ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Semester Thesis

Link Layer Measurements in Wireless Sensor Networks

Roger Kehrer
rkehrer@student.ethz.ch

Prof. Dr. Roger Wattenhofer
Distributed Computing Group

Advisor: Pascal von Rickenbach

Dept. of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich
Summer 2005

Contents

Chapter 1

Introduction

1.1 Motivation

Recent advances in wireless networking and microelectronics have led to the vision of sensor networks consisting of hundreds or even thousands of cheap wireless nodes - each equipped with some memory, a processor, a power unit, and a short range radio - covering a wide range of application domains. The most popular network model (a.k.a. Unit Disk Graphs[2]) assumes that two nodes are only able to communicate directly with each other if they are within a certain distance - the transmission radius of the wireless device. Although this network model has led to many theoretical results it seems to be too idealistic in practice.

We therefore seek for more realistic network models that are more adequate to represent real network conditions. We are particularly interested in metrics such as packet reception rate with respect to distance or radius coverage in regard to radio transmission power setting.

1.2 Assignment

The goal of this thesis is to set up a sensor node test bed using mica2 sensor nodes and to develop a code framework to facilitate subsequent link layer measurements in the test bed. Another important aspect of this thesis is the analysis of the gathered measurements in order to obtain new insights for a more realistic network model.

We partitioned the work in three main parts.

- The first part was to become acquainted with the environment and the sensor nodes.
- The second part was to set up a sensor node test bed to measure the main factors that affects the sending range of the nodes.
- In addition, the third task was an additional task to find out how much a sending Node disturbs the communication of two near nodes.

1.3 Overview

The rest of this Document is structured as follows. In Chapter 2 we characterize the design and the basic ideas. We then present the concrete implementation of the transmission range measurements and the realization of the interference measurements in Chapter 3. The subsequent chapter is about the results of our real world measurements. In Chapter 5 we present an outlook to possible future work and an overview over the known open problems. Finally in Chapter 6 the work is concluded by summarizing the derived results and giving a personal comment.

Chapter 2

Basics and Design

In this chapter we present the design of the framework for transmission range measurements. Furthermore the design of the jamming measurements is shown and a quick summary of the basic hard- and software is given.

2.1 Mica2 Motes

In this thesis we use mica2 sensor nodes. Mica2 motes are wireless sensor nodes produced by Crossbow Technology[3]. They consist of a processor and some memory, a radio unit and an interface to connect it to a so called programming board. The programming board is the interface between the motes and the ethernet. To analyze a packet on a computer connected to a node through the ethernet, you have to run an application called SerialForwarder on the computer. This application forwards the incoming messages to the application and converts them to java objects. Figure 2.1 depicts a mica2 node and a programming board. The first part of getting into the environment means to get into the operating system and the programming language of the mica2 motes.

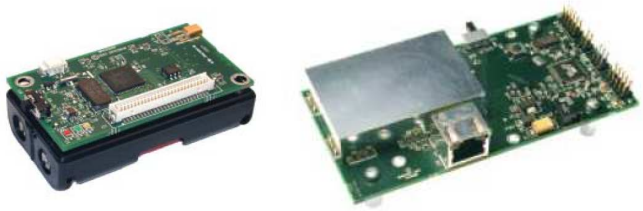

Figure 2.1: A mica2 mote and a programming board. Note that the two figures are not in the same scale and the programming board is about four times bigger than a node.

The operating system running on the mica2 motes is TinyOS[7], an open source operating system designed for wireless embedded sensor nodes that have very limited resources. It has been developed by the computer science division at the University of California in Berkeley[4].

The programming language of the mica2 motes is nesC[6]. NesC is an extension to the C programming language designed to embody the structuring concepts and execution model of TinyOS[1]. One of the major extensions compared to the C programming language is the concept of bidirectional interfaces. These are interfaces which specify a set of functions to be implemented by the interface provider (called commands) and a set of interfaces to be implemented by the interface user (called events). Figure 2.2 shows the concept of the bidirectional interfaces.

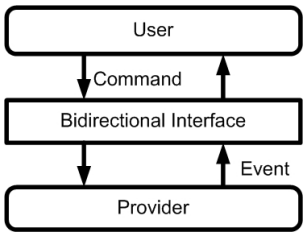

Figure 2.2: The Concept of the bidirectional interface. It connects a user and a provider and there are two method types. The command is implemented by the provider and is called by the user, whereas the event is called by the provider and must be implemented by the user. You can only use the interface as a user if you implement the specified events.

2.2 Transmission Range Measurements

As described in Section 1.2, the task was to find out the transmission range of a node. To get this, we wanted to measure the percentage of packets a node receives in various distances from the sender with various transmission power settings. For this purpose, we decided to use the following experimental setup.

As we wanted to measure the sending range not only in one dimension but in two, we decided to distribute 49 nodes on a regular grid with the sending node in one corner. The experimental setup is shown on Figure 2.3. For the sake of simplicity the nodes were consecutively numbered. This setup provides only one quadrant of the area around the sender, but because of the limited number of available nodes, we could not provide a setup where all four quadrants were covered. Nevertheless, with this setup it is easy to make measurements for all four quadrants, without even moving the nodes. This can be done by rotating all nodes by 90 degrees three times. In doing so, we can do the measurements for all quadrants one after the other and merge them together to a complete result.

As we wanted to measure the percentage of packets received by the nodes distributed on the grid, we had to design a protocol to send messages over the grid and, once the measurements are finished, collect the results again.

First, we did some measurements to find out if the nodes on the grid affect each other. After some tests we found out, that this is not the case and thus

[1]To get into TinyOS and nesC we recommend you to go through the TinyOS tutorial at http://www.tinyos.net/tinyos-1.x/doc/tutorial/.

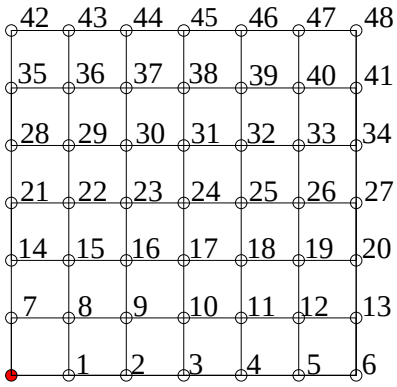

Figure 2.3: The experimental setup of the transmission measurement, where the red node is the sender and the others are receivers. The numbers are the id's of the nodes.

we could broadcast the packets from the sender and it wasn't required to send them sequentially to one node after the other.

So we resolved upon the following protocol:

- Send a packet from the connected computer to the sender to awake it with the right parameters.
- Send a number of packets from the sender to the nodes on the grid.
- Send a packet to each node to get the number of received packets.
- Send the results of each node back to the computer and display the results.

The first two steps consist of sending a packet through the ethernet and invoking the node connected to the programming board, the sender. To minimize the work to be done on the sender we decided to swap out the work to the computer. For this purpose we just send one InitMsg from the computer to the Sender and broadcast the IncMsg just once. This means, that the number of packets to send can be controlled by the computer. This makes things easier to control and to debug. To guarantee, that just one message is sent at one time, the computer waits with the next message until the last was acknowledged.

The third and the fourth steps are again started by the controller after having invoked all the broadcasts. This is done by sending a ResultsMsg to the sender. The sender starts collecting the results from the nodes and returns them back to the controller. The controller gets the results from the sender one after the other and stores them to a file. It terminates, if it has collected the results from all the nodes. Figure 2.4 shows the protocol of the transmission range measurements.

As we are not working in a perfect world and we are expecting high packet loss rates, it is possible, that the transmission protocol of the EchoMsg between the sender and the satellites fails. This is fatal, as it will imply that no packet has reached the satellite. This is the reason why a retransmit protocol was

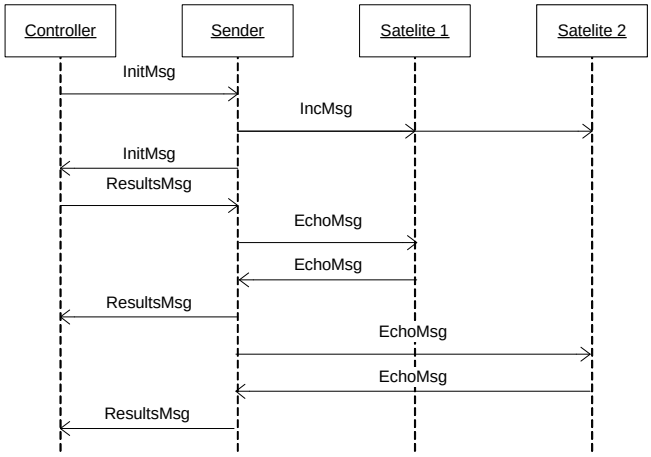

Figure 2.4: The protocol of the transmission range measurement

implemented. If the sender doesn't get an answer on the EchoMsg from a satellite, it retransmits the message several times. Due to the time it consumes to retransmit the packet, it is important to find a good retransmission ratio. The formula for the possibility of a successful transmission is calculated by the following formula: $f(x) = 1-((1-p^2)^x)$ where p is the probability of one message being transmitted and x is the number of retransmissions. Figure 2.5 shows the success probability of the transmission versus the number of retransmits done at different message success rates. In order to get a sufficiently high transmission

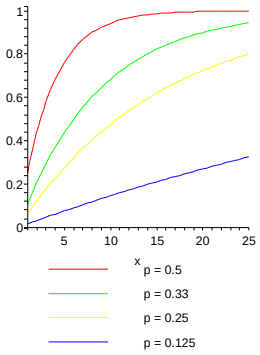

Figure 2.5: The x-axis denotes the number of retransmissions and the y-axis the possibility of a successful transmission of the retransmit protocol respectively. The miscellaneous colors show different success rates of one message.

rate, we decided to choose 20 retransmissions as a reasonable rate. Thus we got even for small transmission rates, like 25%, a success rate of over 60%. In other words, this means, that if we send the ResultsMsg with the same transmission power as the IncMsg, the possibility that the protocol returns for a node a value

smaller than i.e. 10% is very small. Instead of getting the right value, the retransmission protocol fails and the returned value is zero. Although this is not correct, we decided that this is not crucial. If 20 attempts to transmit the result fail, the link is very unreliable and thus it is not wrong to assign it a zero value.

2.3 Interference Measurements

The third task was to find out how much a sending node disturbs the communication between two other nodes. What we wanted to measure was the transmission rate between sender and receiver when the distance between the receiver and the interceptor grows.This was done with several transmission power levels. So we had to implement a sender, a receiver and an interceptor node sending all the time.

Since the radio stack of the nodes implements a collision detection on the MAC-layer to prevent channel utilization if it is already occupied, we had to disable this functionality for the sender and the interceptor. In TinyOS the collision detection is done in the CC1000RadioIntM.nc file by updating a squelch value from time to time and compare it to the actual overhead signal. In the current implementation a simple check decides if the received byte is filled with information or if it is just background noise. When a packet should be sent it checks if the noise ratio is significantly under a defined level. If this is the case, the medium is considered free and the packet is transmitted. Otherwise, after trying several times with a random waiting time in between, the packet is dropped and a fail is returned. As we didn't want the communication to fail because of the sender considering the medium to be occupied but because of the consciously superposition of the two signals at the receiver, the sender as well must not check the channel to be free. Under this circumstances, the receiver gets the two superposed signals and we are able to determine at which distance of the interceptor to the receiver the communication between the latter and the sender is possible.

Chapter 3

Implementation

In this chapter we first give an introduction to the communication stack in TinyOS. Furthermore we present the implementations for the transmission range measurements and the interceptor. The layout of all mentioned messages is described in Appendix A.1.

3.1 Communication in TinyOS

In TinyOS the transmission of a message is done by wiring the component called GenericComm.SendMsg[MessageType] into the code. To send a message, you then call GenericComm.SendMsg(Msg). This method is asynchronous and returns true or false depending on the success of starting the sending process. Once the message has been physically sent, an event called sendDone is triggered. Figure 3.1 shows the model of sending messages in TinyOS. In our protocols there is never more than one message sent at one time and thus we can use these methods to implement it.

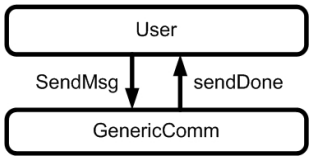

Figure 3.1: The model of sending of messages in TinyOS. The user calls the SendMsg command and after the message has been sent an event called sendDone is triggered.

The mica2 motes are equipped with a Chipcon[1] radio module. The sending power of the motes is set by wiring the radio control module into the code and then calling SetRFPower(value) on it. It is used to set the power of the radio module to a given value that can range from 0 to 255. A value of 0 does not mean, that the power level is set to zero, but that it is lowered by 20 dBm. On the other hand 255 means that the sending power is raised by 5 dBm. For a detailed explanation look at the Chipcon SmartRF CC1000 data sheet[5].

3.2 Transmission Range Measurements

As described in Section 2.2 the implementation consists of three main parts: The controller, the sender and the satellites. In the following sections we will address important implementation issues of all three applications separately.

3.2.1 The Controller

The source code for the controller is written in java and is run on a computer connected to the sender via ethernet and the programming board[1]. The controller is basically a MessageListener[2] that is listening to the messages sent by the sender. When receiving an InitMsg it checks, if it has sent the desired number of InitMsg messages. If this is the case, it sends a ResultsMsg or otherwise it sends an other InitMsg. After having received the ResultsMsg messages from all the nodes, it stores the results and terminates. Figure 3.2 depicts the flow diagram of the controller.

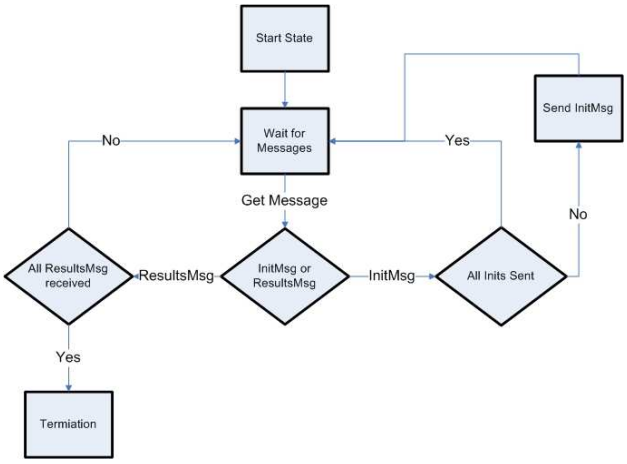

Figure 3.2: The Flow diagram of the controller.

3.2.2 The Sender

The implementation of the sender is the most complex part of the three. As described in Section 3.1, a sendDone event is triggered after every successful transmission of a message. We used the sendDone method to implement the protocol. This is suitable for our problem, because you can place the sending of the next message in the sendDone event handler and thus come up with the sequential nature of the protocol. A list of all methods of the sender and what they do in detail is displayed in Appendix A.3.

The first thing to do is to broadcast the IncMsg. When receiving an InitMsg from the controller, we broadcast the message to the satellites. Here we use the

[1] See Section 2.1 how the connection between a node and a computer is done

[2] MessageListener is a Java interface specified by TinyOS to Listen for incoming messages. See Appendix A.2 for details.

sendDone triggered by the broadcast to send the InitMsg back to the controller.

After having sent all the InitMsg messages the results are collected. This process is started by a ResultsMsg from the controller. An EchoMsg is then sent to the first satellite. This triggers the start of the timer for the retransmission.

When getting an EchoMsg back from a satellite the results are sent back to the controller. We finally send an EchoMsg to the next node and iterate over all nodes by increasing node id's.

If the transmission of the EchoMsg failed, the timer fires without having received the result from the current node. In this case the EchoMsg is resent up to 20 times. If another timeout appears, it returns 0 as a result for the current node under consideration and the timer is stopped. The flow chart of the sender is shown in Figure 3.3.

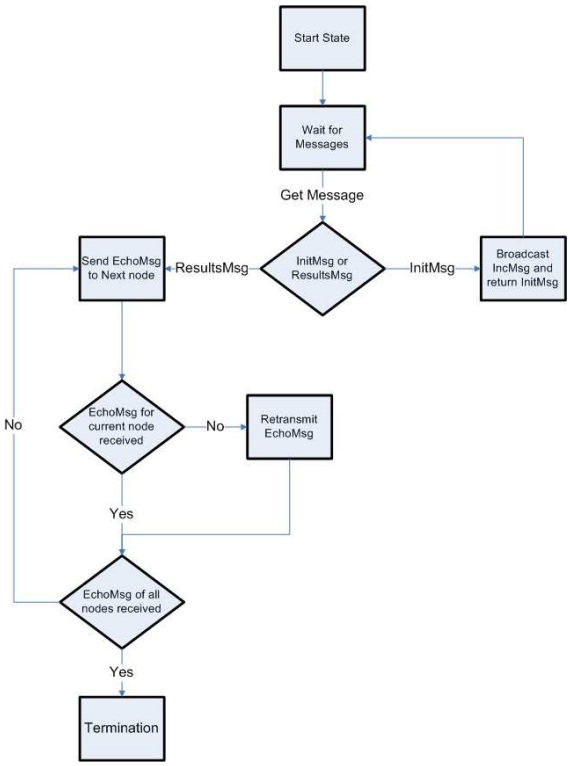

Figure 3.3: Flow diagram of the sender.

The radio power level is given to the sender as a parameter in the InitMsg. Thus, the sender sets the radio power to the desired value before sending the IncMsg to the satellites and resets it to the maximum power level to collect the results from each node to get the highest possible success rate for the retransmission protocol.

3.2.3 The Satellites

The satellites are programmed to wait for two types of messages: The IncMsg and the EchoMsg. Upon receiving an IncMsg it increases the local counter for the received messages of this pass. If it receives an EchoMsg it sends the counter back to the sender. The sending power of the satellites is always set to the maximum power level ,that is, 255.

3.3 Interference Measurements

In Section 2.3 the problem with the collision detection mechanism of TinyOS in this experiment was described. To get around this, we first tried to find an interface, where we could manipulate the stack at the MAC-layer and thus circumvent the collision avoidance checks. But we did not manage to find an interface and a fitting configuration which provides this functionality. Therefore the idea came up to modify the MAC-layer implementation directly.

3.3.1 The Sender

For the sender we had to disable the test for the free medium. In the MAC-layer protocol an event is fired when a test probe from the medium is ready. The parameter of this event is the current RSSI value which is high if the channel is free and low otherwise. This value is then compared to the last probes to decide if the medium can be accessed safely without collision. We simply removed this check to attain our goal. The precise implementation how to disable the collision detection in the MAC-layer is shown in Appendix A.4.

As a sample application we used a simple dummy application that sends the output of an integer counter over the radio and displays the three lowest bit of the current value with the leds.

3.3.2 The Interceptor

The interceptor application node also has to ignore the check for the free medium. Thus, the same changes to the MAC-layer are made as at the sender. Additionally, we made another change: If we sent one message after the other, there would always be a gap between two messages, where the medium is not occupied. To occupy the medium permanently, we tried to have no interruptions in the sending process. We managed this by hacking the state machine of the MAC-layer implementation. Instead of sending a message, it simply sends the same byte over and over again to fill the channel. In Appendix A.5 the concrete changes are described.

3.3.3 The Receiver

As receiver we used the TosBase application that comes with the standard TinyOS installation. It simply relays the incoming messages from the radio to the connected computer. On the computer the received packets are analyzed and the packet reception rate is calculated.

Chapter 4

Results

Initially, we wanted to find a network model for the reception rate dependant on the distance and the relative position to the sender. During our research we found out, that the transmission ratio is not only dependent on the distance from the sender but on several other characteristics.

One factor that influences the transmission range dramatically is, if there is a line of sight between the communicating nodes. This was crucial especially for the first part of our experiments, where we tested the framework inside of buildings. At this point we also found out that it is important for the transmission ratio that the position of the antennas are parallel and that they are standing upright. It is also important that the nodes are standing free. For example, a node standing very close to a wall had lot more problems receiving packets than one in the middle of a room.

But the most significant factor for the transmission range was the height over ground. We found out, that the transmission range of a node lying on the floor was approximately 30 meters. By raising them to 2 meters over ground the sending range increased to over 100 meters. This fact forced us to take a decision about our measurements. We thought it would be the most realistic case to make the measurements when placing the nodes on the ground. Another point is, that it would be easier to find a place to make the measurements with a sending range up to 30 meters.

4.1 Transmission Measurements

4.1.1 Testbed Settings

We did the measurements on a football field. But as the grass disturbed the sending range, we decided to raise the nodes at least a bit over ground by placing them on turned flowerpots as shown in Figure 4.1.

Our results are displayed in Figure 4.2 and 4.3. We sent 5000 packets to each node and recorded the number of packets received. In all the pictures a square represents a node. We repeated all the measurements at least three times and analyzed the results. As the results were very similar in each measurement, we decided that we can merge them together by simply take the average of the individual measurements.

Figure 4.1: One node placed on a turned flowerpot in the grass.

Due to the limited transmission range of the nodes when placed on the floor, it was sufficient to have a square with a side length of 30 meters. With the 49 nodes[1] the distance between two neighboring nodes are always 5 meters.

4.1.2 Measurement Results

The three figures in Figure 4.2 show the measurements given three different radio power levels. With the sending power increasing from left to right the range of the sender also increases. But it does not increase as much as we expected. The sending range with sending power set to 0 is approximately 15 meters whereas with 128 sending power it is about 20 meters and for 255 circa 25 meters. We expected the influence of the sending power to be more severe specially when seeing that the height over ground increases the sending range by a multiple.

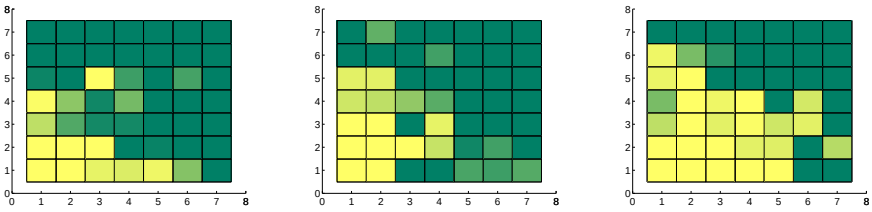

Figure 4.2: The results of the transmission measurements at different power levels. The leftmost graphic is the measurement with radio power level 0, the one in the middle is with level 128 and the rightmost with 255. In all three graphics, the sender is in the left lower corner at coordinate (1/1).

As early tests showed, it is hard to find a place where a node receives just a part of the sent packets, we expected the transmission ratio to be with high probability either 0% or 100%. The measurements showed that the transmission ratio behave in the expected manner. There are just few values lying between 0% and 100%. This somehow correlates to the unit disk graphs, as in this model the nodes are either fully reachable or out of range.

Figure 4.3 shows the transmission rate of a node in all four quadrants. As mentioned in Section 2.2 the measurements are not really done in all four quad-

[1] See Section 2.2 for the details of the model.

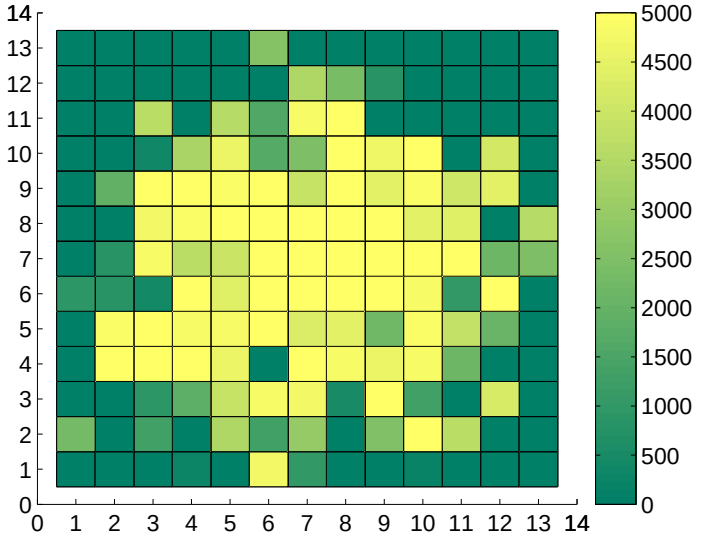

Figure 4.3: The measurements with power level set to 255 of all four quadrants pasted together to one picture. In this configuration the sender is placed in the middle of the area at coordinate (7/7). The colors show the number of packets received by each node according to the colorbar on the right side.

rants concurrently, but the quadrants have been simulated by rotating the nodes and thus changing the orientation. Although all the measurements have been made on the same underground the rates differ significantly for the four quadrants. We do not have an explanation for this phenomenon. We expected the measurements to be more or less equal for all four quadrants and instead they differed noticeable. This phenomenon was even reproducible as the miscellaneous measurements for the quadrants resembled each other.

As mentioned above, the retransmission protocol can fail. This will result in a zero value at one node. In our graphics these are shown as a green quare as the node at coordinate (6/4) in picture 4.3.

4.2 Interference Measurements

4.2.1 Testbed Settings

As described above, the sending range heavyly depends on the hight over ground. To keep the dimension of the measurements at a small scale, we decided to place the nodes on the ground. In doing so, we could make the measurements within just a few meters.

We decided to place the receiver, the sender and the interceptor in a row. The distance between receiver and sender was thereby fixed to five meters. The interceptor was shifted on this line to several positions and we measured the percentage of packets transmitted from the sender to the receiver. The testbed setting is shown in Figure 4.4.

Figure 4.4: The testbed settings of the interceptor measurements. The blue node is the receiver, the green one is the sender and the red one the interceptor. The distance between the sender and the receiver is five meters and the position of the interceptor is variable.

4.2.2 Measurement Results

As described above, the three nodes are arranged in a line with the interceptor node having a variable position. Figure 4.5 shows the percentage of packets received by the sender versus the distance between the interceptor and the receiver. The sending power of the sender was set to 128 whereas the sending power of the interceptor was modified.

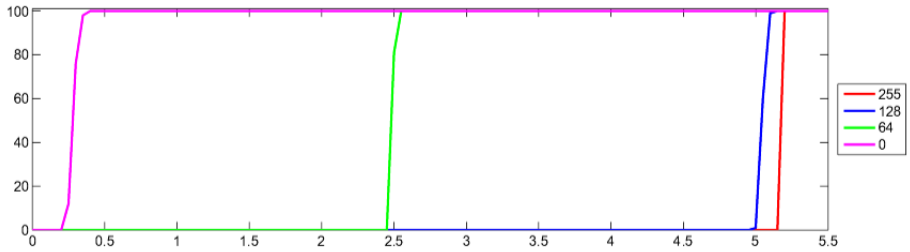

Figure 4.5: The results of the interceptor measurements. On the x-axis the distance from the receiver to the sender is displayed and the y-axis shows the percentage of packets transmitted from the sender to the receiver. The colors show the various power levels of the interceptor.

At all four sending power levels, the step from 0% to 100% is within a few centimeters. This means, that at a certain position, if we moved the interceptor node a few centimeters further away from the receiver, the transmission rate of the sender increased significantly. As one can see, this step is at diverse positions when intercepting with various sending power levels. If intercepting with 128 the step is at 5.05 meters from the receiver. This means, that the interceptor just disturbs the communication when placed closer to the receiver than the sender.

With sending power 64, the step is at 2.45 meters and with 0 sending power it is at 0.25 meters. This means that a node, placed in a distance of more than 0.25 meters from the receiver and sending with power level 0, is not heard by the receiver, if you place a node with sending power 128 in 5 meters distance. The signal sent by the interceptor is then interpreted as noise and hence ignored. It is interesting that the step for power level 255 is not far beyond the position of the sender. This means, that you cannot intercept the communication between two nodes from a distance greater than the distance between the sender and the receiver even with a far higher power level than the sender.

Chapter 5

Outlook

Even though our framework worked good and delivered us some insight on the sending behavior of the mica2 nodes, some problems have arisen during the use of the framework which are not yet solved.

5.1 Open Problems

As mentioned above, the problem with timeouts in the retransmission protocol exists. Although this is not crucial, one could invest some time to solve it. One possibility would be to implement the data collection protocol as a multi-hop protocol. This can be done by implementing a flooding protocol and thus broadcasting the request by the sender and then letting every node who got the request for the first time rebroadcast it. This is not efficient but should be sufficient for our needs. Alternatively one could divide all the nodes into groups and declare a special node to be the manager. The request could then be sent to the representant and it would forward it to the other nodes of its group. However this does not solve the problem of one node being completely out of range of the other nodes. Additionally, the problem only arises when sending the IncMsg with power level 255. Otherwise the nodes receiving just a small percentage are easily reached by the sender with the EchoMsg because these are always sent with full power.

Another problem arises when doing multiple measurements without rebooting the satellites. This means that the nodes have to go back into the initial state after having transmitted the results. But after a failure of the sending of the EchoMsg it can not reset its state because this could lead to a malfunction of the retransmit protocol. If the sending of the ResultsMsg back to the sender failed, it would reset and thus transmit a zero back to the sender in a later step of the retransmission protocol. This means, that the problem is to find the right moment to reset. We now solved this problem by resting the nodes when the first IncMsg is received after having sent the EchoMsg back to the sender. This works most of the time. But there are cases, where this method fails. For example if the measurement is done with low power it can happen that a node, after having returned the EchoMsg for one round, does not get an IncMsg from the next round and then receives once again an EchoMsg. It then returns the value from the last measurement without resetting in between. We resolved this

problem by making measurements with low power level first and then increasing the power. Alternatively one can reboot the nodes after each round which is not very convenient. Another way would be to introduce a counter for the rounds in every IncMsg and ResultsMsg and a node would only reboot when receiving an IncMsg with a higher counter than the previous ResultsMsg. The best solution would be to use a reset message transmitted via multiple hops. This would reduce the chance of this failure but can not eliminate it completely.

5.2 Future Work

There are several things one could do in the future. On the one hand one could buy gps sensors for the nodes. This would help getting around the rigid arrangement of the nodes because the nodes could send their positions back to the sender. Using this, the nodes do not need to be in a predefined place but can be placed arbitrary.

On the other hand one needs a flat area that is big enough to do the measurements. The problem is that the only way to get the power for the programming board was via an electric cable and a power outlet. This makes it hard to find a convenient place to make the measurements. It would be a great benefit to have a portable power supply for the programming board to make the measurements independent from a power outlet.

Chapter 6

Conclusion

6.1 The Results

I think that the initial tasks of this thesis are met in a good manner, as we have learned a lot about the behavior of the mica2 nodes. We have seen that the sending ratio not only depends on the distance as previously expected, but there are also other criterions. I did not know that, i.e., the height over ground is so important for the sending range. On the other hand, the sending power did not influence the sending range as much as we expected.

As the sending range of the mica2 motes is nearly a circle and because of the transmission rates being either 0% or 100%, we can say, that the unit disk graph model is applicable if we do not have obstacles.

6.2 Personal Experience

The work on this thesis was very interesting. It was a new experience to work with an embedded system and his very limited debugging possibilities. The three leds on the nodes don't give a lot of information on the system state and it took a long time to find out how one can debug more or less comfortable. We found out, that the best way to debug the system is to use a second programming board with a TosBase application running on it. One can then collect all packets sent on the channel and display them at the connected computer. This crucially simplifies the work.

Appendix A

Implementation Details

A.1 Message Types

The following messages are used by our applications.

- The InitMsg

```
typedef struct InitMsg {
    uint16_t Number;
    uint8_t signalStrength;
}InitMsg;
```

- The IncMsg:

```
typedef struct IncMsg {
    uint16_t Number;
}IncMsg;
```

- The EchoMsg:

```
typedef struct EchoMsg {
    uint16_t Number;
    uint16_t Sender;
}EchoMsg;
```

- The ResultsMsg

```
typedef struct ResultsMsg {
    uint16_t Number;
    uint16_t Received;
    uint16_t Timeout;
}ResultsMsg;
```

A.2 MessageListener

The MessageListener interface provided by the net.tinyos.message package.

```
package net.tinyos.message;

 // MessageListener interface (listen to TinyOS messages).<p>

 // An interface for listening to messages built from
 // net.tinyos.message.Message

 // @author  David Gay
public interface MessageListener {
     // This method is called to signal message reception. to is
     // the destination of message m.
     // @param to the destination of the message (Note: to is only valid
     //   when using TosBase base stations)
     // @param m the received message
    public void messageReceived(int to, Message m);
}
```

A.3 Methods of the Sender

```
These are the messages implemented by the sender of the
transmission range measurement framework together with a short
description of each of them.

UARTInitReceive.receive(...) {
    //get an InitMsg over the Ethernet,
    //save the parameters and broadcast the IncMsg.
} RadioIncSend.sendDone(...) {
    //The sending of the IncMsg is done.
    //Transmit an InitMsg back to the controller over the Ethernet.
} UARTReceive.receive(...) {
    //get the first ResultsMsg over the Ethernet and
    //send the first EchoMsg to the satellite with Id 1.
} RadioEchoSend.sendDone(...) {
    //The sending of an EchoMsg is done.
    //Start the timer for the retransmission protocol.
} RadioReceive.receive(...) {
    //Get an EchoMsg over the Radio from a node.
    //Send the result back to the controller and stop the timer.
} UARTSend.sendDone(...) {
    //The sending of a ResultsMsg is done.
    //Send a message to the next available node.
} Timer.fired() {
    //The TimeoutTimer of the retransmission protocol. If it fires,
    //it checks if the current node has already responded. If not
    //it retransmits or considers the node to be unreachable
}
```

A.4 How to disable the Collision detection

To disable the collision detection in TinyOS on the mica2 motes, one has to change the following lines in the CC1000RadioIntM.nc file located in \tos\platform\mica2\. On line 961ff there is the following statement:

```
if ((data > (usSquelchVal + CC1K_SquelchBuffer)) &&
            (initRSSIState == PRETX_STATE)) {...}
```

To disable the check one has to change this line to the following:

```
if (initRSSIState == PRETX_STATE) {...}
```

A.5 How to corrupt the medium permanently

To hack the state machine of the MAC-layer of TinyOS on mica2 motes one should change the following lines in the CC1000RadioIntM.nc file located in \tos\platform\mica2\. On line 641ff there is the following state in the state machine:

```
case TXSTATE_DATA:
    if ((uint8_t)(TxByteCnt) < txlength) {
        NextTxByte = ((uint8_t *)txbufptr)[(TxByteCnt)];
        usRunningCRC = crcByte(usRunningCRC,NextTxByte);
        // Time Sync
        signal RadioSendCoordinator.byte(txbufptr, (uint8_t)TxByteCnt);
    }
    else {
        NextTxByte = (uint8_t)(usRunningCRC);
        RadioTxState = TXSTATE_CRC;
    }
    break;
```

If these lines are changed to the following, the state machine never leaves the sending state and it always sends the same byte.

```
case TXSTATE_DATA:
    TxByteCnt = -1; //it always sends the same byte
    if ((uint8_t)(TxByteCnt) < txlength) {
        NextTxByte = ((uint8_t *)txbufptr)[(TxByteCnt)];
        usRunningCRC = crcByte(usRunningCRC,NextTxByte);
        // Time Sync
        signal RadioSendCoordinator.byte(txbufptr, (uint8_t)TxByteCnt);
    }
    else {
        NextTxByte = (uint8_t)(usRunningCRC);
        //RadioTxState = TXSTATE_CRC; //don't leave the state
    }
    break;
```

Bibliography

[1] Chipcon Homepage.

`http://www.chipcon.com/`

[2] B. N. Clark, C. J. Colbourn, and D. S. Johnson. Unit Disk Graphs. Discrete Mathematics, 86:165177, 1990.

[3] Crossbow Technology.

`www.xbow.com`

[4] Homepage of the Computer Science Division at University of California Berkeley.

`http://www.cs.berkeley.edu/`

[5] Mica2 datasheet.

`http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf`

[6] NesC Homepage.

`http://nescc.sourceforge.net/`

[7] TinyOS Homepage.

`http://www.tinyos.net/`