hours
CPU busy, radio on	7 hours
Shallow sleep, 8 LEDs on, no radio	3 hours

Table 2.2: Battery Lifetime, the table is from [3, p. 27]

The Sun SPOT also has a lot of other features and specifications which I will list here [8]:

- 2.4 GHz IEEE 802.15.4 radio with integrated antenna
- AT91 timer chip
- USB interface
- 180 MHz 32 bit ARM920T core - 512K RAM - 4M Flash
- 5 general purpose I/O pins and 4 high current output pins
- The dimensions with the antenna are 7cm x 4cm x 2.3cm

2.5 Limitations

Even though the Sun SPOT does provide a wide variety of features and services, they are not without limits. For this project I am creating a virtual battery instead of using the real battery (for simplicity purposes), which allows me to use the light sensor to simulate a solar collector - hereby recharging the virtual battery based on the readings of the light sensor. Ideally the Sun SPOT could have a real solar collector attached to it, so it would be the actual physical battery that were being recharged by the environment. There has been some work done in this area [9], which concluded that it was possible to have the Sun SPOT remain alive for a long term experiment, without having to recharge manually. [9] also states his research has several potential customers - which indicates that it is actually applicable in the real world to attach a solar collector and create a network not unlike the one created in this project.

2.6 Development Environment

When following the installation guide of the Sun SPOT, it is recommended to use NetBeans as the IDE to create your source code. I have used NetBeans IDE 6.5 during this project. The Sun SPOT package also includes what Sun calls a Sun SPOTManager. The Sun SPOTManager is used to update the Sun SPOT SDK, and it can also be used to get information on a Sun SPOT. Furthermore it features a 'getting started' guide which I thought was very useful. There is also a more practical tool within the manager - called Solarium. Solarium is a simulator capable of simulating a number of Sun SPOTs on your PC, while simulating relevant environmental effects (lighting, movement, temperature, pushing buttons etc.). This has been very useful in the early stages of this project, where I had not yet reached the point where I needed to test my program in real life on more Sun SPOTs. As mentioned Solarium is only a simulator, thus many practical errors were never revealed in this tool, such as lost packages during transmission and keeping LEDs on to name a few (the Sun SPOT has some power-saving feature which in certain situations turn off LEDs when the thread turning them on has died). If a basestation is attached to the computer, the simulated Sun SPOTs are capable of interaction with real life Sun SPOTs through the basestation. This feature is very nice, but during this project it was never relevant. Although Solarium was very useful, I found it lacking. I could not place two Sun SPOTs in the simulator such that they were out of range of communication, which would have been very nice to have been able to do in this project. It also demanded a lot from my CPU, an Intel Core2 2.13GHz. When simulating three Sun SPOTs, both of my cores were constantly at a hundred percent in use. This also limited the testing potential of Solarium, which in the end forced me to find other ways to debug my program.

Seeing as how a Sun SPOT has no display, it is very limited how much information they can show when running an application. At first, I lit certain LEDs to indicate certain sections of the program were being run, to help me debug. When the program started to grow, the LED system would have become too complex if I had limited my debugging solely to this approach. The way I handled this was wiring one of the two buttons, such that when I pressed it, certain information would be relayed back to the basestation and then printed out in the NetBeans console by a desktop application. At times this method could be very time consuming due to the fact that if I wanted more data printed by the desktop application, I had to create more code with the sole purpose of debugging. At some point I calculated that approximately twenty percent of the lines in my source code were in order to extract information from the Sun SPOT in a manner that helped me debug my program.

The Sun SPOT developers guide mentions that you can run the applications in debugging mode, but they also write that some code which runs correctly in a standalone application will cause an exception when running under the debugger [7, p. 48]. I took some time trying to understand this debugger, but I concluded that with this statement it was not worth the effort. In retrospect I might have been wrong - which i will discuss further in chapter 7.

Chapter 3

Energy Aware Routing Protocol (EARP)

EARP is a protocol presented to me by a PhD-student named Mikkel Koefoed Jakobsen. It is a work in progress, and is currently being tested and documented by Jakobsen as a part of his PhD-thesis. He is also currently working on an article about EARP [5] which documents several promising test results and simulations.

3.1 The Basic Idea of EARP

As the name suggests, EARP is a routing protocol that takes current power levels within the system into consideration whenever a package is routed through the WSN. Consider the WSN of figure 3.1, where you wish to route data from node E to the sink.

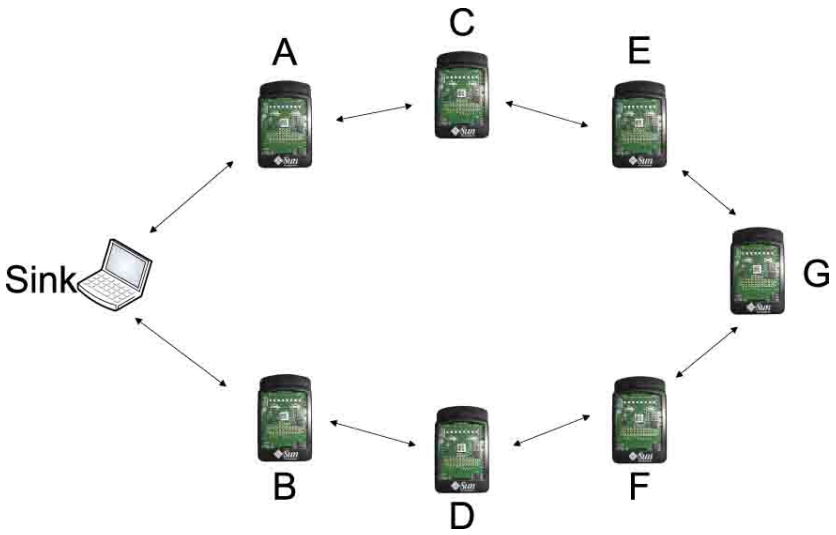

Figure 3.1: A circular wireless sensor network

The shortest path (and hereby the least energy consuming path) is obviously to take C to A to Sink. It is important to note that the nodes have the ability to recharge through a solar collector. Now consider the same case, with the alteration that node C is lying in shade, and is unable to recharge. If data keeps getting routed from E to C to A to Sink, the node C will strain its battery and die sooner than the rest of the WSN - hereby rendering data collection from C impossible until C has been recharged. Due to

this - it is in the interest of the WSN to keep C alive as long as possible, despite the fact that it is in the shade and unable to recharge. This is where EARP steps in. The idea of EARP is to detect that node C is being strained more than the rest of the network, hereby anticipating that C will die first. EARP will then route the data from E to G to F to D to B to Sink instead. This route might be longer and generate a greater power usage, but it will allow C to rest and recharge when possible in order to stay alive and collect data. The nodes which have routed the data from E are all less strained than C. If it turns out that the shade on C was not particularly dense, it will not be viable to continue taking the long way around for long. Eventually the nodes G, F, D and B could become more strained than C due to the extra incoming traffic. EARP will take note of this when it happens, and direct data from E back to the original route, where C is now less strained than G, F, D and B.

One of the more important ideas of EARP, is that it does not have to be the neighbor of E that is strained for EARP to reroute the data. Say that in the previous example, it was node A lying in the shade and not C. Then data from C should of course be routed the long way around (basically the same situation as the previous example), but data from E should not first reach C before it is rerouted - it should be told by C that the route through C contains a strained node. This should prevent E from sending data to C, where C would only send it back again. As stated - this is a very important aspect of EARP, due to the fact that this only requires E to communicate with its neighbors whenever deciding on which way to route a package. This also means that there is no need to determine the entire route to the sink, prior to sending data - the direction of the data will be decided by each node when the data is received.

So in short, EARP may spend more energy in total than other protocols, in order to keep data of neighboring nodes up to date, and routing data might be send through a route containing many nodes, but the idea is that this extra spend energy, is used in a manner to sustain the WSN as a whole - hereby preventing nodes from dying for as long as possible.

3.2 Why EARP

There are several reasons for EARP being the protocol chosen in this thesis. For one it is currently being researched and developed at DTU, and it is therefore very unlikely that EARP has been implemented in a real WSN prior to this assignment - which makes it new and interesting. The Sun SPOTs are adept at visualizing the actual workings of EARP, with the LEDs portraying what is happening, so by using EARP here, there is potential for this thesis to be used as a practical test (and hopefully proof) of the simulated test results.

On the more technical side of things - EARP is very neat compared to other protocols trying to achieve the same goal of longer system lifetime in a WSN [5, II. Related Work]. Other protocols require a complete route from node to sink before they are willing to send data. This demands a complex algorithm in order to ensure the routes are valid. It also requires more stored data in the single nodes than EARP. EARP only requires the nodes to be aware of the immediate next node en route to the sink, before sending data to the sink. This does not require using as many lists as the other protocols, and hereby making the storage space needed on the nodes smaller.

3.3 How is EARP implemented

In order for a node to tell its neighbors how stressed a route through it would be, a node is assigned a numerical value. This value is called the node height, and it is calculated based on three different values; a hops-to-sink value, a depletion value and an augmentation value.

Every node must be assigned a hops-to-sink value. This value indicates the amount of times the data from one node has to "hop" to another node in order to reach the sink, assuming the data takes the shortest possible path. In figure 3.1 A and B have a value of 1, C and D have a value of 2, E and F have a value of 3 and G has a value of 4. If a node is not within reach of the sink, its value is positive infinity. When a hops-to-sink value has been correctly assigned to all the nodes, it is called a distance map.

Every node also has a numerical value representing how much its battery has been depleted, this value is called depletion. The formula for calculating depletion is given in [5, Depletion] and is depicted as formula 3.1

$$depletion = \alpha \left(1 - \frac{Charge_{left}}{Charge_{capacity}}\right) \tag{3.1}$$

α is a constant that [5, V. Results] states (based on simulations) that with a value of 50, had the desired effect on the formula. The depletion is calculated periodically as the node uses power or is being recharged.

The last value a node needs in order to calculate its total height, is the augmentation level. Because a node can only send data to a neighbor if that neighbor is of lower height, there is a possibility that a node can find itself without a route if all of the neighbors have a higher node height (the node is a local minimum). If this is the case, the augmentation level is raised to a value such that at least one neighbor is of lower height. This of course creates a ripple effect throughout the WSN. When one node raises its node height, there is a chance another node will become a local minimum and hereby raising its own augmentation. This will continue until the WSN is settled and every node (not counting the sink) is not a local minimum. It is important to remember to lower this value, once it is no longer necessary to keep it as high as it is in order to avoid being a local minimum. Augmentation is initialized at zero, and is calculated through the formula 3.2

$$augmentation = oldAugmentation + lowestHeightNeighbor - nodeHeight + 1.0 \tag{3.2}$$

With these values of hops-to-sink, depletion and augmentation, a node calculates its node height simply by adding these three values together. Whenever a change is made in the node height, the node should share this change with its neighbors in order to keep them up to date on what route is viable to take. But in order to keep power usage on this sort of communication to a minimum, the implementation features a threshold value. This means that the change in nodeheight has to exceed a certain value before the node deems it necessary to broadcast an update to its neighbors. Figure 3.2 depicts a network with six nodes placed in a line, such that each node has two neighbors excluding the end nodes which only have one neighbor.

As is shown, node A has a hops-to-sink value of one, B has 2, C has 3 and so on. Node A has a very high depletion value compared to the rest of the nodes, which could

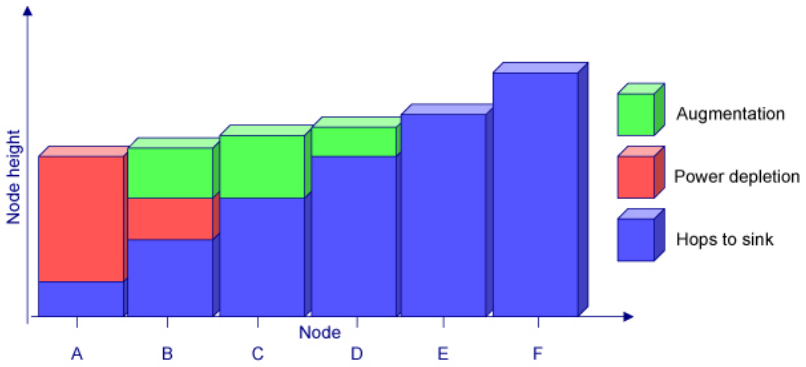

Figure 3.2: Node heights

indicate that A is lying in a shade, having trouble recharging. Node B could also be lying in the shade because it has a depletion value higher than most nodes. We can see that B features an augmentation value, let us look at how this has been generated. Before any augmentation value was calculated, B would not have needed one, because although it is lower than A, it is a little bit higher than C, so at this point it is not a local minimum. C however is a local minimum because both B and D are higher, so C would generate an augmentation value big enough to make C higher than B (which is the lowest height neighbor). Then B would be in the same problem, and generate an augmentation value large enough to make it higher than C. This would continue until B eventually would get higher than A. At this point C would have gotten higher than D which would cause C and D to push each other upwards, until the shown state had been reached. Here no one is a local minimum except for A, but A has direct contact with the sink, so this is not an issue.

3.4 Demonstration

In order to demonstrate EARP in practice, I will refer to [1] which is a video I have attached along with the report. But before we commence with the video, I will explain how to read the LEDs of the Sun SPOTs. Observe figure 3.3.

The first four LEDs indicate the status of the virtual battery, when all four LEDs are completely lit, the battery is fully recharged, and when they are all off the battery is dead. The next four LEDs can display three different kinds of information. At first the eighth LED will glow yellow, which indicates it has no route to the sink yet (in most cases because it has not been initialized yet). The second information they can display, is the hops-to-sink value. The four LEDs depict this value binary in the color red. Lastly the last four LEDs can display when a package is being handled by the node. This is indicated by turning the LEDs green one by one, until all four are green - at which point the package will be handed on to the next node. In the video I will be pressing the left button when I wish for a node to collect data and send it to the sink. In order for the nodes to be close enough to each other for me to video them all in one shot, while still being able to see the LEDs, I have chosen to use a version of my program where each node has a unique set of ports on which to listen and broadcast, which will generate the WSN depicted in figure 3.4.

If this example were to be done without hardcoding, the strength of the radio signals would have to be set to a value so weak, that the radio range would be incredibly inconsistent. The video proceeds like as the following numerated list.

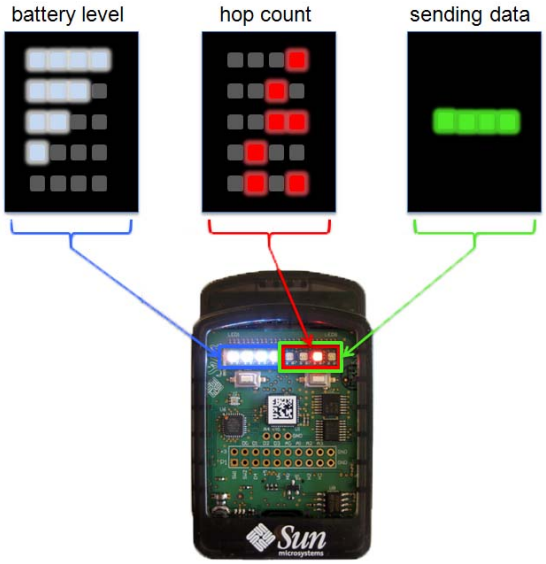

Figure 3.3: How to read the LEDs on the Sun SPOT

1. The video starts off by turning the Sun SPOTs on, where you can see the battery is initialized as fully charged, and the eighth LED is glowing yellow, indicating that the WSN has not been initialized yet. I then press a command on my desktop application, after which you can see the yellow light turning off and red lights depicting the nodes hops-to-sink value is shown for a few seconds.

2. I then send two packages from 3981, which both take the shortest route, which should be sensible as all the nodes should be fully recharged at this point, or at least very close.

3. I also send a package from 2FED which also takes the shortest route

4. Now I put 395D in the "shade", such that the battery of this node will be stressed faster than the rest of the nodes.

5. After letting 395D lose some charge due to shade, I send packages from all over the network, where they all choose to send their data clockwise - hereby sparing 395D of having to send any data, which will cause that node to stay alive for a longer time.

Whenever a package reaches the sink - I have the route printed to the terminal via my desktop application. I have included this output in table 3.1 in chronological order. The actual output has been placed in appendix A.

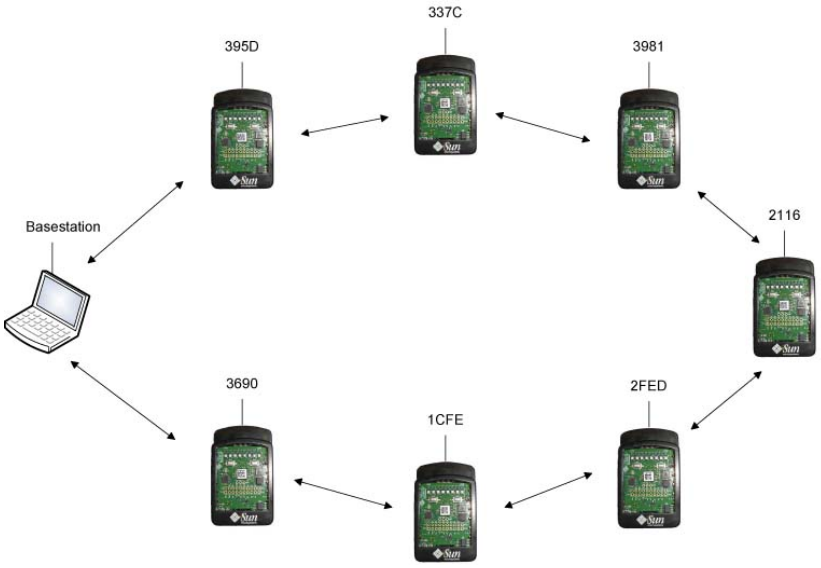

Figure 3.4: A WSN. The last four characters of the Sun SPOTs IEEE address is shown. The first twelve are the same on them all (0014.4F01.0000)

Package number	(Start) Route taken (End)
1	3981 - 337C - 395D - Basestation
2	3981 - 337C - 395D - Basestation
3	2FED - 1CFE - 3690 - Basestation
4	2116 - 2FED - 1CFE - 3690 - Basestation
5	1CFE - 3690 - Basestation
6	337C - 3981 - 2116 - 2FED - 1CFE - 3690 - Basestation
7	3981 - 2116 - 2FED - 1CFE - 3690 - Basestation

Table 3.1: Routes of packages from video

Chapter 4

Wireless Sensor Network Setup

There are a lot of different theoretical approaches to choose from when setting up a WSN. I have put a lot of effort into portraying different kinds of choices, and the consequences of these choices. At first it might be unclear what the different problems might turn out to be, but once I started working with the WSN, the problems presented themselves.

In the early testing stages of setting up a network between multiple SPOT's, it became apparent fairly quickly that some extra attention and resources was to be given to the setup of the network. During testing with only a few sensors (two or three), loss of data during transmission was not detected. However - when the test was upscaled to more SPOT's (more than three), loss of data was so high that simple handshaking algorithms or a queue structure would no longer suffice.

Since the amount of lost data was highly susceptible to the place in which the test was taking place (at DTU, at home, outside etc.) - I assumed that the single reason of the lost data was due to noise on the SPOT's radiofrequency, which is a common high frequency of 2.4GHz. This assumption was proved to be wrong. The SPOT's have ports on which they listen for data and send data. The protocol I use is the build-in radiogram protocol [3, p. 33]. Early on in the project I used the general broadcasting method of the radiogram which does not insure deliverance. This protocol supports a small buffer on a port, which is not meant to handle the communication traffic of a whole network of SPOT's. This results in data being lost in the buffer. This problem was temporarily solved for the small demonstration of seven SPOT's, by assigning special ports for each SPOT on which to broadcast and listen for data, following the model of figure 4.1.

Of course, this was never really a solution since it is far from desirable to manually hardcode the ports on which to send and listen on every single node in a network (which could contain thousands of nodes). I then started working on dynamic solutions to the problem of loss of data, based on the knowledge that using different ports grants the desired effect. I have chosen to use the ports already implemented in the software of the SPOT's, which range from 0 to 255, where ports 0-31 are reserved. This leaves me with 224 different ports to use. I could implement my own ports, but seeing as how this would be fairly complex and time-consuming (not to mention redundant) - I thought the best choice was to make use of the software already implemented. I also used the knowledge that the radiogram protocol has a feature, where it is possible to specify the destination address of the Sun SPOT of a package. This feature insures delivery or notification of failure.

Two different problems presented themselves at this stage:

- The initialization where each node had no information regarding its neighbors, or

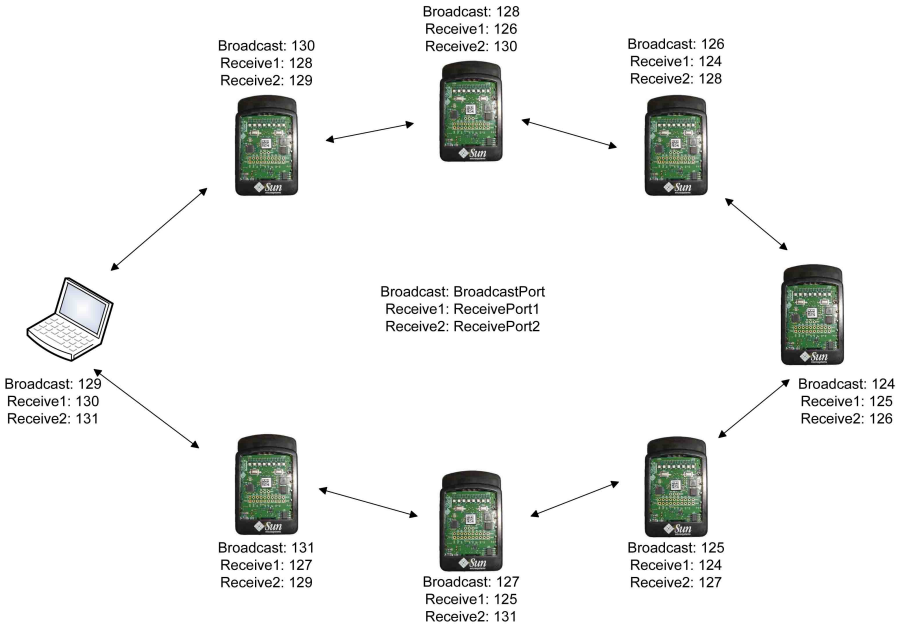

Figure 4.1: Sun SPOTs with hardcoded neighbors

who it is able to communicate with.

- The general setup of the network - i.e. how the network should maintain itself and avoid loss of data once it has been initialized and neighbors have been found.

In order for these problems to be solved the best way possible - it required different solutions for each problem. I will now discuss these solutions as well as other possible approaches.

4.1 Initialization

In order to initialize the network properly, it is important to note the case in figure 4.2.

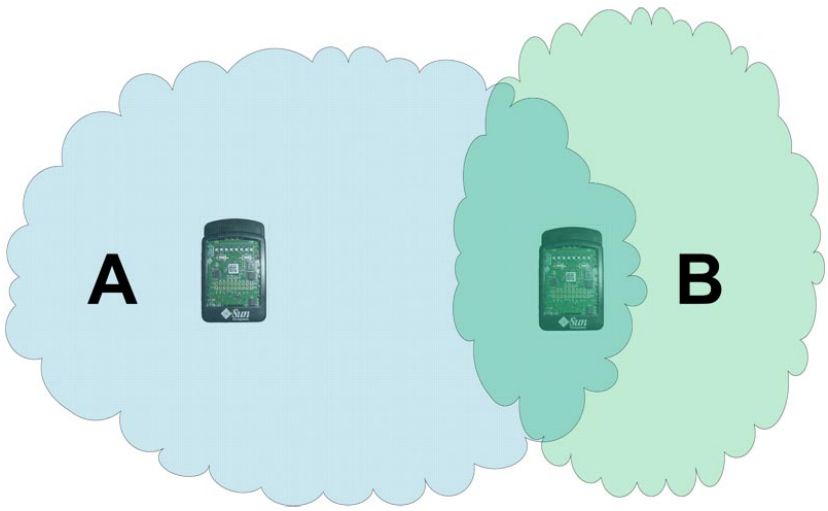

Figure 4.2: The left node A can talk to the right node B, but not the other way around

We see that B is able to hear what A is broadcasting, but B does not have the range to send an acknowledgement back to A. Since A is not able to receive acknowledgement that B is within hearing range, we do not classify A and B as being a neighboring pair, even though B is actually a neighbor of A. I therefore define a neighboring pair of nodes to be two nodes that are both able to hear each other broadcast messages. If one wanted to implement a feature such that the program would support A being a neighbor of B, but B not being a neighbor of A, you could implement a feature where the acknowledgement of B could be rerouted through other nodes in the network. This would however stress the battery of certain nodes even more, and this approach is therefore not a priority in this assignment.

The initialization basically has two jobs to be performed; first it has to assign every node in the network with a hops-to-sink value. Secondly, it is during this phase that the list of neighbors of each node is generated. Although the list of neighbors of a node can be altered even after the initialization, it is still fairly important that nodes do not unintentionally miss information of a new neighbor, as this neighbor could be in the shortest path to the sink, therefore the consequence of not obtaining that node as a neighbor, would be that the hops-to-sink value would be unnecessarily high. The network would also have an increased chance of being split up, where a part of the network would be unreachable.

The initialization process works as follows: Each node except the basestation has a hops-to-sink value of infinity to begin with (indicating it has no route to the sink). The basestation then broadcasts a signal, indicating that every node receiving this signal is directly connected to the sink (hops-to-sink is one), these nodes then send an acknowledgement back to the basestation, and await an affirmation that the basestation is within range of the specific node. If this is the case the node adds the basestation as a neighbor, else the signal is ignored. Then each of these successful nodes broadcasts a new signal, indicating to any receiving node that their hops-to-sink value (i.e. shortest route to sink) could be through here, hereby making their hops-to-sink value two. Now if the receiving node already has a lower hops-to-sink value, the receiving node would send out a reply, and await an answer in order to try and establish the broadcasting node as a neighbor. Else the receiving node would also replace their current hops-to-sink value, and broadcast a new signal stating their new hops-to-sink value. This procedure will eventually end when every node has their minimum hops-to-sink value. The pseudocode for the basestation could look something like the following:

```
broadcastHopsToSink(1)
initializing = true
while(initializing)
    if (got reply)
    send back acknowledgement
        broadcastHopsToSink(1)
    else
        initializing = false
```

While the pseudocode for the nodes would have the following construct

```
hopsToSink = ∞
while(true)
    wait for message with a hopsToSink value
```

```
send back reply and await ackowledgement
if (no ackowledgement is received) then continue
else
    neighbors.add(ackowledgingNode)
    if (message.hopsToSink < hopsToSink)
        hopsToSink = message.hopsToSink
        broadcastHopsToSink(hopsToSink+1)
```

In an ideal environment, this algorithm would produce the minimal hops-to-sink value for every node. However as I have stated earlier, loss of data during broadcasting is an issue. Therefore this algorithm does not guarantee that it is the actual optimal values of hops-to-sink each node will end up with. But it will be an approximation, which is subject to optimization even after the initialization, where the nodes continue to update their list of neighbors. One way to ensure a result closer to perfection during initialization would be to have the basestation broadcast a number of signals at an interval, indicating that the system should repeat the process of initialization. This way the chance of an unintentionally lost edge between two nodes existing is greatly diminished.

Note at the stage of the chinese delegation demonstration, this was actually how the initialization worked. But in order to remove the parts of the program that were hardcoded for demo purposes, the initialization should also set up some sort of system in the network in the ways of communication (using different ports or using build-in protocols).

4.2 General Setup

This section will cover the different approaches I have considered towards solving the problem with loss of data, due to too much traffic on a port. Some of these approaches are similar to each other in various ways, which should come as no surprise as good solutions are often derived of a mix of multiple solutions. That being said, most of these approaches are generally acknowledged and accepted protocols, and are already being used on different systems (the internet, LAN etc.).

During my speculations I have had to take several different factors into consideration. Seeing as how I have a time limit on my project, it has to be taken into consideration that I could run out of time prematurely. Therefore I have prioritized solutions which I know I can use within the time limit. If then there would be time left at the end, I could try to improve this part of the program. Another aspect of my considerations is the fact that this is supposed to work on an arbitrary number of nodes, therefore the program should preferably not be broken by a very large network of nodes. Furthermore, since this is a wireless network of nodes, it would be desirable to conserve as much energy as possible in a section of the network. One more important thing I had to consider, was the fact that it would be a nice feature to take care of nodes being moved, removed or inserted to the system after initialization. Lastly I also considered that this system has to work in real time, meaning that it is a priority that outdated information is not used as new information, i.e. it is important for the broadcasting queues to be very small or even non-existent. However, this is not a critical factor, unless the outdated information is incredible old. This is because outdated information will generally be close to the actual information, and therefore it will not cause the network to break. It might cause a few routings of packages going through stressed nodes, but this error

would be correctly fairly quickly as this is the nature of the EARP algorithm.

As mentioned earlier, I will design my solutions on the fact that I need to ensure neighboring nodes do not communicate on the same ports as other nodes in the list of neighbors. This is based on the fact that this solved the problem during the demonstration. I will however also consider a protocol build in the Sun SPOTs - which to a great extend should work as the other solutions - this specific approach is covered in the following section called *Radiogram Protocol Point-to-Point Connection*. I have 224 unique ports at my disposal, supplied by the build-in software of the SPOTs. Furthermore it should be noted that in this section, I assume that the list of neighbors in each node has been completed during the initialization.

4.2.1 Aloha

The Aloha protocol is a commonly known protocol [4], designed for wireless networks. It has a very straight forward policy; if a node has a desire to send data, it should simply send it. If the node then does not receive an acknowledgement from the receiving node before a certain time period, a collision is assumed to have happened, and the original node will try and send the data after a random amount of time. In figure 4.3 the boxes represent packages, where the gray boxes have caused a collision and must be resend, while the white boxes have succesfully been send and received.

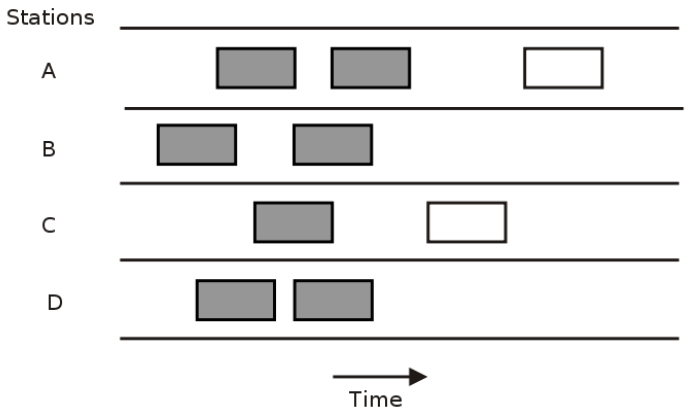

Figure 4.3: Pure Aloha, picture taken from [4]

The pros:

- It is simple to implement
- It does not take a lot of alterations to allow inserting, removing or moving of a node
- It is not going to break down because of a large network of nodes

The cons:

- This algorithm is not very energy-efficient. In a crowded area, a lot of information will be rebroadcast, and therefore a lot of power will be wasted this way.
- It does not really do anything to reduce the amount of lost data.

4.2.2 Ethernet

Ethernet is a lot like Aloha. The difference lies in the fact that Ethernet is used in wired networks, due to the fact that Ethernet is based on CSMACD "Carrier Sense, Multiple Access, Collision Detection". The Multiple Access says that every node in an Ethernet network is connected through a single data path. The Carrier Sense says that a node will only send data if the wire is currently available. Lastly the Collision Detection, this is the part that makes it hard to implement in a wireless network. Collision Detection says that the system is able to detect whenever a collision has occurred on the wire. When a collision occurs, the nodes send out a jamming signal to prevent the other nodes from start sending new packages, until the collision has been resolved. The collision is handled the same way as Aloha - the nodes will wait a random amount of time and then retransmit, hereby making sure that one of the two nodes will gain access (and thereby preventing access from the other node) first, and hence complete the data transfer.

As stated, Ethernet is not desirable in my current situation, due to the fact that collision detection is difficult to build into a wireless network, if you want to do it like in a wired network, and not like aloha (although it is possible).

4.2.3 TCP

TCP "Transmission Control Protocol" is used a lot online, and the general concept could prove advantageous to this project, therefore I chose to include this in my considerations. What I can use is the very basic idea of TCP, which is to break packages up in smaller packages and then transmitting these packages. But seeing as how the packages in my network will most likely always be very small, breaking the packages into smaller parts is irrelevant in this case. However, when the packages are split into smaller sections, they are marked in a way such that the receiving end can tell if it has lost a package. This could be done by numbering the packages, so that if for example a node has received every package from one to seven, and then suddenly receives a package marked nine, it knows that package eight is missing, and then it should ask for the broadcasting node to rebroadcast package number eight. This way lost packages will be retrieved. This would require a list of packages that has been sent for each neighbor - thus a 2-dimensional list of packages, where the packages are identified by a node and a package number. It would also require a list of integers identified by a node, where the integers represent what the number of the last received package from the indexing node is. The pseudocode for a receiving node could be the following:

```
lastReceivedPackageNumber<int>[node]
listOfSentPackages<package>[node][packagenumber]
while (true)
wait for package
if (package.getNumber() - 1 == lastReceivedPackageNumber[package.getNode])
    interpretPackage()
    lastReceivedPackageNumber[package.getNode] = package.getNumber()
else
    int i = lastReceivedPackageNumber[package.getNode]
    for (i; i == package.getNumber(); i++)
        ask for package number i
        interpretPackage
    lastReceivedPackageNumber[package.getNode] = i
```

The pros:

- It is not too complex to implement.
- No acknowledgement from the receiving end is actually required.

The cons:

- It requires a buffer of some size to contain a number of old packages.
- It is somewhat susceptible to the amount of neighbors of a node, since a node is required to save an amount of old packages for each neighbor.
- It does not really do anything to reduce the amount of lost data.
- A node will not discover it has lost a package before the next package is received, by that time the old package will likely be outdated and useless.
- It would require some work to extend TCP to allow inserting, removal or moving a node.

4.2.4 Port multiplexing

Port multiplexing is basically that each node gets a unique port number on which to send and receive. This number would be stored in the list of neighbors along with the other information of a neighboring node. Seeing as how I have 224 different ports to use, this solution would seem to satisfy the network of circa 50 nodes we plan to use in the final demonstration. However since it would be very nice to be able to apply this system to other larger networks than 224 nodes (hundreds or thousands of nodes), I would start by assigning the nodes with those 224 ports, and when they are all used, the oldest ports could be reused again and again. This requires that 224 ports is enough ports to make sure that no neighboring nodes are assigned the same port, and since it would require a very large network, or a large chunk of nodes being placed in the same general area, this could be a promising and simple solution.

The pros:

- As long as a node does not have too many neighbors, this approach should resolve the problem of lost data very thoroughly.
- It is fairly simple to implement.
- It does not take a lot of alterations to allow inserting, removal or moving of a node.

The cons:

- It does not guarantee success when the network is very large or a large chunk of nodes are placed within each other's radio range.

4.2.5 Time division multiplexing

Time division multiplexing is, as the name suggests, somewhat similar to Port multiplexing. In this case, each node is assigned a unique time slot in an interval in which it is allowed to broadcast. This approach has some potential, because it too will eliminate the problem of lost data completely. Furthermore this approach will theoretically not break under a very large network, or even a large chunk of neighboring nodes. This approach will however cause a certain constant delay to the system, even if it is not necessary, due to the fact that a node will have to wait its turn, even if the other nodes have nothing to broadcast In figure 4.4, you can see that if node 1 has just missed its timeslot, it will have to wait for the time slots of nodes 2, 3 and 4 to pass. In large networks, this delay could be substantial. It is also a problem to have all the nodes of the network to be synchronized when it comes to their internal clock. They do not have a central clock to have as a source of synchronization, so this would probably represent a good deal of work to implement.

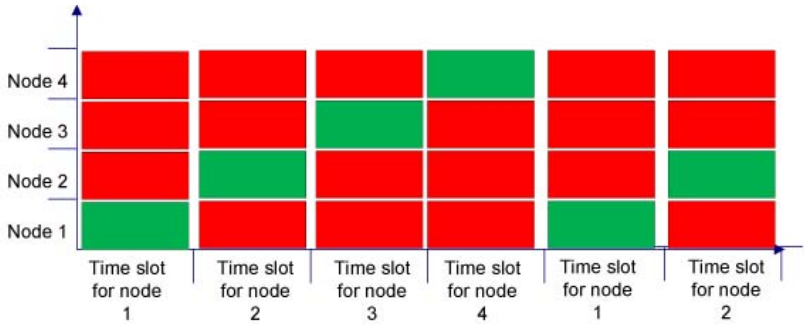

Figure 4.4: Each time slot is reserved for a node

The pros:

- The problem of lost data should be resolved completely.
- Will not break in large networks or a large chunk of neighboring nodes.
- It does not take a lot of alterations to allow inserting, removal or moving of a node.

The cons:

- When dealing with time slots, synchronization is always a problem when there is no central clock.
- The network will be subjected to a constant delay, susceptible to the amount of nodes in the network.

4.2.6 Port multiplexing - Controlled pattern

This approach is an advanced version of the Port multiplexing. The general idea is the same, although instead of simply starting over with the ports when they have all been used - this approach conserves the usage of ports.

In the figure 4.5, a hexagon represents a node, and its neighbors are the nodes touching the hexagon. The letter in the middle represents a port. In the shown example

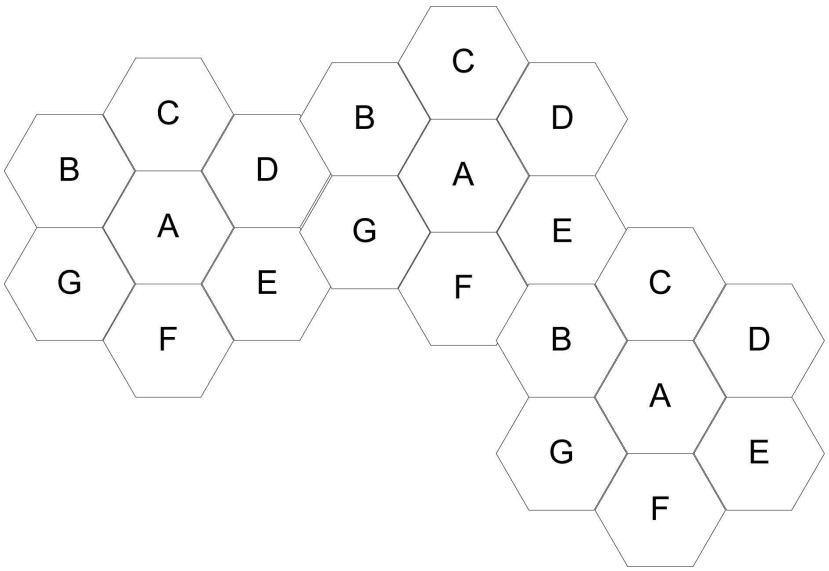

Figure 4.5: 'Clusters' of seven nodes

there are three "clusters" of port patterns, which results in only using seven different ports (A-G) for a network of 21 nodes. The Port multiplexing here would have required 21 different ports. This ensures a minimum amount of ports are in use, and yet no neighboring pair of nodes uses the same port to communicate.

The pros:

- It is more controlled than Port multiplexing when it comes to guaranteeing that no neighboring pair of nodes uses the same port.
- A smaller amount of unique ports are required.

The cons:

- This approach is somewhat complex to implement, as it would take some work ensuring that no neighboring pair of nodes uses the same port.
- It still does not guarantee success if the network is too big, or the amount of neighbors is too big.
- It would take extensive work to allow insertion, removal or moving a node after initialization.

4.2.7 Listen before send with multiple ports (ports as a semaphore)

This approach is based on the idea that a port can be used as a semaphore. All nodes have access to the same ports, and when they wish to broadcast, they go through the ports until they find a port not in use. They then broadcast on this port, that this particular port is now in use (the node takes the "semaphore"). Then the node will broadcast the desired message, and when all is done, it will broadcast that the port is no longer in use (it releases the "semaphore"). Now this approach seems to be very nice, it can handle insertion of new nodes into the network with ease, and with enough ports this should remove the problem of lost data. At first glance this is particularly nice, because it will sort of create the "clusters" of the Port multiplexing - controlled pattern,

with ease. Because any node out of hearing range will not see a port as unavailable if a node far away has claimed it, which means the same ports can be used throughout the network. However, if it turns out at some point that a part of the network contains too many nodes, some nodes may have to wait for a long time in order to send their package, which will cause the information to be outdated. This would require some sort of queue or buffer, in which packages can be stored until a port is available for use. It could also eventually be necessary to simply delete the information in a queue if it is simply too old, which would require some sort of prioritization system. But this approach is not good - even if the above quarrels are worked out - the entire approach fails when we introduce the problem of node B hearing node A, but A not hearing B. Then B could take port 10 and broadcast port 10 is in use, which A will not hear, then right after, A needs to broadcast and will also take port 10. Now the two nodes will broadcast on the same channel, and the original problem is therefore not removed. It could also happen as depicted in figure 4.6

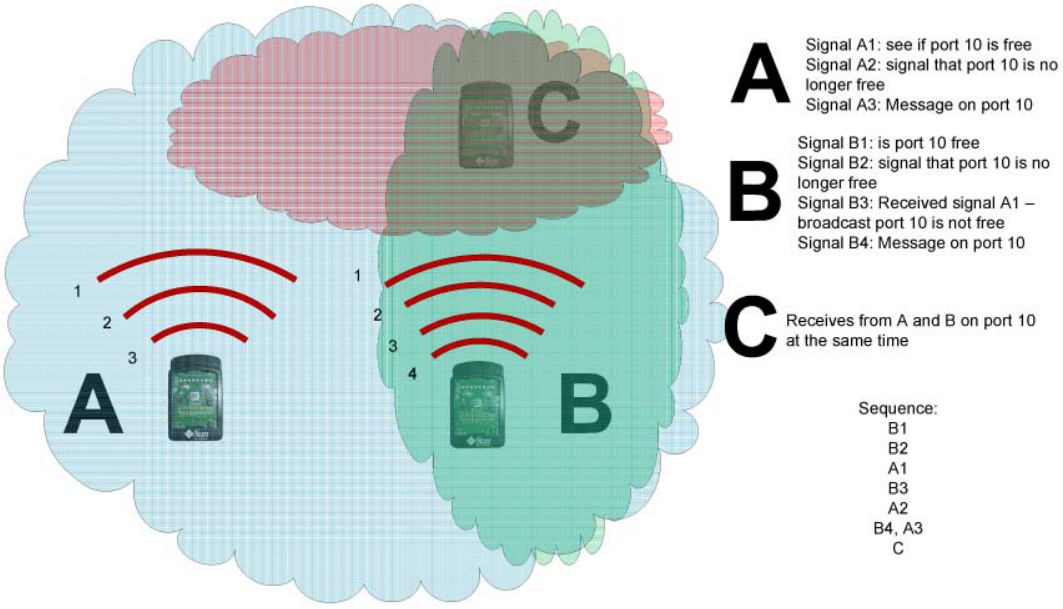

Figure 4.6: Top node C receives two messages on port 10

The pros:

- If there is enough ports, this should be a fairly simple and effective solution
- It is very dynamic when it comes to insertion of nodes or replacing nodes in the network

The cons:

- It is difficult to guarantee that there will not be a lack of ports in an area, which could break the system.
- Some sort of queue or buffer storing old information is needed.
- A system to prioritize between different packages is needed.
- It does not completely remove the original problem.

4.2.8 Radiogram Protocol Point-to-Point Connection

All the previous suggested approaches towards setting up the WSN are based on each node broadcasting to anyone listening on the broadcasting port. As stated in the Sun SPOT section - this method does not provide any guarantee of deliverance or notification of failure. In order to use any of the previous approaches, it is therefore necessary to come up with a method that will act similarly to handshaking. This would most likely involve some waiting, in order to give the potential receiving nodes a chance to respond. It would most likely also involve a number of acknowledgements being sent back and forth, to ensure deliverance or notification of failure. However, the Radiogram protocol does feature a point-to-point connection [3, p. 39]. This feature includes guarantee of deliverance and notification of failure. Since this is a connection to a specific Sun SPOT (identified by the IEEE address), it is required to have the connection opened by both the sending and the receiving Sun SPOT. But the receiving Sun SPOT does not have to identify the sending Sun SPOT. This means that the receiving Sun SPOT just indicates a port to listen on, and will react when a packet with its IEEE address on it arrives. A broadcasting connection is opened this way by indicating an IEEE address and a port number. I decided to test if the problem with lost data due to high traffic on the same port would also be a problem if the connection was opened as a point-to-point connection. Since any Sun SPOT which is not the target IEEE address will disregard the broadcast even before reading the data from the packet - it turned out that when a point-to-point connection is used instead of a general broadcast, the problem with the loss of data is not present. I say this after having tested it on 16 Sun SPOTs in the vicinity of each other - but without any more Sun SPOTs I cannot determine if it will be a problem with a higher number of Sun SPOTs.

When using this approach, each node must be aware of who should receive the information that is about to be sent, i.e. its neighbors - it is no longer being broadcast for anyone to receive and sort out manually. Therefore a list of neighbors with their IEEE address is needed, which will be used whenever a node has to broadcast anything. This means that each node must listen on the same port, and when they wish to broadcast, they have to open a new connection to the specific node. Since the number of neighbors differ from node to node, it is not possible to create each connection to the neighbor as a field in Java ME. Therefore whenever a node has finished broadcasting something to another node - it is crucial that the connection is closed or else errors will occur and exceptions will be thrown.

The pros:

- There is no need to create my own insurance of delivery or notification of failure.
- It does not seem to have the problem with loss of data during high traffic.
- Each node does not have to manually check if a package is meant for it or not - this cuts down on calculation time.
- If it turned out that a large number of Sun SPOTs will in fact cause loss of data, this approach could be combined with one of the previous approaches, due to the fact that ports are also used in this.
- Theoretically it should not require a lot in order to allow insertion, removing or moving a node after initialization.

The cons:

- I am not in complete charge of the handshaking algorithm - which is poorly documented. So it might not work as intended

- It is not absolutely certain if the original problem is eliminated - this can only be done through tests.

4.3 The Selected Setup

The setup I have chosen is the one described as the *Radiogram Protocol Point-to-Point Connection*. One might argue that it would have been wiser to choose an approach that had been tested to work, which all the approaches based on using different ports have been by the Chinese delegation demonstration. But these solutions were all somehow lacking in performance and/or scalability. Furthermore I have also tested the point-to-point connection as well as possible with my available sixteen Sun SPOTs, and it seemed to work just as well as using multiple ports. The fact that this approach has build in handshaking, had a lot of influence in choosing this approach. Build in handshaking, guarantee of deliverance and notification of failure was a lot of what I needed to make the implementation to the Sun SPOT. But as I will discuss later - the notification of failure part is somewhat bugged. Assuming that it was not - it would be very simple to implement the feature of removing, inserting or moving a node within the WSN. As I will discuss in the *Maintenance* section. My program runs two concurrent threads, one which is in charge of reading the light sensor periodically once every second. After which a new node height is calculated, and if the change is node height is significant enough (based on a threshold value) - the node will inform every neighbor of its change in height. This thread looks like the following pseudocode:

```
read light sensor
recharge battery according to the readings
if (|newNodeHeight - lastBroadcastNodeHeight| > threshold)
    for (every neighbor)
        try to open connection to neighbor
        if (connection failed)
            remove neighbor from list of neighbors
        else
            use connection to broadcast new node height
            close connection
```

The other thread is the thread listening for packages being send to the node. It has the following pseudocode:

```
open connection to listen on port 110
wait to receive package
if (package.getNode is not in list of neighbors)
    add package.getNode to list of neighbors
interpretPackage()
```

It should be noted that the method called interpretPackage() is capable of ending up broadcasting information a number of times to neighbors. These connections all follow the pattern that they are opened just before sending a package, and close as soon

as possible after a package has been sent. The following is an example from the source code where a node has received a package, information of a change in node height of another node has occurred:

```
1 else if(pack.getMode() == Consts.NEWHEIGHT && searchForNeighbour(address
      )) {
2         neighbours.put(address, Double.toString(pack.getNewHeight()));
3         calculateAugmentation();
4         calculateNodeHeight();
5 }
```

It should be noted that this code does differ from the pseudocode. This code prevents new neighbors from being discovered due to a change in node height. This has been added in an attempt to keep the WSN more consistent, because it turned out that without this addition to the if-sentence, the neighbors would fall in and out of each ohers radio range so often, it would cause a delay on the system. Ideally it should be as the pseudocode. It can be seen there is a chance that the receiving node is getting a new augmentation value, and therefore also a new nodeheight. Let us assume that it does get a new node height, and the change is big enough to overcome the threshold. In this case a call to `broadcastNodeHeight()` is made, which consists of the following source code:

```
1  Enumeration e = neighbours.keys();
2  while(e.hasMoreElements()) {
3          String address = e.nextElement().toString().toUpperCase();
4          try {
5                  if (!address.equals(basestation)) { //No need to tell
                       the basestation of changes in height
6                  RadiogramConnection s4Con = (RadiogramConnection)
                       Connector.open("radiogram://" + address + ":" +
                       Consts.HOST_PORT);
7                  s4Con.setMaxBroadcastHops(1); //Do not rebroadcast
8                  Radiogram s4dg = (Radiogram)s4Con.newDatagram(2000);
9                  s4dg.reset();
10                 s4dg.writeLong(Consts.NEWHEIGHT);
11                 s4dg.writeDouble(nodeHeight);
12                 s4Con.send(s4dg);
13                 s4Con.close();
14                 }
15         }
16         catch(Exception ex) {}
```

We do indeed see the pattern of a connection being openened to a specific address in line 6, and since it is the same port used at all times in this approach, that port is stored as a constant (port 110). Then it creates a package to send, indicating a new height has been obtained at this node, and after the package is sent, the connection is closed. We also see that the method iterates through the list of neighbors in order to tell every neighbor of the change in node height.

4.4 Maintenance

After the initialization process, it is important to maintain the network by making sure all of the nodes in the network are represented correctly with the right neighbors. When the battery of a Sun SPOT is low on power, it could shorten the range of the radio -

hereby potentially breaking up a neighboring pair of nodes. The environment might also change in the WSN. The landscape could be altered in terms of rocks being placed or removed, vegetation appearing, snow falling etc. All of this has a potential effect on the range of the radios. It could also simply be a matter of a node being removed, inserted or just moved within the WSN. Therefore it is important to continuously update the list of neighbors of each node.

There are two aspects of this problem; the insertion of a node to an area (either by increased radio range or by physically inserting or moving a new node to the area) and the removal of a node in an area (either by reduced radio range or by physically removing or moving the node).

4.4.1 Inserting a node

When a node is inserted into the WSN, the first thing it does is to make a general broadcast, stating that it would like to receive a hops-to-sink value from everyone able to hear it. First it will add all the replying nodes to its list of neighbors, because the reply to the hops-to-sink request is also considered an acknowledgement from the initial broadcast - which is the foundation of a neighboring pair. Immediately after adding a node to the list of neighbors, it will send out an acknowledgement to the new neighbor - telling it that they are indeed a neighboring pair. Then it will choose the smallest value of all the received hops-to-sink values. After which it will broadcast its chosen hops-to-sink value - hereby giving the WSN a chance to adjust itself accordingly to the new node.

If a node has been blocked by some environmental factor, and is suddenly revealed in a manner such that the radio range increases - it should verify its new neighbors and add them to the list of neighbors. Because my chosen way of approach is the point-to-point connection, the node has to detect when its own radio range has been increased, which is a cue to make a general broadcast, indicating a search for neighbors, and then perform a handshake with the nodes that reply. One way of detecting increased radio strength is for the neighbors to keep track of the signal quality of the node. When the quality is notably enhanced, it could let the node know - hereby giving it an indication that its radio range might have increased. This is a fairly tedious approach and requires storage of more data. It also fails if the radio range has been reduced to nothing - which means an empty list of neighbors. In this case no neighbors would be able to indicate a change in the radio strength, and the node would be stuck with an empty list of neighbors - even though in reality it has plenty of neighbors. The approach I have chosen is to periodically do a general broadcast - checking if any new neighbors should have entered the range of the radio. This might cause some redundant broadcasts, but it is simple, and does not rely on other nodes to work.

4.4.2 Removing a node

Whether a node is physically removed from an area, or the radio range of the node has been reduced does not matter because I have chosen point-to-point connection. When a node has been removed or is suddenly no longer able to reach a neighboring node - it will fail in opening a connection to said node. When this failure is detected -it is allowed to try again, but if it continues to fail, it must be assumed that that specific neighbor is no longer in range, and should therefore be removed from the list of neighbors. In theory this is quite simple, but in practice it turns out that the timing of the exception the Sun SPOT throws when failing to open a point-to-point connection with another

Sun SPOT, is quite unfortunate. When the connection times out and thereby failing to open; the Sun SPOT still registers the connection as being opened. Because you cannot have more than one point-to-point connection open on the same port to the same Sun SPOT - this is a problem. The first time the Sun SPOT is removed, it is seemingly done correctly. But when you try to insert the Sun SPOT back into the WSN, exceptions are thrown, stating that point-to-point connections are already open to this specific Sun SPOT. I have yet to figure out a way to tell a Sun SPOT to close the connection after trying (and failing) to open a point-to-point connection. Besides this, it would also be required of the nodes to refresh their hops-to-sink value, because if a node in the shortest route to the sink has disappeared - a new route and new hops-to-sink value is to be found. The node should therefore request the hops-to-sink values of all its neighbors to be reset to infinity, if their shortest path to the sink included the node. It should then request a new hops-to-sink value of its neighbors. Meanwhile the neighbors that reset their hops-to-sink value, should repeat what the node did. Eventually some node will receive a valid hops-to-sink value from a neighbor that did not contain the removed node in its shortest path to the sink, or all the nodes will be reset to infinity - indicating that a cluster of nodes has been cut off from the sink.

Chapter 5

Program Structure

My program is made in Java ME (Java Mobile Edition), which is what is required to run on the Sun SPOTs. It is developed in NetBeans 6.5 which is what Sun Microsystems recommends, when installing the Sun SPOT kit. The program is not particularly large compared to other programs representing the amount of work hours it has received. This is not because the program is shallow, but because of the complexity of the setup and the number of solution models investigated. For each minor problem, a lot of theoretical solutions seemed plausible. But a lot of these solutions turned out to fail in some technical aspect, which would have been very hard to foresee. The entire source code [10] can be retrieved from the added CD.

5.1 UML Diagram

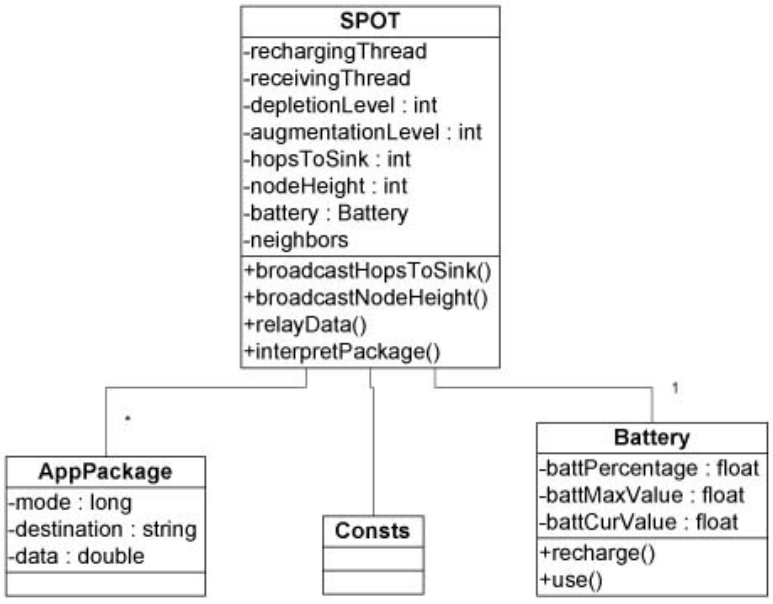

Figure 5.1: UML diagram of the Sun SPOT application

Figure 5.1 shows the UML diagram of the application running on the Sun SPOTs. I have only added the more crucial fields and methods of the classes to this diagram. `Battery` is the virtual battery I have created, which can be used by the SPOT or it can be recharged by the SPOT (it also stores and returns some less interesting values). `Consts`

is a class containing my universal constants. `AppPackage` is a class that represents the data a node will send and receive. It contains a mode, which indicates what information the package holds (a hops-to-sink value, a new node height, data to be relayed to the sink and so on). `AppPackage` also contains its destination and the relevant data. Currently the packages are rather simple, so it would not complicate matters a lot to remove this class and have this work done in the SPOT class. But I created this class with the intension of making the program more susceptible to upgrades and changes - which adding a class like AppPackage certainly achieves.

As hinted - the SPOT class is the most crucial and by far the largest of the classes. It contains two crucial threads; the `rechargingThead` which takes in a reading from the light sensor once every second, and from this reading it recharges or depletes the battery a little (depending on how intense the light is). The other thread is the `receivingThread`. This is the thread that listens on a specific port (port 110 in this case) for any incoming packages. Whenever it receives a package, it will call the method `interpretPackage()`, which then takes the appropriate action depending on the package, this method is explained in the next section *Important Methods*.

5.2 Important Methods

There are a few methods in the program which have larger responsibilities than the others. Although the complexity of the methods is limited, the size of some of them indicate that they are doing more work than other methods. Here I will present some of the larger ones. They all originate in the SPOT class, which is where the main bulk of the work is done, so this makes sense.

5.2.1 interpretPackage()

This method is called whenever the `receiveThread` receives a package. The first two values of the package are always read first. The first value indicates the mode of the package (what kind of information it contains), and the second value either contains all of the information in the package or some of the information - in some cases it is necessary to read in more values from the package. When this is done, a large if-then-else structure has a special action for each mode. This action could be to perform some sort of broadcast, or perform a calculation in order to update the node height for example. When the corresponding actions have been taken, this method is done.

5.2.2 relayData()

This method is not very big - but it is quite crucial as this is the method that generates a new package to be sent to the sink. The only way to invoke this method is to press the left button on the Sun SPOT. In order to visualize sending the data, I have intentionally made a delay of half a second, between lighting the green LEDs. When all four LEDs are green, the method goes to work. It creates a package with the appropriate information and mode, then it finds the lowest height neighbor to be the destination of the package, which is then given as information to the package. At this point the package is ready to be sent, so the battery is drained a little to represent stressing it a little during transmission, and finally a call to the broadcasting method is invoked - creating a broadcast to the receiving node.

5.3 The Desktop Application

In order to receive information and print it out on the desktop - I have created a desktop application to run with the basestation attached. This application consists of a small GUI containing some buttons, that enables me to tell the WSN to perform some action. The single most important button here is the button called Hops To Sink. When I press this button, the basestation sends out a signal telling everyone able to hear it that they are within radio range of the basestation. If a successful handshake is performed, these nodes will then receive a hops-to-sink value of one, and continue the ripple effect of broadcasting hops-to-sink values throughout the entire network. The rest of the buttons are not finished.

Although most modes of the packages being broadcast in the WSN are to be ignored by the desktop application - there are some which indicate that the package is test data, which should be printed for one to see in the terminal. This data could consist of the list of neighbors of a node, the node height of these neighbors, the node height of the node itself, the last node height that node broadcast to its neighbors (which can vary a little from the actual node height - depending on the threshold value) and things of that nature. The source code for the desktop application [11] can be found on the CD.

Chapter 6

Results

Although the program is not currently ready to be deployed in the field, parts of the program does work, and these parts have been demonstrated to work. Furthermore I am aware of what the problem is in my program, and I might have been able to work it out given enough time. But keeping a deadline is also a part of this project, so at some point I had to stop and recognize the fact that I did not have any more time to dedicate towards getting all the wanted features of the program. The mere fact that I know what the current error is, and why that error exists is also a result in itself. I will divulge more attention to this error in chapter seven.

For now I will display my results from different versions of the program. First I had to create a working demonstration model, for a Chinese delegation of scholars to see. After this demonstration, it was clear that I had to put some effort into initializing the WSN. Lastly I had to maintain the WSN, by handling the removal or insertion of nodes.

6.1 Chinese Delegation Demonstration

In order to meet the deadline for the live demonstration of EARP to the Chinese delegation, I had to hardcode a WSN, in a manner such that the nodes would use predetermined ports when communicating with a neighbor, thus preventing loss of data due to high traffic. This proved highly successful in removing the problem with loss of data. In the video [1], it is actually the version of the program I had at the time of the delegation - following the exact pattern of figure 4.1. I have already gone through the details of the video in section 3.4 *Demonstration*, which portrayed the idea of EARP to the letter. This version of the program did not however support the removal or insertion of a node, which were also ignored in the video.

As a result of this demonstration, I figured out that the loss of data was due to high traffic on the ports - this has proven to be quite valuable information to me, in order to advance the program, and in order to remove the hardcoded parts.

6.2 Initialization

The workload of the initialization varies from the different available approaches of the general setup of the network. In my chosen setup, where I use point-to-point connections, the most important part of the initialization is to create a correct distance map (assigning correct values of hops-to-sink). Another important issue to take care of in the initialization process is the creation of the list of neighbors for each node. Because

of my chosen approach, it is rather simple to maintain the list of neighbors, because of the build in handshaking of point-to-point connections. But during initialization, the nodes sense their neighbors by doing general broadcasts, which warrants the need for my own handshaking algorithm in order for a neighboring pair to be sure that they can both hear each other. This feature is implemented in a point-to-point connection, so whenever a node receives a message from a point-to-point connection, it knows it has received something from a neighbor.

It has proved to be quite difficult to perform a practical initialization process of a WSN which is supposed to have nodes farther away than two hops from the sink. This is because there does not seem to be a way to lower the radio strength of the base station, which generally has a range of around 20 meters. Furthermore although the Sun SPOTs have a way of turning down their radio strength, their radio range becomes incredibly inconsistent and the quality of the signal becomes extremely poor, rendering the point-to-point connections practically useless if you want a range below a meter. But even if when you try to aim for a range of a couple of meters, these symptoms still occur. Due to this, it requires an enormous amount of space to correctly set up a WSN where the hops-to-sink value of some nodes is higher than two or three. In the video called 3 SPOTs.avi [2], I have a WSN of three nodes following the setup of figure 6.1

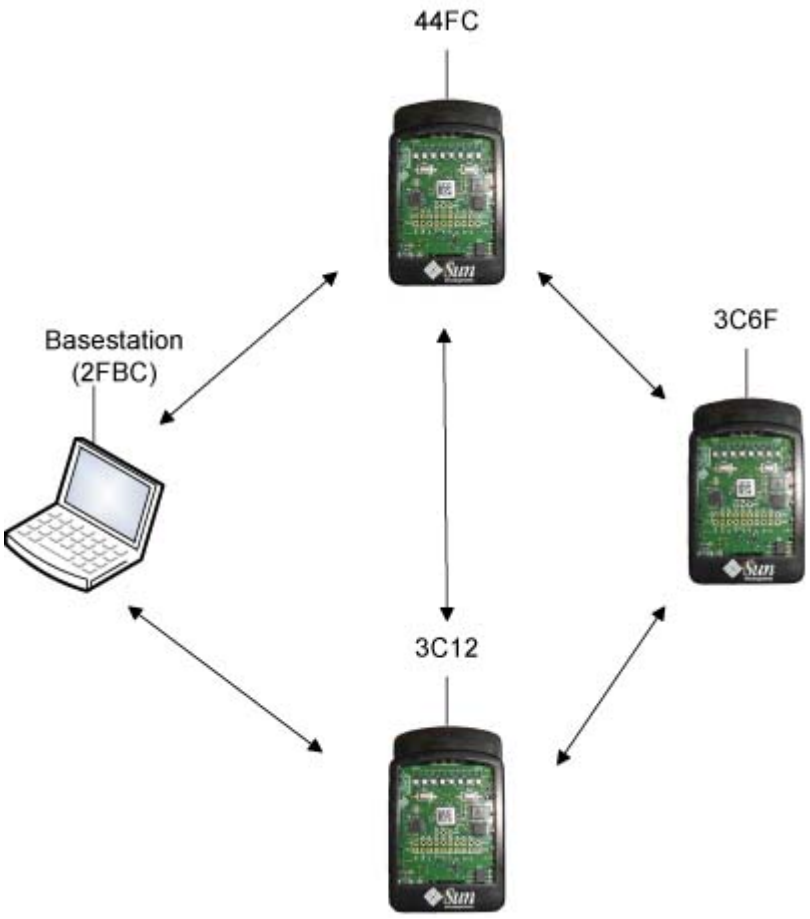

Figure 6.1: A WSN of 3 SPOTs

In order to create this setup, so close to each other with radios at full strength, I had to do a few things. First I have to turn off the basestations broadcast of a hops-to-sink value when a new node enters the WSN and requests one - otherwise I would have to

place 3C6F very far away. Second (and I do this in the video) I only turn on 44FC and 3C12 to begin with, and then initialize them with hops-to-sink values of one via the basestation. After this - I use the insertion feature of the network, by turning 3C6F on after the other two have been initialized. The two nodes with hops-to-sink value one will now tell 3C6F that its current place in the network could place it at a hops-to-sink value of two - which 3C6F will have to accept, because the basestation reply has been turned off. Thus initializing the network of figure 6.1 in an area small enough to video. In theory, this should work just as well with a larger area at your disposal, and with a larger amount of nodes in the WSN.

6.3 Maintenance

In order to maintain the WSN correctly, it needs to recognize when nodes fall in and out of each other's radio range. As I have stated earlier, some problems have occurred here which has not been solved as of now. In short, the problem occurs when a node that has been removed from the WSN tries to reenter the same area, i.e. regain its original neighbors - this happens due to the fact that I have chosen point-to-point connection as my means of communication. The essence of the error is that a new point-to-point connection is established each time a node has to communicate with its neighbor. The reason for this is that this way I am able to detect when that neighbor is out of hearing range. Because then the point-to-point connection will fail to establish. But before it fails to establish, it apparently thinks that the connection is open, and because the connection is not established, no Connection object is created, and thus I cannot shut down the connection because I have no object to perform a close operation on. The result of this, is that the next time the node with the IEEE address of the failed point-to-point connection tries to establish a new connection, an exception is thrown, stating that I cannot have two point-to-point connections on the same port to the same address. This is a problem yet to be solved.

If we overlook the scenario described above - the program does in fact support insertion of nodes and removal of nodes. As explained in the previous section called *Initialization*, inserting a node is as simple as turning it on - it is shown in the video [2]. In appendix B I have added the output for this video. As you can see in the video, I press the right button on the Sun SPOTs several times. Whenever I press this button, data is sent to the desktop application from the Sun SPOT I pressed. If you watch the video, you will see that I press the right button of the two nodes with hops-to-sink one immediately after initializing them. The data I received and I am interested in is shown in tables 6.2 and 6.1

Neighbour 1 is: 0014.4F01.0000.44FC
Neighbour 2 is: 0014.4F01.0000.2FBC

Table 6.1: Data from 3C12

Neighbour 1 is: 0014.4F01.0000.3C12
Neighbour 2 is: 0014.4F01.0000.2FBC

Table 6.2: Data from 44FC

The interesting part of this data is the parts stating the neighbors. We can see at this point they both contain two neighbors, namely each other and the basestation. I

then send a package from 3C12 which is irrelevant, after which I insert 3C6F. This is immediately given the hops-to-sink value two, and should now have the two neighbours of 44FC and 3C12. In order to show this, I press the right button of 3C6F and the output is shown in table 6.3

Neighbour 1 is: 0014.4F01.0000.3C12
Neighbour 2 is: 0014.4F01.0000.44FC

Table 6.3: Data from 3C6F

We see that the two neighbors are indeed 44FC and 3C12. After this I send some packages that show 3C12 is able to send packages that will reach the sink. I then shade a node, this is not currently relevant. But at the 1:41 mark, I again press the right button of 44FC, which then shows table 6.4

Neighbour 1 is: 0014.4F01.0000.3C12
Neighbour 2 is: 0014.4F01.0000.3C6F
Neighbour 3 is: 0014.4F01.0000.2FBC

Table 6.4: Data from 44FC

Where we can see it has indeed added 3C6F as a neighbor. It would seem that 3C6F has been inserted correctly. Right after this, I remove a node from the WSN by turning it off, after which I immediately create a package from 3C6F. Now because the node I turned off had a lower height (more power left in the battery), 3C6F will probably still have that one listed as the best choice to send a package. But we see that the package is correctly routed through 44FC. The route taken was printed out and I have placed it in table 6.5

Data has reached the sink!
Node number 0 in the travelled path is 0014.4F01.0000.3C6F
Node number 1 in the travelled path is 0014.4F01.0000.3C12
Node number 2 in the travelled path is FAILED
Node number 3 in the travelled path is 0014.4F01.0000.44FC
Node number 4 in the travelled path is 0014.4F01.0000.2FBC

Table 6.5: Trying to route through a removed node

The output is not written perfectly, but we see that the FAILED notification occurred after trying to transmit data to the removed node of 3C12 - which is the place where the point-to-point connection will fail to establish. After sending a few packages, I press the right button again on the two nodes left alive in order to see if they indeed have removed the old node from their list of neighbors. The information is shown in tables 6.7 and 6.6

As we can see, 3C12 is indeed removed from the list of neighbors. Hereby ending the video, where I successfully inserted a node and removed a node.

6.4 User's Manual

It is really easy to set up the program. You attach a basestation to your computer, run the desktop application and then you turn on the Sun SPOTs. In order to initialize the system, you press the Hops to sink button in the GUI on the terminal, the system will

Neighbour 1 is: 0014.4F01.0000.44FC

Table 6.6: Data from 3C6F

Neighbour 1 is: 0014.4F01.0000.3C6F Neighbour 2 is: 0014.4F01.0000.2FBC

Table 6.7: Data from 44FC

set itself up. If you wish to create a package on a node to be routed to the sink, simply press the left button of that node. If you wish to see the hops-to-sink value, press the right button on the node, and the LEDs will turn red displaying a number in binary which is the hops-to-sink value.

In order to use the debugging version, the same rules applies as written above, only this time, all the nodes has to stay within range of the basestation, in order to be able to send data directly back.

To use the Chinese delegation version - you have to change the ports used in the class `Consts.Java` between deployment of source code to each node, consistent with figure 4.1.

Chapter 7

Discussion

As I stated in the introduction, the focus of this thesis was highly susceptible to challenges that could occur during the assignment. This has proven to be an advantageous approach, because unforeseen problems and challenges did occur, which were mainly related to getting the hardware to work as intended. But a lot of the original challenges also presented themselves to be correctly regarded as a challenge. These topics included the programs ability to scale the network in a manner such that the amount of nodes in the network had minimum influence on how well the network actually performed. Another topic was the ability of the WSN to perform maintenance on itself - meaning detecting when nodes would fall in and out of radio range of one another.

During my work on the project, it was not always obvious how to approach a problem. A lot of different ways of handling the occurring challenges were considered, and in retrospect they might not all have been the best way of doing things. I have dedicated the section *The Process* to this issue.

As I have stated earlier, my program is not fully functional at this moment, but I am aware of where the faults are, and I believe that I have some good ideas on how to deal with these faults - which I would have done if I had had the time.

7.1 Scalability

A crucial issue of this thesis has been scalability - the ability for the program to work on a WSN with an unknown amount of nodes. The ultimate goal here would be to make the program work completely independent of the amount of nodes in the network, and the amount of neighbors each node had. But to make the program work like this involves a lot more than what is in the scope of this thesis. Doing this would most likely involve some devices with a more expensive radio in it. It would involve doing extended research on how much data could be transmitted over the air in the same area at the same time, before clashes would occur frequently enough to cause errors. Recognizing the fact that this was not the issues of the thesis, the effort in this area was devoted to generating a protocol, which could attempt to come as close as possible to the WSN not being susceptible to the amount of nodes, with the given Sun SPOTs.

The chosen protocol of point-to-point connection was chosen partly because this protocol does seem to be able to handle a very large network. The fact that I tested it on sixteen Sun SPOTs in the vicinity of each other, which seemed to work fine, tells us that the WSN can be so big that every node can have at least fifteen neighbors. This is a pretty substantial amount of neighbors, considering the fact that the Sun SPOTs have a radio range of approximately ten meters or so when outside. Here it is also important

to note that theoretically the WSN can be extremely large - as long as the amount of nodes within the vicinity of each other remains low enough to avoid loss of data due to high traffic over the air. So the program is actually not susceptible to the physical area that the WSN is stretched out on - it is susceptible to the amount of nodes in the large clusters of the WSN, i.e. the amount of neighbors each node has. When this is clear, it could be interesting to discover the limit on the size of the clusters before having too high air traffic. If this was a known value, you could program the nodes to contain a maximum number of neighbors, and if it actually had more neighbors, it should just ignore them. This way you would risk losing crucial edges between nodes, hereby increasing the possibility of having a cluster of nodes isolated from the sink. But the WSN would be able to support more dense clusters.

7.2 Maintenance

Another important issue of the thesis is the fact that the WSN should be able to maintain its own integrity (detecting when nodes fall in and out of radio range of each other). The approaches to this issue vary a lot from the different protocols presented. The complexity of maintenance was highly susceptible to the chosen protocol, which had the effect, that maintenance was highly influential in my decision when choosing point-to-point connection. If I had chosen a protocol based on a general broadcast, I would have to construct my own handshaking algorithm, and in this algorithm, a notification of failure should be used to indicate when a node has fallen out of radio range. When a node is inserted, the approach does not vary a lot from the different protocols. This calls for a general broadcast from the inserted node when it is turned on.

I have handled the complete removal of a node, or the insertion of an entirely new node, and I have covered it in previous sections of the report. But when moving a node around the network when it is turned on the entire time can be trickier. When I sat down and considered how to approach this, I thought of an approach where the node keeps track of how many neighbors it has lost in the recent past. If this number exceeds a certain value based on the total amount of neighbors, then this could indicate that the node has been physically moved - and should therefore do a general broadcast to establish relations with potential new neighbors. This method would not guarantee that a node would detect a physical moving every time, but it would be energy efficient, and hopefully not generate too much traffic over the air. I also came up with a simpler alternative, namely to do a general broadcast periodically in order to sense potential new neighbors. This method would demand more power usage than the other, and more redundant general broadcasts. If the area where the node is placed has a heavy density of nodes, this could potentially cause an overload in air traffic, and the problem with loss of data due to high traffic could become relevant a lot sooner. I have not implemented any of these approaches, but I would probably try implementing the simpler one (doing a general broadcast periodically), in order to determine whether or not the effect on the air traffic would be too great.

7.3 The Process

During the process of the project, I had to consider how I should allocate my time, which tools I should take time to learn in order to save time in the long run and so on. The Sun SPOTs were a completely new tool to me, and even though I have worked

with Java a lot, I had never worked with Java ME (Java Mobile Edition), which I have learned is a limited version of Java, with a smaller library. So I took my time reading the developer's guide [7] and owner's manual [3]. The Sun SPOTManager also came with some tutorials, which I spend time going through. That turned out to be a great benefit in the long run. I was presented with example source code and different techniques for using the hardware at a basic level in these tutorials. I reckon doing these saved me a lot of time getting up to speed on how to program the Sun SPOTs.

There were a few places where in retrospect I think I could have saved some time in the long run if I had spent some more time on those areas. Debugging is one of these areas, where there are several tools available, both in the Sun SPOTManager and in the NetBeans IDE. Even though I felt like I spent a lot of time reading up on EARP, and getting to understand how EARP worked and could be implemented - I might have been able to prevent some early trouble if I had spent some more time on this subject.

7.3.1 Debugging

Because the Sun SPOTs does not have a display, they cannot directly display where an error has occurred the same way as an IDE is able to do in the console. But I was not completely without any form of developed debugging. One tool which I have mentioned is Solarium, which is a tool able to simulate a number of Sun SPOTs. I took my time learning this tool and found it to be very helpful and I believe I saved time in the long run, learning how to use this tool properly. Because in Solarium, the simulated Sun SPOTs does have a terminal in which they can print out to, so that made some problems a lot easier. As I have mentioned, Solarium had problems when I tried to create a WSN, because it was unable to put Sun SPOTs out of radio range of each other. Furthermore the simulated Sun SPOTs did not always act in the same way as the real physical Sun SPOTs. Due to this, Solarium was very useful in the beginning, but as the project stepped along, Solarium became less and less useful as a debugging tool.

Another way of debugging, is presented in the developer's guide [7, p. 47]. It is explained as a way of getting data, which is printed on the Sun SPOT (by a `System.out.println()`), to be automatically sent to the basestation and printed out in the IDE. I did spend some time trying to learn this tool. But after a while of trying, where I could not get it to work, I concluded that it was not worth putting any more effort into. When I later discovered how much time I actually had to spend on getting these printlines printed in the console where I could read them. I think I should have invested some more time in trying to get it to work - because if the tool could work as they stated, I reckon I could have saved many hours of work on debugging. Of course, I never got the tool to work, so I could not say for certain if it would be worth it, and even if it would perform as I had hoped, but if I have to use the Sun SPOTs another time - I will definitely look more into this debugging tool.

7.3.2 System Model as starting point

In the beginning of the project, I was presented with the protocol under development called EARP. EARP had been implemented in a very thorough simulator, which could also visualize a lot of data behind EARP. I was shown this tool and given a presentation of it, to help me understand how EARP worked. It was a good aid in understanding EARP. This simulator had implemented EARP on the nodes of the simulated WSN. I used this implementation to get me started in implementing EARP on the Sun SPOTs.

Looking back it could have been advantageous to spend more time on looking through the simulated implementation - even though it was only implemented for a simulated environment, so it did not take into account some of the problems this practical implementation faced.

Chapter 8

Conclusion

The objective of this project has been to perform a practical implementation of a protocol known as EARP [5] (Energy Aware Routing Protocol), on a small imbedded system known as a Sun SPOT (Sun Small Programmable Object Technology) [6]. Where a virtual battery was to be created, and a solar collector was to be simulated through the light sensor on the Sun SPOT in order to simulate recharging the virtual battery. One of the motivations in doing this practical implementation has been to prove the theory and simulations of EARP is actually valid. Even though the program is not fully functional, it has been able to illustrate the principle and idea of EARP - which was shown to a delegation of Chinese scholars (see video [1]), along with posters explaining the theory of the work behind it. It has been shown on a small network (seven nodes) that EARP does posses the qualities described in [5].

A demonstration of EARP on a network of 30-50 Sun SPOTs to be videotaped was never achieved. It turned out that creating a WSN where the hop count of each node to the sink was larger than two or three, while the radio strength was set to a degree such that the quality of the data was not too low (and too inconsistent) - would have required a minimum area roughly the size of a football field. Since the Sun SPOTs are physically small, it would have been impossible to videotape a WSN of this size, and still being able to see what was happening on the individual Sun SPOTs.

A lot of challenges were presented during this practical implementation - one of which was debugging. In the enclosed CD, I have added the source code of my application in two versions. One version operates at a basic level, with no possibility of retrieving debugging data or anything. The other version is the debugging version, where data can be retrieved from individual nodes as described in the *Results*. I have also enclosed the version shown to the Chinese delegation, which has to be used in a particular special way, as described in *Results - User's Manual*.

A discovery was made that when Sun SPOTs did a lot of general broadcasts over the air on the same port, loss of data was a big issue. In order to remove this problem, I have considered different solutions and approaches as listed in the chapter *Wireless Sensor Network Setup*. The initialization was worked out to be a handshaking algorithm, where a distance map was created (each node was handed a value stating the minimum amount of hops it had to the sink), and each node created a list of neighbors. After the initialization, I chose to have the Sun SPOTs use their build in protocol of point-to-point connection - Which guarantees deliverance and notification of failure. Each time a node wants to communicate to a different node, a point-to-point connection is established, the data is transferred and the connection is closed. At the moment there is a bug when the connection fails to establish. The result of this bug is that a node

cannot be removed from the network and inserted again at the same place. It can only be removed *or* inserted.

8.1 Known errors

At this point, the program does in fact contain errors, some of which have no real effect on the performance. Here is a list of the known bugs and errors. Note that these are only relevant to the finalized version of the program, not the debugging version or the Chinese delegation version:

- A node cannot be removed from a network, and inserted the same place again - this will cause the program to think that two point-to-point connections is about to be established.
- When turning on the nodes, the nodes try to perform some communication which for unknown reasons fails and throws an exception. This does not have any practical effect on the program, other than pressing the right button on a Sun SPOT before it has been assigned a hops-to-sink value (the yellow LED is turned on at this point), will sometimes not work.
- When the right button is pressed telling the LEDs to depict the hops-to-sink value, and the left button is pressed immediately after (to send a package), the lighting of the LEDs will not behave normally - but it does not have any effect beyond the LEDs.
- The only button that has any effect pressing is the *Hops to Sink*. The others worked on earlier stages of the program, but have been deactivated due to malfunctions later on.

8.2 Further Work

In order to continue work with this project - there are several possibilities listed:

- Remove all the bugs and errors stated in the previous section
- Work on actually shooting a video over a longer period of time in a large network consisting of 30-50 nodes.
- The thread of the application running on the Sun SPOT could be parallelized and prioritized further for optimization.
- Work on using the actual battery and a real solar collector instead of using a virtual battery and the light sensor.
- Looking more into using the quality of the packages as an indicator of when two nodes should be a neighboring pair (the Sun SPOTs have the ability to measure the quality through an RSSI value [7, p. 38])

Bibliography

[1] Kristian Hede. Demonstration of earp. Videoclip, June 2009. Stored on the added CD in the folder named Videos under the name EARP Demonstration.avi.

[2] Kristian Hede. 3 spots creating a wsn. Videoclip, June 2009. Stored on the added CD in the folder named Videos under the name 3SPOTs.avi.

[3] Sun Microsystems. *Sun SPOT Owner's Manual - Blue Release 4.0*. 2008.

[4] http://en.wikipedia.org/wiki/alohanet.

[5] Bernhard Fischer Martin R. Jensen Mikkek K. Jakobsen Marcin Szewczyk. Earp: an energy harvesting aware routing protocol based on directed diffusion. 2009.

[6] http://www.sunspotworld.com.

[7] Sun Microsystems. *Sun Small Programmable Object Technology (Sun SPOT) Developer's Guide*. 2008.

[8] http://en.wikipedia.org/wiki/sun_spot.

[9] Vipul Gupta. Experiments with a solar.powered sun spot. 2009.

[10] Kristian Hede. Source code of the sun spot application, June 2009. Stored on the added CD in the folder named Source Code.

[11] Kristian Hede. Source code of the desktop application, June 2009. Stored on the added CD in the folder named Source Code.

Appendix A

Output from video [1]

Data has reached the sink!
Node number 0 in the travelled path is 0014.4F01.0000.3981
Node number 1 in the travelled path is 0014.4F01.0000.337C
Node number 2 in the travelled path is 0014.4F01.0000.395D
Node number 3 in the travelled path is Basestation
Data has reached the sink!
Node number 0 in the travelled path is 0014.4F01.0000.3981
Node number 1 in the travelled path is 0014.4F01.0000.337C
Node number 2 in the travelled path is 0014.4F01.0000.395D
Node number 3 in the travelled path is Basestation
Data has reached the sink!
Node number 0 in the travelled path is 0014.4F01.0000.2FED
Node number 1 in the travelled path is 0014.4F01.0000.1CFE
Node number 2 in the travelled path is 0014.4F01.0000.3690
Node number 3 in the travelled path is Basestation
Data has reached the sink!
Node number 0 in the travelled path is 0014.4F01.0000.2116
Node number 1 in the travelled path is 0014.4F01.0000.2FED
Node number 2 in the travelled path is 0014.4F01.0000.1CFE
Node number 3 in the travelled path is 0014.4F01.0000.3690
Node number 4 in the travelled path is Basestation
Data has reached the sink!
Node number 0 in the travelled path is 0014.4F01.0000.1CFE
Node number 1 in the travelled path is 0014.4F01.0000.3690
Node number 2 in the travelled path is Basestation
Data has reached the sink!
Node number 0 in the travelled path is 0014.4F01.0000.337C
Node number 1 in the travelled path is 0014.4F01.0000.3981
Node number 2 in the travelled path is 0014.4F01.0000.2116
Node number 3 in the travelled path is 0014.4F01.0000.2FED
Node number 4 in the travelled path is 0014.4F01.0000.1CFE
Node number 5 in the travelled path is 0014.4F01.0000.3690
Node number 6 in the travelled path is Basestation
Data has reached the sink!
Node number 0 in the travelled path is 0014.4F01.0000.3981
Node number 1 in the travelled path is 0014.4F01.0000.2116

Node number 2 in the travelled path is 0014.4F01.0000.2FED
Node number 3 in the travelled path is 0014.4F01.0000.1CFE
Node number 4 in the travelled path is 0014.4F01.0000.3690
Node number 5 in the travelled path is Basestation

Appendix B

Output from video [2]

Demo data from: 0014.4F01.0000.3C12
Augmentation is: 0.0
Node height is: 1.6000022888183576
Old node height is: 1.6000022888183576
Depletion level is: 0.6000022888183576
Best neighbour is: 0.0
Decharging by: 120.0
Neighbour 1 is: 0014.4F01.0000.44FC
Neighbour 1 height is: 0.0
Neighbour 2 is: 0014.4F01.0000.2FBC
Neighbour 2 height is: 0.0
Caught java.io.EOFException while reading sensor samples.
Demo data from: 0014.4F01.0000.44FC
Augmentation is: 0.0
Node height is: 1.9249992370605475
Old node height is: 1.8450012207031232
Depletion level is: 0.9249992370605475
Best neighbour is: 0.0
Decharging by: 120.0
Neighbour 1 is: 0014.4F01.0000.3C12
Neighbour 1 height is: 1.6000022888183576
Neighbour 2 is: 0014.4F01.0000.2FBC
Neighbour 2 height is: 0.0
Data has reached the sink!
Node number 0 in the travelled path is 0014.4F01.0000.3C12
Node number 1 in the travelled path is 0014.4F01.0000.2FBC
Demo data from: 0014.4F01.0000.3C12
number 0 routed through 0014.4F01.0000.3C12
number 1 routed through 0014.4F01.0000.2FBC
Caught java.io.EOFException while reading sensor samples.
Demo data from: 0014.4F01.0000.3C6F
Augmentation is: 0.0
Node height is: 2.6000022888183576
Old node height is: 2.6000022888183576
Depletion level is: 0.6000022888183576
Best neighbour is: 0.0

Decharging by: 120.0
Neighbour 1 is: 0014.4F01.0000.3C12
Neighbour 1 height is: 0.0
Neighbour 2 is: 0014.4F01.0000.44FC
Neighbour 2 height is: 1.6049995422363286
Demo data from: 0014.4F01.0000.3C6F
number 0 routed through 0014.4F01.0000.3C6F
number 1 routed through 0014.4F01.0000.3C12
Data has reached the sink!
Node number 0 in the travelled path is 0014.4F01.0000.3C6F
Node number 1 in the travelled path is 0014.4F01.0000.3C12
Node number 2 in the travelled path is 0014.4F01.0000.2FBC
Demo data from: 0014.4F01.0000.3C6F
number 0 routed through 0014.4F01.0000.3C6F
number 1 routed through 0014.4F01.0000.44FC
Data has reached the sink!
Node number 0 in the travelled path is 0014.4F01.0000.3C6F
Node number 1 in the travelled path is 0014.4F01.0000.44FC
Node number 2 in the travelled path is 0014.4F01.0000.2FBC
Demo data from: 0014.4F01.0000.3C6F
number 0 routed through 0014.4F01.0000.3C6F
number 1 routed through 0014.4F01.0000.3C12
Data has reached the sink!
Node number 0 in the travelled path is 0014.4F01.0000.3C6F
Node number 1 in the travelled path is 0014.4F01.0000.3C12
Node number 2 in the travelled path is 0014.4F01.0000.2FBC
Caught java.io.EOFException while reading sensor samples.
Demo data from: 0014.4F01.0000.3C6F
Augmentation is: 0.0
Node height is: 5.275001525878906
Old node height is: 5.275001525878906
Depletion level is: 3.2750015258789067
Best neighbour is: 1.7500000000000007
Decharging by: 120.0
Neighbour 1 is: 0014.4F01.0000.3C12
Neighbour 1 height is: 1.7500000000000007
Neighbour 2 is: 0014.4F01.0000.44FC
Neighbour 2 height is: 7.180000305175781
Demo data from: 0014.4F01.0000.3C6F
number 0 routed through 0014.4F01.0000.3C6F
number 1 routed through 0014.4F01.0000.3C12
Data has reached the sink!
Node number 0 in the travelled path is 0014.4F01.0000.3C6F
Node number 1 in the travelled path is 0014.4F01.0000.3C12
Node number 2 in the travelled path is 0014.4F01.0000.2FBC
Demo data from: 0014.4F01.0000.3C6F
number 0 routed through 0014.4F01.0000.3C6F
number 1 routed through 0014.4F01.0000.3C12
Data has reached the sink!

Node number 0 in the travelled path is 0014.4F01.0000.3C6F
Node number 1 in the travelled path is 0014.4F01.0000.3C12
Node number 2 in the travelled path is 0014.4F01.0000.2FBC
Caught java.io.EOFException while reading sensor samples.
Demo data from: 0014.4F01.0000.44FC
Augmentation is: 0.0
Node height is: 12.084999084472658
Old node height is: 12.084999084472658
Depletion level is: 11.084999084472658
Best neighbour is: 0.0
Decharging by: 120.0
Neighbour 1 is: 0014.4F01.0000.3C12
Neighbour 1 height is: 2.0850028991699237
Neighbour 2 is: 0014.4F01.0000.3C6F
Neighbour 2 height is: 12.084999084472656
Neighbour 3 is: 0014.4F01.0000.2FBC
Neighbour 3 height is: 0.0
Demo -3 data from: 0014.4F01.0000.3C6F
exeption caught: com.sun.squawk.util.SquawkHashtable$HashtableEnumerator@54
Demo -3 data from: 0014.4F01.0000.3C6F
exeption caught: java.lang.IllegalArgumentException: Attempt to open connection twice for Output to 0014.4F01.0000.3C12 on port 124 Demo data from: 0014.4F01.0000.3C6F
number 0 routed through 0014.4F01.0000.3C6F
number 1 routed through 0014.4F01.0000.3C12
number 2 routed through FAILED
number 3 routed through 0014.4F01.0000.44FC
Data has reached the sink!
Node number 0 in the travelled path is 0014.4F01.0000.3C6F
Node number 1 in the travelled path is 0014.4F01.0000.3C12
Node number 2 in the travelled path is FAILED
Node number 3 in the travelled path is 0014.4F01.0000.44FC
Node number 4 in the travelled path is 0014.4F01.0000.2FBC
Demo -3 data from: 0014.4F01.0000.44FC
exeption caught: com.sun.squawk.util.SquawkHashtable$HashtableEnumerator@3e
Demo -3 data from: 0014.4F01.0000.3C6F
exeption caught: com.sun.squawk.util.SquawkHashtable$HashtableEnumerator@55
Demo -3 data from: 0014.4F01.0000.44FC
exeption caught: com.sun.squawk.util.SquawkHashtable$HashtableEnumerator@3f
Demo data from: 0014.4F01.0000.3C6F
number 0 routed through 0014.4F01.0000.3C6F
number 1 routed through 0014.4F01.0000.44FC
Data has reached the sink!
Node number 0 in the travelled path is 0014.4F01.0000.3C6F
Node number 1 in the travelled path is 0014.4F01.0000.44FC
Node number 2 in the travelled path is 0014.4F01.0000.2FBC
Caught java.io.EOFException while reading sensor samples.
Demo data from: 0014.4F01.0000.3C6F
Augmentation is: 0.0
Node height is: 15.790000915527346

Old node height is: 15.850002288818358
Depletion level is: 13.790000915527346
Best neighbour is: 12.790000915527344
Decharging by: 120.0
Neighbour 1 is: 0014.4F01.0000.44FC
Neighbour 1 height is: 12.790000915527344
Demo data from: 0014.4F01.0000.44FC
Augmentation is: 0.0
Node height is: 11.88500213623047
Old node height is: 12.059997558593748
Depletion level is: 10.88500213623047
Best neighbour is: 0.0
Decharging by: 120.0
Neighbour 1 is: 0014.4F01.0000.3C6F
Neighbour 1 height is: 15.850002288818358
Neighbour 2 is: 0014.4F01.0000.2FBC
Neighbour 2 height is: 0.0
Caught java.io.EOFException while reading sensor samples.