



Semester Project Report

# **Wireless Sensor Networks Application for Agricultural Environment Sensing in Developing Countries**

Francois Depienne

Master Candidate in Communication Systems

March 13<sup>th</sup>, 2007

## **Abstract**

COMMONSense Net is a wireless sensor network whose aim is to provide marginal farmers in developing countries with environmental data from their field. The goal is to enable an appropriate water management system given this periodic data.

The project presented in this paper is mainly related to developing a TinyOS2 application for gathering of sensor data towards a sink in the COMMONSense wireless sensor network. Moreover the integration with a MAC/Routing layer has been achieved and a network programming tool has also been integrated. The server side application and user interface has been reprogrammed to work with the new system. Finally, a first deployment of the network is in process in Bangalore, India.

# Table of Contents

Table of Contents .....	3
1 COMMONSense Project Introduction.....	4
2 Wireless Sensor Network Requirements .....	5
3 Global System Overview .....	6
4 Hardware description.....	7
4.1 Sensor Node.....	7
4.2 Antenna.....	8
4.3 Sensors.....	8
4.3.1 Soil Moisture Sensor.....	9
4.3.2 Watermark Sensor.....	10
4.4 Sensorboard.....	12
4.5 Battery.....	14
4.6 Box.....	15
4.7 Total Costs.....	16
5 Software.....	17
5.1 Embedded applications.....	17
5.1.1 CommonSense Application .....	19
5.1.2 Basestation.....	28
5.1.3 Messages.....	30
5.1.4 Deluge .....	34
5.2 Server Side Application.....	36
5.2.1 General Data Logger.....	37
5.2.2 PostgreSQL Database .....	38
5.2.3 Python layer .....	40
5.2.4 Web Interface.....	41
6 Deployment.....	43
7 Main problems .....	43
7.1 Radio .....	43
7.2 Integration with Shockfish MAC .....	43
7.3 Installation of TinyOS 2 and server applications.....	43
7.4 Debugging.....	44
8 Conclusions .....	45
9 References.....	46
10 Appendix.....	47
10.1 TinyOS 1 Installation .....	47
10.2 TinyOS 2 Installation.....	48
10.3 Deluge .....	50
10.4 CVS.....	53
10.5 Compilation on Shockfish Servers .....	55
10.6 Server Installation .....	58
10.7 Server application user's guide .....	59
10.8 Recompiling files from TinyOS 2 Java SDK .....	60

# 1 COMMONSense Project Introduction

## **Water scarcity**

Water consumption around the globe has increased seven-fold in the course of the 20th century. Levels of ground-water are declining. About 1.1 billion people around the world live without satisfactory access to clean water. As world population grows water demand grows as well. According to these facts, the water scarcity problem will increase if we do not act right now.

## **Where to save water?**

Today, agriculture consumes 70% of the fresh water used worldwide by human activity.

The problem of agricultural water management is widely recognized as a major challenge. In developing countries, it is estimated that 40% of the fresh water used for agriculture is lost, either by evaporation, spills or absorption by the deeper layer of the soil.

## **The Indian case**

In India, crop yield heavily depends on unpredictable natural factors like monsoon, floods, and especially drought. The last factor has recently become a major problem for farmers. Additionally, many marginal farmers still make use of archaic agricultural techniques and so do not have the possibility to compete with large western multinationals. In this context, an efficient water management technique to avoid wastes can mitigate the above mentioned effects.

## **What kind of water management?**

One application would be to make use of efficient irrigation. Indeed, only 25% of Indian cultivable surface is using irrigation techniques nowadays. By efficient irrigation, we mean irrigation based on regularly updated soil moisture and ground water data at different spots and different depths in the field. Indeed, different soils let the water percolate faster than others. Moreover different crop types have different moisture requirements for optimal yield.

Moreover, obtaining reliable meteorological data would help farmers to improve their water management.

## **Wireless Sensor Networks as a solution**

The COMMON-Sense project (Community-Oriented Management and Monitoring Of Natural resources via a Sensor network) aims at designing and developing an integrated network of sensors for agricultural water management in the semi-arid rural areas of Chennakeshavapura, a village in the state of Karnataka. Indeed, this system will have an effect on yield at the local level but, in addition, it will allow collecting extensive data that can be used to better understand the effects of water – and other environmental parameters – on agriculture, and thus to develop replicable strategies.

Technically speaking, the COMMON-Sense network consists in a wireless network of ground-sensors that record periodically the state (moisture, watermark, etc.) of the soil. In the system design, sensors record data on a periodic basis and send them in a multi-hop fashion to a centralized processing unit.

## **My assignment**

My job within this project is mainly to develop a TinyOS2 application for gathering of sensor data towards a sink in this wireless sensor network. Moreover I have to integrate the application with a MAC/Routing layer and with a network programming tools called Deluge. The server side application and user interface will have to be reprogrammed to work with the new system. Finally, a first deployment of the network will be done in Bangalore, India.

## 2 Wireless Sensor Network Requirements

Our wireless sensor network consists in spatially distributed autonomous devices equipped with sensors, RF transceivers, microcontrollers and batteries. Hereafter I will describe the optimal properties of this kind of wireless sensor network.

**Self-organization:** The network should auto-detect newly arrived nodes or removed/stolen/broken nodes and adapt the tree to route messages accordingly. After the deployment, no human interaction should be required anymore during the network lifetime (limited by battery life).

**Scalability:** Adding a huge number of nodes should still enable acceptable performance in terms of latency and packet drops.

**Multi-hop transmission:** When the base station is not within communication range, nodes should be able to send their data to neighboring nodes that will forward the messages towards the base station.

**Latency:** The network should forward messages towards the sink with limited latency. This is not the highest priority for our application though.

**Static/Mobile network:** in our case, we are considering a wireless sensor network which has a static topology. However, as mentioned before, nodes may be added or removed at different time instants and the topology must adapt automatically.

**Size:** Sensor nodes for environmental applications should be as small as possible to be “discrete”. Indeed, people might be tempted to steal them or to take them home by pure curiosity [6].

**Price:** since we have a large number of nodes and because our application is agriculture oriented, our nodes must be as cheap as possible. This is of course not the case at this level of research, because we are lacking economies of scale. However, we should keep in mind the financial constraint when choosing each components of the system.

**Robust to physical environment:** Nodes must be robust to work long term in a potentially hostile environment: high humidity during monsoon, extreme heat during daytime, strong electromagnetic fields during thunder, sun rays, presence of animals and human curiosity... all these aspects make it extremely difficult to provide a reliable system.

**Power consumption:** One main challenge is to design low-power hardware components and to develop a software platform that minimizes power consumption. This can be achieved by limiting the time during which the radio chip and the microcontroller are turned on. An efficient MAC layer protocol must be considered for this purpose.

**Low data rate:** For our application (sending of sensor data), we do not need high transmission rates. A few kbps will be sufficient. This also enables lower power consumption and lower bit error rate while transmitting.

### 3 Global System Overview

Now that we have looked at the optimal properties for our sensor network, I will describe our actual system. I will discuss at each step the reasons for our choices and the expected performance.

Our deployment will consist in at least 25 sensor nodes deployed in an area of 1 square kilometer (density of 25 nodes/km<sup>2</sup>), which represents an average distance between neighbors of 250 m. We want our system to operate for a minimum period of time of 6 months without any physical human intervention.

The sensor nodes are running a TinyOS 2 application that will poll the sensors and send the data over the air.

A base station (also a sensor node) will collect the sensor data. It is connected to a server running a Java application that will process the data and store it in a database. A user interface is provided enabling users to display sensor data from the database and interact with the network (send enquiries to sensor nodes...) through predefined commands.

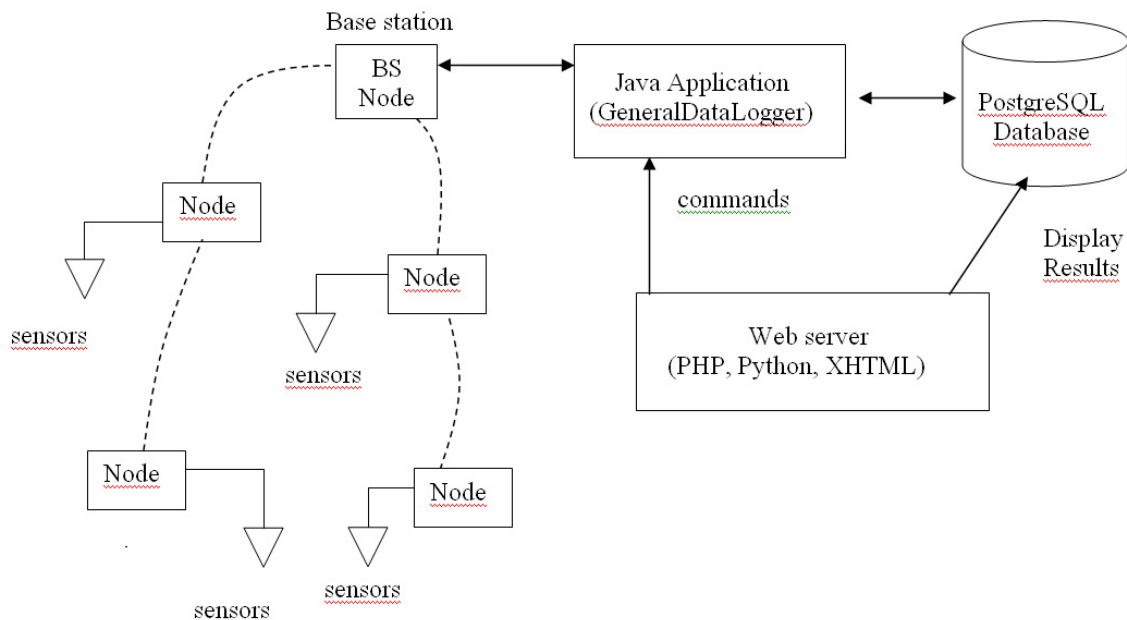


Figure 1: Global System

In the following subsection will describe each part of the system in a detailed fashion.

## 4 Hardware description

Here I will describe the hardware chosen for the project. Most aspects were already decided when I started the project, however I discussed several aspects with Jacques in order to choose the right platform

### 4.1 Sensor Node

There are many TinyOS 2 compatible sensor nodes on the market. Our choice has been mainly oriented by the following criteria:

- low power consumption
- high distance communication range
- low cost
- accessibility of support
- resistance to Indian weather/temperature conditions

The best platform according to these criteria was the **Tinynode 584 platform** developed by **Shockfish**, a company based in Ecublens, Switzerland.

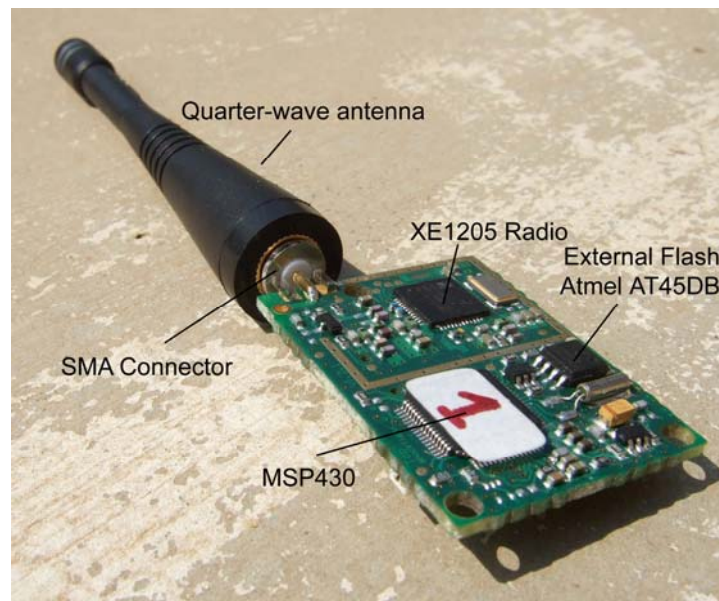


Figure 2: Tinynode

Here is a brief list of important specifications:

#### ***Microcontroller***

It uses the MSP430 microcontroller which is a ubiquitous chip for low-power applications. (8 MHz, 10K RAM, 48K program space). The ADC converter has 12bits, enabling sensor values ranging from 0 to 4095, which is sufficient for our sensor data.

#### ***Transceiver***

The wireless transceiver is a Xemics 1205 multi-channel 868MHz low-power transceiver. This frequency band is an ISM band (Industrial Scientific and Medical band) which is a license-free band. The data rate ranges from 38Kbps to 152.3kbps (default is 76 kbps). The transceiver also enables to

adapt transmission power from 5dBm up to 12dBm, which enables to reduce consumption when nodes are placed near each other. The lower the data rate the higher the communication range, so we can expect to obtain low packet drops at distances of up to 500 meters in rural area. Moreover, we will place our nodes at about 2 meters above the ground so that we reach RF Line of Sight (a Fresnel zone between both communication parties free of obstacles), which improves the transmission quality [9].

#### ***Power Consumption***

We have very low consumption:

Sleep: 0.004mA  
Reception: 16mA  
Transmission: 25mA (0dBm)- 62mA (15dBm)

#### ***Weather conditions***

Finally the tinynode works properly within -40°C to 85°C which is suitable for the most extreme Indian weather conditions.

For more information about the Tinynode platform, please refer to the Fact Sheet (<http://www.tinynode.com>)

## **4.2 Antenna**

We use a regular omni-directional quarter-wave antenna with SMA connector. The tinynodes come with an SMA connector directly soldered on the board. .



**Figure 3: 1/4 wave antenna**

## **4.3 Sensors**

Our sensors will measure two different type of information about the water content inside the soil:

- Volumetric Moisture Content or Volumetric Water Content (VMC/VWC) with soil moisture sensors
- Soil-matrix potential (SMP) with watermark sensors (associated with temperature sensors)

Both will be measured at two depths below the ground: 10cm and 30cm. These heights represent moisture measurement at the “soil surface” as well as measurement deeper at the root level of the plants. Depending on the soil type and density, the water may percolate at different speeds until reaching the water table.

### 4.3.1 Soil Moisture Sensor

Volumetric moisture content  $VMC$  is defined as the fraction of water volume  $V_w$  contained in the total volume  $V_{tot}$  (soil  $V_s$  and water) of the sample.

$$VMC = V_w / V_{tot} = V_w / (V_s + V_w)$$

The corresponding sensor will be EC-5 sensor from Ech2o (<http://www.ech2o.com>). It measures the dielectric constant of the soil in order to find its volumetric water content. Since the dielectric constant of water (80) is much higher than that of air (1) or soil minerals (in between, variable), the dielectric constant of the soil is a sensitive measure of water content.

Compared to other sensors from Ech2o (EC-10 and EC-20), the EC-5 can measure VWC from 0 to 100%, and allows accurate measurement of all soil types and a much wider range of salinities.



Figure 4: Ech2o EC-5 Soil Moisture Sensor

Sensors give a voltage output related to the dielectric constant. Because our ADC is based on 12 bits and  $V_{cc}$  being 2500mV, the actual voltages are obtained by:  $V = \text{Output} * 2500 / (2^{12})$  mV. To obtain the VWC from this voltage, there are equations for different soil types. For all mineral soil types with electrical conductivities from 0.1 dS/m to 10 dS/m (our case), the following calibration equation is the most accurate, according to the Ech2o:

$$VWC = 11.9 * 10E-4 * V - 0.401$$

where  $V$  is the output of the probe in mV when excited at 2500mV.

From our tests we obtained the following values:

Environment	Quantized output (O)	Voltage (V)
Water	950	580mV
Air	270	165mV
Humid ground (after rain)	500-700	300-430mV
Dry ground	430	262mV

However we noticed that these results do not correspond to correct VWC values with the formula from Ech2o. So we decided to work with agronomists in Bangalore in order to calibrate the probes more accurately given these voltages.

### 4.3.2 Watermark Sensor

Although volumetric moisture content give good information about the percentage of water contained in the soil, it does not say how easy this water can be used by a plant, for example. This aspect is very important in agriculture.

Beneath the soil surface, soil particles, water, air and plant roots share the space. Water cannot move freely as it is the case on the surface; its movement is determined by the soil structure. Moreover, water is naturally attracted by the soil particles; this is due to a phenomenon called *capillarity*. The smaller the capillary (or pore) between soil particles is, the harder it will be for the plant to extract the water from it. Thus, when the soil is very wet, all the large pores inside the soil are filled with water, which makes it very easy for the plant to extract. On the contrary, when the soil is very dry, the remaining water is contained in the very small pores and is therefore very difficult for the plant to extract. This is illustrated in figure 5.

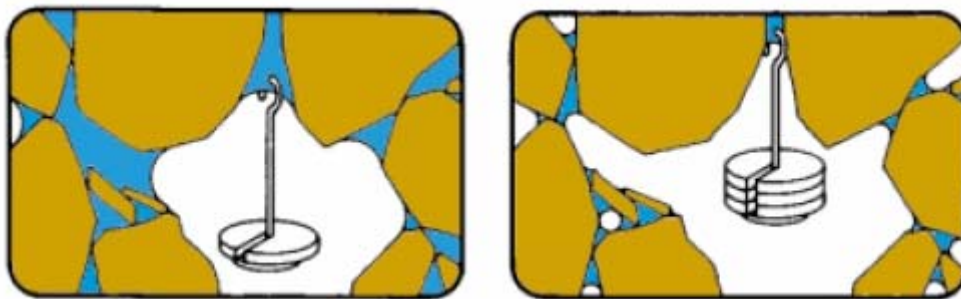


Figure 5: Capillarity and SMP

This phenomenon is quantified by the parameter called *Soil Matrix Potential (SMP)* or *Soil Water Potential* and is defined as the force necessary to remove water from the soil per surface unit. SMP is therefore of the same unit as pressure as it is a force divided by surface. SMP is commonly expressed in kPa (“kilo-Pascal”) which is equal to cbar (“centi-bar”). This scale is usually negative: 0 kPa means that the soil is saturated with water and is therefore immediately available by the plant whereas a -100 kPa measure testifies a very dry soil (the water is very difficult to extract from the pores).

One of the most common methods of measuring SMP is with resistive sensors. A resistive sensor consists of a porous block made of gypsum or fiberglass. The block contains two electrodes connected to a conductive wire. When buried in the ground, the water is free to move in and out of the sensor, until it is at equilibrium with the soil moisture. The electrical resistance of the gypsum block then depends on the soil matrix potential.

The resistance value can however not be read using a simple DC voltage; it must be excited by an AC voltage in order to perform that

Watermarks are basically ameliorated gypsum blocks. The granular matrix is surrounded by a synthetic membrane for protection against deterioration. This makes the Watermark much more durable than a gypsum block. Moreover, watermarks are very cheap.

A disadvantage is that the characteristics between two Watermark sensors can change and that the conversion kOhm to kPa is still soil-specific. However, these changes are much smaller than for gypsum blocks.

The Watermark sensor output is the electrical resistance. This resistance basically depends on the water contained in the pores of the gypsum block (matrix) of the sensor.

Two steps are required to be able to interpret the Soil Matrix Potential (SMP) from the Watermark sensor. The first step consists in reading the electrical resistance of the Watermark sensor; the second one is the conversion from the electrical resistance (in kOhm) into the actual SMP value (in kPa).

The watermark type we are using is the “Irrrometer Watermark Soil Moisture Sensors” (<http://www.irrometer.com>)



**Figure 6: Irrrometer Watermark Sensor**

A non-linear equation has been developed to convert the electrical resistance (in kOhm) of the Watermark sensor into Soil Matrix Potential (in kPa). This conversion equation is the one used by the Watermark electronic reader from Irrrometer.

$$SMP = \frac{4.093 + 3.213 * R}{1 - 0.009733 * R - 0.01205 * T_s}$$

*SMP* is the Soil Matrix Potential in [kPa], *R* is the Watermark resistance value in [kOhm] and *T<sub>s</sub>* is the estimated or measured soil temperature in [°C] near the probe.

However, the coefficients of this equation can be soil-specific; these that are appropriate for a clay soil may not be appropriate for a sandy soil. For a very accurate application Watermark sensors should be calibrated in order to find the optimum coefficients.

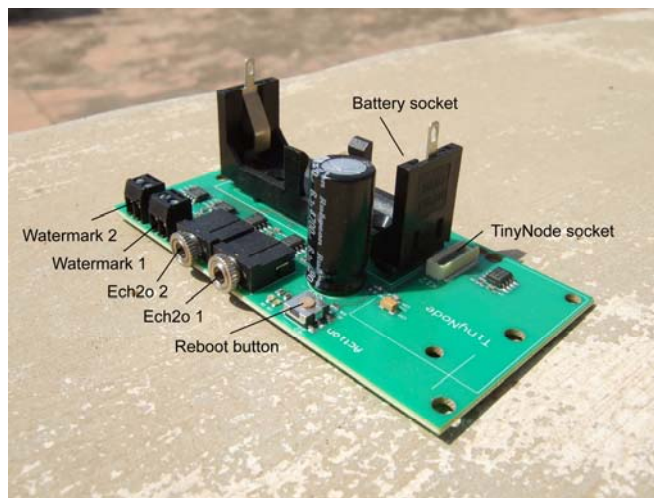
The conversion equation takes into account the effect of temperature. As a matter of fact, as temperature increases the Watermark resistance decreases. Thus, variations in soil temperature can affect Soil Water Potential readings by 1 to 3% per °C. Moreover, the dryer the soil is, the larger the effect of temperature becomes. It is therefore necessary to take into account the effect of temperature.

## 4.4 Sensorboard

The tinynode needs an interface board that provides a power source, circuits and connectors to the sensors. At first we used the standard extension board from Shockfish for this purpose. However, this board was not optimized for our application: on the one hand we needed additional I/O connectors for the sensors. On the other hand, some circuits from the extension board were not useful for us, so that we could reduce the price of the board.

I also identified some coupling issues between two sensors. If the value of one soil moisture sensor increased, the other one increased too. The problem came from the fact that the sensor circuits were interconnected at the excitation line, so we concluded that our final board must have separate circuits for each sensor.

Here after we describe the final board:



**Figure 7: Final acquisition board**

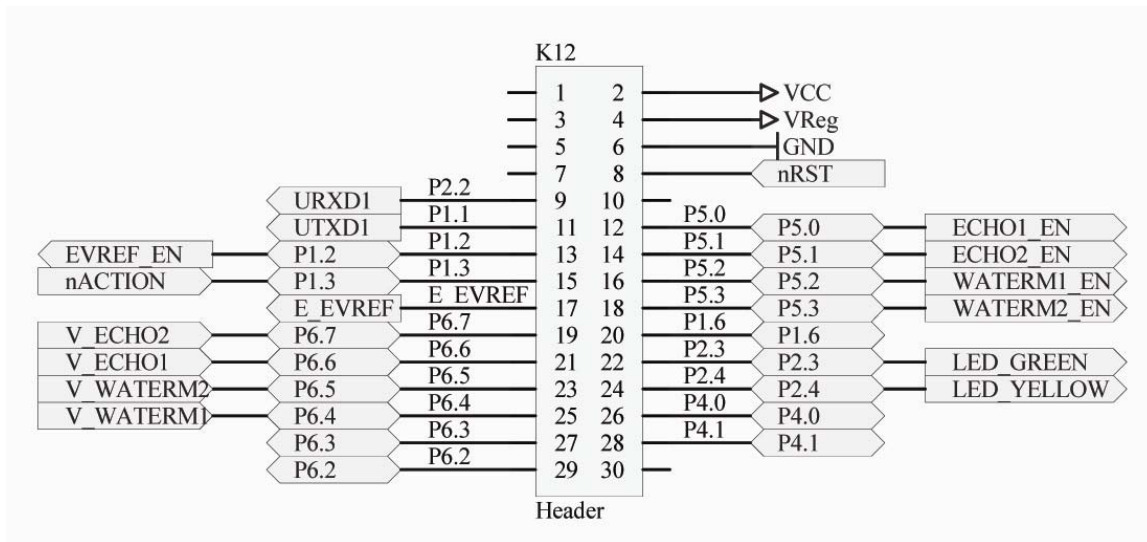
We have:

- two connectors and driver circuits for Ech2o EC-5 soil moisture sensors
- two connectors and driver circuits for Watermark sensors
- socket for a AA lithium 3.6V battery.

Next we provide the schemas of the circuits of the board. These have been designed by Roger from Shockfish given our specifications.

**Tinynode socket:**

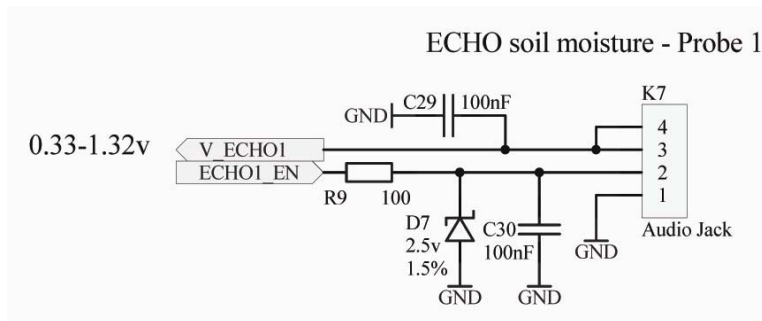
We see that we have mainly I/Os for Leds, the four sensors as well as power source and ground. This schema was useful for me when developing the software driver for the probes.



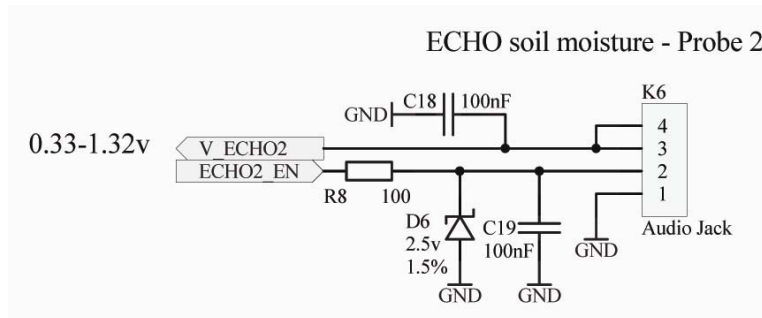
**Figure 8: Tinynode socket pins**

**Echzo Sensor Driver Circuits:**

This circuit enables simple DC voltage application on the probes.



**Figure 9: Echzo Driver Circuit 1**



**Figure 10: Echzo Driver Circuit 2**

### Watermark Sensor Driver Circuits:

Here we need to apply an AC voltage. For this purpose we use a regular 555 timer.

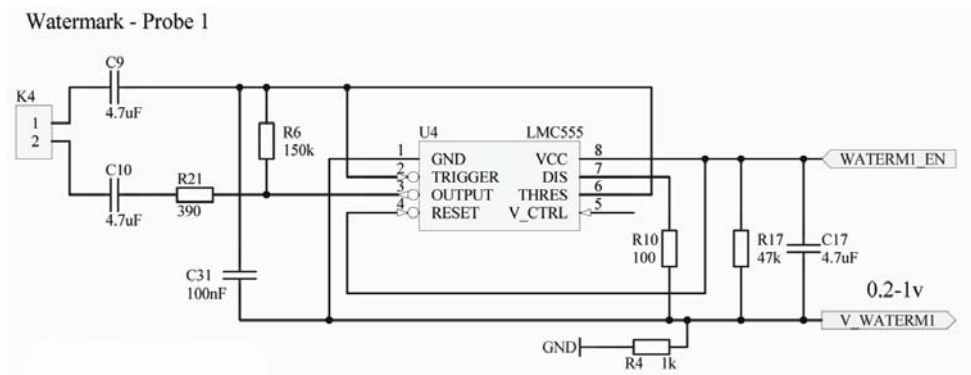


Figure 11: Watermark Sensor Driver Circuit 1

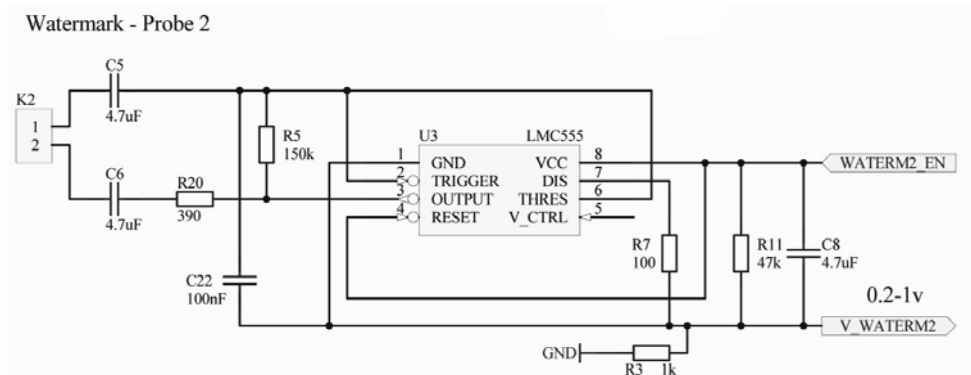


Figure 12: Watermark Sensor Driver Circuit 2

## 4.5 Battery

As a power source we will use a single AA 3.6V battery (Model: ER14505H) that will fit into a dedicated socket.



Figure 13: ER14505H Battery

Our battery has 2700 mAh. For a 6 months autonomy, our platform must consume less than 0.625 mA on average.

## 4.6 Box

To enable a robust system in rural Indian areas, we have to make sure that humidity will not reach the electronic circuits of the board. This can cause short circuits which damage the boards. For this purpose, we use polycarbonate cases from FIBOX (norm IP 66/67)<sup>1</sup>. They are “dust-tight” and “Water from heavy seas or projected in powerful jets shall not enter the enclosure in harmful quantities”. We also included a pressure equalizer plug to avoid condensation inside the box, which would harm our system.

As an additional security, we put silicone inside the screws to improve sealing.



Figure 14: The box

---

### <sup>1</sup> Ingress protection rating

The first digit qualifies the protection against the “access to hazardous parts (e.g., electrical conductors, moving parts) and the ingress of solid foreign objects”.

The second digit corresponds to the protection against harmful ingress of water.

## 4.7 Total Costs

Here is a list of the main hardware components and their price.

Tinynode	150 CHF
Sensorboard	55 CHF
Ech2o sensor	100 CHF
Watermark sensor	50 CHF
Box	30 CHF
Battery	3.5 CHF
<hr/>	
<b>Total:</b>	<b>388.5 CHF</b>

This relatively high price clarifies that production costs of such sensors have to be reduced by approximately 20 times to become affordable for agricultural environment sensing applications. However our research goal is to convince farmers or environment related professionals of the usefulness of sensors through this pilot project.

## 5 Software

Different software aspects had to be implemented in this project

- Embedded code on nodes (NesC)
- Embedded code on the base station (NesC)
- Server data logger application (Java)
- Database for sensor data storage (PostgreSQL)
- Web user interface for applying commands and displaying result charts (Python, PHP, HTML)

The two first parts had to be programmed from scratch for TinyOS 2. The three last aspects had to be reimplemented partially to comply with the new embedded TinyOS2 code on the nodes.

### 5.1 Embedded applications

This part of the code has to take care of the following aspects:

- periodic retrieval of current sensor data
- sending of sensor data to the network
- deal with commands sent from the base station and send reply messages where appropriated
- route the information towards the sink
- discover network neighbors and create a routing tree

All these aspects must be programmed with one main goal in mind: reduce the power consumption to its minimal.

Here after I describe the framework I used while programming:

#### TinyOS 2

TinyOS 2 is a new, open source, energy efficient, small operating system developed by UC Berkeley. It is the most popular platform enabling large scale, self-configuring sensor networks. In theory, Shockfish tinynodes should work “out-of-the-box” with TinyOS 2.

For the installation guidelines, please follow the corresponding appendix of this report.

Between TinyOS 1 and TinyOS 2, some aspects have changed. A code for TinyOS 1 will not work on TinyOS2. For a description of differences see the document: <http://www.tinyos.net/tinyos-2.x/doc/html/overview.html>

#### nesC

nesC is an extended version of the C programming language, adding the structuring concepts and execution model of TinyOS. Because hardware is almost always split-phase rather than blocking, TinyOS uses this model too. For example, to acquire a sensor reading with an analog-to-digital converter (ADC), software writes to a few configuration registers to start a sample. When the ADC sample completes, the hardware issues an interrupt, and the software reads the value out of a data register.

Programs are built out of *components* (modules or configuration). These components are "wired" together to form whole programs.

A *configuration* contains all the wiring to other components that a given component needs to perform its operations.

A *module* contains the corresponding implementation code.

Components can provide or use *interfaces*. An interface contains the routines that a component providing the interface must implement, and that can be called by components using the interface. The routines can be of two types: commands, that clients of the interfaces can invoke to perform an operation, and event, that are returned when a given condition is raised (typically when an operation is complete, a resource becomes ready etc.). The provider of the interface must send a signal corresponding to the event, and the client of the interface must implement an event handler for that signal.

The following conventions are *best practices* for file names:

Interfaces	<interfacename>IF.nc
Main application	<nameofapplication>AppC.nc
Header	<headertype>.h
Public module/configuration:	<modulename>C.nc
Private module/configuration:	<modulename>P.nc

## CVS

TinyOS being developed by a community of programmers, CVS is used to manage source file updates and enabling cooperative programming. For a short introduction about CVS, please see the corresponding appendix.

## 5.1.1 CommonSense Application

This application is the one we install on the sensor nodes. During the course of the project, major structural changes have been made to improve the simplicity, power consumption on the one hand, and for the integration with the MAC/Routing layer provided by Shockfish on the other hand. Please have a look at the chapter “Main problems” to see why we chose to work with the Shockfish MAC/Routing layer.

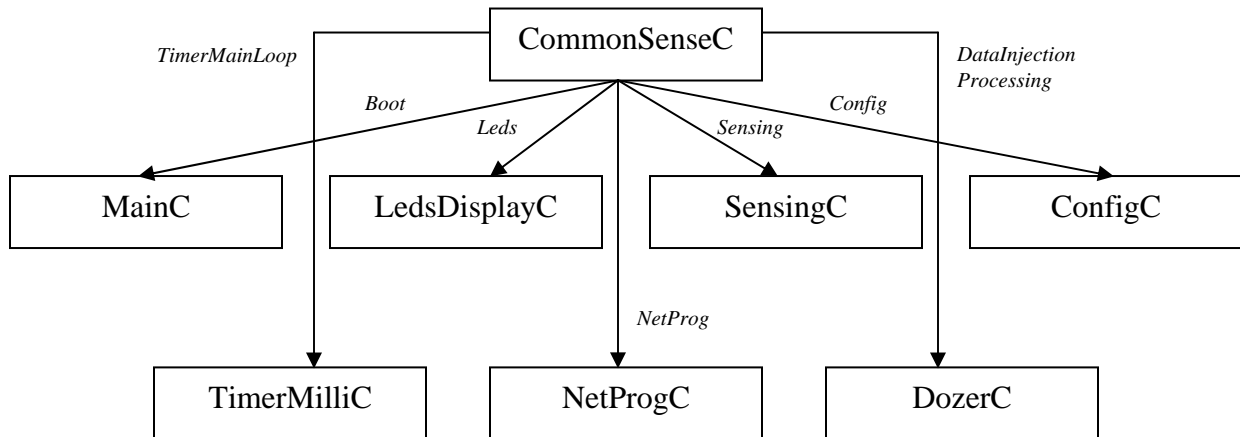
At first, the application was very simple with one module. I then tried to structure the software with more modules to improve visibility and flexibility. However, as a drawback we achieved much higher power consumption (up to 3,5mA). The final application is divided in only a few interconnected modules, representing a well structured and low-power application. Consumption performance of XXXXXXXXXXXXX mA.

Here after I will describe the different modules and wirings. The graphs represent the wirings. Boxes represent components. Arrow labels are interface names provided/used by the components.

### 5.1.1.1 CommonSenseAppC/CommonSenseC

#### Wirings

CommonSenseAppC.nc contains the main wirings of our application, as follows:



*CommonSenseC* module represents the main application.

*MainC* as a wiring, corresponds to a “main” function in C, the bootstrap of the application.

The *LedsDisplayC* configuration is mainly used for debugging purposes. Tossim, a simulation tool for TinyOS 1, has been ported to TinyOS 2, but only for the micaz platform. There is currently no other way for debugging the software with Tinynodes than using Leds. In the final application, Leds will of course all be turned off to reduce power consumption.

*SensingC* is a configuration in charge of the sensing process, the polling of the different sensors.

*ConfigC* is a getter/setter module for the current configuration of the application.

*TimerMilliC* is used for the main scheduling of periodic sensing.

*NetProgC* is used for network programming, the processing commands sent by the server application. It is part of the ShockFish MAC.

*DozerC* is used to send messages on the network (e.g., sensor data). Moreover it schedules time slots where the MSP430 is dedicated only to the radio and time slots for application processing. This is due to problems experienced with the shared SPI between XXXXX. This is part of the Shockfish MAC.

## Application Flow

CommonSenseC loads the configuration file that can either be saved on the external flash or simply from the CommonSense.h header file. In the meanwhile, DozerC starts beaconing the neighbors with regular updates and creates a spanning tree for the routing of packets towards the sink.

Then, if sensing is enabled, CommonSenseC starts the periodic sensing process. The period used is one hour, which is sufficient for our application and enables low power consumption.

Each time the timer is fired, Sensing.startSensing() is called. When the sensing process is over, SensingC signals an event on CommonSenseC providing the sensor payload. CommonSenseC then uses DozerC to send this payload towards the sink.

If a command from the server arrives, NetProg.setParam or NetProg.getParam events are signaled on CommonSenseC. The command processing is done and if applicable, DozerC is used to send a corresponding response towards the server.

## Code Details

Here after I explain how I integrated the Shockfish MAC within CommonSenseC.nc. Other parts of the code are trivial to understand.

*Headers:*

The following header files that are present on the Shockfish server must be included:

```
#include "circqueue.h"
#include "messages.h"
#include "constants.h"
```

*Interfaces:*

You have to use 3 interfaces:

```
module CommonSenseC
{
  uses {
    ...
    interface DataInjection;
    interface NetProg;
    interface Processing;
  }
}
```

### *Synchronized Sensing:*

When booted, I start the main sensing timer. Note that `SENSOR_SAMPLING_INTERVAL_IN_MS` is present in `app_cst.h` in the CommonSense base directory. In the compiled application though, it is present in `constants.h`

```
event void Boot.booted() {
    ...
    call
    TimerMainLoop.startPeriodic(SENSOR_SAMPLING_INTERVAL_IN_MS)
}
```

When the timer is fired, we do not directly start sensing. This is for scheduling purposes with the radio. Indeed, the Shockfish MAC reserves the MSP430 completely for radio when sending. This is due to the synchronization problems with the radio we had at the beginning. So the only thing I do here is setting the variable `doProcessing` to `TRUE`.

```
event void TimerMainLoop.fired() {
    doProcessing = TRUE;
}
```

The event `Processing.doProcessing()` is signaled when the MSP430 is “free to use” for our application. It is signaled with the same frequency than the beaconing of the MAC (defined in `app_cst.h` under `INTERVAL_TIME_IN_MS`). It is at this step that we start one sensing process, if `doProcessing` has been set to `TRUE` before. We then have to set `doProcessing` to `FALSE` to avoid restarting sensing at the next `Processing.doProcessing()` event.

```
event error_t Processing.doProcessing() {
    if (doProcessing) {
        call Sensing.startSensing(config);
        doProcessing = FALSE;
    }
    return SUCCESS;
}
```

### *Netprog:*

`NetProg.setParam` is signaled when a “SET” command arrives. A “SET” command is a command that carries a command type and a value.

```
event error_t NetProg.setParam(uint8_t appCmd, uint8_t val) {
    switch(appCmd) {
        ...
    }
}
```

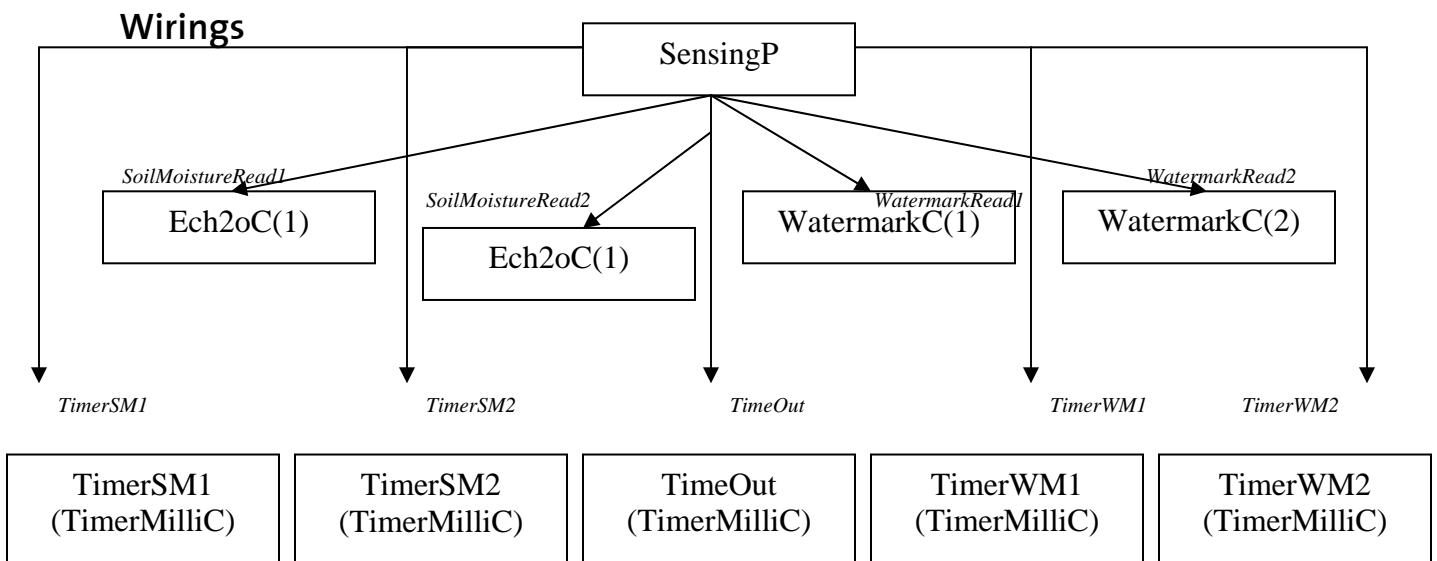
NetProg.getParam is signaled when a “GET” command arrives. A “GET” command is a command that carries only a command type, e.g., getRFPower.

```

event uint8_t NetProg.getParam(uint8_t _cmd) {
    switch(_cmd) {
        ...
    }
}

```

### 5.1.1.2 SensingC/SensingP



*SensingP* is the main sensing module.

*Ech2oC(1)/(2)* are generic configurations for soil moisture sensor drivers

*WatermarkC(1)/(2)* are generic configurations for watermark drivers

*TimerSM1/2* and *TimerWM1/2* are the timers used when polling each sensor

*TimeOut* is a timer used as a time out when a sensor does not respond.

### Application Flow

As soon as *SensingC.startSensing()* is called from *CommonSenseC*, *SensingC* checks the current configuration to extract the “sensorflags” telling which sensors are active. Then it starts polling the corresponding sensors one after the other.

For each sensor, we take a few samples at different intervals of time. The time intervals and the number of samples are the result of discussions with the Hydrum group from EPFL and our observations. We noticed that values obtained by soil moisture sensors are fluctuating (about 5 - 10% with the old board, much less though with the new acquisition board) so that taking an

average over a few samples is necessary. We will also provide the maximal and minimal value obtained within the samples. Additionally, we keep an interval of time between samples, as we recognized that discharging the sensor and the soil takes a while and this may influence the subsequent measurement.

The following values will be used:

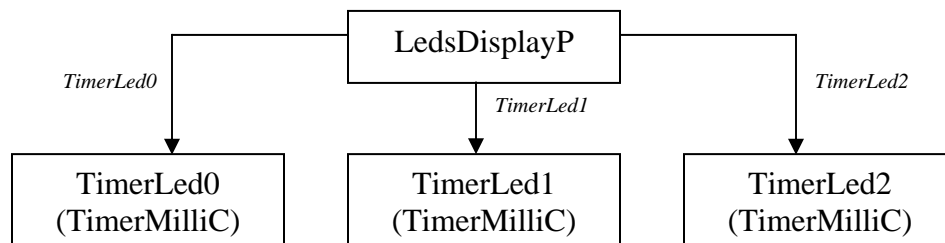
<i>Sensor Type</i>	<i>Number of samples</i>	<i>Interval between samples</i>
Watermark	20	105
Soil moisture	20	105

Each time we poll a sensor, we call a time out timer in parallel which assures the continuity of our application (no deadlock). If the timeout timer is fired, the corresponding sensor value will be 0. This enables to identify non-working sensors from the server side.

When all the active sensors have been polled, SensingP puts the sensor data into a data structure and signals an event on CommonSenseC with this sensor payload.

### 5.1.1.3 LedsDisplayC/LedsDisplayP

#### Wirings



#### Description

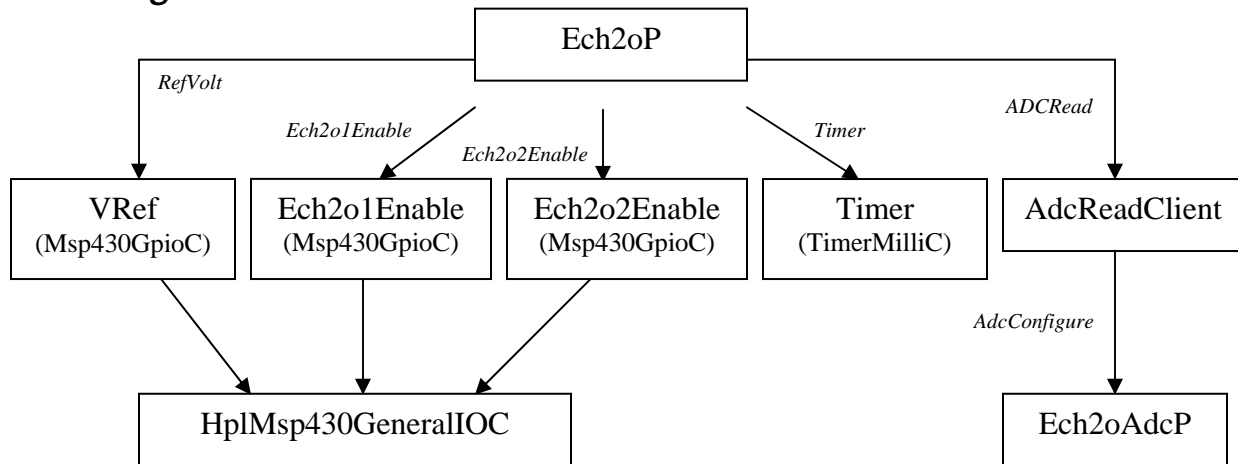
The basic TinyOS2 Leds interface only enabled setting, clearing and toggling the Leds. LedsDisplayP enables Leds to blink (turning a Led on and automatically off again after a few ms). Each Led needs its own timer for that purpose.

### 5.1.1.4 Sensor Drivers

This part describes the sensor drivers I developed for Ech2o and Watermark sensors. These are used by SensingC/P.

#### 5.1.1.4.1 Ech2o Soil moisture

##### Wirings



This corresponds to the high level part of the Ech2o driver. We did not modify any source code from the official TinyOS tree, but added our own modules.

*VRef* is a MSP430 input/output pin for the reference voltage.

*Ech2o1Enable* is a MSP430 input/output pin that is SET (Vcc) when sensor is read, CLR otherwise

*AdcReadClient* deals with the reading and ADC conversion from V\_ECHO<1/2> pin. This pin corresponds to the sensor data voltage.

*Ech2oAdcP* defines the configuration of the ADC port, given the header file Ech2o.h

##### Application Flow

When sensing is required, Ech2oP sets the Ech2o<1/2>Enable as an output pin and sets it to Vcc. This voltage is applied at the sensor during 10-15ms, after which the voltage on pin V\_ECHO2 is read. During this phase about 10mA are consumed.

## Code Details

Here are the pin wirings from Ech2oC, based on Figure 8: Tinynode socket pins

```
Ech2o1Enable -> HplMsp430GeneralIOC.Port50;  
Ech2o2Enable -> HplMsp430GeneralIOC.Port51;  
VRef -> HplMsp430GeneralIOC.Port12;
```

Here is the ADC configuration of Ech2o.h, where we define the corresponding channel.

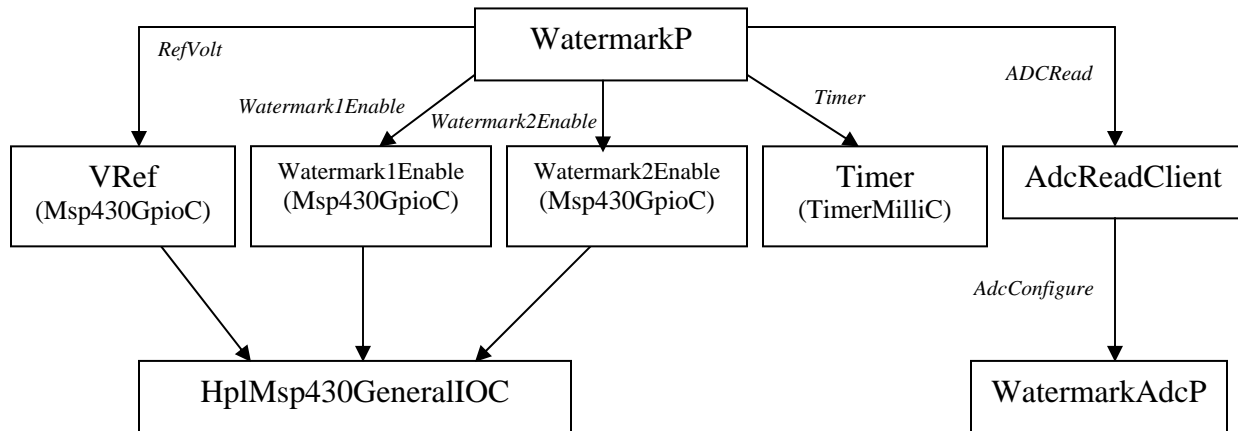
```
const msp430adc12_channel_config_t ech2oAdc1 = {  
    INPUT_CHANNEL_A6, // ADC 6, P6.6  
    REFERENCE_VREFplus_AVss,  
    REFWOLT_LEVEL_2_5,  
    SHT_SOURCE_ACLK,  
    SHT_CLOCK_DIV_1,  
    SAMPLE_HOLD_4_CYCLES,  
    SAMPCON_SOURCE_SMCLK,  
    SAMPCON_CLOCK_DIV_1  
};
```

```
const msp430adc12_channel_config_t ech2oAdc2 = {  
    INPUT_CHANNEL_A7, // ADC 7, P6.7  
    REFERENCE_VREFplus_AVss,  
    REFWOLT_LEVEL_2_5,  
    SHT_SOURCE_ACLK,  
    SHT_CLOCK_DIV_1,  
    SAMPLE_HOLD_4_CYCLES,  
    SAMPCON_SOURCE_SMCLK,  
    SAMPCON_CLOCK_DIV_1  
};
```

### 5.1.1.4.2 Irrrometer Watermark

The driver for the Watermark sensor is very similar to the one for the Ech2o Sensor.

#### Wirings



*VRef* is a MSP430 input/output pin for the reference voltage.

*Watermark<1/2>Enable* is a MSP430 input/output pin that is SET (Vcc) when sensor is read, CLR otherwise

*AdcReadClient* deals with the reading and ADC conversion from V\_ECHO2 pin (see figure XXX). This pin corresponds to the sensor data voltage.

*WatermarkAdcP* defines the configuration of the ADC port, given the header file *Watermark.h*

#### Application Flow

When sensing is required, *WatermarkP* sets the *Watermark<1/2>Enable* as an output pin and sets it to Vcc. This voltage is applied at the sensor during 30ms, after which the voltage on pin V\_WATERM<1/2> is read.

Here are the pin wirings from *WatermarkC*:

```
Watermark1Enable -> HplMsp430GeneralIOc.Port52;  
Watermark2Enable -> HplMsp430GeneralIOc.Port53;  
VRef -> HplMsp430GeneralIOc.Port12;
```

Here is the ADC configuration of *Watermark.h*, with the corresponding channel.

```
const msp430adc12_channel_config_t watermarkAdc1 = {
    INPUT_CHANNEL_A4, // ADC 4, P6.4
    REFERENCE_VREFplus_AVss,
    REFVOLT_LEVEL_2_5,
    SHT_SOURCE_ACLK,
    SHT_CLOCK_DIV_1,
    SAMPLE_HOLD_4_CYCLES,
    SAMPCON_SOURCE_SMCLK,
    SAMPCON_CLOCK_DIV_1
};

const msp430adc12_channel_config_t watermarkAdc2 = {
    INPUT_CHANNEL_A5, // ADC 5, P6.5
    REFERENCE_VREFplus_AVss,
    REFVOLT_LEVEL_2_5,
    SHT_SOURCE_ACLK,
    SHT_CLOCK_DIV_1,
    SAMPLE_HOLD_4_CYCLES,
    SAMPCON_SOURCE_SMCLK,
    SAMPCON_CLOCK_DIV_1
};
```

## 5.1.2 Basestation

The base station is a simple message forwarding application.

Messages received from the radio interface are put added to a circular FIFO message queue and forwarded to the UART (serial) port. All types of AM groups are accepted and forwarded.

Messages from the UART port are considered as commands. Byte 0 is the command type and byte 1 is the command argument. Then they are injected into the network.

Also, due to different AM Types of packets of the same type (e.g. sensor data messages may have AM type 0x80 and 0x81). So what I do is to put both to a single value: 80 becomes 81 and 81 stays 81. I always take the upper value.

### Code details

*Headers:*

```
#include "circqueue.h"
#include "messages.h"
#include "constants.h"
```

*Interfaces:*

```
module BaseStationP {
    uses {
        ...
        interface Processing;
        interface BeaconCommand;
        interface DataInjection;
        interface NetProg;
    }
}
```

*Command Processing:*

When an messages arrives from the UART port, I parse it as a command.

```
event message_t *UartReceive.receive[am_id_t id](message_t *msg,
                                                void *payload,
                                                uint8_t len) {
    ...
    processUARTPacket(ret, FALSE);
}
```

Here I take byte 0 as the command type and byte 1 as the argument, and inject it into the network with BeaconCommand.setCommand.

```
void processUARTPacket(message_t* Msg, bool wantsAck) {
    uint16_t cmd = (uint16_t)Msg->data[0] | (uint16_t)Msg->data[1] << 8;
}
```

```
    call BeaconCommand.setCommand(((xe1205_header_t*)(Msg->data
- sizeof(xe1205_header_t)))->dest,cmd);
}
```

### *Message AM Type parsing:*

As explained above I put messages of the same type to the same AM type by simple incrementation. Note that the types are defined in Dozer (not accessible for us):

```
id = call RadioAMPacket.type(msg);
if (id == AM_BEACON_MSG || id == AM_DATA_MSG || id ==
AM_GET_APP_PARAM_MSG) {
    id++;
}
```

## 5.1.3 Messages

### 5.1.3.1 AM types

According to the Shockfish MAC, all the types must be even numbers. This is due to the fact that if a node has no additional message to send it will send the last message with AM type + 1. That way the receiver knows he can turn off the radio.

The following AM types apply for our messages:

```
enum {
    AM_BEACON_MSG = 0x50,
    AM_DATA_MSG = 0x80,
    AM_GET_APP_PARAM_MSG = 0x84,
};
```

So, beacon messages will have AM type = 0x50 and 0x51, sensor messages AM type = 0x80 and 0x81, command replies AM type = 0x84 and 0x85.

If you want to add a new message type, it must have a EVEN AM type (starting at 0x88).

### 5.1.3.2 Shockfish Header

In every message sent a header is added by the Shockfish MAC. It consists of the following structure:

```
typedef nx_struct shockfish_header {
    nx_uint16_t originatorID;
    nx_uint16_t parentID;
    nx_uint16_t aTime;
    nx_uint24_t tStamp;
    nx_uint8_t padding;
} shockfish_header_t;
```

OriginatorID	is the ID of the node that took the sensor measurement.
ParentID	is the parent of that node in the spanning tree
aTime	is the travel time of this message in the network
tStamp	is the sending time stamp
padding	is one byte added to tStamp to respect the 16 bits structure

### 5.1.3.3 Commands

For the commands to work with NetProg I had to follow the following rules:

#### **Get commands:**

There is a maximum of 16 different GET commands.

The types at the sending side (server) must have the structure 0xA<1-F>. This enables the Shockfish MAC to differentiate them from SET commands. However the uint8\_t\_cmd parameter returned by "getParam" is a truncated version of the type: only <1-F> is returned.

The GET command has no argument. However the sent message will anyways contain a “wasted” parameter byte. In other words you may put a parameter but it will never be used/read.

We use the following types, present defined in CommonSense.h:

```
COMMAND_REPLY_TYPE_GET_RFPOWER = 0xA1,
COMMAND_REPLY_TYPE_GET_SEND_RATE = 0xA2,
COMMAND_REPLY_TYPE_GET_SENSORFLAGS = 0xA3,
COMMAND_REPLY_TYPE_GET_SENSING = 0xA4,
COMMAND_REPLY_TYPE_GET_NUM_READINGS_SM1 = 0xA5,
COMMAND_REPLY_TYPE_GET_NUM_READINGS_SM2 = 0xA6,
COMMAND_REPLY_TYPE_GET_PERIOD_SM1 = 0xA7,
COMMAND_REPLY_TYPE_GET_PERIOD_SM2 = 0xA8,
```

### **Set commands:**

There is a maximum of 16 different SET commands.

The types at the sending side (server) must have the structure 0x9<1-F>. However the uint8\_t appCmd parameter returned by “setParam” is a truncated version of the type: only <1-F> is returned.

The SET command has an argument of 1 byte.

We use the following types, present defined in CommonSense.h:

```
COMMAND_REPLY_TYPE_SET_RFPOWER = 0x91,
COMMAND_REPLY_TYPE_SET_SEND_RATE = 0x92,
COMMAND_REPLY_TYPE_SET_SENSORFLAGS = 0x93,
COMMAND_REPLY_TYPE_SET_SENSING = 0x94,
COMMAND_REPLY_TYPE_SET_NUM_READINGS_SM1 = 0x95,
COMMAND_REPLY_TYPE_SET_NUM_READINGS_SM2 = 0x96,
COMMAND_REPLY_TYPE_SET_RATE_SM1 = 0x97,
COMMAND_REPLY_TYPE_SET_RATE_SM2 = 0x98
```

### **On the network:**

Commands are not independent messages, but they are sent within beacon messages. Note that the commands have been implemented with Little Endian system, meaning that if you have a SET command 0x91 with argument 0x02, it will be sent in the network as 0x0291.

### **Message structure**

```
typedef nx_struct command_message {
    nx_cmd_type_t type;
    nx_cmd_value_t newvalue;      // argument of the command
} command_message_t;

typedef nx_struct command_reply {
    nx_cmd_type_t type;
    nx_cmd_reply_t data;
```

```
    } command_reply_t;
```

### 5.1.3.4 Sensor Data Message

This is the useful payload containing sensor data and sensorflags. I can be extended up to 17 bytes with other sensors.

```
typedef nx_struct sensor_message {
    nx_uint16_t sensorflags;
    nx_uint16_t watermark1;
    nx_uint16_t watermark2;
    nx_uint16_t moist1;
    nx_uint16_t moist2;
} sensor_message_t;
```

### 5.1.3.5 Channel sniffing

If you sniff the channel (e.g., with a basestation) you may see a byte stream similar to this one:

```
00 FF FF 00 0A 06 7D 50 00 A1 6A C1 00 00
00 FF FF 04 24 06 7D 50 00 A1 67 5F 01 00
00 FF FF 04 24 06 7D 50 00 A1 CF BE 01 00
00 00 0A 04 24 1A 7D 85 04 24 00 0A 00 00 00 2D C9 00 A1 00 03
00 FF FF 04 24 06 7D 50 00 A2 CF BE 01 00
00 00 0A 04 24 1A 7D 81 04 24 00 0A 00 00 00 2D C9 00 01 E0 04 15 02 55 03 13 04 05
```

We have 4 beacon messages. Every 4 beacons a sensor data message is sent. We also see a command reply (the 4<sup>th</sup> message)

#### Example of the sensor message:

```
00 00 0A 04 24 1A 7D 81 04 24 00 0A 00 00 00 2D C9 00 01 E0 04 15 02 55 03 13 04 05
```

Synchronization preamble	00	}	From Serial packet
Destination	00 0A		
Source	04 24	}	XE1205 Radio header
Payload Size	1A		
AM Group	7D		
AM Type	81		
OriginatorID	04 24		
parented	00 0A	}	Shockfish header
aTime	00 00		
tStamp	00 2D C9		
Padding	00	}	Sensor payload
Sensorflags	01 E0		
Watermark1	FF FF		
Watermark2	FF FF		
Moist1	03 13		
Moist2	04 05		

**Example of beacon message:**

00 FF FF 04 24 06 7D 50 00 A2 CF BE 01 00

Synchronization preamble	00	}	From Serial packet
Destination	FF FF		
Source	04 24	}	XE1205 Radio header
Payload Size	06		
AM Group	7D		
AM Type	50		
Command argument	00		
Command type	A2	}	Beacon payload
Pseudo-random seed	CF BE		
Hopcount towards sink	01		
Number of children	00		

**Example of command reply**

00 00 0A 04 24 0D 7D 85 04 24 00 0A 00 00 00 2D C9 00 A1 00 03

Synchronization preamble	00	}	From Serial packet
Destination	00 0A		
Source	04 24	}	XE1205 Radio header
Payload Size	0D		
AM Group	7D		
AM Type	85		
OriginatorID	04 24		
Parented	00 0A	}	Shockfish header
aTime	00 00		
tStamp	00 2D C9		
Padding	00		
Command type	A1		
Command argument	00 03	}	Command reply payload

## 5.1.4 Deluge

### What is Deluge?

Deluge enables network programming. It enables to change the software on motes without requiring any physical human intervention. This can be very useful when your motes are spread within a wide area, or installed under the ground etc...

Deluge divides your external flash into partitions (called images) where you can store complete TinyOS applications. You can, over the air, program a new application into one of the partitions. Then you can reboot the mote with one of these images and have the new application running. Note that the complete network will respond to each command and so you will keep a global homogenous state.

### Installation

To install Deluge (under TinyOS<sub>1</sub>), please have a look at the corresponding appendix.

### Problem with TinyOS 2

The problem with Deluge is that it is not yet implemented for TinyOS<sub>2</sub>. So, we have to install it under TinyOS<sub>1</sub>. Moreover, injecting new images (TinyOS 1 or 2 applications) into partitions must be done under TinyOS<sub>1</sub>.

So where is the problem? The problem arises when you are currently running a TinyOS 2 application from an image. If at the server side you try to reboot with another image or want to inject another application, it will not work, because Deluge is not active on the node.

### Our Solution

The solution is to reboot the node with a TinyOS 1 application first, then the usual Deluge commands will work again. To do that, the final solution has been to integrate the command within the Shockfish MAC that will take care of rebooting the node with a given image.

### Code Details

First we need to save current TinyOS configuration data in the internal Flash. If we do not do so, this information will be lost when we reboot. By configuration data, we mean TOS\_NODE\_ID (the current node address) and the AM Group ID. Moreover a CRC applied to NetProg\_TOSInfo data is included:

```
typedef struct NetProg_TOSInfo {
    uint16_t addr;
    uint8_t  groupId;
    uint16_t crc;
} NetProg_TOSInfo;

void writeTOSInfo() {
    NetProg_TOSInfo tosInfo;
    uint16_t crc;
    call IFlash.read((uint8_t*)IFLASH_TOS_INFO_ADDR, &tosInfo,
sizeof(tosInfo));
    tosInfo.addr = TOS_NODE_ID;
```

```

        tosInfo.groupId = 0x7d; // default group in tiny1.x
        crc = computeTosInfoCrc(&tosInfo);
        if (tosInfo.crc == crc)
            return;
        tosInfo.crc = crc;
        call IFlash.write((uint8_t*)IFLASH_TOS_INFO_ADDR, &tosInfo,
sizeof(tosInfo));
    }

```

A second step is to get the address of the image we want to reboot from (e.g., the golden image TOSBOOT\_GOLDEN\_IMG\_ADDR) and write in into the internal Flash at position TOSBOOT\_ARGS\_ADDR.

```

tosboot_args_t args;
args.imageAddr = TOSBOOT_GOLDEN_IMG_ADDR;
args.gestureCount = 0xff;
args.noReprogram = FALSE;
call IFlash.write((uint8_t*)TOSBOOT_ARGS_ADDR, &args, sizeof(args));

```

Finally we reboot:

```

WDTCTL = 0;

```

## How to use

From a user perspective now, we can use this dedicated reboot command in the following way:

It is a simple command like any other, just respecting the `0x1<imagenumber>` to reboot on 12bits> structure (so “i” instead of “A” or “g”).

For example, if you want to reboot with the GOLDEN IMAGE (image 0), you use `0x1000` (in Little Endian: `0x0010`)

Example with the C SDK:

(Note that you have first to define the delay after which the reboot will take place)

```

Delay of 0:
./sfsend localhost 9002 0x00 0xFF 0xFF 0x00 0x0A 0x50 0x7d 0x00 0x00
0x20

```

```

Reboot on Golden Image:
./sfsend localhost 9002 0x00 0xFF 0xFF 0x00 0x0A 0x50 0x7d 0x00 0x00
0x10

```

## 5.2 Server Side Application

On the server side, we have the following applications running:

- General Data Logger (Java application)
- Database for sensor data storage (PostgreSQL)
- Web user interface for applying commands and showing result charts (Python, PHP, HTML)

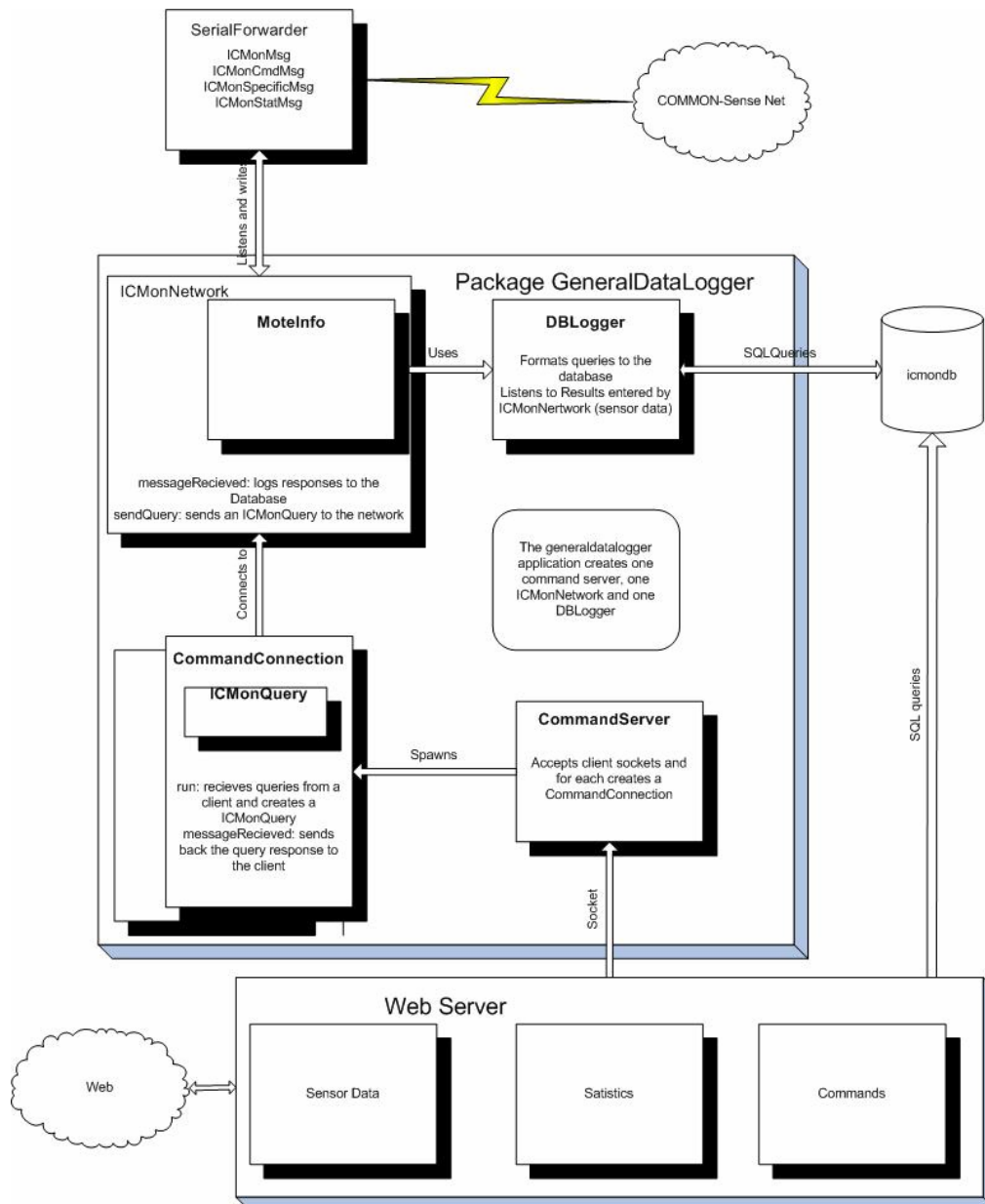


Figure 15: Server Side Applications

## 5.2.1 General Data Logger

The General data logger is a java application whose roles are the following:

### Sensor Messages

- reception of sensor data messages from the UART port
- parsing of sensor messages

### Database

- connecting to a PostgreSQL database
- storing sensor data and mote information into the database

### Commands

- reception of commands through a TCP socket
- parsing of commands
- sending of commands to the serial port

### My Contribution

A long part of the project has been spent in improving the General Data Logger application and adapting it to TinyOS2 and Shockfish MAC. My tasks:

- understanding how General Data Logger works
- installing it
- correct some bugs
- adding new commands
- improving the code structure

### Application Flow:

Here I explain in simplified manner which classes come into play in two scenarios: sending of a command and reception of sensor data.

When a command arrives from the Python layer on the TCP socket created by CommandServer, it is analysed in CommandConnection. Here the corresponding ICMonQuery is created and sent to ICMonNetworkSH. In ICMonNetworkSH a message is displayed on the screen and optionally a log entry in the database is stored. The query is then forwarded to MotelF. From MotelF, it is the standard TinyOS Java SDK stack that will convert the Java packet into a TinyOS readable packet and send it to the COM port.

When a message arrives from the serial port, ICMonNetworkSH and CommandConnection message listeners are signaled. ICMonNetworkSH deals with sensor data messages, verifies the content and signals DBLogger. DBLogger will store the sensor data into the database. CommandConnection deals with incoming answers from commands. It displays a message on the screen and sends an answer to a TCP client (Python layer).

## 5.2.2 PostgreSQL Database

The database stores information about the sensor data and also information the tinynodes related to the network topology.

My contribution here was to

- install PostgreSQL server
- make it work with GeneralDataLogger
- create a new schema and database for the new application

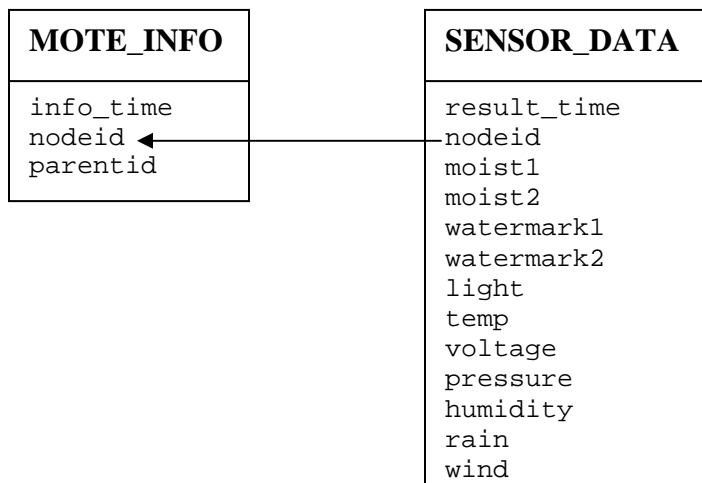
The main database is “icmondb\_rev2”. It contains two tables: `mote_info` and `sensor_data`.

`Mote_info` contains information about the node and the network topology.

`Sensor_data` contains the sensor data received by the nodes.

We decided to store the ADC output voltages. Conversion formulas for soil moisture information will be used on top of the database. This enables more flexibility, as at a present time, the formulas are in calibration phase by agronomists at UAS in Bangalore. However we may change this later. We added already all the sensors we will ultimately use. However, at present time, only watermark and moisture sensor data are stored.

Here is the main schema:



The SQL code to generate the data base:

```
DROP TABLE mote_info;
DROP TABLE sensor_data;

CREATE TABLE mote_info (
    info_time TIMESTAMP,
    nodeid INT PRIMARY KEY,
    parentid INT
)
```

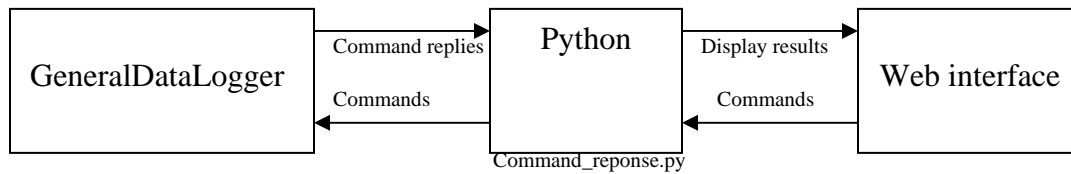
```
CREATE TABLE sensor_data (  
  result_time TIMESTAMP,  
  nodeid INT REFERENCES mote_info(nodeid),  
  moist1 INT,  
  moist2 INT,  
  watermark1 INT,  
  watermark2 INT,  
  light INT,  
  temp INT,  
  voltage INT,  
  pressure INT,  
  humidity INT,  
  rain INT,  
  wind INT  
)
```

### 5.2.3 Python layer

The Python application is placed between the GeneralDataLogger and the web interface. It connects to the TCP socket opened by the Generaldatalogger and forwards the command requests from the web interface. As well it receives command replies from the Generaldatalogger and displays the results on the web interface as a pop-up window. It also enables to generate graphs.

My tasks here were to:

- install Python and make it work with Apache
- learn Python basics
- implement the new commands
- improve the code



The Python file used to send commands via Web interface is `command_response.py`. It is called for example in the following way within a web site to display the corresponding pop-up window:

```
<script language="javascript">

w = window.open("secure/command_response.py?
command=<?php echo $querytype; ?>
&value=<?php echo $value; ?>
&addr=<?php echo $destination; ?>
&hostname=<?php echo $commandhostname ?>",
"Response Window",
config="height=500, width=400, toolbar=no, menubar=no, scrollbars=yes, resizable=yes,
location=no, directories=no, status=no")

</script>
```

The Python layer can also be used without the web interface. One uses the `command_response_shell.py` instead of `command_response.py` in that case. For example to call the command `getRFPower` for node 1025 at the shell:

```
C:\..\www\secure>Command_response_shell.py get_rfpower o 1025 localhost
```

The first argument being the commande name, the second one the corresponding argument (for this command the argument plays no role), the third one is the destination node ID and the last one is the host name where the TCP socket is open on port 9876.

## 5.2.4 Web Interface

The web interface enables to send commands to the nodes.  
A pop-up window appears with the replies.

My contribution here was to create a new interface for commands, which is more user friendly than the original one. I also had to install Apache server with PHP and make it work with Python.

The user first chooses the type of command and the corresponding parameter (where applicable). Then he chooses the destination node ID. Finally he presses the Submit Query button. A pop-up window appears and a Python application is called within that window. After a few seconds replies from all the nodes (where applicable) are shown in the pop-up window.

The screenshot shows the 'COMMON-Sense Net Homepage' with a navigation menu on the left and a main content area. The main content area is titled 'Send a Command to the Motes' and contains two sections: '1. Choose the query' and '2. Choose the destination node ID'. The '1. Choose the query' section has several radio button options: 'Get RF power level', 'Set RF power level' (with a text input field and 'dBm (0, 5, 10 or 15dBm)'), 'Get send interval', 'Set send interval' (with a text input field and 'ms'), 'Get sensing', 'Set sensing' (with 'activate' and 'deactivate' radio buttons), 'Get sensorflags', and 'Set sensorflags' (with checkboxes for Voltage, Humidity, Temperature, Light, Pressure, Soilmoisture 1, Soilmoisture 2, Watermark 1, and Watermark 2). The '2. Choose the destination node ID' section has a text input field and the instruction '(put 65535 to broadcast the query)'. A 'Submit Query' button is located below the input field. The footer contains the W3C XHTML 1.0 logo, the text 'Last modified 19.02.2007, 05:43', and the email 'gansorscope@icavsun1.epfl.ch'.

Figure 16: Command Web Interface

Here is the pop-up window with replies from nodes.

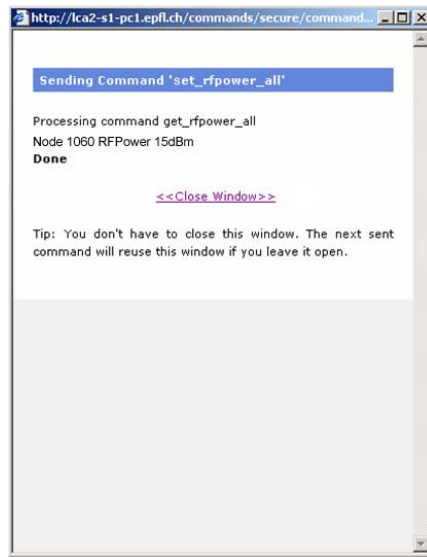


Figure 17: Pop-up window with command replies

## 6 Deployment

The deployment was planned last weekend in the village of Chennakeshavapura, a village in the state of Karnataka. However, due to resistance of farmers and problems of stealth, we could not deploy it there yet. We plan a similar deployment in terms of number of nodes within the IISc campus early next week.

For the time being a small deployment with 5 nodes in the lab showed positive results with more than 98% of packet arrivals. Commands are processed properly and replies reach the web interface with success. I am looking forward to see if the network will scale for the larger deployment.

## 7 Main problems

At every step of this project I had challenging problems to solve. Challenging in the sense that often I had no clue where the problems come from, how I could solve it and where I could find an answer. I often spend hours, days or even months solving some issues. This is of course not an exhaustive list of the problems, but they give a good insight into the type of problems encountered.

### 7.1 Radio

The largest problem we faced was related to the Radio driver. We had inconsistent sending, respectively reception of simple messages between a single node and a basestation. Tests overnight showed that in the best case we had about 90% reception probability, in the worst case about 10%.

We used the standard drivers provided in the TinyOS 2 tree. There were two possible modules enabling radio communications: ActiveMessageC and AMSenderC/AMReceiverC. The latter seemed to have slightly better performance, but when analyzing the underlying layers we discovered that both shared the same drivers. So there was no reason to have real differences. We also changed the packet size, the radio channel, the nodes, the boards, the antennas, the modules and the application. We posted messages on the TinyOS mailinglist without getting replies. It seemed that we were the only ones facing this kind of problem.

We had to choose what to do: either we reprogram the whole MAC layer or we wait for Shockfish to release their MAC/Routing package. We chose the second option after discussions with Shockfish which were very encouraging, as they told us the package would be ready for January. Finally we received a first version mid-february, which delayed our schedule a lot. The results in term of packet drops were excellent though, so that our choice showed to be the right one in the end.

### 7.2 Integration with Shockfish MAC

A few weeks have been spent with Shockfish to understand, implement and improve the behavior of the Shockfish MAC/Routing package. Slowly but surely we managed with Maxime to obtain a working platform. The interaction with Shockfish was very intense and very positive, so that each problem was solved within a few days at most. The interaction and contributions took place in both ways. However the work related to this integration was much more time consuming than expected. The whole software had to be readapted.

### 7.3 Installation of TinyOS 2 and server applications

A few days have been spent to make TinyOS 2 platform working on Linux and Windows. When installation problems arise, you spend your days googling after error messages. Then you find that a single variable in a given line from a given file, hidden in a given directory can solve all your

problems. It is a frustrating job, I must admit, as you do not really learn much and you spend a huge time on that instead of value-adding aspects of the project like implementation or system design. The same challenges arose when trying to install Apache, Python, PostgreSQL and make them all work in harmony together.

## 7.4 Debugging

Debugging TinyOS applications without debugger was a hard task. My tools were only 3 blinking Leds. I believe that a lot of time could have been saved if TOSSIM would have been implemented for the tinynode.

## 8 Conclusions

I implemented an embedded application for TinyOS 2 on the Tinynode platform. A MAC/ Routing layer from Shockfish has been integrated. Server applications have been re-implemented partially to comply with the new software. Results on a small scale network are excellent with more than 98% success rate. A large-scale network will be deployed next week to prove the scalability and the stability of the network in a more hostile environment.

On a more personal point of view, I want to say that the project was really challenging for me. I learned that perseverance is a key quality you need for research/engineering projects. Perseverance even if you have no clue whether you are going into the right direction or not. Often you go home at night with high frustration because you did not advance at all during the day. The next day though, by chance often, you solve the problems. Solving problems by chance is not really the most satisfactory reward you can obtain, but with TinyOS, it often happened that way.

During the project, I learned a lot of new technologies like TinyOS, nesC embedded systems and Python. I also used technologies I already knew like Java, PHP, HTML, database management and Linux. I also got an insight into how to use technology for nature-oriented application. However, the main positive aspect here is that I acquired the knowledge of interfacing all these technologies to create a full working system. This represents, I believe, a key skill for my future engineering career.

During the project I had to interact with people from many different horizons like researchers from IISc (Prabhakar, Sujay...), people from Shockfish (Maxime, Roger), hydrologists from Hydram, TinyOS people at Berkeley, farmers... Working with Jacques was a very positive experience. Jacques knows to motivate you when you feel down and "in despair". He also lets you take initiative, trusts you and does not impose his point of view. I was definitely not an "easy" student, but I tried my best to make his project a success.

What I also really liked in the project was the fact that we apply technology for developing countries. It associates what I learned throughout my studies and a much more down-to-earth aspect: helping people that have real needs for their survival. It was also my first work experience in developing countries, and I believe it made me grow a lot in my vision of life in general. A big "thank you" for that!

## 9 References

- [1] TinyOS website <http://www.tinyos.net>
- [2] TinyOS Installation <http://www.tinyos.net/tinyos-2.x/doc/html/install-tinyos.html>
- [3] The Wasal blog <http://blogs.epfl.ch/wasal>
- [4] Deluge 2.0 : TinyOS Network Programming  
[www.cs.berkeley.edu/~jwhui/research/deluge/deluge-manual.pdf](http://www.cs.berkeley.edu/~jwhui/research/deluge/deluge-manual.pdf)
- [5] Philipp Levis *TinyOS programming*, October 27<sup>th</sup>, 2006
- [6] Prabhakar T V , N V Chalapathi Rao, Sujay M S, Jacques Panchard, H S Jamadagni, Andre Pittet *Sensor Network Deployment For Agronomical Data Gathering in Semi-Arid Regions* in Wisard 2007 workshop at Comsware 2007, 07-08 Jan 2007, Bangalore, India.
- [7] J.-D. Decotignie *REAL-TIME SYSTEMS II, Real-Time Networking: Wireless sensor networks*
- [8] Nicolas Burri, Pascal von Rickenbach, Roger Wattenhofer *Dozer: Ultra-Low Power Data Gathering in Sensor Networks*
- [9] F. Depienne, G. Noubir, Y. Wang. A Platform for Heterogeneous Vehicular Communications and Applications. In *Proceedings of the 1st IEEE Workshop on Automotive Networking and Applications (AutoNet 2006)*, December 1st, 2006, San Francisco, CA, USA
- [10] Hachette, Le guide du routard, Inde du Sud, 2007

## 10 Appendix

### 10.1 TinyOS 1 Installation

You can directly install it from the Shockfish website for Windows only  
<http://www.tinyos.com/index.php?id=100>

Then in the `/etc/profile.d/` directory create a file `tinyos.sh` with the following content:

```
# Java

export JDKROOT=/cygdrive/c/PROGRA~1/Java/jdk1.5.0_02
export PATH="$JDKROOT/bin:$PATH"

# MSPGCC

export MSPGCCROOT=/opt/msp430/bin
export PATH="$MSPGCCROOT/bin:$PATH"

# TINYOS

export TOSROOT=/opt/tinyos-1.x
export TOSDIR=$TOSROOT/tos
CLASSPATH="$TOSROOT/tools/java/javapath`"
CLASSPATH="$`cygpath -w /opt/tinyos-
1.x/contrib/shockfish/tools/java/`; $CLASSPATH"
export CLASSPATH="$`cygpath -w
/projects/ICMon/tools/java/`; $CLASSPATH"

# building TinyNode

export TOSMAKE_PATH="$TOSDIR/../../contrib/shockfish/tools/make"

# MAKERULES

export MAKERULES=$TOSROOT/tools/make/Makerules
```

The installation should work properly.

## 10.2 TinyOS 2 Installation

TinyOS2 can be installed both on Windows (Cygwin) and Linux.

You should install it from the official TinyOS website for Linux or Windows

<http://www.tinyos.net/tinyos-2.x/doc/html/install-tinyos.html>

In general you can follow the installation description from the website, however I encountered several problems, so I give some more details:

### Java

You have to update the environment variables after installing the Java SDK.

Therefore add the directory "JAVAINSTALLPATH"/jdk"VERSION"/jre/bin and "JAVAINSTALLPATH"/jdk"VERSION"/bin to your PATH variable. To do that you can add the following line at the end of the file /home/"USER"/.bashrc :

```
export
PATH=$PATH:"JAVAINSTALLPATH"/jdk"VERSION"/jre/bin:INSTALLPATH"/jdk
"VERSION"/bin
```

### RPMs

You should download RPMs for the TI MSP430 (not Atmel AVR).

You might have to add --ignoreos and --force attributes for the RPM command if an error message appears, e.g.

```
Rpm -ivh --ignoreos --force msp430tools-base-0.1-
20050607.cygwin.i386.rpm
```

When you install tinyos-tools-1.2.3-1.cygwin.i386.rpm, verify that you get a message saying that JNI has been installed. If not, reinstall Java JDK and restart the TinyOS installation from scratch.

If you use the *Debian/Ubuntu* distribution of Linux, you have to convert the RPMs into DEB files or directly convert and install the files. You need the *alien* application for that purpose:

```
Apt-get install alien
```

Then apply the following line to each RPM file to directly install them

```
Alien -i MYFILE.rpm
```

### PATHS

At the end of the installation you need to define Environment Variables. You should add the lines as explained at the end of the file /home/"USER"/.bashrc file. There is no issue if you work under Linux.

However if you work under Cygwin, read the following:

- You have to save the .bashrc file under Unix file format with ANSI encoding. To do this, forget about Notepad. I recommend using Textpad ([www.textpad.com/](http://www.textpad.com/)).

- For the classpath, you will need to use a special command and add ALL the classpaths in one line separated by ":" as shown below:

```
export TOSROOT=/opt/tinyos-2.x
```

```
export TOSDIR=$TOSROOT/tos
export CLASSPATH=`cygpath -wp
$TOSROOT/support/sdk/java/tinynode.jar:/projects/CommonSense/java
:/projects/CommonSense/java/net/tinyos/icmon/generalatalogger/po
stgresql-8.1-405.jdbc3.jar:.`
export PATH=/opt/msp430/bin:$PATH
export MAKERULES=$TOSROOT/support/make/Makefiles
```

For the rest you can follow the guidelines of the installation website.

## Compilation Problem

After installation if you encounter a problem compiling basic applications like BLINK, it might be that the nesC version is not supported by your version of Cygwin. What you can do is install an older version of nesC compiler from Sourceforge ([http://sourceforge.net/project/showfiles.php?group\\_id=56288](http://sourceforge.net/project/showfiles.php?group_id=56288))

## Baud rate Problem

Another problem with the tinynode platform might be the BaudRate of the serial connection. You have to replace the default 115200 by 57600bps. To configure this properly you have to edit two files:

In \$TOSROOT\tos\platforms\tinynode\TinyNodeSerialP.nc, replace:

```
msp430_uart_config_t msp430_uart_tinynode_config = {ubr:
UBR_1MHZ_115200, umctl: UMCTL_1MHZ_115200, ssel: 0x02, pena: 0,
pev: 0, spb: 0, clen: 1, listen: 0, mm: 0, ckpl: 0, urxse: 0,
urxeie: 1, urxwie: 0};
```

By:

```
msp430_uart_config_t msp430_uart_tinynode_config = {ubr:
UBR_1MHZ_57600, umctl: UMCTL_1MHZ_57600, ssel: 0x02, pena: 0, pev:
0, spb: 0, clen: 1, listen: 0, mm: 0, ckpl: 0, urxse: 0, urxeie:
1, urxwie: 0};
```

In \$TOSROOT\support\sdk\java\net\tinyos\packet, replace:

```
Platform.add(Platform.x, "tinynode", 115200);
```

By:

```
Platform.add(Platform.x, "tinynode", 57600);
```

## 10.3 Deluge

Deluge will be installed under TinyOS 1 as there is no version for TinyOS 2 available yet. For a more detailed installation guide, please refer to reference [4].

### Formatting the Flash

The first step is to format the Flash for Deluge:

On directory `tinyos-1.x/apps/TestDeluge/FormatFlash` do the following

```
Make TINYOS_NP=BNP tinynode install bs1,COMX
```

with `X` your com port number

### Installing DelugeBasic

Then we can install DelugeBasic. On directory `tinyos-1.x/apps/TestDeluge/DelugeBasic` do the following:

```
Make TINYOS_NP=BNP tinynode install.Y bs1,COMX
```

with `Y > MAX_SINK_ID` (from `app_cst.h`)  
`X` your com port number

Then you can test whether the basic installation worked properly by:

```
Java net.tinyos.tools.Deluge -ping
```

You should see that images are not yet installed.

### Install a golden Image

Now you need to install a GOLDEN IMAGE (image o), which is a reliable image you want to use when rebooting. We will use DelugeBasic. However this time you have to use a special deluge command to inject the image. First you have to compile DelugeBasic from directory `tinyos-1.x/apps/TestDeluge/DelugeBasic`

```
make TINYOS_NP=BNP tinynode
```

Then enter subdirectory `build/tinynode/` where the file `tos_image.xml` is situated.

```
Java net.tinyos.tools.Deluge --inject -tosimage=tos_image.xml --  
imgnum=0
```

## Install a TinyOS 2 application

Before being able to install our TinyOS2 application, we have to correct some bugs:

In file `$TOS2ROOT/support/make/Makedefaults`, replace:

```
GOALS += tos-ident-flags tos_image
By
GOALS += ident_flags tos_image
```

Next you have to modify file

```
$TOS1ROOT/tools/java/net/tinyos/deluge/TOSBOOTImage.java
```

At line 102 comment out the following lines

```
//nlist=doc.getElementsByTagName("platform");
//tmp = nlist.item(0).getFirstChild().getNodeValue();
//platform = nlist.item(0).getFirstChild().getNodeValue();
```

And add the line instead

```
platform = "";
```

Moreover you have to modify the Makefile of the application:

Add the following line:

```
# Pseudo-Deluge support
PFLAGS += -Wl,--section-start=.text=0x4800,defsym=_reset_vector__=0x4000
```

Also convert any CFLAGS into PFLAGS

e.g.

```
CFLAGS += -I ./Echo2
CFLAGS += -I ./Watermark
```

Become

```
PFLAGS += -I ./Echo2
PFLAGS += -I ./Watermark
```

Now you can compile your Tinyos2 application entering `./make_bin`.

After compiling you still need to modify `tos_image.xml` in the `build/tinynode` subdirectoy. Replace

```
<deluge_support>no</deluge_support>
by
<deluge_support>yes</deluge_support>
```

Your `tos_image.xml` file should look similar to this one:

```
<tos_image>
```

```
<ident>
  <user_hash>39A5F7C9L</user_hash>
  <unix_time>45F25D85L</unix_time>
  <uid_hash>35631EC0L</uid_hash>
  <deluge_support>yes</deluge_support>
  <hostname>"braincell"</hostname>
  <user_id>"root"</user_id>
  <program_name>"CommonSenseAppC"</program_name>
  <platform>tinynode</platform>
</ident>
  <image format="ihex">C4F0D8FCE4A00001E539C
:10CAE0003C53FB230C4D12C30C108E4B00002E53F5
:10CAF0003C53FB230C4D1CF30524CE4A00001E536F
:.....
  </image>
</tos_image>
```

Enter the directory where your tos\_image.xml file. To install it into image 1, enter:

```
Java net.tinyos.tools.Deluge --inject --tosimage=tos_image.xml --imgnum=1
```

To reboot with this newly installed image:

```
Java net.tinyos.tools.Deluge --reboot --imgnum=1
```

## 10.4 CVS

CVS allows you to make changes to source files of software and make it available to all the other programmers on the same project. Each file will have a version.

Groups of files may get a tag (independently of versions of individual files). This represents a “full software version”, in which the compatibility of the files has been verified (or should have).

### Repository

First you have to define the repository, which is the place (usually a distant server) where the files will be available. There are a few ways to do that:

- set the environment variable `$CVSROOT`
- `cvs -d`
- Root file in subdirectory `CVS`

Repository Format:

```
[ :method: ] [ [user] [ :password ] @ ] hostname [ : [port] ] /path/to/repository
```

Example:

```
:ext:username@icsill-cvs.epfl.ch:/cvs
```

:ext: is the method (external), you can have :server:, :pserver:, :local: among others

### CVS directories

Each directory after downloading the files has its own CVS directory (locally) in which you find 3 files:

- Entries: gives the files of that directory, the version and the date where they were uploaded
- Repository: the directory (from the root) corresponding to this directory
- Root: the root :pserver:anonymous@tinyos.cvs.sourceforge.net:/cvsroot/tinyos

### Create a CVS module on the server

A *module* means a complete set of files, like a full application. Place yourself locally in the directory just under the module and enter:

```
cvs import $directory $vendorId $revision
```

### Add a file to the CVS server

Make sure you added the directory first

```
cvs add file  
cvs commit
```

### Copy files from the server to your local directory (checkout)

```
Cvs co module_name
```

### Update your local files from files on the server

```
cvs update -d
```

Overwriting also the files you have modified locally:

```
cvsv update -d -C
```

Take away the tags:

```
cvsv update -A
```

Download versions that correspond to a specific tag

```
cvsv -r $revision
```

### **Check the status of the files**

(which revision, version)

```
cvsv status $nom_des_fichiers
```

FOR WINDOWS :

## 10.5 Compilation on Shockfish Servers

Two files must be present in the root directory: `app_cst.h` and `make_bin`

### **app\_cst.h**

This file must be placed in the root directory of your application. Sensor nodes and base stations must have EXACTLY the same `app_cst.h` file.

It enables to define important system variables for our network:

```
#ifndef __APP_CONSTANTS_H__
#define __APP_CONSTANTS_H__

enum {
    MAX_SINK_ID = 1024,
    INTERVAL_TIME_IN_MS = 3000, // beacon period
    MAX_CHILDREN = 2,
};
uint32_t SENSOR_SAMPLING_INTERVAL_IN_MS = ((uint32_t)4*INTERVAL_TIME_IN_MS);
#endif
```

*MAX\_SINK\_ID*: the upper bound for sink IDs, i.e. base stations must have `TOS_NODE_ID < MAX_SINK_ID`. Accordingly, sensing node IDs must be superior or equal to `MAX_SINK_ID`.

*INTERVAL\_TIME\_IN\_MS* is the beacon send period which are used for synchronizing. Note also that your sensor polling will only be an integer multiple of this value if you use the optimized NesC code (see in the dedicated section).

*MAX\_CHILDREN* will define the maximum number of children each node will have. The number of “processing slots” in the code will depend on this value. The basestation does not consider this value.

*SENSOR\_SAMPLING\_INTERVAL\_IN\_MS* is the sampling period of sensors. If a node does not receive any packet during twice this interval it declares the sender as dead. The base station though does not time out a child.

### **Make\_bin**

This file must be placed in the root directory of your application

You will use it instead of the usual “make tinynode” command if you wish to compile with the Shockfish MAC/Routing layer on the Shockfish server.

```
#!/bin/sh
#
# Build script for CommonSense with Shockfish MAC/Routing layers
#
SSHKEY=../../sshkey/id_rsa
USER_HOME=/home/commonsense
#USER=commonsense@hq.shockfish.com
#PORT=822
USER=commonsense@localhost
PORT=9060

find . | grep "nc$|Makefile|h$|c$" | xargs tar czf src.tar.gz > /dev/null 2>&1
scp -P $PORT -i $SSHKEY src.tar.gz $USER:$USER_HOME
ssh -p $PORT -i $SSHKEY $USER 'build_tinynode'
scp -P $PORT -i $SSHKEY $USER:$USER_HOME/build.tar.gz build.tar.gz
tar xzf build.tar.gz > /dev/null 2>&1
```

You must use a private SSH key (`id_rsa`) to login to the Shockfish server. On the top part of `make_bin`, adapt the relative path to the key. Keep the current username. The password is the same as the username.

The variables `USER` and `PORT` depend on whether you work from EPFL or from IISc.

From IISc (using a tunnel):

```
USER=commonsense@localhost
PORT=9060
```

From EPFL:

```
USER=commonsense@hq.shockfish.com
PORT=822
```

What `make_bin` does is to take all your TinyOS application related files (`.nc`, `.c`, `.h` and `Makefile`) in all subdirectories starting from `$PWD` and put them into a tarball file. It then uploads the file on the Shockfish server.

Note: Make sure all your modules are all present within the subdirectories of `$PWD` (i.e. put all your personal modules within the application subdirectories and adapt the `Makefile` accordingly with e.g. `PFLAGS += -I./MyMoistureSensor/`).

Then it uses SSH to launch to remote compilation. You will see all the remote compilation messages on your shell screen, which enables you to debug properly.

Finally it will copy the compiled files (binaries) back to your local directory and untar them.

## Tunnel Configuration

From EPFL, no configuration is required. The following explanations only apply to IISc using a ssh tunnel.

1) Download and install the Putty suite

[www.putty.nl/download.html](http://www.putty.nl/download.html)

2) Save `key.ppk` (on the CommonSense CD) somewhere in your home directory

3) Set up a new SSH connection

- open putty
- in the Session tab: enter `lca2srv2.epfl.ch` as hostname
- in the Connection tab: enter `jpanchar` as auto-login name
- in the SSH - Auth tab: enter the path to the key.ppk file
- in the SSH - Tunnels tab:
  - enter `9060` into Source port
  - enter `hq.shockfish.com:822` into Destination
  - check "Local"
  - click on Add
- in the Session tab: enter a name for the session and click on Save
- click on open
- when prompted, enter `tinynos20`

## cygminires.dll error message

If you get an error message about `cygminires.dll`, you must also (re)install "cygminires":

Launch the cygwin setup file from <http://www.cygwin.com/setup.exe> (or the local `setup.exe` file on your disk)

In the package list that will appear, select the radio button "Keep", click on view to obtain "Full".

Choose the minires package and click on the corresponding "New" column to obtain "reinstall" and check the "source" box. Click next to install the package. It should work now.

## Compiling and installing

Now that you configured `make_bin`, `app_cst.h` and the tunnel, you can apply the following lines for the base station and the nodes

Compile:

```
./make_bin
```

Install:

```
make reinstall.Y bsl,/dev/ttyS<x>
```

with

- `Y < MAX_SINK_ID` for a sink, `Y >= MAX_SINK_ID` for a node.
- `<x>` is the serial port ID you have connected your node to.

## 10.6 Server Installation

The following elements must be installed. They are all available in the CommonSense CD. I also include the links, if you want to install newer versions.

**EasyPHP 2** (including Apache 2.2.3, PHP 5.2.0)

<http://www.easyphp.org/index.php3>

Copy the www directory from the CommonSense CD into the www directory of Apache.

**Python 2.4 :**

<http://code.enthought.com/enthon/>

**Mod\_python 3.3.1**

<http://www.modpython.org/>

**Matplotlib > 0.71:**

<http://sourceforge.net/projects/matplotlib>

**PIL 1.1.6:**

<http://www.pythonware.com/products/pil/>

**PostgreSQL 8.2 :**

<http://www.postgresql.org/>

During the installation you may have to delete manually the following files from your Windows directory: libeay32.dll , ssleay32.dll

You also need to add C:\Program Files\PostgreSQL\8.2\bin to the PATH variable.

To create the database you need to do the following in a cygwin shell:

Enter the directory where you have icmon.sql file (included in the CommonSense CD) and enter:

```
createdb -U icmon icmondb_rev2
psql -e icmondb_rev2 icmon < icmon.sql
```

## 10.7 Server application user's guide

Given a properly installed server (good luck!), here is how to use the whole server application.

### Basestation

- Connect the Basestation node to the server with a serial cable (or USB to Serial)

### SerialForwarder

- open a cygwin session  
- enter the following command to launch the SerialForwarder

```
java net.tinyos.sf.SerialForwarder
```

- In the window that opens, under "Mote communication", enter\$

```
serial@comX:57600
```

where comX is your serial port connected to the base station. To find out what is the port name you go to "Device Manager" of your Windows OS.

- Restart the server: packets should arrive

### Listener

If you want to see the raw bytes arriving at the base station, the Listener is for you!

- open a cygwin session  
- enter:

```
Java net.tinyos.tools.Listen
```

### Database Server

- launch the PostgreSQL database server (in the programs menu)

### GeneralDataLogger

- open a cygwin session  
- enter the directory `/"somepath"/net/tinyos/icmon/generaldatalogger/`  
- enter

```
Java net.tinyos.icmon.generaldatalogger.GeneralDataLogger 0x7d  
icmondb_rev2 tinynode
```

With `0x7d` the AM group we use in our network  
`icmondb_rev2` is the database we use to store the sensor data  
`Tinynode` is the platform

### Apache

In EasyPHP, just launch the Apache Server.

### Firefox

In Firefox, enter the address `http://localhost/` and you enter the CommonSense website.  
On the top menu, choose "commands".

## 10.8 Recompiling files from TinyOS 2 Java SDK

If you want to make changes to MotelF or any file from the original TinyOS2-Java SDK, you have to include this modification in the tinyos.jar file included in the classpath. To do that, apply the change to the source file you want, then go to

```
$TOSROOT/support/sdk/java
```

Launch

```
make
```

Then, in the same directory you must create a new JAR file with the compiled classes

```
Jar cvf tinyos.jar net/
```

Normally this jar file is already in your class path. If not, you have to add it.