Evaluation of a Web-based Sensor Network Interface

Marcus Chang

Technical Report no. 06/01
ISSN: 0107-8283

Dept. of Computer Science
University of Copenhagen • Universitetsparken 1
DK-2100 Copenhagen • Denmark

Evaluation of a Web-based Sensor Network Interface

Marcus Chang

Graduate Student
Dept. of Computer Science
University of Copenhagen

Abstract

This paper evaluates the wireless sensor network testbed at Harvard University from a user's perspective qualitatively by evaluating the web-based user interface and quantitatively by running several pre- and custom-made programs on the testbed. The purpose of this evaluation is to provide feedback to the "TinyOS Testbed Working Group" community and to contribute to the development of the WSN infrastructure at DIKU.

Contents

1 Introduction

A Wireless Sensor Network (WSN) is characterized by a network of small autonomous units equipped with sensors and radio. These units form a wireless network and solve a given task in collaboration with each other. The units are often referred to as motes, since the vision of WSN is to have dust sized units, and are typically equipped with limited processing and storage capabilities. The motes are usually powered by their own power supply which makes power consumption a key factor in WSN. Since it is often impossible to recharge or replace batteries in a fully deployed WSN, the power supply puts and effective limit to the motes lifespan. The abillitie to turn on components only when they are needed, and otherwise have them turned off, is known as duty-cycling, and forms an integral part in WSN, since the lowered power consumption often increases the lifespan of a mote by several orders of magnitude. The collected data from the sensors are typically either stored in the network for later retrieval or transmitted through the network to a basestation which collects the data. Since the power needed to transmit data does not depend linearly on the transmitted distance, but rather on some power n of the distance (where $n > 1$), data are often routed through the network instead of directly to the basestation in order to save power.

WSN has been used with success in several real-life deployments, with the Great Duck Island habitat monitoring as one of the most famous. In the Great Duck Island experiment biologist wanted to monitor the habitat of the Leach's Storm Petrel. This was done by monitoring the microclimate in the nesting burrows by placing motes equipped with temperature, humidity, barometric pressure and infrared sensors inside the burrows. The readings were then forwarded to the basestation using either a star-topology network or a mesh-topology network. The reason for using two different kind of network topologies was to determine which one was better under real-life circumstances. Besides from being a biology project, the experiment also served as a testing environment for a WSN. Giving researches the abilitie to test power consumption, mote durablility and different network topologies all under real-life conditions. The problem with this approach in WSN development is the cumbersome and expensive task of programmming individual motes, placing them inside nesting burrows and reclaiming them after use. [1]

1.1 Motivation

In the research of network topology, power consumption, applications and WSN in general, the time consuming task of programming individual motes, deployment and retrieval from a real life environment and the economical cost of loosing motes are all barriers that moves the focus away from the actual research. Further more the down payment for a WSN easily mounts to several thousand dollars, effectively confining the majority of people to do research on simulators or networks consisting of just a few motes.

In order to do more effektive research on WSN the problems mentioned above must be dealt with appropriately. One should be able to program all the motes simultaneously and without the need to interact individually with every single mote. Deployment and retrieval should also be made transparent to the programmer, and reusability is mandatory. A team at Harvard University lead by Prof. Matt Welsh have implemented a WSN testbed called MoteLab [2]. The testbed makes it possible to program an entire WSN remotely and since the motes are permenantly placed, deployment and retrieval are no longer an issue. The usefulness of such an infrastructure is obvious, and the same MoteLab infrastructure has already been implemented at Massachusetts Institute of Technology and a similar infrastructure is in development at the Dept. of Computer Science at the Copenhagen University (DIKU).

1.2 TEP

The purpose of this paper is to evaluate the MoteLab infrastructure at Harvard University from the end users perspective.[1] This will include all the steps from finding the appropriate site, to running actual programs on the testbed. The goal is to find weaknesses in the implementation and suggestions for improvements. Hence this paper will function both as a draft for an actual TinyOS Enhancement Proposal (TEP) [3] and as a contribution to the infrastructure development project at DIKU.

1.3 Development platform

The platform used for code development was a Windows XP with the the **TinyOS 1.1.0 Installshield** installation (with **Cygwin** and **Java SDK**) and upgraded with the **TinyOS 1.1.11Feb2005cvs** RPM. Additionally the **SUN Javax** library was added.

1.4 Acknowledgements

Graduate student Jan Flora and Esben Zeuthen for lending me their ”Directed Diffusion” implementation and subsequently help in adapting the program to the MicaZ mote. Prof. Matt Welsh and the people at Harvard University for letting me use their MoteLab testbed and answering my questions. Ass. Prof. Philippe Bonnet for supervising this project.

This paper

Section 2 contains a description of the MoteLab software and how to obtain it as dictated in the TEP guidelines. Section 3 contains the preliminary work of finding a WSN testbed and how to get started using it. In section 4 several programs are tested on MoteLab. Section 5 describes the work of going from a theoretical implementation to an actual hardware specific implementation. Section 6 lists the suggested recommendations found during the evaluation. And finally section 7 concludes the paper.

The raw data obtained during this project can be found at

```
/net/urd/home/disk14/marcus/projekter/motelab
```

on the DIKU computer network. The files are:

- startup-data-4325.zip - Startup
- startup-data-4326.zip - Startup
- startup-data-4327.zip - Startup
- 1source-data-4335.zip - Startup using radio with 1 source
- 2source-data-4357.zip - Startup using radio with 2 sources
- 3source-data-4363.zip - Startup using radio with 3 sources
- rf-powerManage.log - power profile for Startup with radio
- uart-powerManage.log - power profile for Startup with UART
- sn-g9-s7-data-4127.zip - SensorNode with gateway = 9 and source = 7
- sn-g9-s7-data-4128.zip - SensorNode with gateway = 9 and source = 7
- sn-g9-s7-data-4143.zip - SensorNode with gateway = 9 and source = 7
- sn-g9-s27-data-4125.zip - SensorNode with gateway = 9 and source = 27

[1] It was originally intended to include an evaluation and comparison with the testbed at Massachusetts Institute of Technology as well, but since the team at MIT have just implemented the same system developed at Harvard University there was no point in doing so.

2 Implementation

2.1 Hardware

The testbed at Harvard University consist of 30 MicaZ motes. The MicaZ mote is equipped with an Atmel ATMEGA128L processor running at 7.3 MHz, 128 KB of read-only program memory, 4 KB of RAM and a Chipcon CC2420 radio operating at 2.4 GHz.[2] The radio are IEEE 802.15.4 compliant and has a transmission rate of 250 kbps. Each mote is connected to a Crossbow MIB600 ethernet gateway board which power the motes and provides remote reprogramming and logging capabilities [5]. Due to the size of the Crossbow MIB600 programming board the motes do not have any sensor board attached. The 30 motes and programming boards are distributed over three floors with 2, 25 and 3 motes on the first, second and third floor respectively at the Maxwell Dworkin building at Harvard University. In order to meassure real power consumption a digital multimeter Keithley 2701 has been attached to mote no. 3 and it is capable of sampling at 250Hz. [6] [2]

2.2 Software

The MoteLab software controlling the infrastructure of the testbed consist of: **MySQL Database Backend**, **Web Interface**, **DBLogger** and **Job Daemon**. These four modules run on a central server and utilizes a **MySQL Database Server** [7] and a **TinyOS SerialForwarder** connected to each mote [8].

Web Interface

This module consist of PHP pages and Javascripts which creates, edits and schedules jobs, displays a schedule of the testbed, provides user information and enables download of logged data. A job consist of one or more executables and java classes. Each job also has a certain number of motes (possible them all) associated with it and a mapping of motes to executables. Finally a job also contains other configuration parameters such as enabling of power profiling.

MySQL Database Backend

The database contains all the information needed for testbed operation. This includes testbed states which are: user information, access rights, mote state, information about uploaded files, job properties and testbed schedule. And job-generated data, which for each instance of a job consist of a table for each message type associated with that job.

DBLogger

A Java program called **DBLogger** connects to each motes data logging port, and parses messages sent over the serial port and inserts them into the database. Each table consist of the inspected message's structure, identifikation of the mote from which the message originated from, a timestamp and a global sequence number.

Job Daemon

The **Job Daemon** is a Perl script run as a **cron** job. The daemon sets up experiments by reprogramming the motes, starting system components such as the **DBLogger** and **SerialForwarder**. Afterwards its responsible for closing down the afore mentioned programs, stopping motes, killing processes and dumping data from the database into an easy downloadable file.

[2]These specifications are slightly lower than what Crossbow Technology Inc. states on their homepage for the MicaZ mote [4], probaly because the motes at Harvard University are of an older version.

2.3 SourceForge

All the software behind MoteLab have been made open source and are available for download at SourceForge at http://sourceforge.net/projects/syrah/

3 Preliminary work

3.1 Finding a testbed

As a novice WSN programmer the first task is to find a testbed. Which did not seem to be a problem since searching for the keywords "sensor", "network" and "testbed" in Google yielded the Harvard MoteLab as the third choice. [6] In comparison the MoteLab implemented at MIT did not show up at all. Not even after adding the keyword "MIT" to the search. One could argue that people with the need for a WSN testbed must be so involved in the community that finding a testbed would not be a problem, but nevertheless search engines is the most popular way to find things on the Internet.

3.2 Getting access

After finding the homepage for the testbed, the next objective was getting access. This information was directly available from the MoteLabs's frontpage and the account creation was done manually by mailing a request to Prof. Welsh. Besides providing information on how to get access, the frontpage also has a concise description of the site, the usage of the WSN testbed, how to get access to the source code, a description of the MoteLab hardware and a description of what software the motes can run. Links to a graduate course that had used the testbed, Prof. Welsh personal information page and a map of the deployed motes and their status was also available from the frontpage.

After logging in several pages become available: user info, schedule, create job, edit job and home.

The "user info" page gives information about the user's account name, account type, testbed quota, database handle and the abilitie to change password. It also describes how to gain access to the MySQL database where all the generated data are being stored and how to connect directly to each mote's **SerialForwarder** through a TCP connection.

The "schedule" page contains a table of scheduled jobs and free timeslots. Each timeslot is 5 minutes wide and the standard user is granted at 30 minutes quota. A job is scheduled by choosing the job, choosing which part of the motelab the job should run at and last by choosing one or more vacant timeslots.

The "create job" page is divided into 4 sub-pages: Description, files, motes and options. On the "description" page one designates the job a name and a description. On the "files" page one can upload executables and java class files, and associate each job with one or multiples files. The uploaded files must be named and optionally described. On the "motes" page one can choose to either run one program on all available motes, distribute several binaries evenly throughout the testbed or finally select which program should run on individual motes. The last page "options" is a checkbox for enabling high resolution (250 Hz) power consumption meassurements on the specific mote no. 3.

The "edit job" holds a list of all the jobs the user has created, and the job editing is similar to the job creation process.

The "home" page displays a status over a currently running job, scheduled jobs and a list of completed jobs with the task number, job name, start and completion time and the generated data as a zip archive.

3.3 Getting Started

Compiling for the MicaZ

The obvious way to get to know the MoteLab is by running some simple applications on the testbed. And the easiest way to do this is by using the standard applications distributed with **TinyOS**. Since the target platform is the MicaZ mote the straightforward approach to obtain executeables, is by running a "make micaz" inside the selected application directory.

This did not work however, since all the applications inside the **/tinyos-1.x/apps** folder is configured to use the older **Makerules** file from the **/tinyos-1.x/apps** folder (which does not include support for the MicaZ mote) instead of using the newer **Makerules** file from the **/tinyos-1.x/tools/make** folder. After changing the **Makefile** to use the newer **Makerules**, there were no problems compiling the code.[3] Alternative one could use the software branch Crossbow Technology Inc. has made for the MicaZ mote, which include support for different sensor boards and several other applications not originally supplied by TinyOS [9].

Creating a job

After obtaining the executeables the next step is to try it out. After inserting a name and a description for the job, one then has to upload one or more program and class files. Once uploaded however the files remain on the MoteLab and can subsequently be used directly in other jobs.

At this point it is not quite obvious what the class files are for, only that at least one must be associated with each job. If one have followed the TinyOS tutorial it says in lesson 6 how to communicate with motes by using the **Message Interface Generator (MIG)** to generate Java class files that correspond to Active Message types [10]. This would be the most obvious choice for a class file, although a bit odd that such a file is an absolute necessity upon job creation. Only after reading the MoteLab article [6] and reading the assignment [11] for the graduate course, is it clear that the MIG generated Java class is used by the MoteLab program **DBLogger** to parse packets read over the serial line into the database. Hence the Java class facilitates logging of data. This does however raise the question why a job demands at least one Java class associated with it since not all applications would require the logging of data.

After uploading and selecting program and class files, each program must be associated with none, some or all motes. This can be done by either choosing to run one program on all motes, distribute the selected programs evenly throughout the testbed or finally by mapping each program to none or more motes. At this point one encounters a small bug in the Javascript if multiple programs have been selected and added at once, instead of being added individually one at a time. The motes designation page thus behave as only one program has been selected, and it is only possible to run one of the selected programs on all of the motes.

Finally to complete the job creation one can choose to enable power consumption meassurements on mote no. 3.

Scheduling a job

Once a job has been created it can be scheduled for execution. The schedule consist of 5 minutes timeslots, and each scheduled job occupies at least one timeslot. Since a standard

[3]This was probaly caused by compiling on a system which originally was running TinyOS version 1.1.0 and subsequently upgraded to version 1.1.11, but developing under a Windows environment one does not have that much of an option.

user has a 30 minutes quota, one can either schedule up to 6 jobs at once or schedule 1 job to run for 30 minutes. After a job has been scheduled it can be deleted (from the schedule) as well. This leads however to a minor display bug, since the name of the job still occupies the vacant timeslot. Besides from choosing vacant timeslots one also gets the opportunity to choose if the job should be run on MD All, MD West og MD East. Assuming this partitioning is aimed at the actual motes it is a bit odd that a job can be scheduled to run on one partition, even though the mapping inside the job do not contain any motes from this subset. This also leads to a minor display bug, since after scheduling two jobs to run on its own partition, the timeslot is still flagged as vacant although it is not.

Interacting with the motes

During the execution of a job, it is possible to connect to each motes **SerialForwarder** and thereby send and receive data directly to each mote. It should be noted though, that even if two way commmunication is possible, it is not possible to connect a terminal directly to the **SerialForwarder**. Only Active Message packets can be communicated this way.

Logging data

As mentioned above logging is done solely through the inspection of Active Message packets send through the serial line which matches an uploaded Java class. This implies that the only way to log data in the MoteLab database is by wrapping an Active Message around the data and transmitting this message to the UART address. In other words in order to have something stored in the database one must first create an Active Message, write the data into a predefined struct, use the struct as payload in the Active Message and finally transmit the Active Message through the TinyOS radio stack, making it impossible to use the radio and communicate over the UART at the same time. Furthermore the struct must be known by the MoteLab as a Java class file, or else no logging is done if the class is not recognized.

This rather complex scheme to enable logging makes the wording on the frontpage sound a bit odd,

"During the job all messages and other data are logged to a database which is presented to the user upon job completion and then can be used for processing and visualization. In addition, simple visualization tools are provided via the web interface for viewing data while the job is running." [6]

since nothing is actually logged unless the user specifically does so in the program, and it does not say anywhere on the MoteLab page how this should be done. Finally the simple visualization tools provided by the web interface are simply not present. The only way to gain access to the logged data, is either by establishing a remote connection directly to the MySQL database or by downloading the tables as a zip archive.

Debugging

One of the biggest problems when working with a remote system is the lack of ground truth. When running programs on a local mote there are several ways to debug. First, motes usually are equipped with LEDs and reset buttons, which would indicate if a mote is running or not, and the abilitie to reset a mote at any given time gives plenty of insigt in were the program might have failed. Second, by connecting a terminal directly to a mote it is possible to gain insigt in what is transmitted over the UART. When running a job on MoteLab the only way of knowing if a program worked or not is by inspecting the logged data. But in the case of absent data debugging becomes ekstremely tedious since the source of the problems could be in the communication path as well as the rest of the program. This could be the Java class having a slightly different struct than the one in the Active Message, the Active Message not having the correct number, communication between the mote and UART conflicts with communication on the radio, and the UART not wired in correctly.

4 Running programs on MoteLab

4.1 Startup sequence

In order to implement duty-cycling in a WSN the motes involved must be synchronized to some extent. It is thus important to understand the startup sequence in the network. Since the programming of the motes is done from a centralized server one would expect the motes to be fairly synchronized from start. In order to test the startup sequence a simple program was written. The program **Startup** consist of a single task that dumps the system time[4] with an incremental serial number to the serial port. This task i posted every second from the moment the mote starts up. This simple test should show if the motes internal counters are reset upon every startup and in which order the individual motes become operational. Table 1 shows a typical output from the program.

Only the SerialNumber and counter field has been set by the program. The insert_time, motelabMoteID and motelabSeqNo have been inserted automaticaly by the MoteLab software. The first one notices is that all the motes except number 4 appears to be relatively synchronized, eg. they all share the same SerialNumber. For some reason mote no. 4 always seemed to be 5 seconds behind all the other motes. The second thing that comes to mind is that the actual logging in the database apparently also is 5 second behind since the first 5 packets has been lost for all the motes except mote no. 4. This appears to be correlated since subsequent tests with other programs on motes different than no. 4 does not show this delay. Third, the counter values are not strictly increasing but they are close enough to each other to conclude that the counters indeed are reset upon startup. Not shown by this single table is that multiple runs of the **Startup** program shows that the actual logging happens in a random order, but the actual mote startup and presumably programming happens in a fixed sequence. This becomes evident when multiple outputs are sorted by counter values, as shown in table 2.

Although a definite sequence cannot be resolved certain patterns a clearly visable when multiple tables are compared, e.g. motes no. 1, 8 and 27 always shows with low counter values while motes no. 5, 16 and 18 always shows with high counter values.

The source code to **Startup** can be found at appendix B.

4.2 Packet loss

In order to estimate the kind of reliability required by the network protocol for a given task, it is necessary to at least know the average packet loss on a typical hop between two motes. To test this the previous program was altered to use the radio broadcast address instead of the serial port address. On the receiving end the standard TinyOS basestation **TOSBase** was used. To represent a typical good connection mote no. 21 and 26 was chosen as source and sink respectively.[5]

As shown in table 3 a simple link between two motes has an average packet loss at about 1.2%. This is under the condtitions that no interferens are present and the two motes are located close to each other. Hence this value can be used as a lower bound for the average packet loss. By adding more sources to the experiment, packet loss due to interferens and packet drops due to the sink mote being busy can be meassured. First

[4]Where system time is the number of clock cycles since activation.

[5]See appendix A for an overview map of the main testbed floor.

```
+--------------+---------+----------------+---------------+--------------+
| serialNumber | counter | insert_time    | motelabMoteID | motelabSeqNo |
+--------------+---------+----------------+---------------+--------------+
|            6 | 5651491 | 20050524204814 |             8 |            1 |
|            6 | 5738176 | 20050524204814 |            23 |            2 |
|            6 | 5735183 | 20050524204814 |            25 |            3 |
|            6 | 5775394 | 20050524204814 |            12 |            4 |
|            1 | 1643141 | 20050524204814 |             4 |            5 |
|            6 | 5711171 | 20050524204814 |             1 |            6 |
|            6 | 5722608 | 20050524204814 |            27 |            7 |
|            6 | 5748432 | 20050524204814 |            24 |            8 |
|            6 | 5811612 | 20050524204814 |             7 |            9 |
|            6 | 5826802 | 20050524204814 |            11 |           10 |
|            6 | 5802348 | 20050524204814 |            13 |           11 |
|            6 | 5781203 | 20050524204814 |            20 |           12 |
|            6 | 5834463 | 20050524204814 |            15 |           13 |
|            6 | 5828920 | 20050524204814 |            10 |           14 |
|            6 | 5855207 | 20050524204814 |             6 |           15 |
|            6 | 5797985 | 20050524204814 |            18 |           16 |
|            6 | 5843257 | 20050524204814 |             2 |           17 |
|            6 | 5832293 | 20050524204814 |            30 |           18 |
|            6 | 5797995 | 20050524204814 |            19 |           19 |
|            6 | 5869989 | 20050524204814 |            26 |           20 |
|            6 | 5870113 | 20050524204814 |             9 |           21 |
|            6 | 5830254 | 20050524204814 |            22 |           22 |
|            6 | 5902901 | 20050524204814 |            17 |           23 |
|            6 | 5896098 | 20050524204814 |            21 |           24 |
|            6 | 5966188 | 20050524204814 |             5 |           25 |
|            6 | 5971111 | 20050524204814 |            16 |           26 |
|            6 | 6053939 | 20050524204814 |            28 |           27 |
+--------------+---------+----------------+---------------+--------------+
```

Table 1: Startup - sorted by insertion time

```
+--------------+---------+----------------+---------------+--------------+
| serialNumber | counter | insert_time    | motelabMoteID | motelabSeqNo |
+--------------+---------+----------------+---------------+--------------+
|            1 | 1661202 | 20050524210514 |             4 |            6 |
|            6 | 5649836 | 20050524210514 |             8 |            1 |
|            6 | 5708278 | 20050524210514 |            27 |            4 |
|            6 | 5709842 | 20050524210514 |             1 |            2 |
|            6 | 5731832 | 20050524210514 |            25 |            5 |
|            6 | 5737274 | 20050524210514 |            23 |            3 |
|            6 | 5765969 | 20050524210514 |            24 |            8 |
|            6 | 5779161 | 20050524210514 |            12 |           11 |
|            6 | 5782345 | 20050524210514 |            20 |           15 |
|            6 | 5790864 | 20050524210514 |            18 |            7 |
|            6 | 5791277 | 20050524210514 |            19 |           10 |
|            6 | 5807213 | 20050524210514 |            13 |           21 |
|            6 | 5823458 | 20050524210514 |            11 |           12 |
|            6 | 5826729 | 20050524210514 |            22 |           16 |
|            6 | 5828478 | 20050524210514 |             7 |           13 |
|            6 | 5830392 | 20050524210514 |            15 |           20 |
|            6 | 5843399 | 20050524210514 |             2 |            9 |
|            6 | 5847329 | 20050524210514 |            10 |           17 |
|            6 | 5855491 | 20050524210514 |            30 |           18 |
|            6 | 5865597 | 20050524210514 |             6 |           14 |
|            6 | 5870793 | 20050524210514 |             9 |           19 |
|            6 | 5878013 | 20050524210514 |            26 |           24 |
|            6 | 5886822 | 20050524210514 |            21 |           25 |
|            6 | 5907124 | 20050524210514 |            17 |           22 |
|            6 | 5937056 | 20050524210514 |             5 |           23 |
|            6 | 5963549 | 20050524210514 |            16 |           26 |
|            6 | 6023597 | 20050524210515 |            28 |           27 |
+--------------+---------+----------------+---------------+--------------+
```

Table 2: Startup - sorted by counter

Source(s)	Send	Received	Lost	%
1	1813	1790	23	1.3
2	2396	2119	277	11.6
3	1815	1347	468	25.8

Table 3: Packet Loss

mote no. 9 and was added and second mote no. 1. As shown in the table the packet loss increases to 11.6% with 2 sources and to 25.8% with 3 sources. Disregarding each motes counter drifting apart from each other over time, the transmissions occurs almost simultaneously, hence the packet losses observed are good estimates on the upper bound for 2 and 3 motes broadcasting within close range.

In a multihop network with multiple packets being routed simultaneously one should not expect a very high transmission success even after just a few hops unless some sort of reliability is added to the network protocol.

4.3 Power Consumption

One of the key constraints in a WSN is the limited power supply. Although power consumption can be estimated based on the power ratings given by the manufacture, the only way to know for sure if a certain program behaves as intended with regards to power savings, is by doing real meassurements on the mote during the progams execution. To test this both **Startup** programs was executed on mote no. 3 with power meassurement enabled, in order to get an idea of the actual difference in power consumption with and without the radio enabled.

The actual data was not logged in the database but only archived for download from the homepage. The format of the file is shown below. The timestamp appears to be in seconds and the reading in Amps, but nothing official can be found on the homepage.

```
# Data from job 1030 run on Wed May 25 14:26:17 2005
# Column format
# <timestamp> <current reading> <sample number>
+0.000000 +2.39584278E-02 +00000
+0.003556 +2.41000596E-02 +00001
+0.007078 +2.38464903E-02 +00002
+0.010583 +2.39264444E-02 +00003
+0.014075 +2.42611114E-02 +00004
...
```

Disregarding the last column and choosing a small time interval, e.g. 2 seconds, the data files are easily converted to graphs with GnuPlot [12].

Figure 1 shows the power profile of the **Startup** program only using the serial port and figure 2 shows the same program using the radio instead. Not surprisingly there is a power increase each time the mote transmits a message over the radio as seen in figure 2. What is surprising is the factor 6 difference between running with the radio enabled or disabled and the relatively small difference between transmitting and listening on the radio.

4.4 TinyDB

The purpose of MoteLab is to enable developers to run full blown programs on a fully deployed WSN. The only way to evaluate the testbed in this regime is to actually run real programs. A typical task for a WSN is to perform some meassurements and then send the data through the network to a base station where the data can be stored. This can be done by letting all motes sample data continously and only when a given condition is fulfilled the data is routed to the sink mote. Ideally this should be done with the lowest duty cycling possible. For this evaluation purpose **TinyDB** fits very well since it provides the above mentioned functionality with a simple user interface.

Unfortunately it was not possible to get **TinyDB** to work on MoteLab. After the modifications was made to the **Makefile** both the mote program and user front-end compiled without incident. But upon running the program on the testbed it was not possible

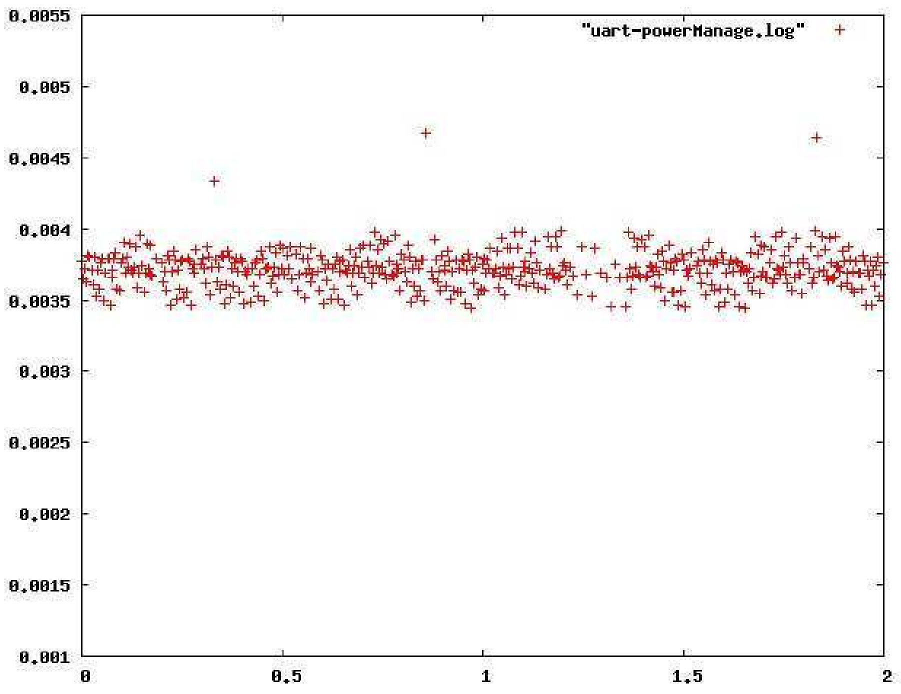

Figure 1: Power Profile with UART - plotted with GnuPlot

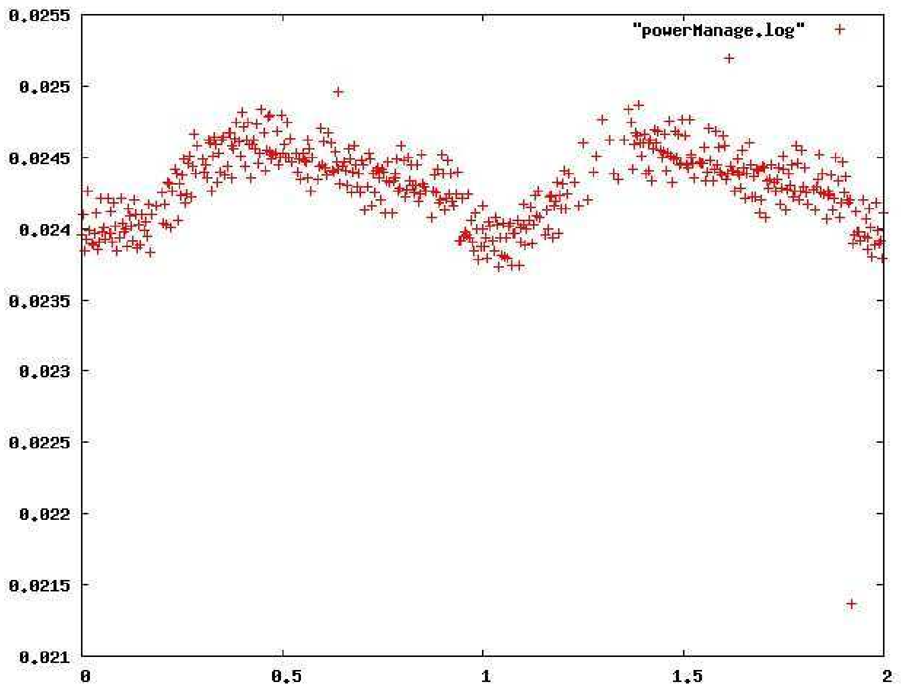

Figure 2: Power Profile with radio - plotted with GnuPlot

to establish commmunication to the different motes with the user interface. Further more by placing **TOSBase** programs on central motes it should be possible to intercept control packets, but no packets were received at all.

This could be the result of a badly configured TinyOS production environment since there had been problems with other network related tools, e.g. the **SerialForwarder**, but the lack of any packets at all points to an incompatability between **TinyDB** and the MicaZ motes, possible due to the newer radio (compared to the Mica2 mote). Apparently a similar problem was posted to the TinyOS-help mailing list half a year ago, which supports the former assumption, but the lack of debugging tools makes this an unsolved question [13].

4.5 Surge-View

Since there might be some incompatibility problems with the MicaZ mote and older TinyOS programs, it would make sense to try and run some programs written by the manufacture. One of the more advanced programs contributed by Crossbow Technology Inc. is the **Surge-View** program. In essence it is at program designed to test a given mote setup by establishing a routing tree for the network and then continously route sensor readings back to the mote connected to the user interface, for all the motes in the network [14].

This program did however not work at all. Since the user interface was a pre-compiled binary the development platform could not be at fault. Like with the **TinyDB** case several **TOSBase** motes was placed in the network and as before no packets were interceptet at all.

4.6 Sensornode program

In the light of these failures to run a program that resembles a real problem solving one, a custom program **SensorNode** was made. The program establishes a routing tree rooted at a predetermined sink. The tree is build by the sink broadcasting tree-establishing-packets. Motes that receive this packet determines, based on the link quality and the distance to the sink, if it should use the mote as parent. Once a mote has choosen a parent it too broadcasts a tree-establishing-packets, but with the distance to the root set accordingly. A mote only updates its parent when the distance to the root has improved. This should also help against loops in the routing tree. Based upon the meassurements performed above regarding packet loss, precautions were made to the network protocol by adding implicitly acknowledgements. Since a parent is choosen based upon the link quality, asynchronic links can be minimized by setting the quality thresshold high. Assuming synchronic links, a mote resends a packet to it's parent until it either receives the same packet from the parent or receives a new packet that needs to be forwarded. This packet drop was choosen to simplify the program and minimize the buffer need. In it's current form the program will try to resend a packet ten times within two seconds.

Originally the program was intended to only send data to the sink when a certain condition was met on the sensors. But since the motes do not have any sensors, the program was changed to use a predetermined source and repeatedly send data to the sink every third second. Also to ensure that the routing tree eventually is established the sink repeats it's tree-establishing-packet every tenth second. This will of course conflict with the data packet every thirty second but for this evaluation purpose this is ignored. As the program is in it's current form the motes wont change their parent eventhough no acknowledgements are received. This should of course be changed to include some sort of timeout of the parent, but for now mote loss is not supported.

A test of the program with mote no. 7 as source and mote no. 9 as sink yielded the following table as result:

```
+------------------+-------+------+---------------+----------------+
| lastSampleNumber | light | temp | data          | insert_time    |
+------------------+-------+------+---------------+----------------+
|                1 |   374 |  404 |               | 20050503181441 |
|                2 |   217 |  307 |               | 20050503181443 |
|                3 |   219 |  305 |               | 20050503181446 |
|                4 |   214 |  308 |               | 20050503181449 |
|                5 |   214 |  310 |               | 20050503181452 |
|                6 |   217 |  304 |               | 20050503181454 |
|                7 |   218 |  305 |               | 20050503181459 |
|                8 |   213 |  310 |               | 20050503181501 |
|                9 |   214 |  308 |               | 20050503181504 |
|               10 |   218 |  304 |               | 20050503181507 |
|               11 |   215 |  306 |               | 20050503181511 |
|               12 |   214 |  311 |               | 20050503181512 |
|               13 |   216 |  307 |               | 20050503181515 |
|               14 |   218 |  304 |               | 20050503181518 |
|               15 |   214 |  307 |               | 20050503181522 |
|               16 |   214 |  311 |               | 20050503181524 |
|               17 |   217 |  306 |               | 20050503181527 |
|               18 |   218 |  304 |               | 20050503181530 |
...
```

where the trivial fields have been omitted. The data field contains the route taken but for some reason this field does not get converted to ASCII characters. But with a hex-editor the route can be deciphered to:

```
7 -> 24 -> 19 -> 20 -> 5 -> 16 -> 9
```

which by inspection of the MotaLab map seems to be a plausable route. Several consecutive test showed close to no packet loss.[6] The mote no. 9 was chosen as sink since it's location forms a bottleneck in the network, which might become a source of a possible network partitioning. Besides the sink sending data over the serial port for logging, each mote was instructed to send it's depth to the serial port, in order to establish some sort of graphical representation of the routing tree. Unfortunately this output must have conflicted with date forwarding, since very few of these packets eventually got logged in the database.

The source code to **SensorNode** can be found at appendix C.

5 From teori to application

One of the main advantages of a WSN testbed, is the ability to bridge the gap between running programs on a simulated sensor network like the TOSSIM simulator [15] to running programs in a real life experiment. One could say that this application is the most important, since it allows a programmer to develop code not possible to simulate on TOSSIM, e.g. a new MAC-layer, and still be able to remove bugs before an actual real life implementation is performed, which could be a costly affair without proper debugging.

To this end graduate student Jan Flora and Esben Zeuthen has been kind enough to lend me their support, since they have made a TinyOS implementation of "two-phase pull Directed Diffusion" [16] as part of their master thesis. Their program was developed entirely on the TOSSIM simulator, and is stable in it's current state.[7] It was however not part of the design phase and development process that the implementation eventually should be able to run on real motes.

[6]One of these tests uses mote no. 27 as source, with the same results.

[7]Since the master thesis has not been submitted yet, it is not possible to show the program's sourcecode.

The first step in the adaption process was to make the program able to compile to the MicaZ platform. One of the things that needed to be done was to change certain unit types inorder for them to be compatible with the mote hardware. The second thing was the use and allocation of buffers. In it's native form the compiled program would exceed the memory requirements by a factor 2. This emphasizes the fact that motes indeed have limited memory capacity and that other hardware restrictions might apply.

The second step was to actual get some confirmation that the program indeed was running. This could either be done by logging data or by connecting to each mote through the **SerialForwarder**. Since the program originally did not contain any form for user interface the easiest way would be to log data. This could either be done directly by the program or indirectly by intercepting messages sent over the radio with the **TOSBase** program. The program was modified to include a command to send packets directly to the serial port and **TOSBase** motes were placed at strategical chosen positions. Both approaches requires the presence of Java class files with data structures matching the Active Message payload. This lead to the discovery of a bug in the program since the payload size was set incorrectly, leading to packets being send over the serial port, but not being recognized and parsed by MoteLab. This was discovered by modifying a local version of the **SerialForwarder** program, making it dump the raw packets to a terminal, disregarding the payload structure. By setting one mote to act as both subscriber and publisher, it was shown that the program indeed was running, that the communication to the serial port was working and that at least some of the program logic was working, since data was received as expected.

The third step was to actually have two motes, a publisher and a subscriber, communicate with each other. This was however not possible since the radio apparently did not work as intended. The motes did run but no radio transmissions were interceptet at all by the **TOSBase** motes. The component responsible for calling the radio was tested seperately to see if it was just bad wirering or something similar, but surprisingly the radio did work in this simple setup. The lack of debugging tools were a problem at this point, since it was not possible to get an indication of where in the program the problem resided. Under normal conditions debugging code could be used, but since the only way of logging data is by using the radio stack, the debug messages themselfes might cause problems.

We were not able to find the reason behind this problem in the given time, but more testing will most certainly reveal the problem.

6 Recommendations

6.1 Documentation

One of the biggest problems when working with MoteLab, was the lack of documentation or rather the difficulty in finding it. Article [2] and the student assignment [11] gave and excellent inside into the workings of MoteLab and references to them both would help new users alot. But of course some real documentation or a how-to would be preferred. This should of course include how the actual logging takes place, since from the current wording on the homepage one gets the impression that logging is done automatically without the need of any user actions, e.g. like making suitable class files. Second it should be mentioned eksplicitly that the motes do not have any sensor boards attached, since this might be the last assumption one would make about a sensor network testbed. On a side note (since it is trivial to mention that information on a homepage needs to stay up-to-date), really important information like what kind of motes that are currently installed and being used at the testbed, must be kept updated. Finding out that the testbed in fact used MicaZ motes and not Mica2 as the homepage said was really difficult, especially since the lack of debugging tools often leaves guessing as the only resort.[8]

[8]This information was changed though during this evalutaion.

6.2 Jobs

The first impression of the whole job scheduling process was that it was nice and intuitive (except the use of class files), and it made running batches of jobs easy. In the long run however the lack of the ability to remove jobs from the job list, and remove, rename or delete uploaded files, makes it difficult to keep the different files and jobs sorted, and mistakes are easily made. Second, the way jobs are scheduled makes it hard to work interactively since one has to wait until a job has completed until it can be modified, unless one would resort to deleting a job in process and risk another user taking the now vacant timeslot. A better way to do this scheduling would be to add the option of booking motes, instead of booking a job, so one could better utilize the reserved time. Also, when the testbed is used as a development platform and not just as a batch test platform, the whole graphical user interface becomes a bottleneck. Preferably some sort of command line interface would be more efficient, similar to typing "make install" when one work locally on a mote attached physically to one's computer.

6.3 Interaction

By using **SerialForwarders** directly on each mote and making them publicly available makes it in teori easier to use standard TinyOS programs. The drawback of this is however the lack of debugging abilities, since the **SerialForwarder** only works with Actice Messages, and as discussed in the last section, these messages are not suitable for debugging messages when the radio is at fault. A better solution would have been to make the serial port directly available over TCP with some sort of real serial port forwarder. This would enable the user to attach any kind of tools including terminals and **SerialForwarders** directly to the motes, and there by gaining access to the raw data coming from the serial port. In the absence of class files this raw data could also be logged automaticaly, since output of any kind is invalueable during debugging. One would thus be able to use both the serial port and the radio at the same time without the two conflicting with each other.

6.4 Logging

The idea of logging all data in a relational database and subsequently making the database avaible with remote login is excellent. However storing each data structure of each job of each run in a seperate table is not very efficient. Since the payload size is fixed a generic table could be made to contain all records of a given job, and the job execution number and data structure type could be attributes in this table. Cleaning up this table could be done with some cascade deletes on the run number.

7 Conclusion

The purpose of this paper was to evaluate the MoteLab testbed at Harvard University both in regards of finding weaknesses and places for improvement to the MoteLab software, but also as a contribution to the current development of a WSN infrastructure at DIKU. The former was done by testing several different programs on the testbed with as little knowledge of the inner workings as possible, and thus using the discovered problems as basis for recommendations for improvement. The latter was done by supplying the working group with firsthand experience of a working testbed including some of the material discovered during the evaluation.

A 2nd floor at the Maxwell Dworkin Laboratory

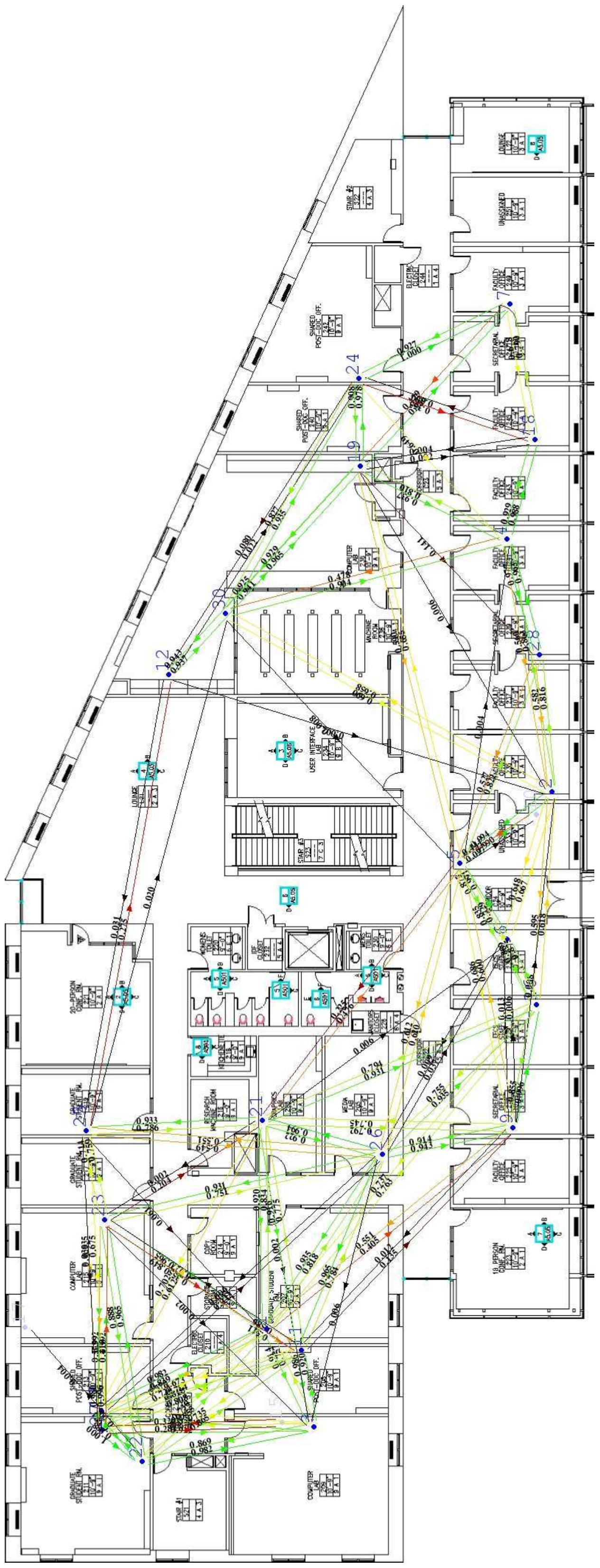

Figure 3: Connectivity and link quality graph

B Startup

B.1 Header files

```
struct StartupMsg
{
    uint16_t serialNumber;
    uint32_t counter;
};

enum {
  AM_STARTUPMSG = 10,
};
```

B.2 Startup.nc

```
includes StartupMsg;

configuration Startup { }
implementation
{
  components Main, StartupM
             , TimerC
 , SysTimeC
             , GenericComm as Comm;

  Main.StdControl -> StartupM;
  Main.StdControl -> TimerC;

  StartupM.SysTime -> SysTimeC;
  StartupM.Timer -> TimerC.Timer[unique("Timer")];
  StartupM.CommControl -> Comm;
  StartupM.SendMsg -> Comm.SendMsg[AM_STARTUPMSG];
}
```

B.3 StartupM.nc

```
includes StartupMsg;

module StartupM
{
  provides interface StdControl;
  uses {
    interface Timer;
    interface StdControl as CommControl;
    interface SendMsg;
  interface SysTime;
  }
}
implementation
{
  uint16_t serialNumber;
  uint32_t counter;
  TOS_Msg msg;
```

```
/**
 * Used to initialize this component.
 */
command result_t StdControl.init() {

  call CommControl.init();

  atomic {
    serialNumber = 0;
    counter = 0;
  }

  return SUCCESS;
}

/**
 * Starts the CommControl components.
 * @return Always returns SUCCESS.
 */
command result_t StdControl.start() {
  call Timer.start(TIMER_REPEAT, 1000);
  call CommControl.start();
  return SUCCESS;
}

/**
 * Stops the SensorControl and CommControl components.
 * @return Always returns SUCCESS.
 */
command result_t StdControl.stop() {
  call Timer.stop();
  call CommControl.stop();
  return SUCCESS;
}

task void Task() {
  struct StartupMsg *pack;
  atomic {
    pack = (struct StartupMsg *) msg.data;
    serialNumber++;
    pack->serialNumber = serialNumber;
pack->counter = call SysTime.getTime32();
  }

  call SendMsg.send(TOS_BCAST_ADDR,
   sizeof(struct StartupMsg),
   &msg);
}


/**
 * Signalled when the previous packet has been sent.
 * @return Always returns SUCCESS.
 */
```

```
  event result_t SendMsg.sendDone(TOS_MsgPtr sent, result_t success) {
    return SUCCESS;
  }

  /**
   * Signalled when the clock ticks.
   * @return Always returns SUCCESS.
   */
  event result_t Timer.fired() {
post Task();

    return SUCCESS;
  }

}
```

C SensorNode

C.1 Header files

DataMsg.h

```
// DataMsg.h

enum {
  BUFFER_SIZE = 14
};

struct DataMsg {
uint16_t sender;
    uint16_t receiver;
    uint16_t sourceMoteID;
    uint16_t lastSampleNumber;
uint16_t light;
uint16_t temp;
    uint8_t data[BUFFER_SIZE];
};

enum {
  AM_DATAMSG = 10,
};
```

TreeMsg.h

```
// TreeMsg.h

struct TreeMsg {
    uint16_t depth;
    uint16_t time;
    uint16_t sourceMoteID;
};

enum {
  AM_TREEMSG = 32
};
```

SensorNode.h

```
// SensorNode.h

enum {
  GATEWAY = 7,
  SOURCE = 27,
  STRENGTH = 215
};
```

C.2 SensorNode.nc

```
// SensorNode.nc

includes DataMsg;
includes TreeMsg;
```

```
configuration SensorNode { }
implementation
{
  components Main, SensorNodeM
           , TimerC
      , PhotoTemp
           , GenericComm as Comm;

  Main.StdControl -> SensorNodeM;
  Main.StdControl -> TimerC;

  SensorNodeM.Timer -> TimerC.Timer[unique("Timer")];
  SensorNodeM.ResendTimer -> TimerC.Timer[unique("Timer")];
  SensorNodeM.CommControl -> Comm;
  SensorNodeM.ReceiveDataMsg -> Comm.ReceiveMsg[AM_DATAMSG];
  SensorNodeM.ReceiveTreeMsg -> Comm.ReceiveMsg[AM_TREEMSG];
  SensorNodeM.SendDataMsg -> Comm.SendMsg[AM_DATAMSG];
  SensorNodeM.SendTreeMsg -> Comm.SendMsg[AM_TREEMSG];

  SensorNodeM.Temp -> PhotoTemp.ExternalTempADC;
  SensorNodeM.Light -> PhotoTemp.ExternalPhotoADC;
  SensorNodeM.TempStdControl -> PhotoTemp.TempStdControl;
  SensorNodeM.LightStdControl -> PhotoTemp.PhotoStdControl;

}
```

C.3 SensorNodeM.nc

```
// SensorNodeM.nc

includes DataMsg;
includes TreeMsg;
includes SensorNode;


module SensorNodeM
{
  provides interface StdControl;
  uses {
    interface Timer;
    interface Timer as ResendTimer;
    interface StdControl as SensorControl;
    interface ADC as Temp;
    interface ADC as Light;
   interface StdControl as TempStdControl;
  interface StdControl as LightStdControl;
    interface StdControl as CommControl;
    interface SendMsg as SendDataMsg;
    interface SendMsg as SendTreeMsg;
    interface ReceiveMsg as ReceiveDataMsg;
    interface ReceiveMsg as ReceiveTreeMsg;
  }
}
```

```
implementation
{
  TOS_Msg treeReceiveBuffer, treeSendBuffer;
  TOS_Msg dataReceiveBuffer, dataSendBuffer;
  TOS_Msg uartBuffer;

  TOS_Msg *treeReceiveBufferPtr, *treeSendBufferPtr;
  TOS_Msg *dataReceiveBufferPtr, *dataSendBufferPtr;
  TOS_Msg *uartBufferPtr;

  uint8_t treeDepth;
  uint16_t parent;
  uint16_t time;
  uint16_t signalStrength;

  uint16_t dataSeq;
  uint16_t vectorClock[32];
  uint8_t resend;
  uint8_t printData;
  uint8_t printTree;

  uint16_t light;
  uint16_t temp;

  /**
   * Used to initialize this component.
   */
  command result_t StdControl.init() {
uint8_t i;

    //turn on the sensors so that they can be read.
  call TempStdControl.init();
  call LightStdControl.init();

    //turn on the radio and UART
    call CommControl.init();

if (TOS_LOCAL_ADDRESS == GATEWAY) {
treeDepth = 0;
parent = 0;
time = 0;
dataSeq = 0;
} else {
treeDepth = 99;
parent = 99;
time = 0;
dataSeq = 0;
}

for (i = 0; i < 32; i++) {
vectorClock[i] = 0;
}

atomic {
```

```
printData = 0;
printTree = 0;
resend = 0;
light = 0;
temp = 0;
}

treeReceiveBufferPtr = &treeReceiveBuffer;
treeSendBufferPtr = &treeSendBuffer;

dataReceiveBufferPtr = &dataReceiveBuffer;
dataSendBufferPtr = &dataSendBuffer;

uartBufferPtr = &uartBuffer;

    return SUCCESS;
  }

  /**
   * Starts the SensorControl and CommControl components.
   * @return Always returns SUCCESS.
   */
  command result_t StdControl.start() {
  call TempStdControl.start();
  call LightStdControl.start();
    call CommControl.start();

if (TOS_LOCAL_ADDRESS == GATEWAY) {
    call Timer.start(TIMER_REPEAT, 10000);
} else {
    call Timer.start(TIMER_REPEAT, 3000);
}

    return SUCCESS;
  }

  /**
   * Stops the SensorControl and CommControl components.
   * @return Always returns SUCCESS.
   */
  command result_t StdControl.stop() {
call TempStdControl.stop();
call LightStdControl.stop();
    call Timer.stop();
    call CommControl.stop();

    return SUCCESS;
  }

  task void relayDataTask() {
    struct DataMsg *pack;
uint8_t jump;

atomic {
    pack = (struct DataMsg *) dataSendBufferPtr->data;

pack->sender = TOS_LOCAL_ADDRESS;
pack->receiver = parent;
jump = pack->data[1] + 1;
pack->data[0] = treeDepth;
pack->data[1] = jump;
pack->data[jump] = TOS_LOCAL_ADDRESS;

call SendDataMsg.send(TOS_BCAST_ADDR,
sizeof(struct DataMsg),
dataSendBufferPtr);
}
  }

  task void sendDataTask() {
    struct DataMsg *pack;

atomic {
dataSeq++;

    pack = (struct DataMsg *) dataSendBufferPtr->data;

pack->sender = TOS_LOCAL_ADDRESS;
pack->receiver = parent;
     pack->sourceMoteID = TOS_LOCAL_ADDRESS;
pack->lastSampleNumber = dataSeq;
pack->light = light;
pack->temp = temp;
pack->data[0] = treeDepth;
pack->data[1] = 2;
pack->data[2] = TOS_LOCAL_ADDRESS;

call SendDataMsg.send(TOS_BCAST_ADDR,
sizeof(struct DataMsg),
dataSendBufferPtr);
}
  }

  task void plantTreeTask() {
    struct TreeMsg *pack;

atomic {
if (TOS_LOCAL_ADDRESS == GATEWAY) {
time = time + 1;
}

    pack = (struct TreeMsg *) treeSendBufferPtr->data;
     pack->depth = treeDepth + 1;
pack->time = time;
    pack->sourceMoteID = TOS_LOCAL_ADDRESS;

call SendTreeMsg.send(TOS_BCAST_ADDR,
sizeof(struct TreeMsg),
treeSendBufferPtr);
}
  }


  /**
   * Signalled when data is ready from the ADC.
   * @return Always returns SUCCESS.
   */
  async event result_t Light.dataReady(uint16_t data) {

atomic light = data;
call Temp.getData();

    return SUCCESS;
  }

  async event result_t Temp.dataReady(uint16_t data) {

atomic {
temp = data;
resend = 10;
printData = 1;
post sendDataTask();
call ResendTimer.start(TIMER_ONE_SHOT, 200);
}

    return SUCCESS;
  }

  /**
   * Signalled when the previous packet has been sent.
   * @return Always returns SUCCESS.
   */
  event result_t SendDataMsg.sendDone(TOS_MsgPtr sent, result_t success) {

atomic {
if (printData == 1) {
printData = 0;
call SendDataMsg.send(TOS_UART_ADDR,
sizeof(struct DataMsg),
dataSendBufferPtr);
}
}

    return SUCCESS;
  }

  event result_t SendTreeMsg.sendDone(TOS_MsgPtr sent, result_t success) {

atomic {
if (printTree == 1) {
printTree = 0;
```

```
call SendTreeMsg.send(TOS_UART_ADDR,
sizeof(struct TreeMsg),
treeSendBufferPtr);
}
}

    return SUCCESS;
  }


  /**
   * Signalled when the clock ticks.
   * @return SUCCESS.
   */
  event result_t Timer.fired() {

if (TOS_LOCAL_ADDRESS == GATEWAY) {
printTree = 1;
post plantTreeTask();
} else if (TOS_LOCAL_ADDRESS == SOURCE && parent < 99) {
call Light.getData();
} else {
}

return SUCCESS;
  }

  event result_t ResendTimer.fired() {

atomic {
if (resend > 0) {
resend--;
call SendDataMsg.send(TOS_BCAST_ADDR,
sizeof(struct DataMsg),
dataSendBufferPtr);

call ResendTimer.start(TIMER_ONE_SHOT, 200);
}
}

return SUCCESS;
  }

  /**
   * Signalled when data message received.
   * @return The free TOS_MsgPtr.
   */
  event TOS_MsgPtr ReceiveDataMsg.receive(TOS_MsgPtr m) {
TOS_Msg *tmp;
struct DataMsg *pack;

atomic {
pack = (struct DataMsg *) m->data;
```

```
if (vectorClock[pack->sourceMoteID] < pack->lastSampleNumber) {
if (TOS_LOCAL_ADDRESS == pack->receiver) {
vectorClock[pack->sourceMoteID] = pack->lastSampleNumber;

if (TOS_LOCAL_ADDRESS == GATEWAY) {

tmp = dataSendBufferPtr;
dataSendBufferPtr = m;
m = tmp;

printData = 1;
post relayDataTask();
} else {

tmp = dataSendBufferPtr;
dataSendBufferPtr = m;
m = tmp;

resend = 10;
post relayDataTask();
call ResendTimer.start(TIMER_ONE_SHOT, 200);
}
}
} else if (vectorClock[pack->sourceMoteID] == pack->lastSampleNumber) {

// message from child, send ack
if (TOS_LOCAL_ADDRESS == pack->receiver) {

tmp = dataSendBufferPtr;
dataSendBufferPtr = m;
m = tmp;

post relayDataTask();

// message from parent, cancel resend and print local copy to uart
} else {
resend = 0;

tmp = dataSendBufferPtr;
dataSendBufferPtr = m;
m = tmp;
     }
 }
}

    return m;
  }

  event TOS_MsgPtr ReceiveTreeMsg.receive(TOS_MsgPtr m) {
struct TreeMsg *pack;

pack = (struct TreeMsg *) m->data;

if (TOS_LOCAL_ADDRESS != GATEWAY) {
```

```
if (treeDepth > pack->depth && m->strength > STRENGTH) {

treeDepth = pack->depth;
parent = pack->sourceMoteID;
time = pack->time;
signalStrength = m->strength;

printTree = 1;
post plantTreeTask();
} else if (parent == pack->sourceMoteID) {
time = pack->time;
post plantTreeTask();
}
}

    return m;
  }

}
```

References

[1] Joseph Polastre John Anderson Robert Szewczyk, Alan Mainwaring and David Culler. An analysis of a large scale habitat monitoring application. *ACM Sensys Conference*, November 2004.

[2] Pat Swieskowski Geoff Werner-Allen and Matt Welsh. Motelab: A wireless sensor network testbed. *IPSN'05, SPOTS*, 2005.

[3] Philip Levis. Tep structure and keywords. *http://cvs.sourceforge.net/viewcvs.py/tinyos/tinyos-1.x/beta/teps/txt/tep1.txt?view=markup*, 2004.

[4] Crossbow Techonology Inc. Mote-kit2400 datasheet. *http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Kit_Datasheet.pdf*, 2005.

[5] Crossbow Techonology Inc. Mib600ca datasheet. *http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MIB600CA_Datasheet.pdf*, 2005.

[6] Pat Swieskowski Geoff Werner-Allen and Matt Welsh. Motelab homepage at harvard. *http://motelab.eecs.harvard.edu/*, 2005.

[7] MySQL AB. Mysql database server. *http://www.mysql.com/products/mysql/*, 2005.

[8] Bret Hull and David Gay. Serial-line communication in tinyos-1.1. *http://www.tinyos.net/tinyos-1.x/doc/serialcomm/index.html*, 2003.

[9] Crossbow Techonology Inc. Crossbow's tinyos applications for motes. *http://www.xbow.com/Support/Support_pdf_files/xbow_5020-0315-01_A.tgz*, 2005.

[10] TinyOS. Tinyos tutorial. *http://www.tinyos.net/tinyos-1.x/doc/tutorial/index.html*, 2003.

[11] Matt Welsh. Assignment #1 - multihop data-collection protocol for sensor networks. *http://www.eecs.harvard.edu/ mdw/course/cs263/assn1/index.html*, 2004.

[12] Thomas Williams and Colin Kelley. Gnuplot. *http://www.gnuplot.info/*, 2004.

[13] Harri Siirtola. Tinydb doesn't work with micaz? *https://mail.millennium.berkeley.edu/pipermail/tinyos-help/2004-November/006564.html*, 2004.

[14] Crossbow Techonology Inc. Surge-view. *http://www.xbow.com/Support/downloads.htm#surgeview*.

[15] Philip Levis and Nelson Lee. Tossim: A simulator for tinyos networks. *http://www.cs.berkeley.edu/ pal/pubs/nido.pdf*, September 17, 2003.

[16] Ramesh Govindan Chalermek Intanagonwiwat and Deborah Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*, August 2000.