RSSI Testing

# Introduction

The final step of our thesis was to integrate our work with that of the master students. The subject of their research was dynamic event positioning. This means that when an event occurs at a node (the sensors exceed a certain value), a *mobile node* should be sent to it, to investigate the situation, for example by taking a picture of the environment. To be able to direct this *mobile node* to the correct coordinates, its position should be known. The best solution according to the masters is to use RSSI readings and then triangulate these.

To know the distance between nodes, we need to know the relation between distance and RSS. The purpose of this work was to write an application so that we could gather some data about this relationship. If we want to know the distance to a certain node, we need to know the standard RSSI value for that distance.

The second piece of functionality this application will demonstrate is that we can put the intelligence in the mobile node. In the application of our master students, all the data would get routed to a base station which processes this data, calculates the correct direction for the mobile node to follow from it, and then sends these directions back to the mobile node.

The application we built is merely for testing purpose only, it has too little functionality to be of any practical value, but is has been a helpful tool in helping us collect the RSSI data.

The application actually consists of 2 applications, namely one for the anchor node which detects the event and broadcasts a warning signal, and one for the mobile node which detects this warning and tries to find the anchor node.

# Anchor node

## Functional description

This mode checks it sensors for a threshold, when one of the sensor values exceeds this threshold, it signals a warning to the other nodes.

## Anchornode.H

In this header file we declare some constants and the packet format.

enum {

/\* Default sampling period. \*/

DEFAULT\_INTERVAL = 512,

AM\_OSCILLOSCOPE = 0x93,

MAX\_SENS\_VAL = 1000

};

DEFAULT\_INTERVAL is our sampling interval, MAX\_SENS\_VAL is the threshold for our sensors. If it exceeds this, we start broadcasting.

This is our format of our payload. It contains the id of the mote needing sending, and a bool warning which signals whether there is a problem or not.

typedef nx\_struct oscilloscope {

nx\_uint16\_t id; /\* Mote id of sending mote. \*/

nx\_bool warning; /\* Warning or Not \*/

} oscilloscope\_t;

## AnchornodeAppC

#include "printf.h"

configuration AnchorNodeAppC { }

implementation

{

components AnchorNodeC, MainC, ActiveMessageC, LedsC,

new TimerMilliC(), new DemoSensorC() as Sensor,

new AMSenderC(AM\_OSCILLOSCOPE), PrintfC;

AnchorNodeC.Boot -> MainC;

AnchorNodeC.RadioControl -> ActiveMessageC;

AnchorNodeC.AMSend -> AMSenderC;

AnchorNodeC.Timer -> TimerMilliC;

AnchorNodeC.Read -> Sensor;

AnchorNodeC.Leds -> LedsC;

AnchorNodeC.PrintfControl -> PrintfC;

AnchorNodeC.PrintfFlush -> PrintfC;

}

This is al pretty easy, we add the interfaces for the radio, boot, timers, sensors, leds and the printf library.

Note that we do an include for the printf library.

## AnchorNodeC

Here we will discuss the actual implementation of the program.

### Preprocessor

The necessary header files of course.

#include "Timer.h"

#include "AnchorNode.h"

#include "printf.h"

### interfaces

This is the signature of our applications, here we declare the interfaces we use.

We provide the same interfaces as the one we wire in our app file.

module AnchorNodeC

{

uses {

interface Boot;

interface SplitControl as RadioControl;

interface AMSend;

interface Timer<TMilli>;

interface Read<uint16\_t>;

interface Leds;

interface SplitControl as PrintfControl;

interface PrintfFlush;

}

}

### Variables

implementation

{

message\_t sendbuf;

bool sendbusy;

/\* Current local state - warning, node id \*/

oscilloscope\_t local;

Here we declare the necessary variables. Sendbuf is the buffer of our radio and is standard practice in any application that uses the radio or the uart.

Sendbusy is a guard variable which enforces that only one function has access to the radio at a given time.

Local is where we keep our data, this will be used as the payload for the radio packet

### Events & Calls

// Use LEDs to report various status issues.

void report\_problem() { call Leds.led0Toggle(); }

void report\_sent() { call Leds.led1Toggle(); }

void report\_event() { call Leds.led2Toggle(); }

event void Boot.booted() {

local.id = TOS\_NODE\_ID;

if (call RadioControl.start() != SUCCESS)

report\_problem();

if (call PrintfControl.start() != SUCCESS)

report\_problem();

}

Used for debugging and for signaling that the mote has booted

Some events to start the radio, timer and the printf service.

void startTimer() {

call Timer.startPeriodic(DEFAULT\_INTERVAL);

}

event void RadioControl.startDone(error\_t error) {

startTimer();

}

event void RadioControl.stopDone(error\_t error) {

}

event void PrintfControl.startDone(error\_t error) {

printf("Hi I am writing to you from my TinyOS application!!\n");

call PrintfFlush.flush();

}

event void PrintfControl.stopDone(error\_t error) {

printf("This should not be printed...");

call PrintfFlush.flush();

}

event void PrintfFlush.flushDone(error\_t error) {

}

We start the timer with the interval defined in our header file, the interval is DEFAULT\_INTERVAL.

event void Timer.fired() {

if ( local.warning == 1 )

{

if (!sendbusy && sizeof local <= call AMSend.maxPayloadLength())

{

memcpy(call AMSend.getPayload(&sendbuf), &local, sizeof local);

if (call AMSend.send(AM\_BROADCAST\_ADDR, &sendbuf, sizeof local) == SUCCESS)

sendbusy = TRUE;

}

if (!sendbusy)

report\_problem();

}

if (call Read.read() != SUCCESS)

report\_problem();

}

The timer.fired event is fired when a certain amount of time has passed by. If local.warning = TRUE then we start sending our emergency broadcast. But first me must check whether our data will fit in one packet. The memcopy is used to copy local into our sendbuffer.

We send our packet with the AMSend interface, which is single-hop and use the AM identifiers… Our first argument is the address to which we want to send. In this case we used the broadcast address.

Finally we call Read.read() which makes our sensor start reading its value. It will signal the Read.readDone event when its ready, which is explained below.

This event is signaled when the radio has finished sending its packet. The error status is defined in error. We set the sendbusy variable back to FALSE.

event void AMSend.sendDone(message\_t\* msg, error\_t error) {

if (error == SUCCESS)

report\_sent();

else

report\_problem();

sendbusy = FALSE;

}

This event signals when the sensors has finished sampling its data. The data is passed in the data argument.

event void Read.readDone(error\_t result, uint16\_t data) {

if (result != SUCCESS)

{

data = 0xffff;

report\_problem();

}

if ( data > MAX\_SENS\_VAL )

local.warning = 1;

printf("Sensorvalue = %u\n",data);

printf("Warning = %u\n",local.warning);

call PrintfFlush.flush();

}

If the value of data exceeds our threshold we set local.warning to true. We also print some lines for debugging purposes.

## Conclusion

This should be easy enough to understand, if you however for some reason don’t, it might be a good idea to read some of the basic tutorials in the TinyOS documentation.

# BlindNode

## Functional description

This application listens for the broadcasts sent by the anchor nodes. When it receives one, it calculates the RSSI of the message and stores this data. Since RSSI values are very fluctuating, we need to apply a filter to it. In our case we took the average of a number of readings.

With this data the mobile node should be able to determine where it should go to. Since we only get data from one node we can only determine the distance between the two, but not their angle, or their exact position to each other.

We added a very simple mechanism through which the mobile node can come closer to its anchornode. Note that this routing mechanism is very simple, it was merely intended as a helpful tool for our master students.

## Blindnode.h

This event is roughly the same as AnchorNode.h

The only thing we add is the constant:

NREADINGS = 10

This defines the number of RSSI readings we need before we calculate the average.

## BlindnodeAppC

The app file is also simple and adds few little interfaces. We the CC2420ActiveMessageC which provides the CC2420Packet interface. With this interfaces we can measure our RSSI.

## BlindNodeC

### Variables

implementation

{

int8\_t RSSIvals[NREADINGS];

int8\_t reading = 0;

int8\_t RSSIval\_filter\_old;

int8\_t RSSIval\_filter\_new;

int8\_t direction = 1;

bool wait = FALSE;

bool first = TRUE;

RSSIvals is the array that stores the RSSI readings. RSSIval\_filter\_old & new are the results we get from our filter function. Direction defines the direction we should head out.

The bool wait is necessary for when the mobile node is moving to its destination, any RSSI measurements then are pretty useless.

### Events

event void Notify.notify( button\_state\_t state ) {

if ( state == BUTTON\_PRESSED ) {

wait = FALSE;

printf("Userbutton is pressed\n");

call PrintfFlush.flush();

}

}

This event is fired when the user presses the userbutton (telosb platform only!). This is used to tell the application that the mote has reached its destination. Masterstudent Man Hun Wong will interface a mobile robot to the mote, so this should become the event where the robot signals that the movement is done.

A simple averaging filter

void filter(){

int16\_t temp\_total = 0;

for(i=0;i<NREADINGS;i++){

temp\_total += RSSIvals[i];

}

RSSIval\_filter\_new = temp\_total / NREADINGS;

}

This function determines in which direction the mobile node should travel.

void movement(){

if (first){

RSSIval\_filter\_old = RSSIval\_filter\_new;

first = FALSE;

}

printf("RSSIval\_filter\_new = %i\n", RSSIval\_filter\_new);

if (RSSIval\_filter\_new < RSSIval\_filter\_old){

printf("RSSI is verslechterd!\n");

direction++;

if (direction >= 5)

direction = 1;

}

else {

printf("RSSI is verbeterd!\n");

}

printf("Druk op de knop als je klaar bent!\n");

if (direction == 1)

printf("Vooruit\n");

else if (direction == 2)

printf("Achteruit\n");

else if (direction == 3)

printf("Links\n");

else if (direction == 4)

printf("Rechts\n");

RSSIval\_filter\_old = RSSIval\_filter\_new;

}

Our simple algorithm does the following:

It compares the new RSS readings to the old.

If they improve: maintain same direction

Else: move in a new direction

Our application simply cycles between forward, backward, leftward, rightward.

While this is a very simple inefficient way of working, it does the trick.

This event is signaled whenever the radio receives a message. It turns on a led for indication.

event message\_t\* Receive.receive(message\_t\* msg, void\* payload, uint8\_t len) {

oscilloscope\_t \*omsg = payload;

report\_received();

if ( !wait ){

local.warning = omsg->warning;

if ( local.warning == 1 )

RSSIvals[reading++] = call CC2420Packet.getRssi(msg);

if ( reading == NREADINGS ){

filter();

reading = 0;

wait = TRUE;

if (RSSIval\_filter\_new > 10){

printf("Destination reached!\n");

printf("Proficiat!\n");

}

else

movement();

call PrintfFlush.flush();

}

}

return msg;

}

We take a RSSI reading with the CC2420Packet.getRSSI command and put it in the RSSIvals array.

If we have enough readings we filter them.

If our RSSI is good enough we are around our destination, else we need to move.

## ResultS

### output application

# RSSI readings

## INtroduction

For our master student Man Hung Wong, it was very important that he had RSSI readings for a given distance so he could, determine where his mobile node was.

## Method

We used the applications we described in the above sections. We set the NREADINGS constant to 10 which would mean that the mobile node takes 10 RSSI readings before he averages them. We distanced our motes 1 meter separated from each other. We then noted the averaged reading; we repeated this process 5 times. This process was repeated but for different distances, we incremented 1 meter per turn, until we reached 10 meters. For each step then averaged the results and subtracted 45 from these. This is because the CC2420 radio has a 45dBm offset.

These measurements were performed about 50 cm from the ground. We turned off any nearby radio signal, such as Wi-Fi, Bluetooth. These could interfere with our signal.

We then regressed this data to get a mathematical equation which describes the relationship between distance and RSSI.

## Results

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Distance (m) | Measurement 1 | Measurement 2 | Measurement 3 | Measurement 4 | Measurement 5 | Average | to Dbm |
| 1 |  |  |  |  |  |  | -51 |
| 2 | -31 | -28 | -17 | -19 | -28 | -24,6 | -69,6 |
| 3 | -24 | -10 | -23 | -13 | -10 | -16 | -61 |
| 4 | -19 | -26 | -21 | -26 | -17 | -21,8 | -66,8 |
| 5 | -25 | -33 | -32 | -28 | -28 | -29,2 | -74,2 |
| 6 | -18 | -29 | -22 | -21 | -22 | -22,4 | -67,4 |
| 7 | -25 | -21 | -25 | -30 | -31 | -26,4 | -71,4 |
| 8 | -16 | -19 | -29 | -27 | -25 | -23,2 | -68,2 |
| 9 | -23 | -23 | -23 | -24 | -28 | -24,2 | -69,2 |
| 10 | -25 | -32 | -21 | -17 | -11 | -21,2 | -66,2 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Distance (m) | Measurement 1 | Measurement 2 | Measurement 3 | Measurement 4 | Measurement 5 | Average | To dBm |
| 1 |  |  |  |  |  |  | -51 |
| 2 | -19 | -12 | -15 | -13 | -11 | -14 | -59 |
| 3 | -22 | -25 | -19 | -16 | -16 | -19,6 | -64,6 |
| 4 | -18 | -16 | -16 | -18 | -21 | -17,8 | -62,8 |
| 5 | -16 | -20 | -30 | -20 | -20 | -21,2 | -66,2 |
| 6 | -20 | -24 | -24 | -24 | -18 | -22 | -67 |
| 7 | -26 | -25 | -22 | -27 | -20 | -24 | -69 |
| 8 | -23 | -21 | -22 | -22 | -22 | -22 | -67 |
| 9 | -22 | -37 | -1 | -23 | -19 | -20,4 | -65,4 |
| 10 | -20 | -20 | -20 | -22 | -26 | -21,6 | -66,6 |