# Wireless Sensor Networks Lab
# Final Project
# Winter Term 2010/11

Waldemar Haffner, David Meister, Florian Reith, Michael Strosche, Ralf Zimmermann

Technische Universität Darmstadt

**Abstract:** This report describes the system that was to be implemented as the final project of the Wireless Sensor Networks Lab at Technische Universität Darmstadt in the Winter Term of 2010/11. The lab was organized and held by Pablo Guerrero, Arthur Herzog, Christian Seeger, Kristof van Laerhoven, Eugen Berlin and Francois Philipp of the Databases and Distributed Systems Group headed by Prof. Alejandro Buchmann, Ph.D.

## 1 Introduction

The goal of this year's wireless sensor networks lab (WSN) is to build a sensor network to measure CO2, humidity and temperature in the surPLUShome [1] at the campus. The sensor node that we will use is the Zoleria Z1 [2].

We start by briefly describing the requirements that the project must fulfill. Then we describe our implementation and evaluate the results gathered from the live system run at the surPLUShome on campus. Finally we draw our conclusions on the project.

## 2 Requirements

The requirements for the project are divided into different working packages. They were specified in the task sheet [3] and are summarized in this section.

## 2.1 Communication

Florian Reith will work on this working package. His adviser is Francois Philipp.

The task of this working package is to develop an communication driver that helps to save energy, because Scopes, one of the used frameworks, supports only NullMAC. An improvement could be to use XMAC or LPP. This driver has to be integrated into the whole framework.

## 2.2 Network Reprogramming

Ralf Zimmermann will work on this working package. His adviser is Pablo Guerrero.

Since the sensor nodes are also placed in inaccessible places, they have to be programmed over the network. Deluge is an OTA updater for the operating system (OS) on the nodes, but it has to be integrated into the OS.

## 2.3 Querying

Waldemar Haffner will work on this working package. His adviser is Christian Seeger.

To gather informations from the network we use the frameworks TikiDB and Scopes. They have to be restructured and need a cleanup to work and fit onto the nodes. To save bandwidth and energy the queries have to be defined and optimized. Since the sensors varies from node to node, the system must handle heterogeneous nodes.

## 2.4 Sensing

David Meister will work on this working package. His advisers are Kristof van Laerhofen and Eugen Berlin.

In order to measure various environment values (temperature, humidity, CO2) corresponding sensors have to be selected, purchased and soldered onto the nodes. To read the sensor values drivers have to be developed and integrated into TikiDB.

## 2.5 Team Leader & Web Interface

Michael Strosche will work on this working package. His adviser is Arthur Herzog.

The different roles must be coordinated and communication between students and advisers and the architects should go over one person. A documentation must be written. To display the gathered sensor values a web interface has to be developed.

# 3 Implementation

Here we describe how the components that make up the system are actually implemented in software.

## 3.1 Communication

### 3.1.1 Introduction

The final task for this years wireless sensor networks lab was to deploy a network of Zolertia Z1 wireless sensor nodes in the surPLUShome solar house. The network was to be used to gather data over the course of several weeks, but as the sensor nodes run on battery power the lifetime of the network is limited. While it is possible to replace the batteries some of the nodes were to be placed in difficult to access locations, which made energy efficiency a very important design requirement. The data gathering software was to be based on TikiDB, which uses Scopes as its communication subsystem. The goal of the communication project was to make Scopes more energy efficient to extend the network lifetime.

### 3.1.2 Preliminary Considerations

The largest energy consumer in wireless sensor network nodes is usually the radio transceiver. In order to minimize energy consumption we had to minimize the network traffic and deactivate the radio as much as possible. Switching the radio on and off is managed by Radio Duty Cycling (RDC) protocols, but in the default configuration Scopes does not use any RDC protocol (NullRDC). For a large improvement in energy efficiency we had to reconfigure Scopes to use XMAC or LPP instead of NullRDC. Based on our experience from previous exercises, where XMAC performed better than LPP, we decided to use X-MAC. Additional improvements were to be achieved by using the most energy efficient routing protocol provided by Scopes (flooding

or selfur) and minimizing the amount of messages sent for scope maintenance.

### 3.1.3 Simulation

Running tests on real wireless sensor network nodes is very time consuming and complicated, because the test results have do be logged decentralized. We therefore decided to run simulations before testing the software on real hardware and chose Cooja as the simulator. As Cooja does not support the Z1 platform we had to use the similar sky platform, which is based on the same CPU and radio transceiver. We used energest with average energy consumption rates taken from the Zolertia Z1 data sheet for energy consumption simulations.
The surPLUShome is a small house build mostly out of wood and materials, that absorb radio signals only slightly. We therefore assumed that all nodes will be able to reach the base station and most of the other nodes directly and used a one hop scenario for the simulations. In cooperation with the queries project, which worked on reducing redundancies between TikiDB and Scopes, it was early decided to use a single scope for all nodes in the solar house and distinct between nodes equipped with different sensors using TikiDB queries instead of scopes. In the scenario used for our simulations a single base station was used to open a scope and send queries in fixed intervals. Nine nodes in a one hop neighborhood to the base station joined the scope and answered the queries. This setup was very similar to the one we planned to use in the solar house, although the number of nodes used in the solar house would be higher.

### 3.1.4 Operation Testing

To test the functionality of different configurations we used our simulation setup to measure the message loss from the base station to the nodes (query loss) and from the nodes to the base station (reply loss). The loss of queries was considered more harmful, because it tends to affect several nodes at once, leading to the loss of several data records, where a single reply loss results in only one lost data record. The results of our measurement are shown in figure 1, where queries is the percentage of query messages received by the nodes, replies is the percentage of replies the base-station received from nodes who received the query and data records is the percentage of replies the base-station should have received in case all query and reply messages were received.

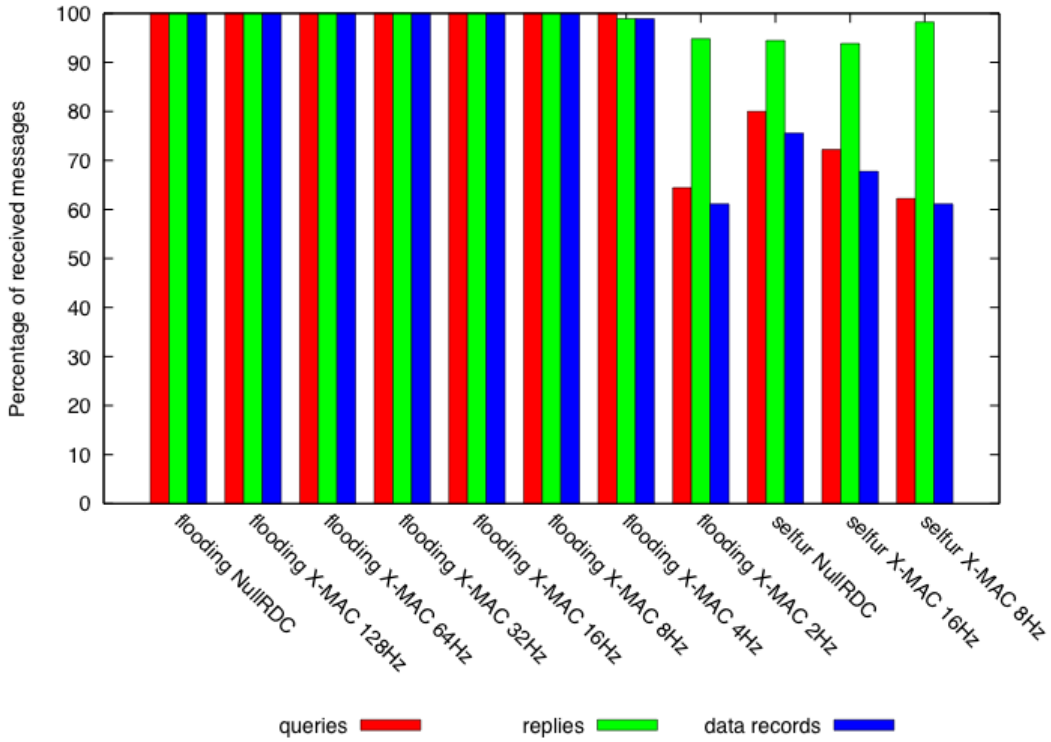


Figure1: Message loss for different configurations

In the default configuration Scopes uses flooding and NullRDC. This configuration showed no message loss in our measurements. Changing the RDC mechanism to X-MAC worked fine for channel check rates from 128Hz down to 8Hz, but we experienced increasing message loss when we lowered the channel check rate to 4Hz and 2Hz. Additionally some messages were delivered more than once with channel check rates of 4Hz and 2Hz. Once we changed the routing protocol from flooding to selfur the network suffered heavy message loss with both RDC mechanisms.

### 3.1.5 Energy Consumption

To find out which configuration is most energy efficient we used energest to simulate the energy consumption of the network for the configurations already used in operation testing. Due to the nature of our application scenario and test case the results are almost linear with time. It is therefore sufficient to show the results for 10 minutes of simulation time.

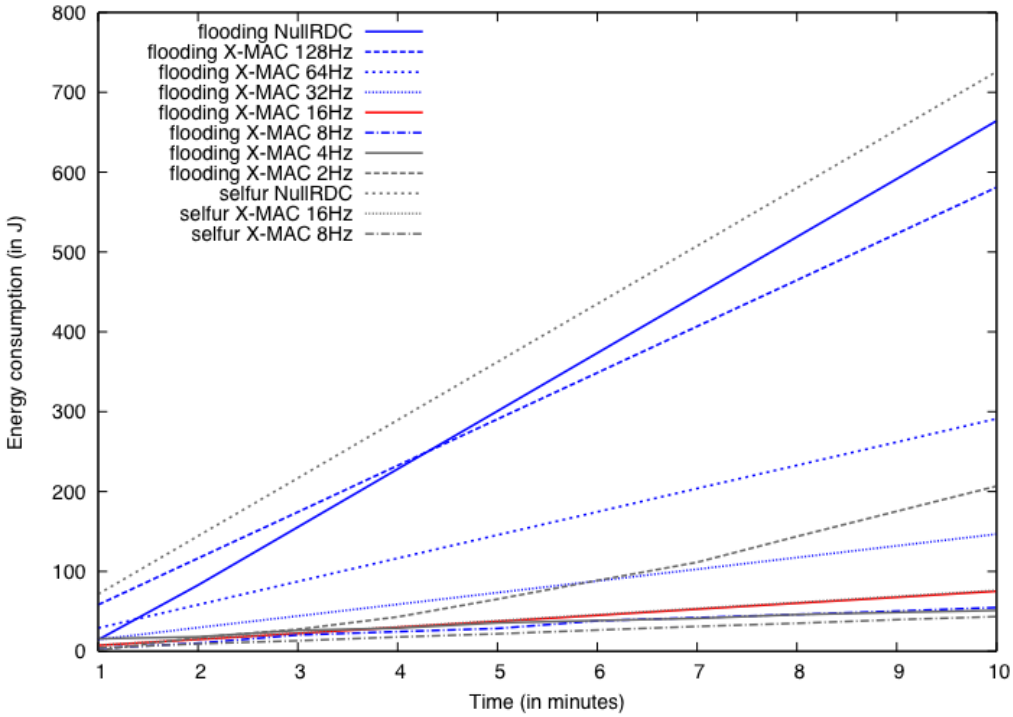

Figure 2: Energy consumption of a 10 node network with different configurations

Keep in mind, that some configurations suffered from message loss and therefore consumed less energy than they would have if they delivered all messages, while the low check rate X-MAC configurations delivered some messages several times. All configurations with malfunctions are shown in gray, the configuration shown in red is the one we chose for the final setup. As expected NullRDC performs considerably worse than X-MAC, and lowering the channel check rate decreases the energy consumption. Surprisingly flooding and selfur show similar performance for some X-MAC configurations and flooding performs even better than selfur with NullRDC. We think the reason for this is our one hop scenario, where all nodes receive all messages sent by other nodes and the rebroadcasting overhead created by flooding is similar to the tree maintenance overhead created by selfur.

### 3.1.6 Final Configuration

We decided to use X-MAC with a channel check rate of 16Hz, flooding as routing protocol and CSMA for the final network setup. Additionally we optimized the scopes and query time to life to reduce the number of keep alive and query messages that have to be broadcasted by the base-station. These final settings were tested on a small testbed of real Zolertia Z1 nodes and showed good performance.

The channel check rate was chosen for safety reasons. While 8Hz still performed well in the simulation we were concerned about possible differences between the simulated sky platform and the real Zolertia Z1

nodes and therefore decided to trade of some energy efficiency for higher reliability. By using CSMA, which resolves some collisions, instead of NullMAC, which does not resolve collisions and is used in the Scopes default configuration, we further improved the reliability of the network communication.
With selfur and low X-MAC channel check rates the network would have been ready for multi hop extension and a slight improvement in energy efficiency would have been gained, but unfortunately we were not able to solve the message loss issues with selfur and therefore decided to use flooding.

### 3.1.7 Problems and Challenges

When we tried to run TikiDB on Zolertia Z1 nodes for the first time the network suffered from 100% message loss. Further testing showed, that the whole Contiki rime stack was not working properly. In the end it turned out, that although the official Contiki build already contains some Zolertia Z1 related files it does not yet support Zolertia Z1 nodes. The problem was solved by using a working Contiki build from Zolertia.
Contiki's netflood protocol uses a 16-bit sequence number to detect duplicate messages and ignores those messages instead of rebroadcasting them. Unfortunately the struct in which netflood connections are stored remembers only 8-bit of the sequence number. To prevent the protocol from failing after 255 messages a bugfix provided by the project supervisors had to be included in our finale image.
Another issue occurred when we tried to set the time to life (ttl) for the TikiDB scope. The nodes are supposed to leave the scope after the ttl expires, they however did not leave. Investigation of the issue showed that a strange Zolertia Z1 behavior caused the nodes to stay in the scope. The time to life was stored and sent over the network in an array of 2 uint_8 values, which was messed up during the network transmission and led to the nodes receiving a longer ttl than set by us. The problem was solved by replacing the array with a single uint_8 and changing the resolution from seconds to minutes.

## 3.2 Network Reprogramming

Because the Network Reprogramming didn't work properly in this project because the updates were too large, it was moved into a separate project. The documentation is shipped in a separate document.

## 3.3 Querying

### 3.3.1 Initial assignment

**Code cleanup**
Most important point in assignment was code cleanup of Scopes and TikiDB. Especially code size reduction was important, since motes have very limited memory. First of all it was necessary to get a rough overview over the stack. Identify modules, sub modules and their responsibilities. Next step was to get them compiled. This task wasn't that easy, since it was required to run the stack on Contiki 2.5. Scopes and TikiDB were written for Contiki 2.4, so changes in 2.5 influenced the stack due to API changes in Contiki. Changes that had to be made were mostly correction of include files and adjusting changed data types. Next problem was that the linker messaged about "text segment exceeded". Such messages are exactly the reason, why code had to be cleaned up. It was simply too large.
Scopes framework has a sub module called "repository". It is responsible to assign motes to a scope, based on predefined criteria. In original state the criteria could be almost arbitrary complex. This code was replaced by a much simpler implementation, which is less versatile. It assigns motes to a scope based only on node_id. This solution fully satisfies our needs and saves a lot of code space.
In TikiDB was the sensortable significantly reduced. This affected not only the lookup tables and access functions, but also the long switch-case cascades in "queryresolver" submodule. In "db" and "dbms" were all aggregation functions removed, since we don't need them. This optimization also heavily affected "querymanager", which was responsible for interpreting such complex queries. Among with query optimization "querymanager" was also lightened, because it now don't have to manage queries' lifetime.
Among those "big" optimizations a lot of other tweaks were made, like removing and inlining of one-line-functions, to save couple of ASM bytes and stack increase.

**Query definition**
Query definition was straight forward. Since no special requirements were formulated and only four fields are interesting/readable, all fields are packed into one query which is executed every 15 minutes.

**Query optimization**
Queries are optimized by means of definition/interpretation/execution. That means, that all unneeded features were removed. Namely query lifetime and aggregation support. Since we decided to use netflooding as routing protocol,

no other query optimizations were made.

**Query dissemination**
Query distribution works as follows: every node starts a process when it joins a scope. This process checks the node for a query. If none is present, it sends a special request to the base station. Base station answers this request by sending query definition to the network. Child nodes, which already have the query simply ignore the package and the child node, which sent the request accepts and stores the query definition. This algorithm ensures, that no child node "hangs around" in case it once had to be rebooted. If a child node leaves a scope, it stops the "query-watch" process and drops the query to avoid sending unneeded data and lower the power consumption. Another case is, if base station had to be restarted (i.e. watchdog). If that happens and base station is back, it waits for SCOPE_TTL amount of time to make sure, all children dropped their queries and left the old scope.

### 3.3.2 Additional assignment

**DB-Writer**
DB-Writer is a software piece, that collects the data delivered to the base station and stores it in a MySQL database. To accomplish this task, the "serialdump" program, which is part of Contiki, was modified. It serves now as serialdump and MySQL client at once. It buffers data incoming through serial port until it reads a newline char. Otherwise it would be quite difficult to parse the data. After a line was read, it its being parsed. From the parsed data a SQL statement is being constructed and submitted to the target database.

**Restructuring Scopes + TikiDB stack**
Scopes and TikiDB were originally developed by two independent development teams. It was evident, because in both modules had parts which logically belonged to other module. Scopes had a simple query system implemented while TikiDB had a simple datalink layer. This fact makes it difficult to bring this modules together. One approach called "wrapper" was initially delivered. It worked fine but it introduced undesired dependencies between those three modules. Following figure illustrates these dependencies. As can be seen, TikiDB depends on wrapper/client code. That means, if you want to write a new client that uses TikiDB, you have to make a copy of it and intrusive change the code. This circular dependency was resolved by using the Dependency Inversion Principle.

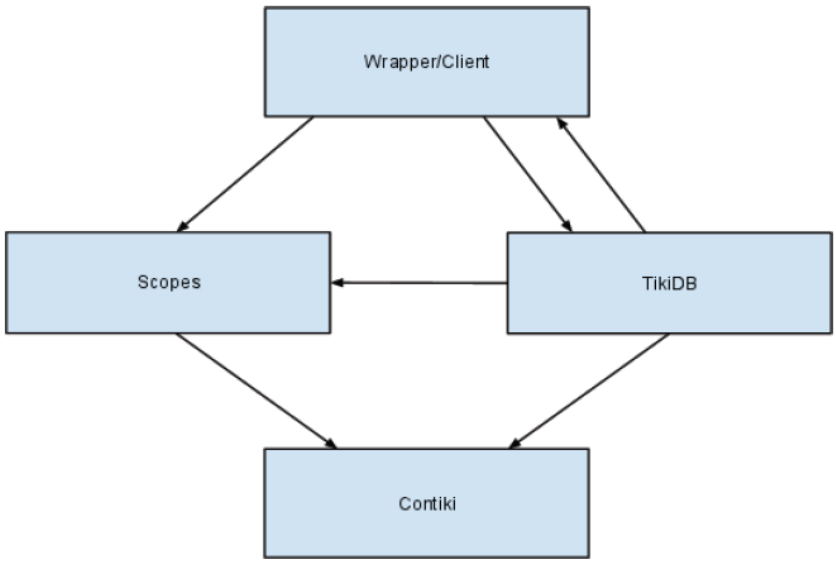

Figure 3: Dependency in the code

**Testing**
Really a LOT of!

### 3.3.3 Result
Final system works fine on Z1 motes. Scopes and TikiDB are now compatible with Contiki 2.5. Room for extensions is there too. Measurements between first compilable version and final result give a difference of about 3KB space.

## 3.4 Sensing

### 3.4.1 Identification and purchase of extra sensors
As already mentioned there is a need to measure humidity, temperature and CO2. Since the Z1 is equipped with a temperature sensor, only humidity and CO2 sensors had to be purchased. In the beginning of the project was also a light sensor desired. So there is additional the opportunity to easily use this sensor (a driver is provided). The most simple way to capture values from the analog world (everything needed is continuous) is to use an ADC (analog-digital converter). Hence we have chosen a humidity sensor with a linear output voltage – the Honeywell HIH-5030. CO2 sensors are very expensive and there is often a necessity to calibrate them. Therefore we bought a complete sensor station containing temperature, humidity and CO2 sensors – Voltcraft CO-50. Two nodes are connected to them, the other uses the internal temperature sensor and the Honeywell humidity sensor.

### 3.4.2 Installation (connection/soldering)

The Z1 provides a couple of connectivities for digital buses, I/Os for ADCs and general purposes. Overall are seven AD channels available. The humidity is soldered to AD channel number one with a 65 kOhm resistor. The sensor station has a output connection using the I2C protocol. We have measured a clock with two kHz and data packets on another pin with an oscilloscope. Unfortunately the I2C port on the Z1 is already used by the two internal sensors. So we build an intermediate system with a pic micro-controller. This controller samples the data packets and stores the sensor values internally. Since we have not known before how the values are presented and in which packet, we have logged a couple of data and tried to find a correlation between them and the displayed values on the station. We have found this correlations, but the sampled values are not the same as the displayed, therefore we have added computations to get the exact results in a simple manner. These final values are then transmitted via UART to the node.

### 3.4.3 Development of drivers

The drivers in Contiki are divided into two different categories. There are driver for different microcontroller like UART and driver for a special platform like the temperature sensor. As we added external sensors, we developed only driver that are suitable for the Z1 platform. They can be found in Contiki under /platform/z1/dev. New developed driver are added to the Makefile, too. The Sky node is similar to the Z1 node because both use the same micro-controller. So I have copied the configuration routine for ADC and adapted it to the Z1 platform. These routine files are labeled z1-sensors.{c|h} and contains methods to configure a special ADC channel and to get the current status of it. The developed driver just defines the channel and some other parameter and offers the sensor methods to configure and to read the captured value. This is done for the humidity and the light sensor. The more difficult part is to get the sensed values from the sensor station. As mentioned above we sampled these values with an intermediate system. This micro-controller has a short routine that triggers data on a positive clock edge. We have seen that nine different packets are sent because nine equal bytes appeared periodically. These header are followed by a couple of another bytes of which two seemed to be data. Then we printed them out on a console and tried to find a correlation by increasing the temperature, CO2 or humidity values. After many measured and logged data we have found this correlation and adjusted the code to store these exact values. We have used then the simplest way to provide these data to the Z1 node by sending them via UART. The UART driver can be found under /cpu/msp430/dev/uart1x.c. Unfortunately this driver has been copied by the original developers from the sky node without adjusting them. So I corrected it to give us the possibility to

receive packets. The packets sent by the pic controller invokes an interrupt that is handled by a developed method. This method in /platform/z1/dev/sensorstation.{c|h} stores the packets internally. Each transmission is separated by a special byte, followed by the values for CO2, humidity and temperature. Since this complete step is not very quick, all values are stored in a struct and are replaced with new arriving data. The sensor station driver provides a method to configure the UART and to return this sensor struct. All values are not needed in such a short interval that this steps suffice related to speed.

### 3.4.4 Calibration

The datasheets for additional sensors gives linear correlations for the measured voltage to the current values. But the conversion is not very accurate. The sensor station itself is already calibrated, so I have used its current values to calibrate the other additional sensors. This calibration is done by simple multiplication or shifting the ADC values. For the lack of being able to handle floating point numbers, I converted the sensed values in such a way that only integer values are used. The temperature for example leads from 24.5 degree to 2450. Besides this integer value is only one value to transmit to other nodes, instead of using two values for comma separated data.

### 3.4.5 Integration of sensing into TikiDB

TikiDB has one part that is responsible for reading sensor data, the queryresolver. There are already a couple of sensors used by the Sky mote before. So I have replaced one light sensor with the new CO2 sensor method and reused the given temperature and humidity methods. The queryresolver itself resolves the incoming queries by us of queryresolver-util. In the queryresolver sensors are started by calling their calibration methods. The data scheme can be found in the header file where I have added the CO2 sensor, too. For every query the resolver uses a switch-case selection to call the responsible sensor method. This methods are inside the queryresolver-util file and invokes depending on what node Contiki is running, the sensor station read function or just the described ADC function. The returned sensor values then are stored in the corresponding column of TikiDB.

### 3.4.6 Problems and challenges

To create and use the ADC part is not very difficult because it is done by simply configure the converter and it runs without use of Contiki in the background. The values are stored in a defined memory and can be collected every time. The more difficult part is the sensor station. As already explained we have spent much time with finding a correlation between displayed and

triggered data. After this has succeed we have tried to find a way to send data to the Z1 node. The first trial was reusing the I2C. The Z1 acts as a Master for the given temperature and humidity sensor, which are digital I2C slave sensors. Unfortunately the trial interrupted or disturbed by the other sensors in such a way that their values occurred instead the sensor station ones. Further both micro-controller work in another voltage domain so that the pic detected incoming messages but the Z1 not. Thereafter we decided to use the UART, as it is very simply, but we have got some troubles, too. The arriving of bytes invokes an interrupt, but only one byte has been arrived in a sending interval. We have then detected that an overflow error occurs every second time. Normally the UART is as quick as the sending part. Especially is this case, as the MSP430 is much faster as the pic one. But it seems that Contiki itself is not quick enough to handle this, so we added a pause of a few milliseconds between every packet until no error occurs and every data has arrived correctly.

## 3.5 Web Interface

### 3.5.1 Analysis of an existing Web-Frontend

Instead of developing a new Web-Frontend from scratch we used an existing Web-Frontend which has been developed in summer term 2008 for the Wireless Sensor Networks Lab. This Web-Frontend used a dynamic web page written in the programming language php [5] to generate a dynamic web page which displayed the monitored data. This data was stored in a MySQL database [6] that was read by methods already included in php. The Layout of this web page was made by combining tables and div-tags which were positioned by commands specified in an external css-file.

When a user opened this web page he could see a graph which showed the recorded data of one node and one sensor (i.e. humidity). Both parameters could be set by selecting the according node/sensor. The graph was painted by a JavaScript-library called "PlotKit" [7] that can plot an array of values into a coordinate system. The values were also displayed in a table.

The user could also set the time in which the web page should refresh itself by running a JavaScript command to load new data from the database, so data could be displayed in nearly real-time.

### 3.5.2 Development of a new Web-Frontend

On basis of the above described Web-Frontend we build a new Frontend. This Frontend uses also a css-file to define some basic design parameters, but the major design is made by using tables where specific elements were placed.


Figure 4: Design of the Web-Frontend

On the right size is a small area where the user can set the refresh-time of the page an a button to submit this value.

In the center of the page is the graph which shows the recorded data of the selected nodes and the selected sensor. Below the graph the user can see a list of CheckBoxes where he can see which nodes have data packets in the database and where he can select nodes to display their data in the graph. In this area he can also select which type of sensor information should be displayed in the graph and a button to submit this data so that the new data can be displayed.

To display the data of the selected sensors we decided to use the "Annotated Time Line" graph in the "Google Chart Tools" API [8]. We decided to use this API, because it can display multiple information in different colors at a time (i.e. the temperature of 13 nodes simultaneous) and the user can zoom and scroll into the data by using the scrollbar directly under the graph, so that he can zoom into a region of interest. This is very useful if he wants to display a large dataset. In such a case the user could not see small changes, because the resolution of the graph is to low.

The data for the graph is also based on a MySQL database and the php-script directly connects to it. The list of nodes who have sent packets is received by sending an SQL command to list all nodeIDs. This list is then filtered so that only IDs with values between 50 and 80 are displayed, because other nodeIDs are invalid. Then the script loads all data packets from the user selected nodes and groups them by the timestamp which indicated when they were the data packets were written into the database. Based onto this informations the php-script generates an array for the Google-API which consists of the timestamp of the event and the sensor-values of the selected nodes, where each node has a single column. If a node has no information at a timestamp when another node has data, then the previous known value is chosen for this node.

All data a user submits (refresh-time, nodeIDs and sensor-type) are transmitted to the php-script by using a HTML-formular. Then the php-script generates a php-session for the user, if there is no active session for this user, and writes the submitted data into this session. This guarantees that even when the web page is refreshed by the user (F5-button in the browser) he can still see the nodes he has selected. Otherwise this information would get lost and the user would have to set this informations every time the page reloads. By using php-sessions it is also guaranteed that multiple users can simultaneous display the data of the nodes with individual settings.

### 3.5.3 Problems

A major problem is that we depend to 100% on Google. This means that if Google decides to shut down this API we could no longer display data with it. Another point is that we need access to the internet to load the API, because it can't be downloaded to have it offline. So this Frontend can't be used in a network that is not connected to the internet.

# 4 System Run

Here we describe our two tests at an office and at the surPLUShome and evaluate the results.

## 4.1 Deployment Map

Here we describe how the nodes were deployed in our two test scenarios.

### 4.1.1 Test-Deployment

13 nodes were flashed with their firmware and 12 of them were placed on a table. the other node, who served as the base-station, was connected to the NSLU2-Router [4], which runs a linux. This router collects all data via the result-writer and writes them into a database.

In this test we collected data about the temperature, humidity and CO2.

### 4.1.2 Final-Deployment

19 nodes were flashed with their firmware and all of them were placed on specific locations. These locations were chosen by the architects and are described later. One node (R1) was the base-station and was connected to the NSLU2-Router as in 4.1.1. Because the solar house was not connected to the internet, we could not write the data directly into a database. We decided to connect a laptop to the router which served as a database-server, so that the router could write the collected data into it.

The system will run over a time of two weeks. Then we will collect the data and evaluate them.

The positions of the nodes were chosen as follows:

**Subsystem "Raum"**


Figure 5: Node positions in Subsystem "Raum"

This system used only six nodes. Node 54 is the base-station of the whole system. These nodes monitor the following parameters:

| node-Name | node-ID | temperature | humidity | CO2 | rssi |
|---|---|---|---|---|---|
| R1 | 54 | **x** | **x** | **x** | **x** |
| R2 | 55 | **x** | **x** | **x** | **x** |
| R3 | 51 | **x** | **x** | | **x** |
| R4 | 52 | **x** | **x** | | **x** |
| R5 | 53 | **x** | **x** | | **x** |
| R7 | 56 | **x** | **x** | | **x** |

## Subsystem "Kühldecke"


Figure 6: Node positions in Subsystem "Kühldecke"

This system used six nodes. These nodes monitor the following parameters:

| node-Name | node-ID | temperature | humidity | CO2 | rssi |
|---|---|---|---|---|---|
| K1 | 57 | **x** | **x** | | **x** |
| K2 | 70 | **x** | **x** | | **x** |
| K3 | 59 | **x** | **x** | | **x** |
| K4 | 60 | **x** | **x** | | **x** |
| K5 | 61 | **x** | **x** | | **x** |
| K6 | 62 | **x** | **x** | | **x** |

**Subsystem "Fassade"**

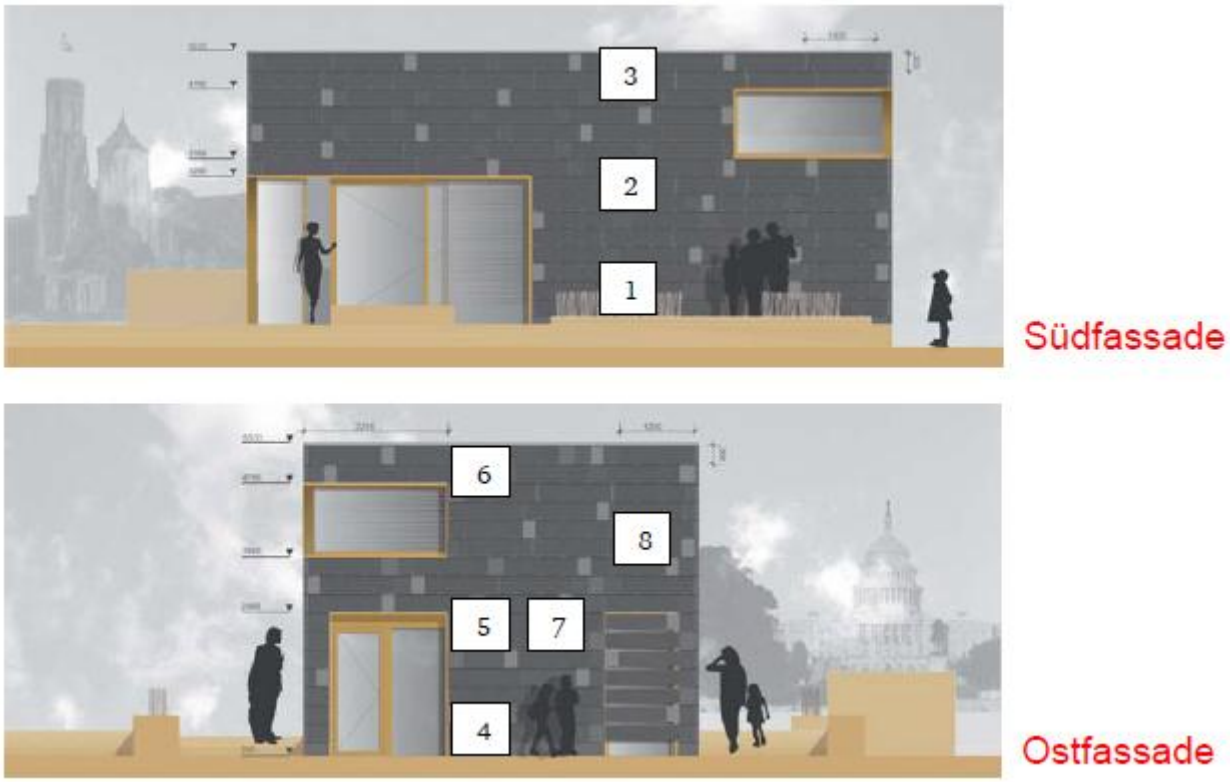


Figure 7: Node positions in Subsystem "Fassade"


Figure 8: Node positions in Subsystem "Fassade"

This system used seven nodes on the north- and east-side of the building. These nodes monitor the following parameters:

| node-Name | node-ID | temperature | humidity | CO2 | rssi |
|---|---|---|---|---|---|
| F4 | 63 | **x** | **x** | | **x** |
| F5 | 64 | **x** | **x** | | **x** |
| F6 | 65 | **x** | **x** | | **x** |
| F7 | 66 | **x** | **x** | | **x** |
| F8 | 67 | **x** | **x** | | **x** |
| F11 | 68 | **x** | **x** | | **x** |
| F12 | 69 | **x** | **x** | | **x** |

## 4.2 Results

Here we describe the results gathered from our deployments.

### 4.2.1 Test-Deployment

This test run showed that the system can run successfully over a long period of time. But we also noticed that the root node restarted very often. These reboots were unexpected to us, but we think that this is a general problem of Contiki, because this operating-system is still in beta-phase. We could not debug this problem because we had no debugging-tools for it.

A second problem was, that we had a very high amount of wrong data like illegal node-ids and illegal data-values like negative humidity-values. These problems may occur when more than one node is sending data to the base-station.

### 4.2.2 Final-Deployment

At first we describe the results of the deployment for each one of the three subsystems and give then a short result for the whole system.

All descriptions concerning the number of data packets related to a version of the database where all wrong values were deleted. Wrong values are illegal sensor values like negative humidity and CO2 values and illegal nodeID's.

## Subsystem "Raum"

By looking at the amount of data packets recorded by the database we could make the following observations:

- Each of the nodes R3, R4 and R7 have sent nearly 300 packets. This is a very low value compared to the other nodes in this subsystem. We think that this is caused by the direction of the base-station antenna, so that it could not receive the data packets well.
- Each of the nodes R2 and R5 have sent nearly 700 packets. This is a very high value. So we think that the communication is best in the upper half of the room.
- The highest amount of data packets in the database is caused by node R1, because it is the base-station. This high amount is also caused by the fact that TikiDB prints out the collected data of the root-node every time it received new data by other nodes.

By looking at the sensor-values we could see that the values of the two CO2-Sensors on the nodes R1 and R2 have nearly the same values over the whole time. This observation could also be made by looking at the temperature and humidity measured by all nodes. We think that this shows clearly that the ventilation of the room is very good.

Another observation was that the change of day and night could be clearly seen in the data-values.

## Subsystem "Kühldecke"

By looking at the amount of data packets recorded by the database we could make the following observations:

- Each of the nodes K1, K3 and K4 has sent between 700 and 900 packets. This is a very high value compared to the other nodes in this subsystem. Compared to the result of the nodes R2 and R5 in subsystem "Room" we think that the communication is best in the upper half of the room on the east-side of the building.
- Each of the nodes K2, K5 and K6 has sent between 300 and 500 packets. This is a very low value compared to the other nodes in this subsystem. This observation seems to support the previous hypothesis that the communication is best in the upper half of the room on the east-side of the building.

By looking at the sensor-values we could see that the values of all sensor values of all nodes have nearly the same values over the whole time. The

only anomaly can be seen on 19.06. at 10 am when the temperature difference between east and west is between 6 and 10 °C. But after 12 - 14 hours the ventilation system has balanced these temperatures.

## Subsystem "Fassade"

By looking at the amount of data packets recorded by the database we could make the following observations:

- It can be seen that if a node has a lower position it's amount of data packets is very high. But if a node has a higher position it's amount of data packets is lower. So node F4 has a higher amount of data packets than node F6.

By looking at the sensor-values we could see that the change of day and night could be clearly seen. Another observation by comparing the values of the east- and north-side of the building is, that the temperatures on the east-side are higher than on the north-side. This is because on the sun will never shine directly onto the north-side.

## Total System

It can be seen that there are two major holes in the database which last over several days. This may be because of many reboots of the base-station. The reboots could also be observed in the test deployment, but in this scenario the system rebooted successfully. The reason for this failure can not be found directly. We think that it is because of the Contiki operating system. Similar problems occurred in the development phase, but we thought we had eliminated them successfully.

The poor yield of data packets (about 50% of the values entered in the database were valid) is probably due to the environment. Incorrect data values result from the fact that data packets may have collided.

A major surprise was that the rssi-value was not recorded in the database. This is because the router ran the wrong version of the result-writer.

Despite all the problems the system collected accurate and responsible data values when it ran properly.

# 5 Conclusion

The Contiki operating system is still in beta-phase and had many bugs which had to be fixed before it could be used on the Zoleria Z1 nodes. This had caused most of the problems we had to deal with. These problems could have been solved in a better way than by using the try&error-method if we could use a debugging-tool. Instead we had to use the printf-method to look where the problem may be.

# 6 Literature


[1] http://www.solardecathlon.tu-darmstadt.de/home/home.de.jsp

[2] http://www.zolertia.com/products/Z1

[3] http://www.dvs.tu-darmstadt.de/teaching/wsnlab/2011/slides/09.final.pdf

[4] http://de.wikipedia.org/wiki/NSLU2

[5] http://www.php.net

[6] http://www.mysql.de

[7] http://www.liquidx.net/plotkit/

[8] http://code.google.com/intl/de-DE/apis/chart/interactive/docs/gallery/annotatedtimeline.html