ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Forest Fire Detection with Wireless Sensor Networks

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ευάγγελος Ν. Σιμόπουλος

Επιβλέπων : Γεώργιος Ε. Οικονομάκος
Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2013

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Forest Fire Detection with Wireless Sensor Networks

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ευάγγελος Ν. Σιμόπουλος

Επιβλέπων : Γεώργιος Ε. Οικονομάκος
Επίκουρος Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 4η Σεπτεμβρίου 2013.

....	....	....
Γεώργιος Οικονομάκος	Κιαμάλ Πεκμεστζή	Δημήτριος Σούντρης
Επίκουρος Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2013

....
Ευάγγελος Ν. Σιμόπουλος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Τα κατανεμημένα συστήματα είναι συστήματα υλικού και λογισμικού, στα οποία, στοιχεία ενσωματωμένα σε υπολογιστές, επικοινωνούν και συντονίζουν τις ενέργειές τους με ανταλλαγή πληροφοριών μεταξύ τους. Ένα κατανεμημένο σύστημα μπορεί να αντιμετωπιστεί ως ένα παράλληλο σύστημα το οποίο βασίζεται σε αυτόνομα υπολογιστικά μέρη (με ενσωματωμένη CPU, αποθηκευτικό χώρο, τροφοδοσία, διεπαφές δικτύου κλπ.) συνδεδεμένα σε ένα δίκτυο (τοπικό, ευρείας περιοχής ή το Διαδίκτυο).

Σκοπός αυτής της διπλωματικής εργασίας είναι η θεωρητική κατασκευή και προσομοίωση της λειτουργίας ενός δικτύου αυτόνομων υπολογιστικών μονάδων, ασύρματα συνδεδεμένων μεταξύ τους, με στόχο την πλήρη παρακολούθηση και προστασία μιας δασικής έκτασης από πυρκαγιές. Το δίκτυο αυτό αποτελείται από μικρές μονάδες, χαμηλού κόστους και ενεργειακής κατανάλωσης, με ενσωματωμένους αισθητήρες. Τέτοια δίκτυα είναι γνωστά ως Ασύρματα Δίκτυα Αισθητήρων (WSN), οι κόμβοι των οποίων ονομάζονται motes. Βασική δομική μονάδα θα αποτελέσει η ενσωματωμένη πλατφόρμα CM5000 της Advanticsys, κατασκευασμένη σε αντιστοιχία με την πλατφόρμα ανοιχτού λογισμικού TelosB/Tmote Sky του Πανεπιστημίου Berkeley. Ολοκληρώνεται γύρω από τον μικροεπεξεργαστή μικτών σημάτων MSP430 της Texas Instruments και τον πομποδέκτη CC2420 της Chipcon, ενώ διαθέτει ενσωματωμένους αισθητήρες θερμοκρασίας, υγρασίας και φωτεινότητας.

Ο προγραμματισμός των motes θα γίνει με τη βοήθεια του TinyOS, ενός λειτουργικού συστήματος ανοικτού κώδικα κατάλληλα σχεδιασμένου για τον προγραμματισμό ενσωματωμένων συστημάτων, γραμμένο σε μια παραλλαγή της γλώσσας προγραμματισμού C, με την ονομασία nesC (**n**etwork **e**mbedded **s**ystems **C**). Το TinyOS θα εγκατασταθεί σε μια διανομή Linux. Για λόγους ευκολίας χρήσης και συμβατότητας επιλέχθηκε η διανομή Ubuntu 13.04. Για την προσομοίωση της λειτουργίας του δικτύου θα χρησιμοποιηθεί ο προσομοιωτής Cooja και μέσω αυτού ο MSPSim.

Λέξεις Κλειδιά: Κατανεμημένα Συστήματα, Ενσωματωμένα Συστήματα, Ασύρματα Δίκτυα Αισθητήρων, WSN, Δασική Έκταση, Πυρκαγιά, Παρακολούθηση, Προσομοίωση, CM5000, TelosB, Μικροελεγκτής Μικτού Σήματος, MSP430, CC2420, TinyOS, nesC, Cooja, MSPSim, Eclipse, Yeti 2

Abstract

Distributed systems are hardware and software systems, in which, components installed on computers, communicate and coordinate their actions by exchanging information. A distributed system can be seen as a system that relies on standalone computing parts (with onboard CPU, storage space, power supply, network interfaces etc.) connected to a network (local, wide area or the Internet).

The scope of this thesis is the theoretical construction and simulation of operation, of a network of standalone computing units, wirelessly connected to each other, targeting the full monitoring and protection of a forest area from fires. This network will be composed of small, low cost and low power consuming modules with onboard sensors. Such networks are known as Wireless Sensor Networks (WSN), the nodes of which are called motes. The basic structural module will be the embedded platform CM5000 by Advanticsys, built in accordance with the open source platform TelosB/Tmote Sky of the University of California, Berkeley. It is built around the MSP430 mixed signal microprocessor by Texas Instruments and the CC2420 transceiver by Chipcon, while it embeds temperature, humidity and light sensors.

The programming of the motes will be done with the help of TinyOS, an open source operating system, specifically designed to program embedded systems and written in a dialect of the C programming language, called nesC (**n**etwork **e**mbedded **s**ystems **C**). TinyOS will be installed on a Linux distribution. For ease of use and compatibility reasons, Ubuntu 13.04 was chosen. For the simulation of the network's operation, the simulator Cooja and within it MSPSim, will be used.

Keywords: Distributed Systems, Embedded Systems, Wireless Sensor Networks, WSN, Forest Area, Fire, Monitoring, Simulation, CM5000, TelosB, Mixed Signal Microprocessor, MSP430, CC2420, TinyOS, nesC, Cooja, MSPSim, Eclipse, Yeti 2

Acknowledgments

First and foremost, I would like to thank Prof. George Economacos for being my supervisor, for his support and eagerness to help me and for our excellent cooperation throughout the duration of this thesis.

I want to express my gratitude to the TinyOS and ContikiOS communities, and their developers, with whom I had the pleasure to discuss many of the problems I faced.

Last, but by no means least, I would like to thank my mother for her silent but endless support and encouragement throughout all these years of study.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1: Introduction

"I'm not sure what solutions we'll find to deal with all our environmental problems, but I'm sure of this: They will be provided by industry; they will be products of technology. Where else can they come from?

George M. Keller, Nation's Business
12 June 1988 (S&S)

1.1 Introduction

In times where technology evolves on a daily basis, new inexpensive solutions can be created. These new tools overcome the existing ones by requiring less work while achieving better results and offering more functionality. These new technologies also allow the conception of new tools to previously unsolvable problems.

Wireless Sensor Networks (WSN) are wireless networks formed by small low cost autonomous devices called motes, with the ability to sense the surrounding environment. An extension to WSN that adds the ability to act besides sensing over the environment is called Wireless Sensor and Actor Networks (WSAN). Both WSAN and WSN are possible solutions for several problems. Their main characteristics are easy deployment and low cost, while having the ability to sense and act without human intervention makes their usage highly attractive in many applications. They are being adopted in several fields of work. Some examples include: creating effective irrigation systems, fire alarms, structure health monitoring and medical or military applications. Throughout this thesis we will deal with WSNs for forest fire detection. Our exact purpose will be to design a WSN to monitor a forest area, acquire readings from appropriately placed sensors and transmit these readings over-the-air so as to detect a fire when it starts and prevent it from spreading.

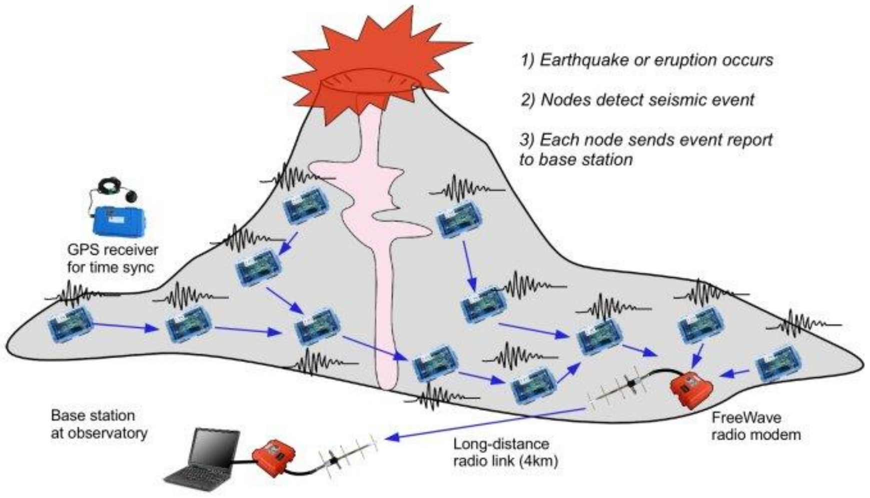
1.2 Monitoring

Monitoring can be defined as the act of continuously observing something. It generally means to be aware of the state of a system. Environmental monitoring describes the processes and activities that need to take place to characterise and monitor the quality of the environment.

When we refer to monitoring we can differentiate two types: active and passive. The difference between these two types is that while active monitoring necessarily involves human presence, being performed through field visits to the monitored environment, passive monitoring is done by autonomous systems not requiring human intervention. In this case, the monitoring system is placed in the environment, automatically acquiring data and either storing it locally for later retrieval or sending it to a remote system.

Figure 1.2.1: Active monitoring (left) vs. passive monitoring (right)

Several applications of WSNs in monitoring exist such as animal monitoring used by biologists to study animals in the wild, structure health monitoring used to ensure buildings or bridges condition, volcano monitoring used to study the seismic activity of volcanic areas and obviously forest monitoring mainly used for forest fire detection.


1) Earthquake or eruption occurs
2) Nodes detect seismic event
3) Each node sends event report to base station
GPS receiver for time sync
Base station at observatory
Long-distance radio link (4km)
FreeWave radio modem
Figure 1.2.2: Harvard’s volcano monitoring [8]

1.3 Motes and Sensors

We will cover the motes in more detail in Chapter 2 but an introduction here is necessary, as they are the basic building block of a WSN. A sensor node (also known as mote) may be described as a small low-cost device with the ability to perform some processing, gather sensory information and communicate with other connected nodes in the network [12]. A mote is a node but a node is not always a mote. Its main components are a microcontroller, transceiver, external memory, power source and one or more sensors. A typical mote can be seen in the picture below.

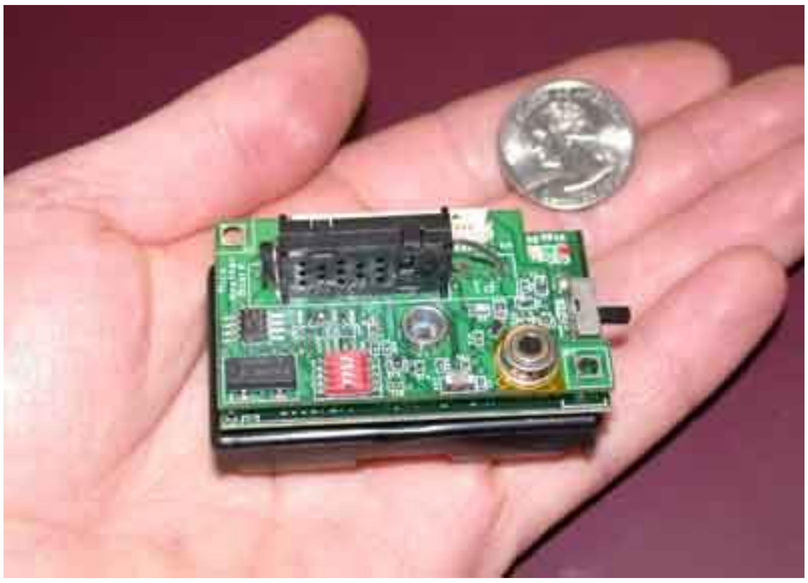


Figure 1.3.1: Typical mote

The microcontroller and memory provide computational power and storage space respectively, while the power source – usually a battery – provides energy supply to the mote, making it autonomous. The mote captures data through the acquisition system composed of a set of sensors. These may be embedded directly in the mote or a separate sensor board connected to the mote via its I/O ports. Sensors of any type (e.g. temperature, humidity, light, acceleration etc.) can be connected depending on the type of data we intend to capture. Using a transceiver, the communication module allows data to be wirelessly transmitted and received between nodes. The typical architecture of a mote is depicted below [12]. Again, more on this in Chapter 2.

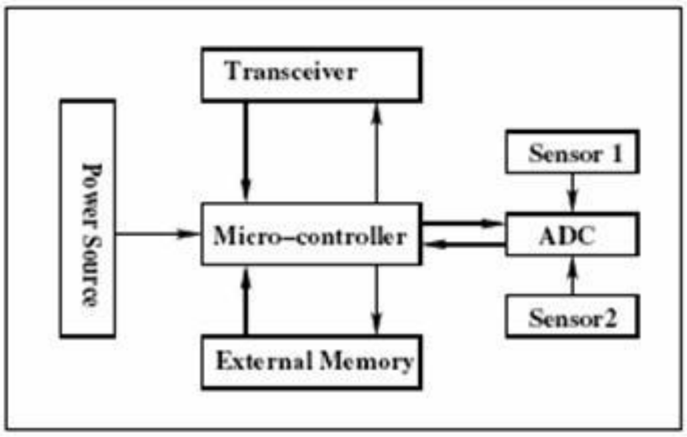


Figure 1.3.2: Typical mote architecture

1.4 Wireless Sensor Networks

Wireless sensor networks (WSN) are wireless networks formed by motes. The wireless and routing technologies in motes allow them to be deployed creating a WSN, where each node may capture environmental information and share it with all other motes. The system’s cost can be highly reduced by avoiding cabling and instead use wireless technology. This also allows both a more flexible deployment and lower maintenance costs.

WSNs intend to provide a low cost solution to problems such as monitoring large areas, difficult access or hazardous environments. These networks can replace expensive active monitoring with cost effective passive monitoring. It is possible to set the motes to capture data for a certain period of time and transmit it to be stored in a central node called sink, where a person could be in order to access and monitor the captured information. The biggest challenges that WSN designers are faced with nowadays are energy efficiency, routing and security [11]. They are presented in more detail below.

1.4.1 Energy Efficiency

Energy management and consumption are critical challenges for WSNs as motes require energy to operate each of their composing parts and be autonomous. The main objective of studies conducted in this field is to maximize the motes' lifetime. All motes' components require a certain amount of energy to operate even when it comes to small amounts. The connection of motes to a power source such as a power socket, implies the use of cables, thus nullifying the benefits of wireless technology.

Most motes nowadays are battery powered, allowing them to be autonomous and wireless but also limiting their lifetime. What WSN designers can do to maximize a mote's lifetime is to minimize its hardware energy consumption. The power usage can be reduced by putting motes into sleep mode - a state where all mote's activity is stopped and all of its composing parts are switched off - or even by putting a single component to sleep when not in use (e.g. switch off the radio transceiver), thus reducing its duty cycle - the percentage of time during which a device is working.

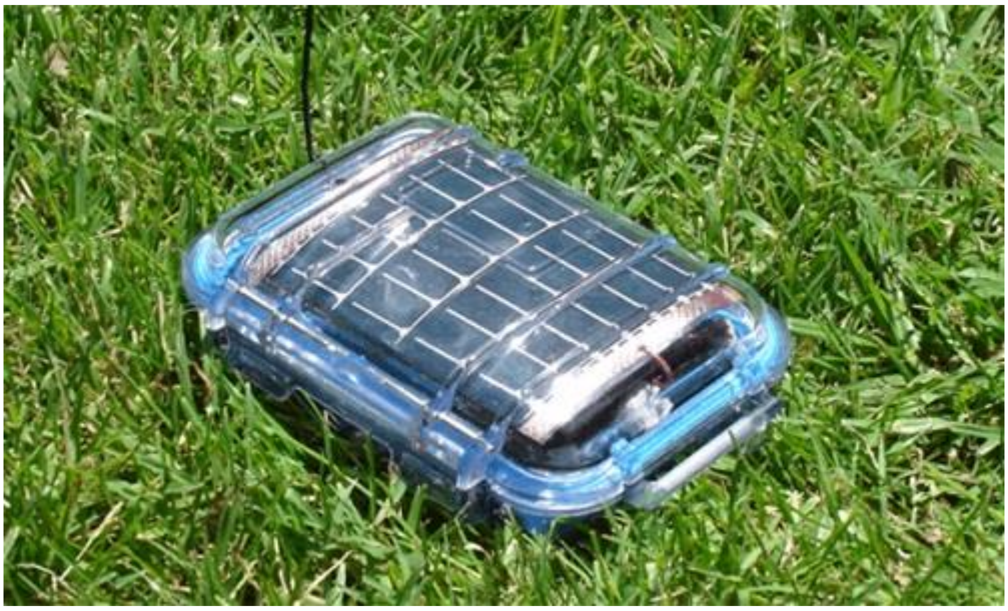


Figure 1.4.1: Heliomote [13]

Research is being done to find alternative or complementary power sources to batteries. Environmental energy harvesting methods are being studied as they allow the mote to collect energy from the environment. Two of the aforementioned methods include solar cells, that allow the conversion of sunlight to electricity through solar panels, and piezoelectric ceramic materials that convert environment vibrations to electricity. The use of energy harvesting techniques turns everlasting mote lifetime into a possibility. Some commercially available products already exist, such as the Heliomote. As will be shown in Chapter 2, the mote that will be used for this thesis is powered by a pair of batteries. We will not go into the process of dealing with alternative power sources for two reasons. In order to use a solar cell a mote has to be put under direct sunlight, which means on top of a tree. But doing so, i) increases the distance of the mote from the ground and because a fire always starts from the ground up, the mote gets slower in detecting changes in light flux or temperature, thus increasing the duration of the crucial first time detection, and ii) may cause the mote and its sensors to overheat during the hot summer months and provide false sensor readings. Piezoelectric generators do not apply to our case too, as they are best suited to seismic oriented applications.

Figure 1.4.2: PMG37 Microgenerator

1.4.2 Routing

Routing collected information between sensor nodes in WSNs presents several challenges. The different kinds of network topologies and their requirement for different routing protocols, the possibility that nodes are randomly deployed or large in quantity are some of the faced problems. Energy and computation constraints also impose new requirements to routing algorithms. A system failure or power shortage may turn off nodes, requiring new routes to be calculated so as to maintain network connectivity between the rest operating nodes.

Requirements such as low energy and memory consumption mean limited routing tables and new algorithms. Several routing protocols have been specifically designed for WSNs in order to appropriately fulfill these special needs. The existing routing protocols are categorized according to the network structure in which they operate and the protocol operation. Depending on the network structure they can be classified as flat, hierarchical or location-based routing. Depending on their operation they can be multipath-based, query-based, negotiation-based, QoS -based or coherent-based [11].

1.4.3 Security

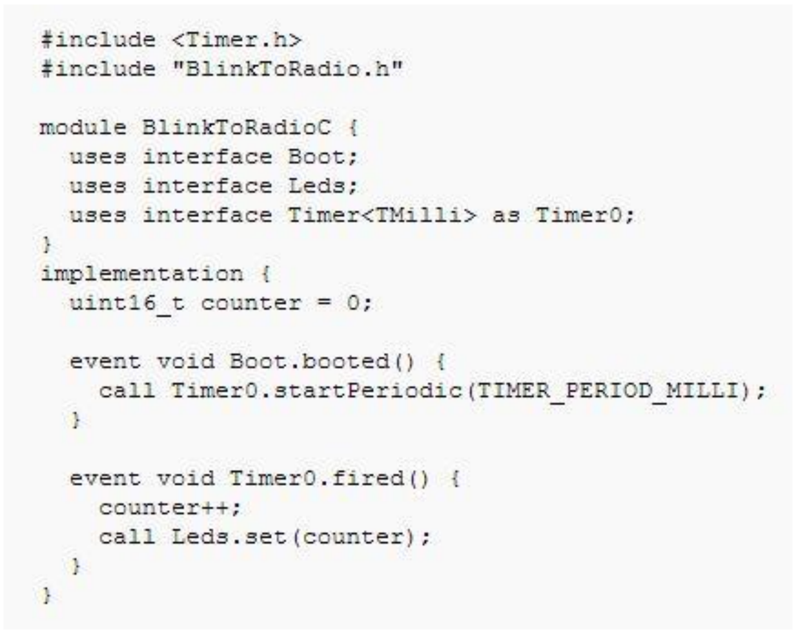
The use of wireless technology in WSNs has numerous benefits but it also introduces several security threats that need to be considered. Motes' characteristics of limited computing power and low energy resources represent a challenge in producing an effective security solution.

Attacks against WSNs are divided into two types: attacks against the security mechanisms and against basic mechanisms. Some of the common WSN attacks are denial of service (DoS), attacks on information in transit, blackhole/sinkhole attacks, hello flood attacks or wormhole attacks. Most of those are caused when a malicious node sends false information to other nodes thus compromising the system. Detecting mechanisms to solve these attacks are still being developed.

1.5 Operating Systems

Due to specific requirements and constraints of sensor nodes and wireless sensor networks, operating systems have been created specifically targeting embedded platforms, their needs and objectives. Reconfiguration, energy awareness and optimization, self-configuration, multi-hop communications, memory and computation power constraints, are some of the requirements these operating systems need to address.

Some of the most popular operating systems used, are Nano-RK [21] developed at Carnegie Mellon University, SOS [22] developed at University of California Los Angeles, MANTIS [23] developed at the University of Colorado, BTNut [24] developed at ETH Zurich, Contiki [25] at Swedish Institute of Computer Science and, the most widely used, as well as the one that will be used in this thesis, TinyOS [9] created at the University of California Berkeley.



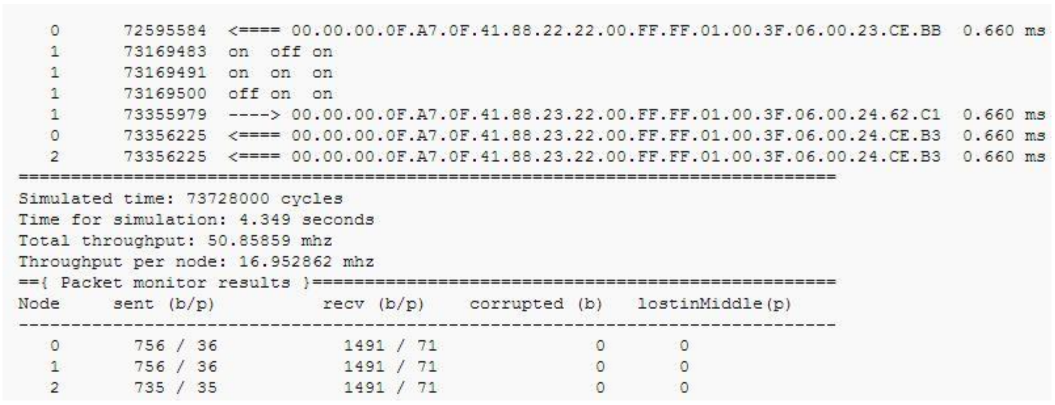
```
#include <Timer.h>
#include "BlinkToRadio.h"

module BlinkToRadioC {
  uses interface Boot;
  uses interface Leds;
  uses interface Timer<TMilli> as Timer0;
}
implementation {
  uint16_t counter = 0;

  event void Boot.booted() {
    call Timer0.startPeriodic(TIMER_PERIOD_MILLI);
  }

  event void Timer0.fired() {
    counter++;
    call Leds.set(counter);
  }
}
```

Figure 1.5.1: TinyOS

TinyOS is an open source operating system featuring a component-based architecture minimizing memory usage and providing an event-driven execution model allowing fine-grained power management and scheduling flexibility. Software programs developed in TinyOS are programmed using nesC, an extension to the C programming language. We will examine both TinyOS and nesC, in depth, in Chapter 3.

1.6 Simulators

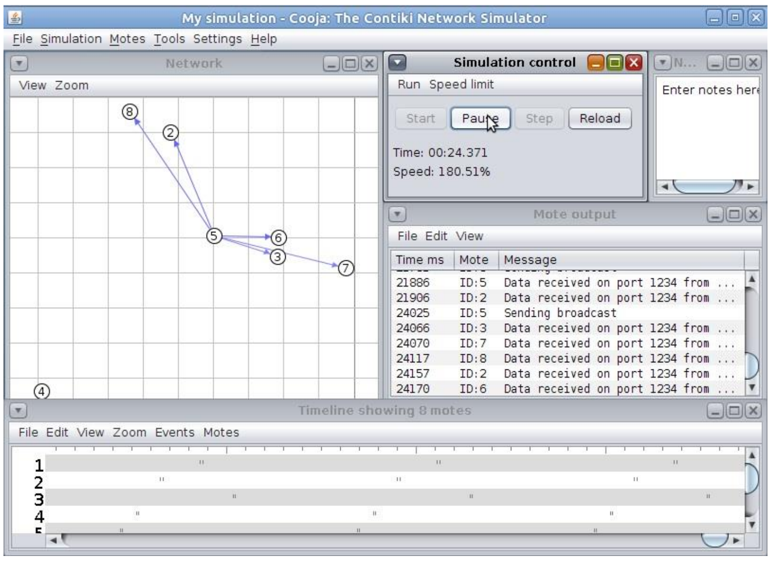
Simulators are software platforms specifically designed to simulate a WSN's or even a single mote's behavior. These platforms allow testing a developed program without having to install the software in the actual motes or, as in our case, without even having any physical sensor node. Simulators are immensely time-saving when one needs to examine the characteristics and operational parameters of a WSN involving hundreds or thousands of motes, prior to its installation.



```
   0      72595584  <==== 00.00.00.0F.A7.0F.41.88.22.22.00.FF.FF.01.00.3F.06.00.23.CE.BB  0.660 ms
   1      73169483  on  off on
   1      73169491  on  on  on
   1      73169500  off on  on
   1      73355979  ----> 00.00.00.0F.A7.0F.41.88.23.22.00.FF.FF.01.00.3F.06.00.24.62.C1  0.660 ms
   0      73356225  <==== 00.00.00.0F.A7.0F.41.88.23.22.00.FF.FF.01.00.3F.06.00.24.CE.B3  0.660 ms
   2      73356225  <==== 00.00.00.0F.A7.0F.41.88.23.22.00.FF.FF.01.00.3F.06.00.24.CE.B3  0.660 ms
==============================================================================
Simulated time: 73728000 cycles
Time for simulation: 4.349 seconds
Total throughput: 50.85859 mhz
Throughput per node: 16.952862 mhz
=={ Packet monitor results }==================================================
Node     sent (b/p)          recv (b/p)    corrupted (b)   lostinMiddle(p)
------------------------------------------------------------------------------
   0       756 / 36            1491 / 71              0       0
   1       756 / 36            1491 / 71              0       0
   2       735 / 35            1491 / 71              0       0
```

Figure 1.6.1: Avrora simulator

Using a simulator, it is possible to monitor and analyze every single mote in a simulated network and its response during its life cycle. Energy consumption, packets received, sent or dropped and the mote's LEDs status are some of the variables usually observed. A large number of simulators exist, some of them are: TOSSIM the native simulator from TinyOS, Avrora [26] developed at the University of California Los Angeles, Cooja originally created at the Swedish Institute of Computer Science as a Contiki simulator but now able to simulate nodes programmed in the TinyOS operating system as well [1], and MSPSim, a MSP430 simulator [5], also developed at SICS.



Time: 00:24.371
Speed: 180.51%

```
Time ms  Mote  Message
21886    ID:5  Data received on port 1234 from ...
21906    ID:2  Data received on port 1234 from ...
24025    ID:5  Sending broadcast
24066    ID:3  Data received on port 1234 from ...
24070    ID:7  Data received on port 1234 from ...
24117    ID:8  Data received on port 1234 from ...
24157    ID:2  Data received on port 1234 from ...
24170    ID:6  Data received on port 1234 from ...
```

Figure 1.6.2: Cooja simulator

Chapter 2: Motes

2.1 Introduction

The term "mote" was coined by researchers in the Berkeley NEST (now WEBS and CENS projects) to refer to spatially distributed autonomous devices which use sensors to cooperatively monitor physical and/or environmental conditions (e.g. temperature, sound, pressure, vibration) at different locations. Practical WSN nodes, henceforth "motes", currently range in size from disc-shaped boards having diameters less than 1 cm to enclosed systems with typical dimensions less than 5 cm square.

Each mote is composed of a microcontroller, transceiver, memory, power source and one or more sensors, either embedded or external to the sensor board. The motes function within a WSN and typically fulfil one of two purposes: either data logging, processing (and/or transmitting) sensor information from the environment or acting as a gateway in the adhoc wireless network formed by all the motes to pass data back to a, usually but not necessarily unique, collection point.

In this chapter we present a brief review of several frequently used WSN motes, compared and contrasted under a number of different parameters. Then, we will delve into the details of Advanticsys CM5000 [27], a TelosB/Tmote Sky based mote, examining each of its structural components and explaining why it is our mote of choice for this thesis.

2.2 Common mote platforms

TelosB/Tmote Sky: Wireless sensor modules developed from research carried out at University of California Berkeley and currently available in similar form factors from Crossbow and Advanticsys.

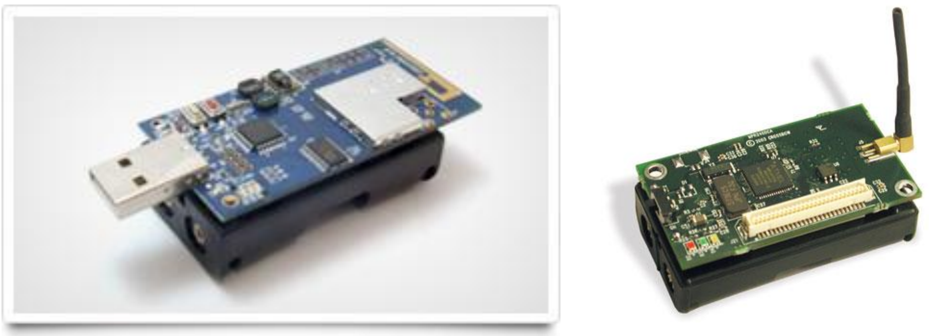


Figure 2.2.1: TelosB/ Tmote Sky (left) & MicaZ (right) motes

MicaZ: Second and third generation wireless sensor networking mote family from Crossbow.

SHIMMER: (Sensing Health with Intelligence, Modularity, Mobility and Experimental Reusability) is a wireless sensor platform designed to support wearable applications and is mainly used in the medical field.

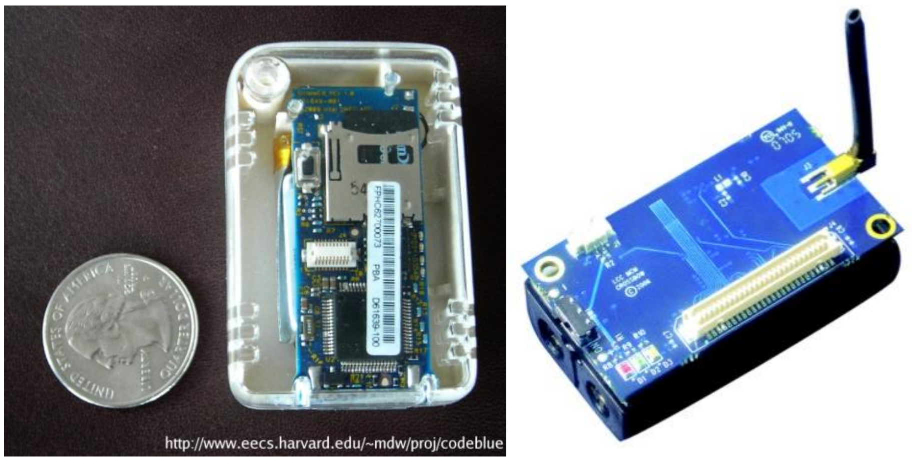


Figure 2.2.2: SHIMMER (left) & IRIS (right) motes

IRIS: The latest wireless sensor network module from Crossbow. Incudes several improvements over the Mica2/MicaZ family of products. Improvements include increased transmission range.

2.2.1 Physical Characteristics

The first parameter which may dictate mote selection for a given application is physical size. Table 2.2.1 provides an overall comparison of the physical dimensions of the motes in the previous section. This table also lists the motes' weight, which can be a decisive factor when choosing a certain WSN, especially in applications where the motes are components of a mobile unit or are integrated into wearable health monitoring solutions.

Mote Platform	WxLxH [cm]	Weight w/o batt [g]	Weight with batt [g]
TelosB/Tmote Sky	3.2 x 6.6 x 0.7	14.93	63.05
MicaZ	3.2 x 5.7 x 0.6	15.70	63.82
SHIMMER	2 x 4.4 x 1.3	4.87	10.36
IRIS	3.2 x 5.7 x 0.6	21.29	69.40

Table 2.2.1: Physical characteristics of common mote platforms [3]

The SHIMMER platform's advantage is obvious. Its small dimensions and low weight make it much more suitable than the other in medical oriented applications. When a mote has to be part of a wearable application, its size and weight are of the utmost importance. Its low weight also minimizes the effect of the mote’s inertial mass when using the mote's embedded accelerometer. In our case, weight and size is not going to be a deciding factor as the motes will be stationary and placed on trees.

2.2.2 Processor and Memory

Table 2.2.2 reviews the microprocessor specifications (bus width and processor clock speed) for each of the respective motes examined. It also provides information on available on-board memory for each mote platform. There is a variety here in available memory sizes, possibly a reflection of their different application spaces.

Mote Platform	Microprocessor	Bus [bits]	Clock [MHz]	RAM [KB]	Flash [KB]	EEPROM
TelosB/ TmoteSky	Texas Instruments MSP430F1611	16	4	10	48	1M
MicaZ	Atmel Atmega 128L	8	8	4	128	512K
SHIMMER	Texas Instruments MSP430F1611	16	8	10	48	none
IRIS	Atmel Atmega 1281	8	8	8	640	4K

Table 2.2.2: Microprocessor & memory specifications [3]

In addition to these on-board memory capabilities, some sensor nodes also allow the option of saving data to additional external non-volatile memory.

2.2.3 Communications Capabilities

The TelosB/Tmote Sky, MicaZ and SHIMMER motes, employ the 802.15.4 compatible CC2420 radio chip from Texas Instruments, while the IRIS Mote uses (again a 802.15.4 compatible chip) Atmel's AT86RF230. These two radios are packet level radios, with a maximum packet length of 127 bytes. In addition to the CC2420, the SHIMMER mote also contains a second radio chip, a class 2 Bluetooth radio compatible with the Mitsumi WML-C46 series. Table 2.2.3 lists the operating specifications of the three radios and Table 2.2.4 gives the power consumption of each radio in sleep mode/switched off, idle/receive mode and when transmitting at a specified power level.

Radio Module	Frequency [MHz]	Modulation	Data Rate	Tx Power [dBm]	Rx Sensitivity [dBm]
TI CC2420	2400 - 2483.5	QQPSK	250 Kbps	-24 - 0	-95
Atmel AT86RF230	2405 - 2480	QQPSK	250 Kbps	-17 - 3	-101
Mitsumi WML-C46	2400 - 2483.5	GFSK	721 Kbps	-6 - 14	-82

Table 2.2.3: Radio chip specifications [3]

The CC2420 is a very popular chip for use on wireless sensor nodes, being used on three of the motes considered here. The CC2420 was the first 802.15.4 radio chip to be widely available in the market. 802.15.4 is very suitable for use in WSNs due to its very low power and flexibility. A feature of the CC2420 lacking on the other radios, is its support for encryption using AES 128. This feature can greatly reduce the cost, both in terms of power and latency, of securing WSN communications.

Radio Module	Sleep [µA]	Idle/Rx [mA]	Tx [mA]
TI CC2420	1 - 426	18.8	17.4
Atmel AT86RF230	0.02	15.5	16.5
Mitsumi WML-C46	50 - 1400	40	60

Table 2.2.4: Radio chip power consumption [3]

The WML-C46 is a class 2 Bluetooth radio, with a range of approximately 10 meters. WSNs were not considered as a target for Bluetooth when it was being designed and as a result it is not ideally suited for use with them, being overly complex for most applications. However, the presence of Bluetooth allows it to address a current problem faced by 802.15.4 devices, which is interoperability with existing devices. For many applications a Bluetooth enabled mobile phone or laptop can be a very convenient device to use for data aggregation or network querying.

2.2.4 Sensor Support

The TelosB/Tmote Sky offers a versatile set of onboard sensors, namely humidity, temperature and light sensors. In addition to the onboard sensors, the TelosB/Tmote Sky provides access to 6 ADC inputs, a UART and I2C bus and several general purpose ports. The MicaZ motes do not have onboard sensors. However, Crossbow offers an extensive set of sensor boards that connect directly to the MicaZ mote and are capable of measuring light, humidity, temperature, pressure etc. Additionally, actuators such as relays and buzzers can be attached too, in case of a WSAN. Intel's SHIMMER mote incorporates a 3 axis accelerometer and allows connection of other sensors through its expansion board. As in MicaZ, more types of sensors (most of them medically oriented) are available. The IRIS mote, in Crossbow tradition, does not offer any embedded sensor capabilities. However, it is equipped with a 51-pin expansion connector that existing MicaZ compatible, Crossbow sensor boards can be connected to.

2.2.5 Power Specifications

Both the TelosB and Tmote Sky boards are typically powered from an external battery pack containing two AA batteries. AA cells may be used in the operating range of 2.1 to 3.6V DC, however the voltage must be at least 2.7V when programming the microcontroller flash or external flash. MicaZ and IRIS motes are also powered by a set of two AA batteries in an attached battery pack. The SHIMMER mote is powered by a rechargeable 450mAh Li-Ion battery. The Shimmer design also includes a Texas Instruments BQ-24080 Smart Li Charger for battery management.

2.2.6 Price

Current (August 2013) pricing information for a single mote is shown in Table 2.2.5.

Mote Platform	Price
TelosB/Tmote Sky	77 €
MicaZ	77 €
SHIMMER	199 €
IRIS	87 €

Table 2.2.5: Cost per mote

2.3 Advanticsys CM5000

Taking into consideration the analysis done in section 2.2 we make the decision to use the TelosB/Tmote Sky platform for our purpose. In order to design and simulate our WSN, we will be using more than 10 motes. SHIMMER is the easiest one to leave out because of its high price and medically oriented field of applications. IRIS is another mote we won't consider. We can see from Table 2.2.4 that its radio transceiver may be the most frugal in terms of power consumption, but only by a little and as will be shown in Chapter 4, our application will not be using the radio for long periods of time. The motes will operate mostly in sleep mode, except when they wake up, sample their sensors and send their readings using their radio. That means that the microprocessor's behavior has to be taken into consideration as well, and by doing that, the IRIS's advantage over a TelosB mote turns into a drawback, if we consider that when active, the Atmega 128 microprocessor draws 7.6mA of current instead of 1.8mA of a MSP430, and when idle, 3.3mA instead of 5.1μA of a MSP430 [3]. The same applies in the case of the MicaZ mote. Tmote Sky uses the same radio chip as Micaz but the latter is equipped with an Atmega 128 instead of a MSP430. The Atmega is a faster processor so it is best suited to more CPU intensive applications.

Another important drawback shared by the MicaZ and IRIS motes, is their lack of onboard sensors. This, not only adds to the platform the cost of separate sensor boards, but it makes them harder to program, pack and simulate as well. Modern simulators may be easier to calibrate and configure than it was in the past, but in order to do so, a programmer needs to be acquainted not only with TinyOS and nesC but possibly Java (in the case of Avrora and Cooja) and Python (in the case of TOSSIM) too.

None of the things mentioned above will be a problem in the case of a Tmote Sky mote. It is equipped with onboard sensors, making it the cheapest to buy, and has the lowest microprocessor power consumption, thus prolonging the mote's life expectancy. Its sensors are also perfectly suited to our case, as temperature, humidity and light are crucial variables to be measured when trying to detect a forest fire. The mote also works perfectly (expect for a small sensor misbehavior) with the Cooja and MSPSim simulators that we will be using.

The Advanticsys CM5000, based on the original open-source TelosB/Tmote Sky, will be our mote of choice.

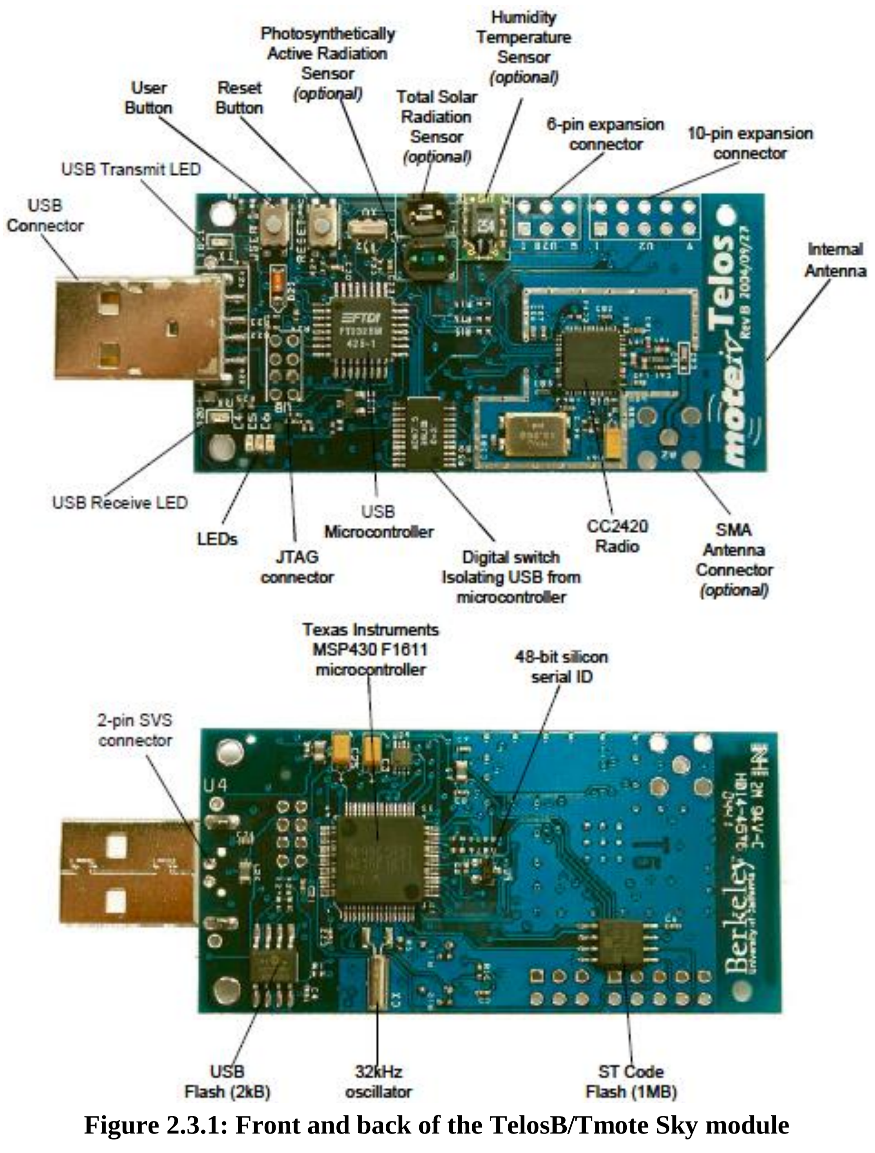

Photosynthetically Active Radiation Sensor (optional)
Humidity Temperature Sensor (optional)
User Button
Reset Button
Total Solar Radiation Sensor (optional)
6-pin expansion connector
10-pin expansion connector
USB Transmit LED
USB Connector
Internal Antenna
USB Receive LED
LEDs
JTAG connector
USB Microcontroller
Digital switch Isolating USB from microcontroller
CC2420 Radio
SMA Antenna Connector (optional)
Texas Instruments MSP430 F1611 microcontroller
48-bit silicon serial ID
2-pin SVS connector
USB Flash (2kB)
32kHz oscillator
ST Code Flash (1MB)

Figure 2.3.1: Front and back of the TelosB/Tmote Sky module

It has the following general characteristics [27]:

- Texas Instruments MSP430F1611 Microcontroller (MCU)
- Texas Instruments CC2420 Radio Transceiver
- Sensirion SHT11 Temperature & Humidity Sensor
- Hamamatsu S1087 & S1087-01 Light Sensors
- User & Reset Buttons
- 3 x LEDs (RGB)
- USB Interface
- 2 x AA Battery Holder
- TinyOS & ContikiOS compatible

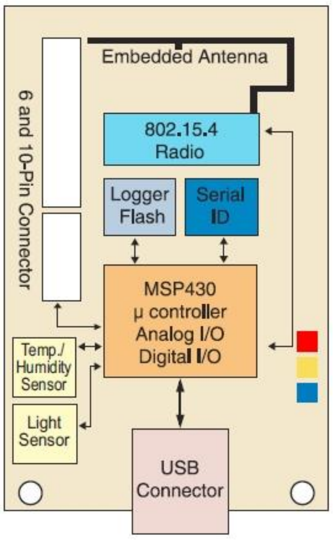


Figure 2.3.2: CM5000 Block diagram [14]

2.3.1 Texas Instruments MSP430F1611

The low power operation of the TelosB module is due to the ultra-low power Texas Instruments MSP430F1611 microcontroller featuring 10kB of RAM, 48kB of program flash memory and 128B of information storage. This 16-bit RISC processor features extremely low active and sleep current consumption that permits Telos to run for months on a single pair of AA batteries.

Figure 2.3.3: TI MSP430F1611

The MSP430 includes three clock sources [15]:

- LFXT1CLK: Low frequency/high frequency oscillator that can be used either with low frequency 32768Hz = 32KHz watch crystals, or standard crystals or resonators in the 450KHz to 8 MHz range.
- XT2CLK: Optional high frequency oscillator that can be used with standard crystals, resonators, or external clock sources in the 450KHz to 8MHz range.
- DCOCLK: Internal digitally controlled oscillator (DCO) with RC-type characteristics.

There are three clock signals available. The one that we are interested in is the Master Clock (MCLK). MCLK is software selectable and is derived from one of LFXT1CLK, XT2CLK or DCOCLK. It is used by the CPU and the system. By default it is sourced from DCOCLK and its default operating frequency in the case of a TelosB mote is 4.15MHz, while it may operate up to 8MHz. In order to achieve frequencies higher than 4.15MHz, a certain minimum amount of supply voltage is necessary. Figure 2.3.4 shows the connection between operating frequency and minimum supply voltage.

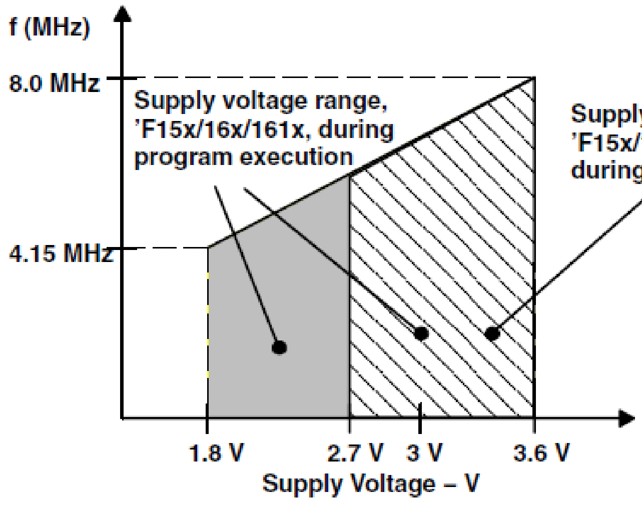


Figure 2.3.4: CPU Frequency vs. Minimum supply voltage

Although it is software configurable, we will not modify the MCLK from its default value, because of the non CPU-intensive nature of our application. If one would like to do so and provided they are using TinyOS, the necessary component to look for [15] would be MSP430ClockC and more specifically its MSP430ClockInit interface. Operating the mote at higher frequencies might be necessary for CPU-intensive programs, but doing so increases the mote's power consumption, while our main objective is to keep it as low as possible.

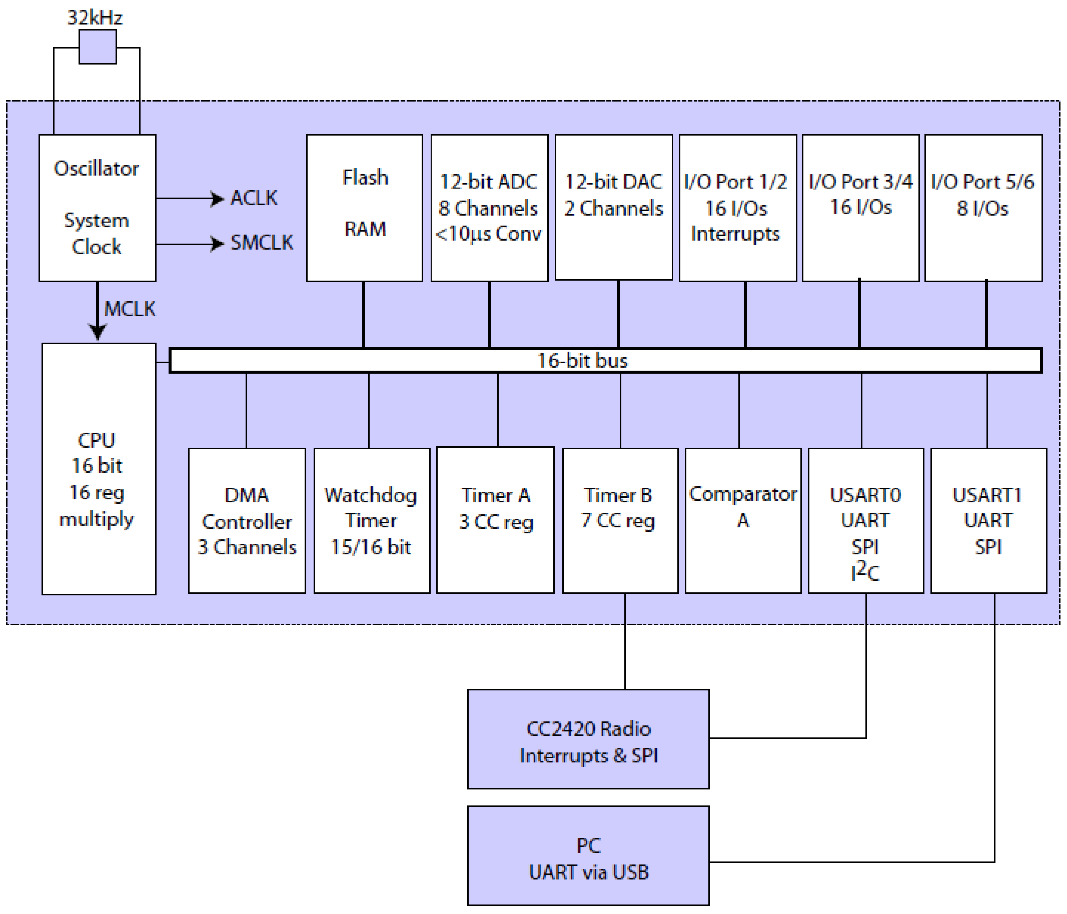


Figure 2.3.5: Block diagram of the TI MSP430 microcontroller and its connection to other peripherals in the Telos module

The DCO may be turned on from sleep mode in 6µs, however 292ns is typical at 20^{o}C. When the DCO is off, the MSP430 operates off an eternal 32768Hz watch crystal. In addition to the DCO, the MSP430 has 8 external ADC ports and 8 internal ADC ports. The internal ports may be used to read the internal thermistor or monitor the battery voltage. A variety of peripherals are available including SPI, UART, digital I/O ports, Watchdog timer and Timers with capture and compare functionality. The F1611 also includes a 2-port 12-bit DAC module, Supply Voltage Supervisor and 3-port DMA controller.

The MSP430 has one active mode and five software selectable modes of operation. An interrupt event can wake up the device from any of the five low-power modes, service the request, and restore back to the low-power mode on return from the interrupt program. As developers, we will not have to worry about MCU management at all in most situations. TinyOS handles everything for us automatically. The low-power modes range from LPM0, which disables only the CPU and main system clock, to LPM4, which disables the CPU, all clocks and the oscillator, expecting to be woken by an external interrupt source. The power parameters of the MSP430F1611 MCU can be seen below (computed at 3.0V supply voltage):

State	Current draw [µA]
Active	1800
Sleep [LPM3]	5.1

Table 2.3.1: Current draw during Active and Sleep modes [15]

2.3.2 Texas Instruments CC2420

The CC2420 is a true single-chip 2.4GHz IEEE 802.15.4 compliant RF transceiver designed for low-power and low-voltage wireless applications. CC2420 includes a digital direct sequence spread spectrum baseband modem providing an effective data rate of 250Kbps. The CC2420 provides extensive hardware support for packet handling, data buffering, burst transmissions, data encryption, data authentication, clear channel assessment, link quality indication and packet timing information. These features reduce the load on the host controller and allow CC2420 to interface low-cost microcontrollers. It is based on Chipcon's SmartRF – 03 technology in 180nm CMOS.

The CC2420 is controlled by the TI MSP430 microcontroller through the SPI port and a series of digital I/O lines and interrupts. The radio may be put to sleep for low power duty cycled operation. The transceiver also has software configurable output power, which the transmission range is obviously dependent on. We will examine this in more detail in later chapters, where we deal with different network topologies and where transmission range will be a critical parameter in reducing overall power consumption. Its main features are summarized below

Frequency Band	2400 ~ 2483.5MHz	IEEE 802.15.4 Compliant
Sensitivity	-90dBm(min), -95dBm(typ)	Receive Sensitivity
Transfer Rate	250Kbps	IEEE 802.15.4
RF Power	-25dBm ~ 0dBm	Software Configurable
Range	~100m (outdoor), 20~30m (indoor)	Longer range possible with optional SMA antenna attached
Current Draw	Receive mode: 18.8mA	
	Transmit mode: 17.4mA	0 dBm
	Idle Mode: 426μA	Oscillator & Voltage Regulator On
	Power Down mode: 20μA	Voltage Regulator On
	Off mode: 1μA	Voltage Regulator Off
RF Power supply	2.1V ~ 3.6V	CC2420 Input Power
Antenna	Dipole Antenna/ PCB Antenna	
Encryption	Hardware MAC encryption AES-128	
Buffer	128(RX) + 128(TX) data buffering	

Table 2.3.2: CM5000 radio transceiver's specifications [16]

2.3.3 Sensirion SHT11

The SHT11 is a relative humidity and temperature sensor manufactured by Sensirion AG. The sensor integrates sensor elements plus signal processing on a tiny footprint and provides a fully calibrated digital output. The calibration coefficients are stored in the sensor's onboard EEPROM. A unique capacitive sensor element is used for measuring relative humidity while temperature is measured by a band-gap sensor. SHT11 is produced using a CMOS process and is coupled with a 14-bit A/D converter [19] [27].

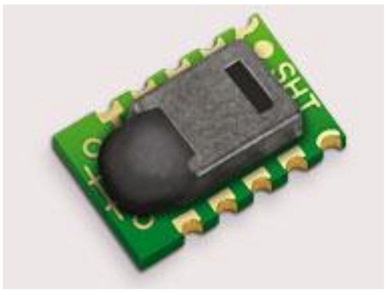


Figure 2.3.6: Sensirion SHT11

Temperature		
Operating Range	-40 - +123.8°C	
Accuracy	± 0.4°C	Typical
Resolution	0.01°C	Typical
Humidity		
Operating Range	0 – 100% RH	
Accuracy	± 3% RH	Typical
Resolution	0.05% RH	Typical

Table 2.3.3: Sensirion SHT11 operational parameters

The maximal accuracy limits for relative humidity and temperature are depicted below:

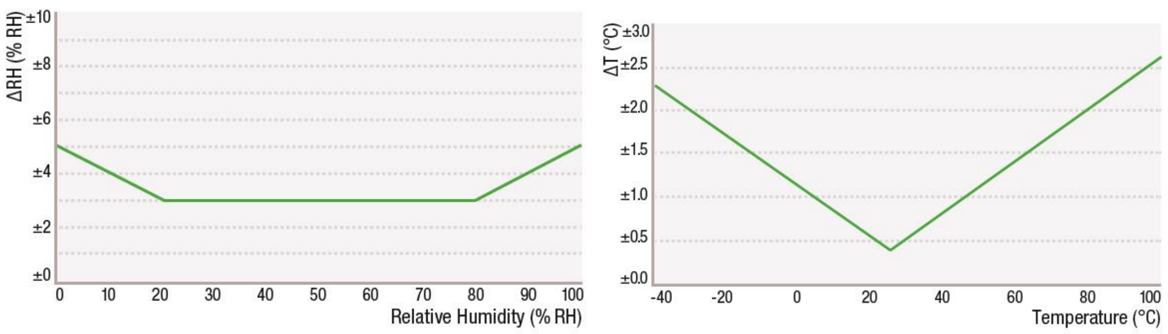


Figure 2.3.7: Maximal RH (left) and Temperature (right) tolerance [19]

The power characteristics can be seen below:

Parameter	Conditions	min	typ	max	Units
Power Supply		2.4	3.3	5.5	V
Supply Current	measuring		0.55	1	mA
	average	2	28		μA
	sleep		0.3	1.5	μA

Table 2.3.4: SHT11 supply characteristics

The average is calculated at one 12 bit measurement per second [19]. In the case of a TelosB mote running on 2 x AA batteries, at a 3V voltage, the current drawn when the sensor is measuring has to be a little less than 0.55mA (measured at 3.3V). We can only make an assumption here, but due to the current value being small and its active duration being equally small, it is safe to assume a 0.5mA current consumption when measuring, without loss in accuracy.

2.3.4 Hamamatsu S1087 & S1087-01

The integrated light sensors S1087 and S1087-01 are ceramic package photodiodes that offer low dark current. The ceramic package used is light-impervious, so no stray-light can reach the photosensitive area from the side or backside. This allows reliable optical measurements in the visible to infrared range, over a wide dynamic range from low light levels to high light levels. The S1087 senses photosynthetically active radiation while the S1087-01 senses the entire visible spectrum including infrared [20].

Figure 2.3.8: Hamamatsu S1087

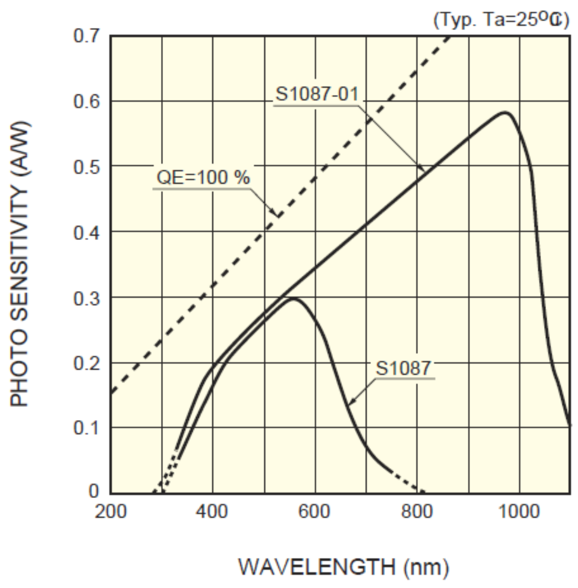


Figure 2.3.9: Spectral Response

The photodiodes are directly connected to the microcontroller's ADC and create a current through a 100kΩ resistor.

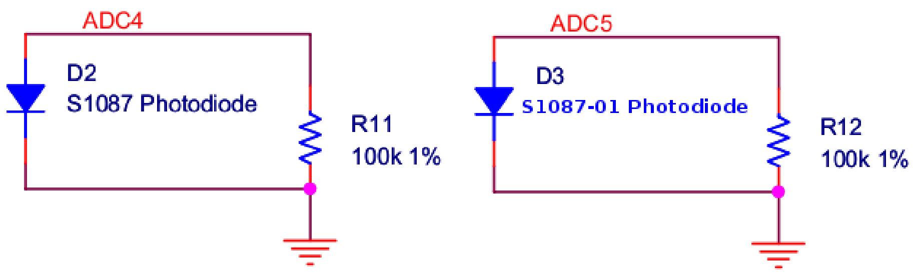


Figure 2.3.10: S1087 (left) & S1087-01 equivalent circuits [27]

S1087 (T = 25°C)		
Spectral Response Range	320 – 730nm	
Peak Sensitivity Wavelength	560 nm	
Dark Current	10pA	
Short Circuit Current	160nA	100lx
S1087-01 (T=25°C)		
Spectral Response Range	320 – 1100nm	
Peak Sensitivity Wavelength	960nm	
Dark Current	10pA	
Short Circuit Current	1.3µA	100lx

Table 2.3.5: S1087 & S1087-01 specifications

Apart from some minimal leakage current, the light sensors consume essentially no power.

2.3.5 External Flash

TelosB uses the ST M25P80 40Mhz serial code flash for external data and code storage. The flash holds 1024kB of data and is decomposed into 16 segments, each 64kB in size. The flash shares SPI communication lines with the CC2420 transceiver, so care must be taken when reading or writing to flash.

ST M25P80 (3V)	
Active Current (Read)	4mA
Active Current (Write/Erase)	20mA
Standby Current	8μA
Deep Power Down Current	1μA

Table 2.3.6: ST M25P80 power states [14]

We can see that using the external flash can take its toll on power consumption. We will not be using the external flash but there is something worth mentioning. The ST M25P series of code flash always start in the standby state. For low power applications, the flash must be sent a command at boot time to place it in the deep power down mode. Fortunately, if using TinyOS, the flash is automatically put into deep power down mode, but during the mote's power analysis we have to include the 1μA current into our calculations nonetheless.

Chapter 3: TinyOS & nesC

3.1 Introduction

This chapter aims to provide an introduction to the software that will be used in this thesis. At first we will give a high-level overview of TinyOS and the nesC language. Then we will go into a level sufficient for writing applications. An overview of the Cooja and MSPSim simulators will follow and the chapter will conclude with a brief presentation of the Yeti 2 plugin for the Eclipse IDE. While this chapter will be more thorough than a plain tutorial, the diversity and complexity of the examined software leaves several topics outside its scope. Should the reader, after reading this chapter, be interested to learn more, there are several sources covering the abovementioned software in greater detail [7] [10].

3.2 TinyOS

TinyOS is a free software and open source software component-based operating system and platform targeting wireless sensor networks. It started as a collaboration between the University of California, Berkeley in cooperation with Intel Research and Crossbow Technology and has since grown to be an international consortium, the TinyOS Alliance.

TinyOS differs from most other operating systems in that its design focuses on ultra-low-power operation. Rather than a fully-fledged processor, TinyOS is designed for the small, low-power microcontrollers, motes have. Furthermore, it has very aggressive systems and mechanisms for saving power. It defines a concurrent execution model, so developers can build applications out of reusable services and components without having much to worry about unforeseen interactions. TinyOS runs on over a dozen generic platforms and its structure makes it reasonably easy to port to new ones.

TinyOS applications and systems, as well as the OS itself, are written in the nesC language. nesC is a C dialect with features to reduce RAM and code size, enable significant optimizations and help prevent low-level bugs like race conditions. At a high level, TinyOS provides three things to make writing systems and applications easier:

- A component model, which defines how to write small, reusable pieces of code and compose them into larger abstractions.
- A concurrent execution model, which defines how components interleave the computations as well as how interrupt and non-interrupt code interact.
- Application programming interfaces (APIs), services, component libraries and an overall component structure that simplify writing new applications and services.

The component model is grounded in nesC. It allows us to write pieces of reusable code which explicitly declare their dependencies. For example a generic user button component that tells us when a button is pressed sits on top of an interrupt handler. The component model allows the button implementation to be independent of which interrupt that is, so that it can be used on many different hardware platforms without requiring complex callbacks or magic function naming conventions. We will examine the basic component model later on in this chapter.

The concurrent execution model enables TinyOS to support many components needing to act at the same time while requiring little RAM. First, every I/O call in TinyOS is split-phase [7]. That means, rather than block until completion, a request returns immediately and the caller gets a callback when the I/O completes. Since the stack isn't tied up waiting for I/O calls to complete, TinyOS only needs one stack and doesn't have threads. Instead, as we will explain later on, TinyOS uses tasks, which are lightweight deferred procedure calls. Any component, can post a task, which TinyOS will run sometime later. Because low-power devices must spend most of their time asleep, they have low CPU utilization and so in practice, tasks tend to run as soon as they are posted (within a few milliseconds). Furthermore, because tasks can't preempt each other, task code doesn't need to worry about data races.

Finally, TinyOS itself has a set of APIs for common functionality, such as sending packets, reading sensors and responding to events. It also provides a component structure and component libraries. For example, Hardware Abstraction Architecture (HAA) defines how to build up from low-level hardware (e.g. a radio chip) to a hardware-independent abstraction (e.g. sending packets). This part lies beyond the scope of this thesis though. As far as the installation is concerned, there are several installation guides available online for all modern operating systems. We will be using TinyOS version 2.1.2, installed on a machine running Ubuntu 13.04. Both of these versions are the latest as of July 2013.

3.3 nesC

nesC (network embedded systems C), pronounced "NES-see", is a component-based, event-driven programming language used to build applications for the TinyOS platform. Program structure is the most essential and obvious difference between C and nesC. C programs are composed of variables, types and functions defined in files that are compiled separately and then linked together. nesC programs are built out of components that are connected ("wired") by explicit program statements; the nesC compiler connects and compiles these components as a single unit. The nesC compiler loads and reads in nesC components, which it compiles to a C file. This C file is passed to a native C compiler, which generates a mote binary [7].

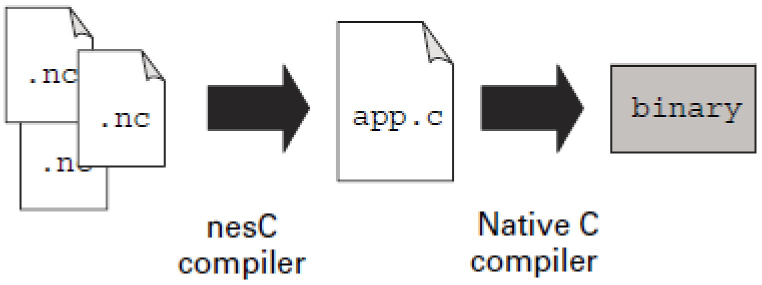


Figure 3.3.1: The nesC compilation model. The nesC compiler loads and reads in nesC components, which it compiles to a C file. This C file is passed to a native C compiler, which generates a mote binary.

3.3.1 Components and Interfaces

Whereas C programs are composed of functions, nesC programs are built out of components that implement a particular service (e.g. change the state of an LED). Furthermore, C functions typically interact by calling each other directly, while the interactions between components are specified by interfaces. Components define two scopes: one for their specification which contains the names of their interfaces, and a second scope for their implementation. A component provides and uses interfaces. The provided interfaces are intended to represent the functionality that the component provides to its user in its specification. The used interfaces represent the functionality the component needs, to perform its job in its implementation.

Interfaces are bidirectional: they specify a set of commands, which are functions to be implemented by the interface's provider, and a set of events, which are functions to be implemented by the interface's user. In other words, the interface's user makes requests (calls commands) on the interface's provider and the provider makes callbacks (signals events) to the interface's user. Commands and events themselves are like regular functions (they can contain arbitrary C code); calling a command or signaling an event is just a function call. For a component to call the commands in an interface, it must implement the events of that interface. A single component may use or provide multiple interfaces and multiple instances of the same interface. The set of interfaces which a component provides, together with the set of interfaces that a component uses is considered that component's signature [10].

There are two types of components in nesC: modules and configurations. Modules provide the implementations of one or more interfaces. Configurations are used to assemble other components together, connecting interfaces used by components to interfaces provided by others. Every nesC application is described by a top-level configuration that wires together the components inside.

3.3.2 An Example Application

Let's try to clear things up with a very basic example application: Example. We will enrich it with more elements step by step. This application turns on an LED as soon as the mote powers up. It is composed of two components: a module, called “ExampleC.nc” and a configuration, called “ExampleAppC.nc”. Remember that all applications require a top-level configuration file, which is typically named after the application itself. In this case, ExampleAppC is the configuration file for the Example application and the source file that the nesC compiler uses to generate an executable file. ExampleC, on the other hand, actually provides the implementation of the Example application. Or, to put it simply, ExampleAppC is used to wire (using interfaces) the ExampleC component (module) to other components that the Example application requires.

The reason for the distinction between modules and configurations is to allow a system designer to build applications out of existing implementations. For example, a designer could provide a configuration that simply wires together one or more modules, none of which they actually designed. Likewise, another developer can provide a new set of library modules that can be used in a range of applications. It should also be mentioned that while one could name an application’s implementation module and associated top-level configuration anything, to keep things simple, it is common naming convention to name the module file after the application name, ending with the letter C, and the configuration file ending with the letters AppC. If there are more than one modules or configuration files, there are several conventions used in TinyOS specified in TinyOS Enhancement Proposal (TEP) 3 [9].

```
module ExampleC {
   uses interface Boot;
   uses interface Leds;
}
implementation {
   event void Boot.booted() {
      call Leds.led1On();
   }
}
```

```
configuration ExampleAppC {
}
implementation {
   components MainC;
   components ExampleC;
   components LedsC;

   ExampleC.Boot -> MainC.Boot;
   ExampleC.Leds -> LedsC.Leds;
}
```

The nesC compiler compiles a nesC application when given the file containing the top-level configuration. Let’s start with that. The first thing to notice is the keyword configuration, which indicates that this is a configuration file. Within the first pair of empty braces, it is possible to specify uses and provide clauses or as we defined earlier, the component’s signature. There is no need to do that in this configuration. A configuration can use and provide interfaces, or said another way, not all configurations are top-level applications.

The actual configuration is described within the pair of curly brackets following the keyword *implementation*. The components lines specify the set of components that this configuration references. In this case those components are MainC, ExampleC and LedsC. There should be no confusion here; the ExampleAppC component is not the same as the ExampleC component. Rather, the ExampleAppC component (configuration) is composed of the ExampleC component (module) along with MainC and LedsC.

The remainder of the ExampleAppC configuration consists of connecting (wiring) interfaces used by components to interfaces provided by others. An interface is denoted by the form Component.Interface. The last two lines wire interfaces that the ExampleC component uses to interfaces that the MainC and LedsC components provide. The MainC.Boot interface is part of TinyOS's boot sequence and enables the mote to be initialized. The LedsC.Leds interface gives the user control over the mote's LEDs. nesC uses arrows to bind interfaces to one another. The right arrow A -> B means "A wires to B". The left side of the arrow (A) is a user of the interface, while the right side of the arrow (B) is the provider. A full wiring is A.a -> B.b which means that the interface *a* of component *A* is wired to interface *b* of component *B*. Naming the interface is important when a component uses or provides multiple instances of the same interface as we will see later on. When a component only has one instance of an interface, we can elide the interface name. For example, the interface name Leds doesn't have to be included in LedsC:

```
ExampleC.Leds -> LedsC;  // Same as  ExampleC.Leds -> LedsC.Leds
```

Because ExampleC only uses one instance of the Leds interface, this line should also work:

```
ExampleC -> LedsC.Leds;  // Same as  ExampleC.Leds -> LedsC.Leds
```

The direction of a wiring arrow is always from a user to a provider. If the provider is on the left side, we can use a left arrow. For ease of reading, however, most wirings are left to right. To sum up, the ExampleC.Leds -> LedsC.Leds line wires the Leds interface used by the ExampleC component to the Leds interface provided by the LedsC component. The ExampleC.Boot interface is wired accordingly.

3.3.3 Commands and Events

If we take a look at the ExampleC module's signature, we can see that it uses the Leds and Boot interfaces. This means that ExampleC may call any command declared in the interfaces it uses and must also implement any events declared in those interfaces. Let's take a look at those interfaces:

```
interface Boot {
   event void booted();
}
```

```
interface Leds {
// Turn LED n on, off, or toggle its present state.
async command void led0On();
async command void led0Off();
async command void led0Toggle();

async command void led1On();
async command void led1Off();
async command void led1Toggle();
async command void led2On();
async command void led2Off();
async command void led2Toggle();

/* Get/Set the current LED settings as a bitmask. Each bit
corresponds to whether an LED is on; bit 0 is LED 0, bit 1 is LED
1, etc.
*/
async command uint8_t get();
async command void set(uint8_t val);

}
```

The first thing to notice is that the Leds interface does not include any events, so ExampleC doesn't need to implement any in order to call the Leds commands. Additionally, ExampleC must implement a handler for the Boot.booted() event. So here is what the last two lines of the ExampleC code do: when the mote boots, or better, when the event Boot.booted() occurs, or even better, when the component that provides the Boot interface (MainC), makes a callback (signals the booted() event) to the component that uses the Boot interface (ExampleC), then the LED1 of the mote is being turned on, or better, the command Leds.led1On is called, or even better, the component that uses the Leds interface (ExampleC), makes a request (calls the led1On command) to the component that provides the Leds interface (LedsC). The keywords event and call help us understand, where an event is signaled or a command is called, in our code.

Now let's make our application a bit more interesting. We will modify it so that when the mote boots up, LED0 and LED1 will start flashing periodically with a period of 1 and 2 seconds respectively:

```
module ExampleC {
   uses {
      interface Boot;
      interface Leds;
      interface Timer<TMilli> as Timer1;
      interface Timer<TMilli> as Timer2;
   }
}
```

```
implementation {
   event void Boot.booted() {
      call Timer1.startPeriodic(512);
      call Timer2.startPeriodic(1024);
   }

   event void Timer1.fired() {
      call Leds.led0Toggle();
   }
   event void Timer2.fired() {
      call Leds.led1Toggle();
   }
}
```

```
configuration ExampleAppC {
}
implementation {
   components MainC, LedsC;
   components ExampleC as App;
   components new TimerMilliC() as TimerA;
   components new TimerMilliC() as TimerB;

   App.Boot -> MainC;
   App.Leds -> LedsC;
   App.Timer1 -> TimerA;
   App.Timer2 -> TimerB;
}
```

Starting with the configuration we can see a couple of changes. First of all, the components MainC and LedsC are declared with the same *components* keyword, separated by a comma. That's practical for reducing the number of lines of code. Then we see the declaration of two instances of a timer component called TimerMilliC which will be referenced as TimerA and TimerB. This is accomplished via the *as* keyword which denotes simply an alias. We also created an alias of the ExampleC module, called App and we used it to do the wiring a few lines below. In general, the *as* keyword can be used both for components and interfaces and makes the signature a bit clearer to the reader by using appropriately named aliases. In the case of ExampleC, it was optional, but in the case of TimerMilliC it was mandatory as we instantiated it twice.

Before checking the wiring, we should take a look at the ExampleC module. We used a set of brackets after the *uses* keyword, and declared all the interfaces inside. That helps up to avoid the repeated typing of the uses keyword. The ExampleC module uses two instances of the interface Timer<TMilli>, provided by the TimerMilliC component, using the names Timer1 and Timer2. The <TMilli> syntax simply denotes that Timer is a generic interface, that is, it takes a single type as a parameter which defines what type of timer it is. This one in particular is a timer that takes its parameter expressed in milliseconds. Since the

ExampleC module uses the Timer interface more than once, its signature must use the as keyword.

Back to the wiring, it is obvious that we elided the interface names on the right side of the arrow. As the TimerMilliC components each provide a single instance of Timer, it does not have to be included in the wirings. However, as ExampleC has two instances of Timer, eliding the name on the user side (left) would be a compile-time error, as the compiler would not know which instance of Timer is being wired. Looking over the ExampleC's implementation, we can see that since ExampleC uses the Timer interface, it must implement handlers for the Timer.fired() event. So what happens in this application is this: when the mote is powered up, two periodic timers are being initiated: Timer1 with a period of 0.5sec and Timer2 with a period of 1 sec. The periods are the numbers passed on as parameters to the startPeriodic command (1 sec = 1024ms) [7]. When Timer1 fires (i.e. every 0.5sec) the state of LED0 is toggled from off to on and vice versa. That means, LED0 blinks every 1 second and similarly, LED1 blinks every 2 sec.

3.3.4 Tasks and Split-Phase Operations

All of the code we've looked at so far is synchronous. It runs in a single execution context and does not have any kind of preemption. That is, when synchronous code starts running, it does not relinquish the CPU to other code until it completes. This simple mechanism allows the TinyOS scheduler to minimize its RAM consumption and keeps sync code very simple. However, it means that if one piece of sync code runs for a long time, it prevents other sync code from running, which can adversely affect system responsiveness. For example, a long running piece of code can increase the time it takes for a mote to respond to a packet.

So far, the code we have seen uses direct function calls. System components such as the boot sequence or timers, signal events to a component, which takes some action (perhaps calling a command) and returns. In most cases, this programming approach works well. Because sync code is non-preemptive, however, this approach does not work well for large computations. A component needs to be able to split a large computation into smaller parts, which can be executed one at a time. Also, there are times when a component needs to do something, but it is fine to do it a little later. Giving TinyOS the ability to defer the computation until later can let it deal with everything else that's waiting first.

Tasks enable components to perform general-purpose "background" processing in an application. A task is a function which a component tells TinyOS to run later, rather than now, and is declared in the implementation module using the syntax:

```
task void taskname() { ... }
```

where taskname( ) is whatever symbolic name we want to assign to the task. Tasks must return void and may not take any arguments. To dispatch a task for (possibly later) execution, we can use the syntax:

```
post taskname();
```

A component can post a task in a command, an event, or even another task. The post operation places the task in an internal task queue which is processed in FIFO order. When a task is executed, it runs to completion before the next task is run. Therefore, a task should not run for long periods of time. Tasks do not preempt each other, but a task can be preempted by hardware interrupts. If one needs to run a series of long operations, they should dispatch a separate task for each operation, rather than using one big task. The post operation returns an error_t, whose value is either SUCCESS or FAIL. A post fails if and only if the task is already pending to run (it has been posted successfully and has not been invoked yet).

Another worth-mentioning characteristic of nesC is Split-Phase Operations. Because nesC interfaces are wired at compile time, callbacks (events) in TinyOS are very efficient. In C, and in most C-like languages, callbacks have to be registered at run-time with a function pointer. This can prevent the compiler from being able to optimize code across callback call paths. Since they are wired statically in nesC, the compiler knows exactly what functions are called where and can optimize heavily.

The ability to optimize across component boundaries is very important in TinyOS, because it has no blocking operations. Instead, every long-running operation is split-phase. In a blocking system, when a program calls a long-running operation, the call does not return until the operation is complete. The program therefore, blocks. In a split-phase system, when a program calls a long-running operation, the call returns immediately and the called abstraction issues a callback when it completes. This approach is called split-phase because it splits invocation and completion into two separate phases of execution. Here is a simple example of the difference between the two:

Blocking	Split-Phase

If (send( ) == SUCCESS) {
  sendCount++;
}
	send( );
	
	void sendDone(error_t val) {
	  if (val == SUCCESS) {
	    sendCount++;
	  }
	}

Split-phase code is often a bit more complex than sequential code. But it has several advantages. First, split-phase calls do not tie up stack memory while they are executing.

Second, they keep the system responsive; there is never a situation when an application needs to take an action but all of its threads are tied up in blocking calls. Third, it tends to reduce stack utilization, as creating large variables on the stack is rarely necessary.

The command Timer.startOneShot is an example of a split-phased call. This command starts a timer that will fire only once, sometime in the future. The user of the Timer interface calls the command which returns immediately. Sometime later (specified by the argument), the component providing Timer, signals the Timer.fired event. To execute the same code in a system with blocking calls, a program might use sleep instead.

Blocking	Split-Phase

state = WAITING;
operation1( );
sleep(512);
operation2( );
state = RUNNING;
	state = WAITING;
	operation1( );
	call Timer.startOneShot(512);
	
	event void Timer.fired() {
	   operation2( );
	   state = RUNNING;
	}

We will not be especially mentioning them, but tasks and split-phase operations are going to be present throughout our code. This section’s goal was to get the reader accustomed to these concepts.

3.3.5 Radio Communication

Radio communication between motes is the fundamental and most important aspect of our application. Although we are going to examine the radio operation of our motes in the next chapter, we think it is necessary to make an introduction here, due to the importance of it.

TinyOS provides a number of interfaces to abstract the underlying communications services and a number of components that provide these interfaces. All of these interfaces and components use a common message buffer abstraction, called message_t, which is implemented as a nesC struct (similar to a C struct). The message_t struct is defined as [7]:

```
typedef nx_struct message_t {
   nx_uint8_t header[sizeof(message_header_t)];
   nx_uint8_t data[TOSH_DATA_LENGTH];
   nx_uint8_t footer[sizeof(message_footer_t)];
   nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
```

Before examining the message_t struct, we should take a look at the variable types used in nesC. Rather than the standard C names of int, long or char, TinyOS code uses more explicit types, which declare their size. In reality, these map to the basic C types, but do so differently for diferent platforms. TinyOS code avoids using int for example because it is platform specific. For example, on Mica and TelosB motes, int is 16 bytes long, while on the IntelMote2, it is 32 bits. Additionally, TinyOS code often uses unsigned values heavily, as wraparounds to negative numbers can often lead to very unintended consequences. The commonly used types are summarized below:

	8 bits	16 bits	32 bits	64 bits
Signed	int8_t	int16_t	int32_t	int64_t
Unsigned	uint8_t	uint16_t	uint32_t	uint64_t

Table 3.3.1: Commonly used types in nesC [10]

There is also a bool type. We can use the standard C types, but doing so might raise cross-platform issues. Most platforms support floating point numbers (float almost always, double sometimes).

Returning to the message_t struct, we see that it is composed of 4 fields. The nx_ prefix is specific to the nesC language and denotes a network type. Network types have the same representation on all platforms. The nesC compiler generates code that transparently reorders access to nx_ data types and eliminates the need to manually adjust endianness and alignment (e.g. extra padding in structs present on some platforms). The header, footer and metadata fields are all opaque and must not be accessed directly. It is important to access the message_t fields only through Packet, AMPacket and other such interfaces, as will be shown. There are a number of interfaces and components that use message_t as the underlying data structure. Let’s take a look at some of those, to familiarize ourselves with the general functionality of the communications system:

- Packet: provides the basic accessors for the message_t abstract data type. This interface provides commands for clearing a message’s contents, getting its payload length and getting a pointer to its payload area.
- Send: provides the basic address-free message sending interface. This interface provides commands for sending a message and canceling a pending message send. The interface provides an event to indicate whether a message was sent successfully or not. It also provides convenience functions for getting the message’s maximum payload as well as a pointer to the message’s payload area.
- Receive: provides the basic message reception interface. This interface provides an event for receiving messages. It also provides, for convenience, commands for getting a message’s payload length and getting a pointer to the message’s payload area.

Since it is very common to have multiple services using the same radio to communicate, TinyOS provides the Active Message (AM) layer to multiplex access to the radio. The term “AM type” refers to the field used for multiplexing. AM packets also include a destination field, which stores an “AM address” to address packets to particular motes. Additional interfaces were introduced to support the AM services:

- AMPacket: similar to Packet, it provides the basic AM accessors for the message_t abstract data type. This interface provides commands for getting a node’s AM address, an AM packet’s destination, and an AM packet’s type. Commands are also provided for setting an AM packet’s destination and type, and checking whether the destination is the local node.
- AMSend: similar to Send, it providesthe basic Active Message sending interface. The key difference between AMSend and Send is that AMSend takes a destination AM address in its send command.

Let’s suppose that we want to create and send a message over the radio, and that our message’s payload is composed of two fields of data. Rather than directly writing and reading the payload area of the message_t with this data, we will use a structure to hold them and then use structure assignment to copy the data into the message payload area. Using a structure allows reading and writing the message payload much more conveniently when our message has multiple fields or multi-byte fields, like uint16_t or greater, because we can avoid reading and writing bytes from/to the payload area using e.g. indices and then shifting and adding. Even for a message with a single field, a designer should get used to using that structure because if they ever add more fields to the message or move any of the fields around, they will need to manually update all of the payload position indices if they read and write the payload at a byte level. The following defines a message structure with a uint16_t data1 field and a uint32_t data2 field in the payload:

```
typedef nx_struct ExampleRadioMsg {
   nx_uint16_t data1;
   nx_uint32_t data2;
} ExampleRadioMsg_t;
```

Instead of rewriting our Example application, we will walk through the steps necessary to send the message over the radio, mentioning the code lines that need to be added in each step. We will use the AMSend interface to send packets as well as the Packet interface to access the message_t abstract data type. We also need an interface to start the radio, so we will use the SplitControl interface, provided by the ActiveMessageC component. Our module’s (ExampleC) signature is modified like this:

```
module ExampleC {
   ...
   uses interface Packet;
   uses interface AMSend;
   uses interface SplitControl as RadioControl;
}
```

Note that the SplitControl interface has been renamed to RadioControl. SplitControl is a general interface used for starting and stopping components, but creating an alias with the name RadioControl is a good way to remind us that this particular instance of SplitControl is used to control the radio, or in other words the ActiveMessageC component.

We need a message_t to hold our data for transmission. The declaration needs to be added in the implementation block of ExampleC:

```
implementation {
   message_t pkt;
   ...
}
```

Next we need to handle the initialization of the radio. It is our choice to start the radio when the system boots so we must call the RadioControl.start command, inside the Boot.booted event. Now there is something worth mentioning. We plan to send the message over the radio every time Timer2 fires (i.e. every 1 sec) but the radio can't be used until it has completed starting up. Due to the split-phase nature of TinyOS, when the program calls the RadioControl.start command, the call returns immediately, but the radio signals that it has completed starting through the RadioControl.startDone event. To ensure that we don't start using the radio before it is ready, we need to postpone starting Timer2 until after the radio has completed starting. That means moving the call to start Timer2, which is now inside the Boot.booted event, to RadioControl.startDone. So the Boot.booted event looks like this:

```
   event void Boot.booted() {
      call Timer1.startPeriodic(512);
      call RadioControl.start();
   }
```

Inside the module's implementation we also have to implement the RadioControl.startDone and RadioControl.stopDone event handlers, which have the following bodies:

```
event void RadioControl.startDone(error_t err) {
   if (err == SUCCESS) {
      call Timer2.startPeriodic(1024);
   }
   else {
      call RadioControl.start();
   }
}

event void RadioControl.stopDone(error_t err) {
}
```

If the radio is started successfully, RadioControl.startDone will be called with the error_t parameter set to a value of SUCCESS. Then it is appropriate to start the timer. If, however, the radio does not start successfully, then it obviously cannot be used so we try again to start it. This process continues until the radio starts, and ensures that the node software doesn't run until the key components have started successfully. If the radio doesn't start at all, a human operator might notice that the LEDs are not blinking as they are supposed to, and might try to debug the problem. For simplicity reasons, we will not use the RadioControl.stop command, so the stopDone event is never going to happen, that's why there is no code in this block.

Since we want to transmit our message every time Timer2 fires, we need to add some code to the Timer2.fired event handler:

```
event void Timer0.fired() {
    ...
    ExampleRadioMsg_t* exmplpkt = (ExampleRadioMsg_t*) (call
    Packet.getPayload(&pkt,sizeof(ExampleRadioMsg_t)));
    exmplpkt->data1 = DATA;
    exmplpkt->data2 = DATA;
    call AMSend.send(AM_BROADCAST_ADDR, &pkt,
    sizeof(ExampleRadioMsg_t);
}
```

This code gets the packet's payload portion and casts it to a pointer to the previously declared ExampleRadioMsg type. It can now use this pointer to initialize the packet's fields, and then send the packet by calling the AMSend.send command. The packet is sent to all nodes in range by specifying AM_BROADCAST_ADDR as the destination address.

There is one more event we need to worry about, AMSend.sendDone. This event is signaled after a message transmission attempt. We'll toggle LED2 if the transmission was successful:

```
event void AMSend.sendDone (message_t* msg, error_t err) {
    if (err == SUCCESS) {
        call Leds.Led2Toggle();
    }
}
```

Returning to the configuration ExampleAppC, we have to declare new components and wire the provided with the used interfaces. ActiveMessageC is a singleton component that is defined once for each type of hardware platform. AMSenderC is a generic parameterized component. The new keyword indicates that a new instance of AMSenderC will be created. The AM_RADIO parameter indicates the AM type of the AMSenderC. We can define this parameter along with others, in the accompanying header file as we will see in the next chapter. The implementation block of the ExampleAppC configuration file now looks like this:

```
implementation {
   ...
   components ActiveMessageC;
   components new AMSenderC(AM_RADIO);
   ...
   App.Packet -> AMSenderC;
   App.AMSend -> AMSenderC;
   App.RadioControl -> ActiveMessageC;
}
```

Receiving a message over the radio works similarly. This, along with other parameters of the radio operation will be examined in the next chapter.

3.4 Cooja

Cooja is a java-based simulator initially developed for simulations of sensor nodes running the Contiki operating system, but now able to simulate TinyOS motes as well [1]. Cooja simulates networks of sensor nodes where each node can be of a different type, differing not only in on-board software, but also in the simulated hardware. Cooja is flexible in that many parts of the simulator can be easily replaced or extended with additional functionality.

A simulated node in Cooja has three basic properties: its data memory, the node type and its hardware peripherals. The node type may be shared between several nodes and determines properties common to all these nodes. For example, nodes of the same type run the same program code on the same simulated hardware peripherals. Nodes of the same type are initialized with the same data memory, except for the node ID. During execution however, the data memories of the nodes will eventually differ after reacting to external stimuli.

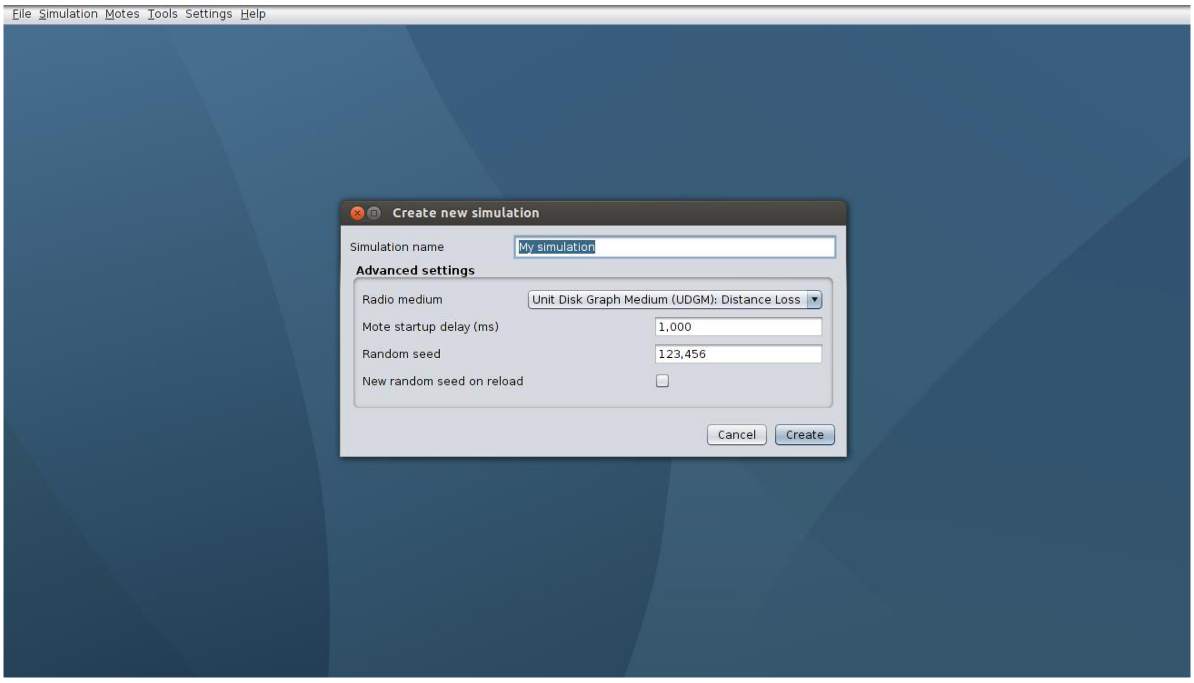
By clicking on File -> New Simulation the new simulation wizard starts up. Here the user can adjust some basic simulation settings, namely the preferred radio medium which determines the radio surrounding behavior, the mote startup delay which is the time difference between the startup of the first and last mote, and the random seed which controls the random behavior such as various delays, node positions etc.

The wireless messages can be sent on different radio mediums; the simulator proposes four wireless channels that are: No Radio Traffic, Unit Disk Graph Medium (UDGM) – Constant Loss, Unit Disk Graph Medium (UDGM) – Distance Loss, and Directed Graph Radio Medium (DGRM) [1].

- No Radio Traffic does not permit the radio communication on the channel and therefore cannot be employed to simulate WSNs.
- UDGM – Constant Loss is a wireless channel model where the transmission range is modelled as an ideal disc where all nodes outside of it do not receive packets, while those within receive all messages. The predefined maximum transmission range is

multiplied by the ratio of the current output power to the maximum output power of the simulated device and the resulting transmission power is compared to the distance in the simulation. For example, if the transmission range of the mote is 200m and the current output power is half of the maximum, the disc has a radius of 100m.

- UDGM – Distance Loss is a radio medium similar to the previous one but it extends it in two ways. First, the interferences are now considered and, in case of interfered packets, they are lost due to the interference range which is larger than the transmission range. Second, the success ratio of the transmission and reception can be set: a packet is transmitted or received on the basis of two probabilities, SUCCESS_RATIO_TX (if unsuccessful, no device receives the packet) and SUCCESS_RATIO_RX (if unsuccessful, only the destination of the packet does not receive it).
- DGRM is a model that creates the topology of nodes through edges. It lets the programmer fully customize the mote-to-mote relations.



Simulation name: My simulation

Advanced settings

Radio medium	Unit Disk Graph Medium (UDGM): Distance Loss
Mote startup delay (ms)	1,000
Random seed	123,456
New random seed on reload	☐

Figure 3.4.1: New Simulation wizard

The configuration of Cooja is flexible so that many parts of the simulator can be replaced or extended with additional functionality. Example parts are the radio mediums just described, the interfaces and plugins. The interfaces represent some properties of the node such as the position, the serial port and the user button state. The plugins are used to interact with a simulation. They often provide the user with a graphical interface to observe something of interest in the simulation.

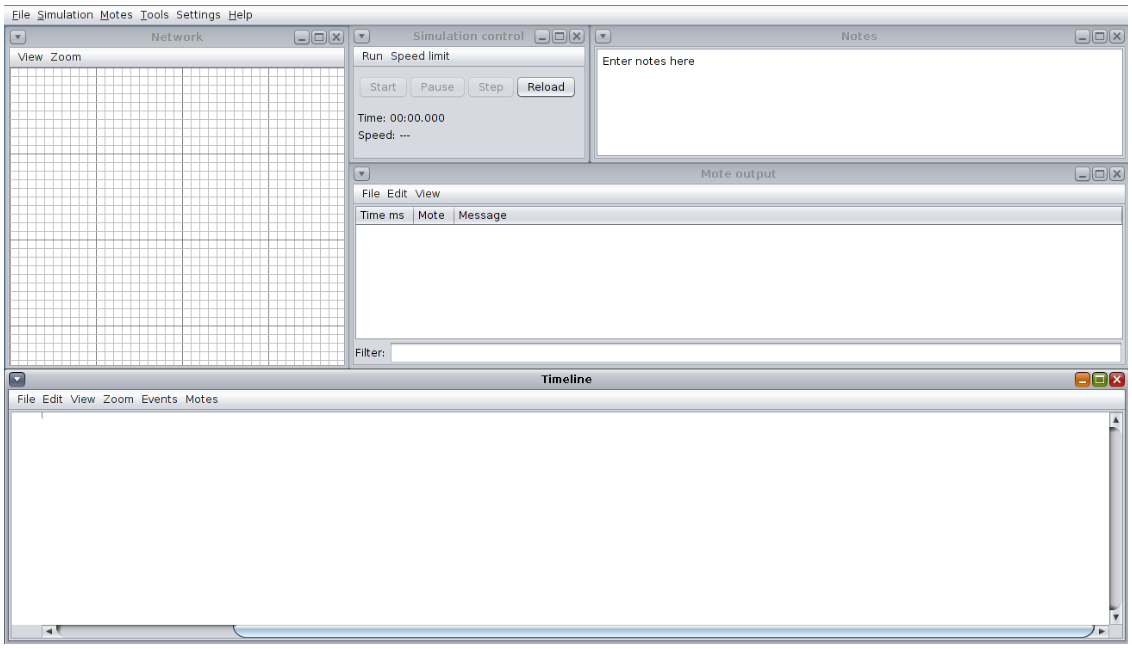
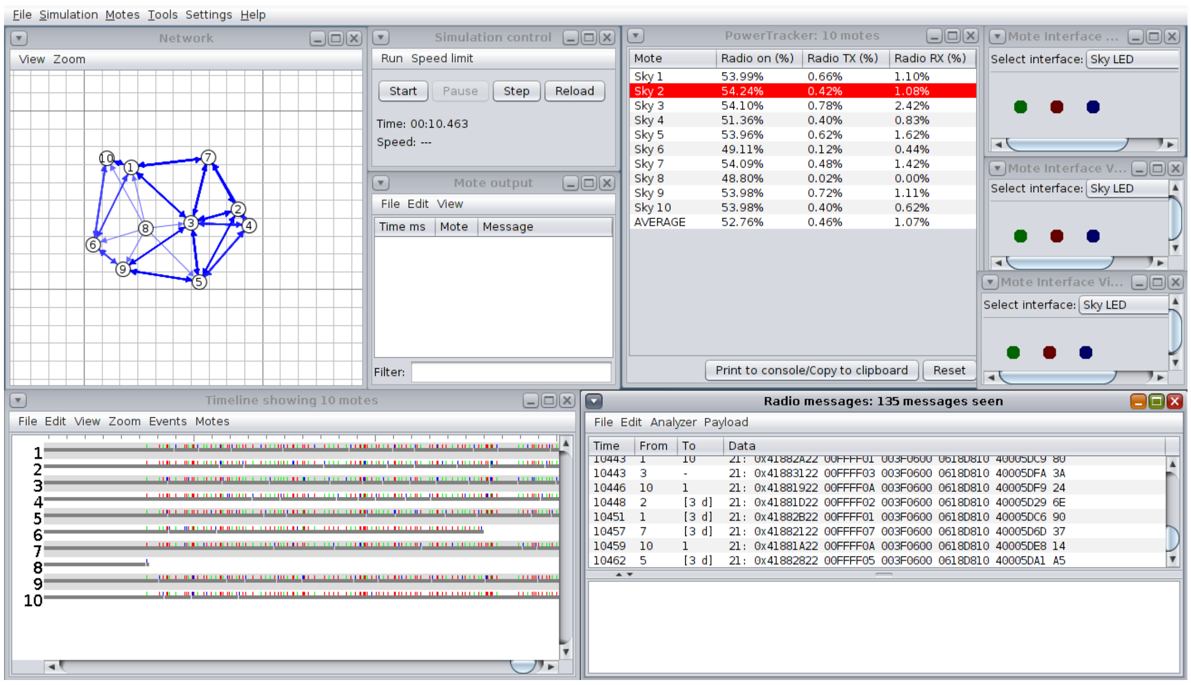


Figure 3.4.2: Default plugins

The default plugins are the Network Visualizer, the Timeline, the Mote Output, the Simulation Control and the Notes. The Network Visualizer simply lets us configure the network's topology. The user can drag and drop the nodes, change the transmission and interference range and show mote information such as LED states, position, mote IDs etc. The Timeline displays the radio state for each mote through different colors: on (grey), off (no color), packet transmission (blue), packet reception (green) and interference (red). It also displays the LED state of each mote. The Mote Output displays the log output for all simulated motes. The Simulation Control controls starts, pauses, stops and reloads the simulation, and changes the simulation speed.


File Simulation Motes Tools Settings Help
Network
View Zoom
10
1
7
2
3
4
8
6
9
5
Simulation control
Run Speed limit
Start Pause Step Reload
Time: 00:10.463
Speed: ---
Mote output
File Edit View
Time ms Mote Message
Filter:
PowerTracker: 10 motes
Print to console/Copy to clipboard
Reset
Mote Interface ...
Select interface: Sky LED
Mote Interface V...
Select interface: Sky LED
Mote Interface Vi...
Select interface: Sky LED
Timeline showing 10 motes
File Edit View Zoom Events Motes
1
2
3
4
5
6
7
8
9
10
Radio messages: 135 messages seen
File Edit Analyzer Payload

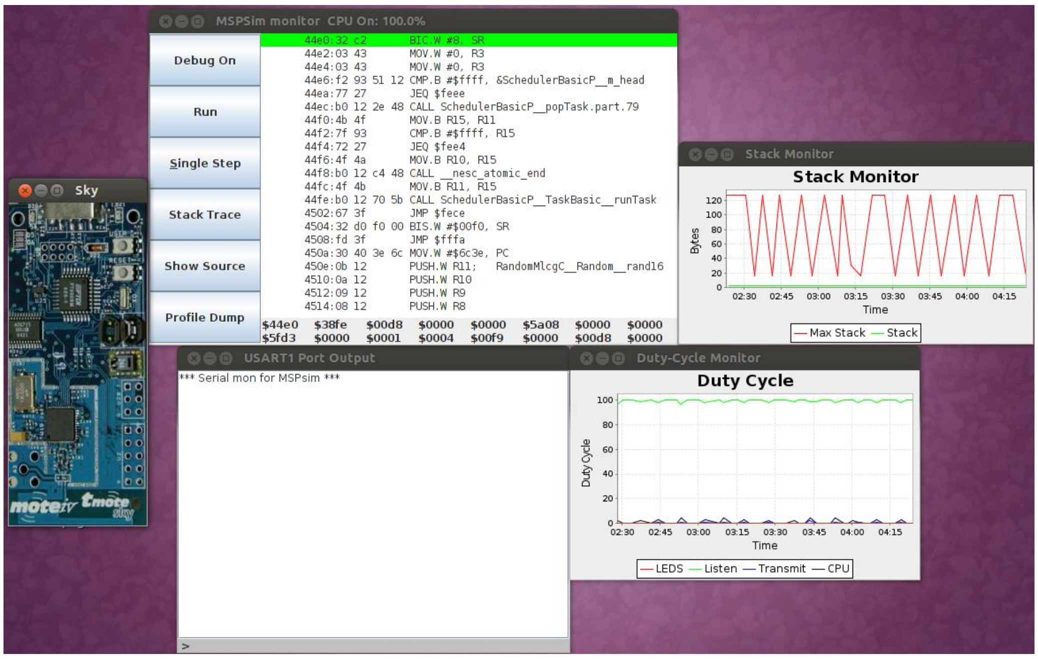
Mote	Radio on (%)	Radio TX (%)	Radio RX (%)
Sky 1	53.99%	0.66%	1.10%
Sky 2	54.24%	0.42%	1.08%
Sky 3	54.10%	0.78%	2.42%
Sky 4	51.36%	0.40%	0.83%
Sky 5	53.96%	0.62%	1.62%
Sky 6	49.11%	0.12%	0.44%
Sky 7	54.09%	0.48%	1.42%
Sky 8	48.80%	0.02%	0.00%
Sky 9	53.98%	0.72%	1.11%
Sky 10	53.98%	0.40%	0.62%
AVERAGE	52.76%	0.46%	1.07%

```
Time   From  To     Data
10443  1     10     21: 0x41882A22 00FFFF01 003F0600 0618D810 40005DC9 80
10443  3     -      21: 0x41883122 00FFFF03 003F0600 0618D810 40005DFA 3A
10446  10    1      21: 0x41881922 00FFFF0A 003F0600 0618D810 40005DF9 24
10448  2     [3 d]  21: 0x41881D22 00FFFF02 003F0600 0618D810 40005D29 6E
10451  1     [3 d]  21: 0x41882B22 00FFFF01 003F0600 0618D810 40005DC6 90
10457  7     [3 d]  21: 0x41882122 00FFFF07 003F0600 0618D810 40005D6D 37
10459  10    1      21: 0x41881A22 00FFFF0A 003F0600 0618D810 40005DE8 14
10462  5     [3 d]  21: 0x41882822 00FFFF05 003F0600 0618D810 40005DA1 A5
```

Figure 3.4.3: Extended plugins

We will also be using the Radio Messages and PowerTracker plugins. The Radio Messages plugin displays all radio messages exchanged between motes along with the contents of the message. The PowerTracker is extremely useful and is essentially a mote radio duty cycle, showing a list of all motes, along with Radio On, Radio RX and Radio TX percentages.

3.5 MSPSim

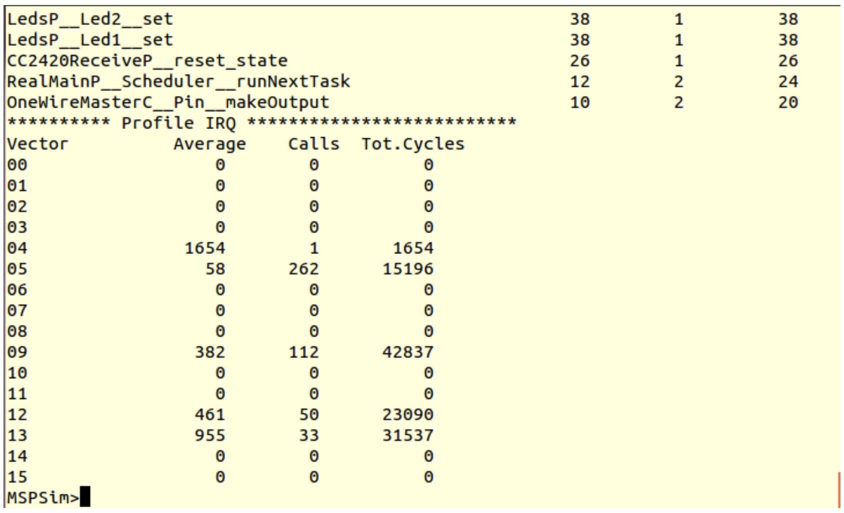


```
44e0:32 c2        BIC.W #8, SR
44e2:03 43        MOV.W #0, R3
44e4:03 43        MOV.W #0, R3
44e6:f2 93 51 12 CMP.B #$ffff, &SchedulerBasicP__m_head
44ea:77 27        JEQ $feee
44ec:b0 12 2e 48 CALL SchedulerBasicP__popTask.part.79
44f0:4b 4f        MOV.B R15, R11
44f2:7f 93        CMP.B #$ffff, R15
44f4:72 27        JEQ $fee4
44f6:4f 4a        MOV.B R10, R15
44f8:b0 12 c4 48 CALL __nesc_atomic_end
44fc:4f 4b        MOV.B R11, R15
44fe:b0 12 70 5b CALL SchedulerBasicP__TaskBasic__runTask
4502:67 3f        JMP $fece
4504:32 d0 f0 00 BIS.W #$00f0, SR
4508:fd 3f        JMP $fffa
450a:30 40 3e 6c MOV.W #$6c3e, PC
450e:0b 12        PUSH.W R11;   RandomMlcgC__Random__rand16
4510:0a 12        PUSH.W R10
4512:09 12        PUSH.W R9
4514:08 12        PUSH.W R8
$44e0  $38fe  $00d8  $0000  $0000  $5a08  $0000  $0000
$5fd3  $0000  $0001  $0004  $00f9  $0000  $00d8  $0000
```

```
*** Serial mon for MSPsim ***
```

Figure 3.5.1: Standalone MSPsim running

MSPSim [5] is a cycle-accurate Java based simulator of the MSP430 microcontroller. It is able to simulate all motes that embed an MSP430 MCU. It can be run from a terminal or from within Cooja by right clicking a mote and choosing MSP Cli. During startup MSPSim is composed of five windows showing a picture of the mote, the duty cycle of various components, the serial output, a stack monitor and a main control window.



```
LedsP__Led2__set                                        38        1        38
LedsP__Led1__set                                        38        1        38
CC2420ReceiveP__reset_state                             26        1        26
RealMainP__Scheduler__runNextTask                       12        2        24
OneWireMasterC__Pin__makeOutput                         10        2        20
********** Profile IRQ **************************
Vector         Average    Calls  Tot.Cycles
00                   0        0           0
01                   0        0           0
02                   0        0           0
03                   0        0           0
04                1654        1        1654
05                  58      262       15196
06                   0        0           0
07                   0        0           0
08                   0        0           0
09                 382      112       42837
10                   0        0           0
11                   0        0           0
12                 461       50       23090
13                 955       33       31537
14                   0        0           0
15                   0        0           0
MSPSim>
```

Figure 3.5.2: Output of the Profile command in a terminal

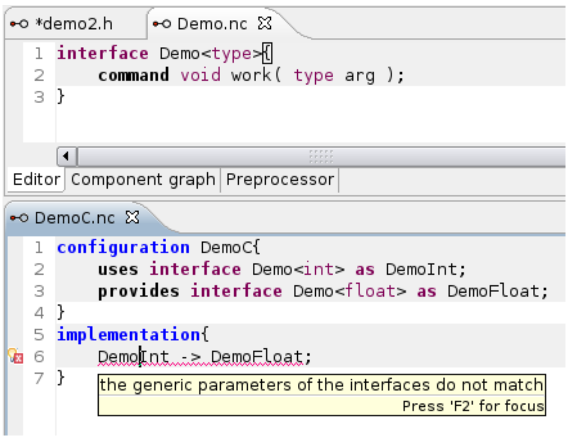
Several commands can be given in the terminal as well, giving the user the opportunity to observe the duty cycle of the radio and the MCU (in numbers), several variables during program execution and information on various mote components. When run from within Cooja, a single window appears which essentially works like a terminal.

3.6 Yeti 2 plugin for Eclipse IDE

The TinyOS 2.x Plugin for Eclipse, nicknamed “Yeti 2”, was developed by the Distributed Computing Group at ETH Zurich [28]. The plugin aims to provide developers with all the convenient functions expected from a modern development environment. It can be a very useful tool when building an application from scratch or when analyzing existing code.

Its main features include:

Error detection, for example syntactical errors or errors that occur when wiring interfaces and components.



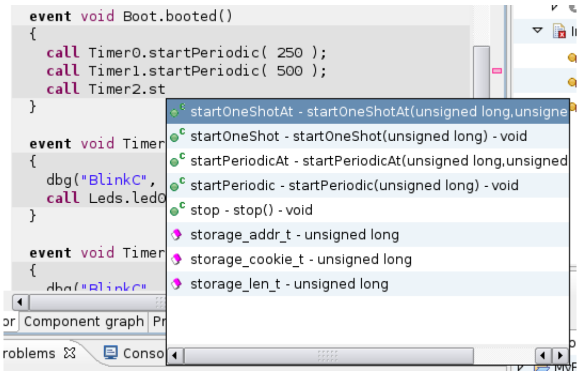
```
interface Demo<type>{
    command void work( type arg );
}
```

```
configuration DemoC{
    uses interface Demo<int> as DemoInt;
    provides interface Demo<float> as DemoFloat;
}
implementation{
    DemoInt -> DemoFloat;
}
```

the generic parameters of the interfaces do not match
Press 'F2' for focus

Figure 3.6.1: Yeti2 Error Detection

Code completion which can be activated by pressing Ctrl + Space. The plugin internally builds a model of each file. That model tells where and what is available at different locations.

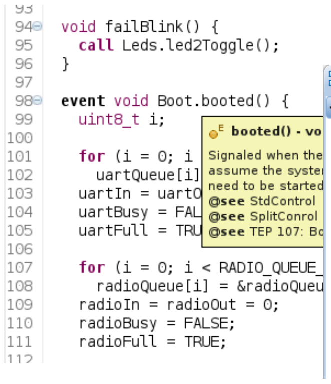


```
event void Boot.booted()
{
  call Timer0.startPeriodic( 250 );
  call Timer1.startPeriodic( 500 );
  call Timer2.st
}

event void Timer
{
  dbg("BlinkC",
  call Leds.led0
}

event void Timer
{
  dbg("BlinkC"
```

```
startOneShotAt - startOneShotAt(unsigned long,unsigne
startOneShot - startOneShot(unsigned long) - void
startPeriodicAt - startPeriodicAt(unsigned long,unsigned
startPeriodic - startPeriodic(unsigned long) - void
stop - stop() - void
storage_addr_t - unsigned long
storage_cookie_t - unsigned long
storage_len_t - unsigned long
```

Figure 3.6.2: Yeti 2 Code Completion

nesC documentation which is activated when the mouse rests over an item. A hover pops up and shows the nesC documentation associated with that element.

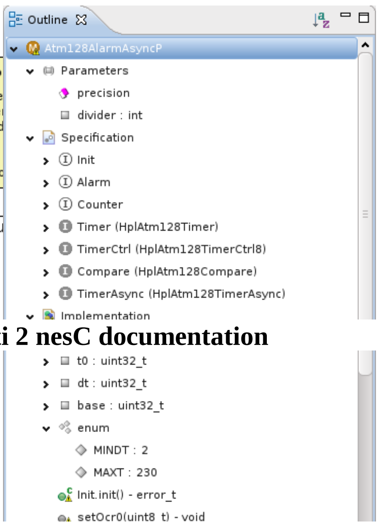


```
void failBlink() {
  call Leds.led2Toggle();
}

event void Boot.booted() {
  uint8_t i;

  for (i = 0; i
    uartQueue[i]
  uartIn = uartO
  uartBusy = FAL
  uartFull = TRU

  for (i = 0; i < RADIO_QUEUE_
    radioQueue[i] = &radioQueu
  radioIn = radioOut = 0;
  radioBusy = FALSE;
  radioFull = TRUE;
```

booted() - vo
Signaled when the
assume the syste
need to be started
@see StdControl
@see SplitConrol
@see TEP 107: Bo

The elements of every file whether

Figure 3.6.3: Yeti 2 nesC documentation



```
Atm128AlarmAsyncP
  Parameters
    precision
    divider : int
  Specification
    Init
    Alarm
    Counter
    Timer (HplAtm128Timer)
    TimerCtrl (HplAtm128TimerCtrl8)
    Compare (HplAtm128Compare)
    TimerAsync (HplAtm128TimerAsync)
  Implementation
    t0 : uint32_t
    dt : uint32_t
    base : uint32_t
    enum
      MINDT : 2
      MAXT : 230
    Init.init() - error_t
    setOcr0(uint8_t) - void
```

Figure 3.6.4: Yeti 2 Outline

it is a component or an interface, get represented in the Outline.

The plugin can also show the contents of a file as a graph.

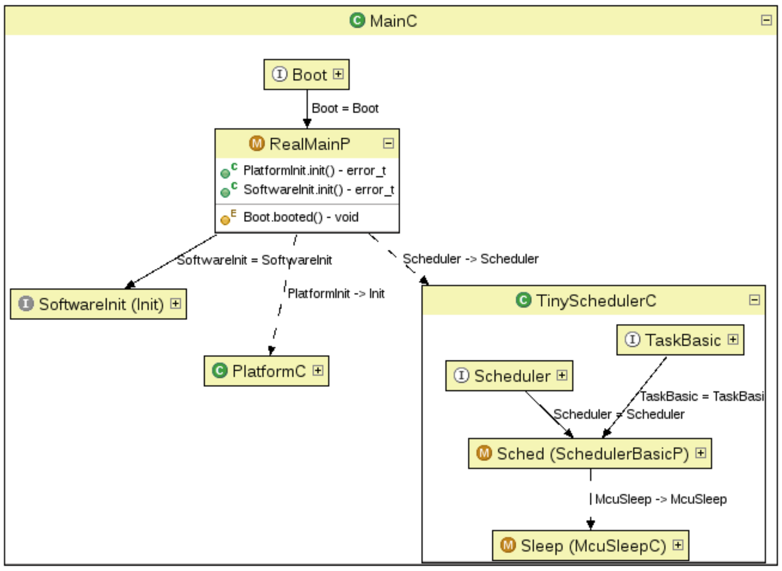


Figure 3.6.5: Yeti 2 Graph Creation

Chapter 4: WSN 1.0

4.1 Introduction

This is the chapter in which we actually build our application. We will start by programming motes that perform simple but fundamental tasks. Then we will combine a number of these tasks to create motes that operate as a WSN and after a series of measurements and observations we will try to extend and enhance our code, so as to make the network's operation more efficient.

4.2 Creating the mote

4.2.1 Sensor operation

The most important attribute of a mote functioning in a forest fire detecting WSN, is its ability to utilize the sensors attached to it. In our case the CM5000 mote is equipped with three onboard sensors, as described in Chapter 2: two light sensors and a temperature/relative humidity sensor. Let's start with the light sensors first.

The Hamamatsu S1087 and S1087-01 sensors provide visible and infrared light values respectively. We are only going to use the first one as it adequately fits our needs. As shown in figure 2.3.10 the photodiode is directly connected to the microcontroller's ADC and creates a current through a 100kΩ resistor.

$$I = \frac{V_{sensor}}{100000}$$

According to the graph provided by Hamamatsu (fig. 2.3.9), we can deduct a formula, which is essentially an approximation at a specific operating temperature, to linealyze the output current vs. the incident light level. The temperature of 25^{o}C is a good theoretical operating temperature of the WSN, but if we wanted to be more accurate we would have to extract the most appropriate constant in the range or ranges that the sensors are going to be working; the only available graph in the datasheet was the one drawn at the above mentioned temperature and since it suits us we will proceed with that. The incident light level is measured in lux and the output current in A.

$$lx = 0.625 \times 10^6 \times I \times 10^3 = 0.625 \times 10^9 \times \frac{V_{sensor}}{10^5} = 6250 \times V_{sensor}$$

The V_{sensor} value must be calculated by first of all obtaining the raw ADC count value of the sensor. This value can vary depending on the microcontroller's configuration. The default TinyOS-2.x configuration assumes the following values for a TelosB/Tmote Sky platform:

$$V_{sensor} = \frac{ADC_{value}}{4096} \times V_{ref}$$

V_{ref} is the voltage level of the internal reference voltage generator [27]; it is software configurable, its default value is 2.5V and it is defined (and can be adjusted) in the MSP430ADC12.h header file, located in the /tos/platform/msp430 folder in the TinyOS installation directory. Substituting to the equation above we obtain:

$$lx = 6250 \times \frac{ADC_{value}}{4096} \times 2.5 = 3.815 \times ADC_{value}$$

Moving to the Sensirion SHT11 things are simpler. According to the datasheet [19], the raw readings can be converted to SI units as follows:

- Temperature

$$T = d_1 + d_2 \times SO_T$$

V_{DD}	d_1(°C)	d_1(°F)		SO_T	d_2(°C)	d_2(°F)
5V	-40.1	-40.2		14bit	0.01	0.018
4V	-39.8	-39.6		12bit	0.04	0.072
3.5V	-39.7	-39.5				
3V	-39.6	-39.3				
2.5V	-39.4	-38.9				

Table 4.2.1: d_1 and d_2 in relation to VDD and ADC bits of operation

For TelosB motes the sensor is coupled with a 14-bit converter and in our case it is powered by 2xAA batteries connected in series, providing a supply voltage V_{DD} = 3V. SO_T is the raw ADC value of the sensor. Thus the formula is:

$$T = -39.6 + 0.01 \times SO_T$$

- Relative Humidity

$$RH = c_1 + c_2 \times SO_{RH} + c_3 \times {SO_{RH}}^2$$

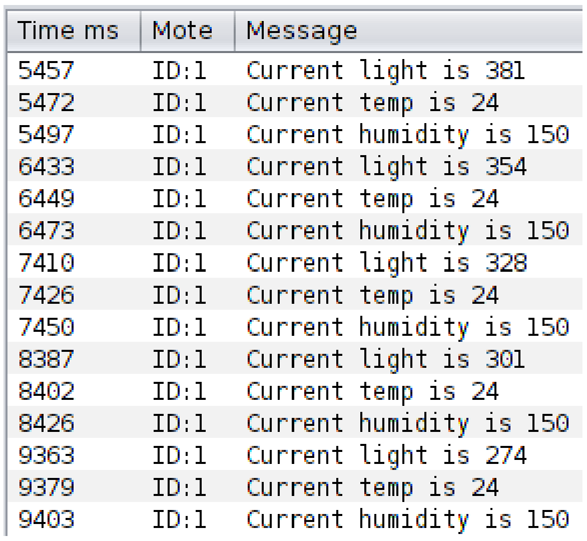
SO_{RH}	c_1	c_2	c_3
12bit	-2.0468	0.0367	-1.5955E-6
8bit	-2.0468	0.5872	-4.0845E-4

Table 4.2.2: c_1, c_2 and c_3 for 8 or 12bit ADC [19]

The default value for a CM5000 mote is 12bits [14], and SO_{RH} is the raw output of the sensor, so the formula is:

$$RH = -2.0468 + 0.0367 \times SO_{RH} - 1.5955 \times 10^{-6} \times {SO_{RH}}^2$$

To test the equations above, we program and simulate a mote, which periodically samples its sensors, converts the raw data to SI units and sends these values to its serial port. This is what the output looks like in Cooja:



```
Time ms  Mote   Message
5457     ID:1   Current light is 381
5472     ID:1   Current temp is 24
5497     ID:1   Current humidity is 150
6433     ID:1   Current light is 354
6449     ID:1   Current temp is 24
6473     ID:1   Current humidity is 150
7410     ID:1   Current light is 328
7426     ID:1   Current temp is 24
7450     ID:1   Current humidity is 150
8387     ID:1   Current light is 301
8402     ID:1   Current temp is 24
8426     ID:1   Current humidity is 150
9363     ID:1   Current light is 274
9379     ID:1   Current temp is 24
9403     ID:1   Current humidity is 150
```

Figure 4.2.1: Mote's Output

Temperature is at a stable 24^{o}C, relative humidity is also stable at 150% and light intensity is decreasing at a rate of 27lux/sec, until it reaches zero, then starts from 977lux all the way down to zero again. There are no actual sensors to gather data from, and Cooja uses parameters defined in its source code. One could modify these predefined values by modifying Cooja's source code, but that would set the modified values valid for the whole simulation. In other words, as of the latest version of Cooja included in Contiki 2.6, it is not possible to adjust the default sensor values just for a subsection of the simulation area, but only for the area as a whole. As will be shown later, this will not prevent us from simulating the WSN's operation, as we will use "dummy motes" with predefined sensor values. What we wanted to examine through this test was whether our formulas produced values within an acceptable range. The value of relative humidity is acceptable too. Relative humidity measures the current absolute humidity relative to the maximum for that temperature, so a value of 150%, although large, is possible. We will not use it for the rest of this thesis though, as in the case of a fire, temperature is quicker to change and that should be enough. One could use the relative humidity measurement to acquire a scope for the forest area as a whole, but since we have no physical motes and Cooja produces only stable values, we will leave it out.

4.2.2 Power Consumption

The typical operation of a fire-detecting mote as part of a WSN, would include sampling its sensors, sending the sensor data over the radio and receiving data from nearby motes to retransmit it to other nearby motes. Let's create a mote that samples its sensors once every minute and sends a message containing the sensor data, while having its radio always on listening to other motes and retransmitting their messages. A simplified state machine of this Prototype mote is depicted in figure 4.2.2. The mote boots and starts listening. It periodically samples its sensors and sends the data over the radio. In the event of a received message from another mote, it forwards the message to other nearby motes. In order to calculate the expected battery life of this (and any) mote, we must first measure each component's duty cycle in all possible power states.

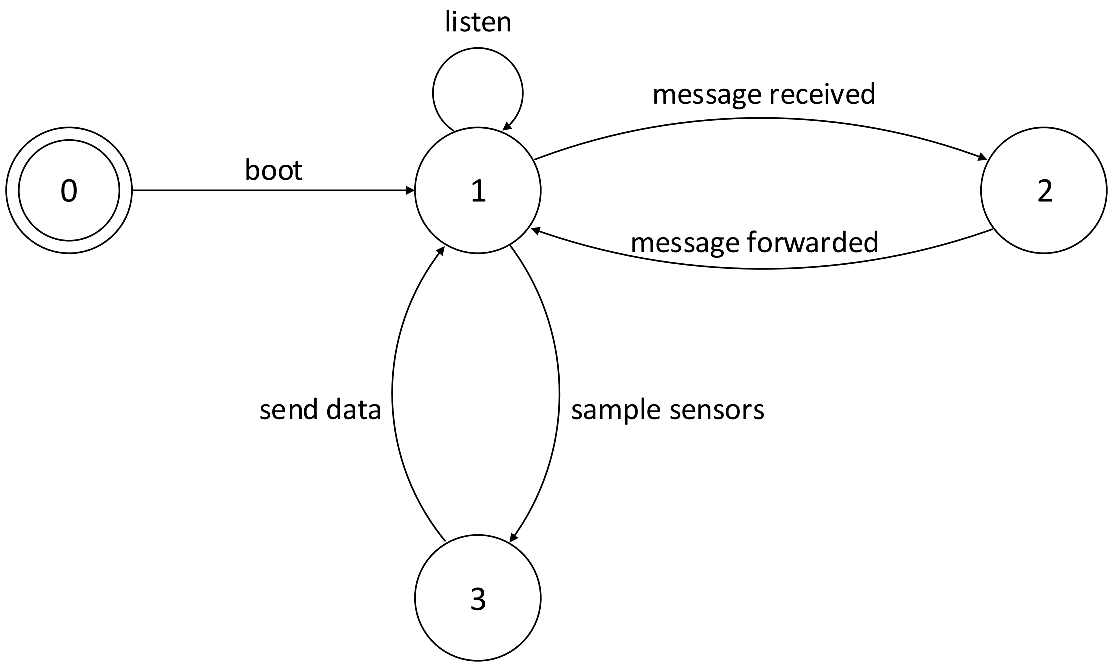
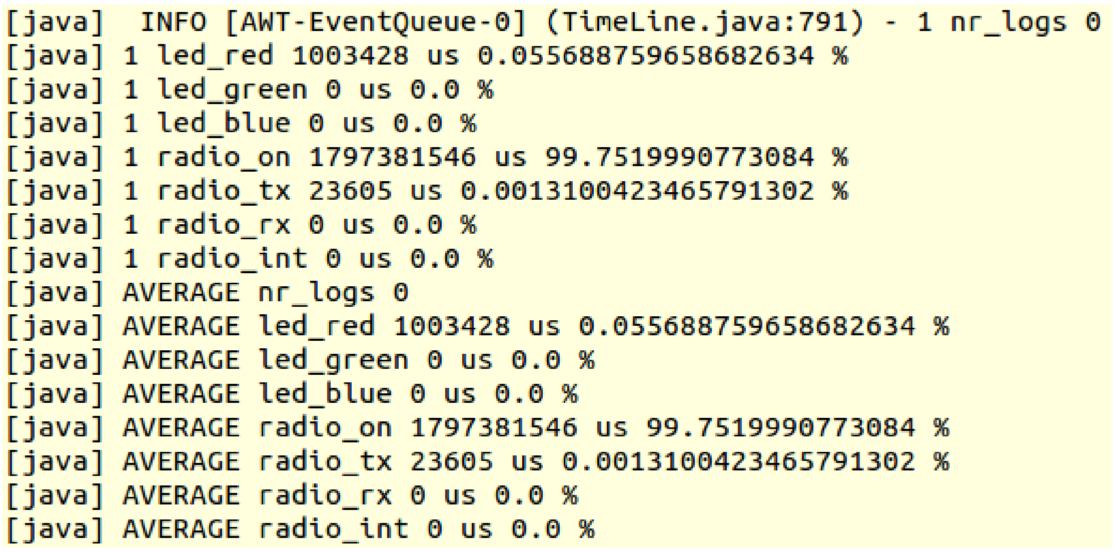


Figure 4.2.2: Simplified state machine of the Prototype mote

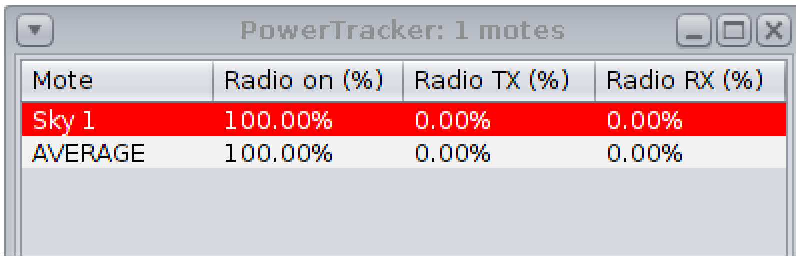
The simulation was run for 30 minutes; adequate time for the duty cycle to reach a steady value. The only parameter not clearly visible in Cooja is the sensors' duty cycle, but there is a good workaround to obtain it [2]. When the program calls the command to start sampling the sensors, we turn on LED 0, and when the sampling is completed and the packet is about to be sent, we turn it off. The LEDs are generally visible in Cooja's Timeline plugin, which apart from showing the LED and radio states, has the ability to print statistical simulation facts, to the console. The light sensor's current consumption is negligible so only the temperature sensor contributes to the power consumed. So, printing the statistics in the command line we get:



```
[java]  INFO [AWT-EventQueue-0] (TimeLine.java:791) - 1 nr_logs 0
[java] 1 led_red 1003428 us 0.055688759658682634 %
[java] 1 led_green 0 us 0.0 %
[java] 1 led_blue 0 us 0.0 %
[java] 1 radio_on 1797381546 us 99.7519990773084 %
[java] 1 radio_tx 23605 us 0.0013100423465791302 %
[java] 1 radio_rx 0 us 0.0 %
[java] 1 radio_int 0 us 0.0 %
[java] AVERAGE nr_logs 0
[java] AVERAGE led_red 1003428 us 0.055688759658682634 %
[java] AVERAGE led_green 0 us 0.0 %
[java] AVERAGE led_blue 0 us 0.0 %
[java] AVERAGE radio_on 1797381546 us 99.7519990773084 %
[java] AVERAGE radio_tx 23605 us 0.0013100423465791302 %
[java] AVERAGE radio_rx 0 us 0.0 %
[java] AVERAGE radio_int 0 us 0.0 %
```

Figure 4.2.3: Command line statistics for the Prototype mote

Furthermore, because this mote is simulated alone, radio transmission happens only once every 30 seconds and its duty cycle as seen in figure 4.2.4, is less than 0.01%, so Cooja's PowerTracker plugin calculates the transmit-state duty cycle to be actually 0.00% and the receive-state to be 100.00%. In general, we are going to use two decimal digits for duty cycle values so in this particular example we will consider the transmit-state (TX) duty cycle to be zero.



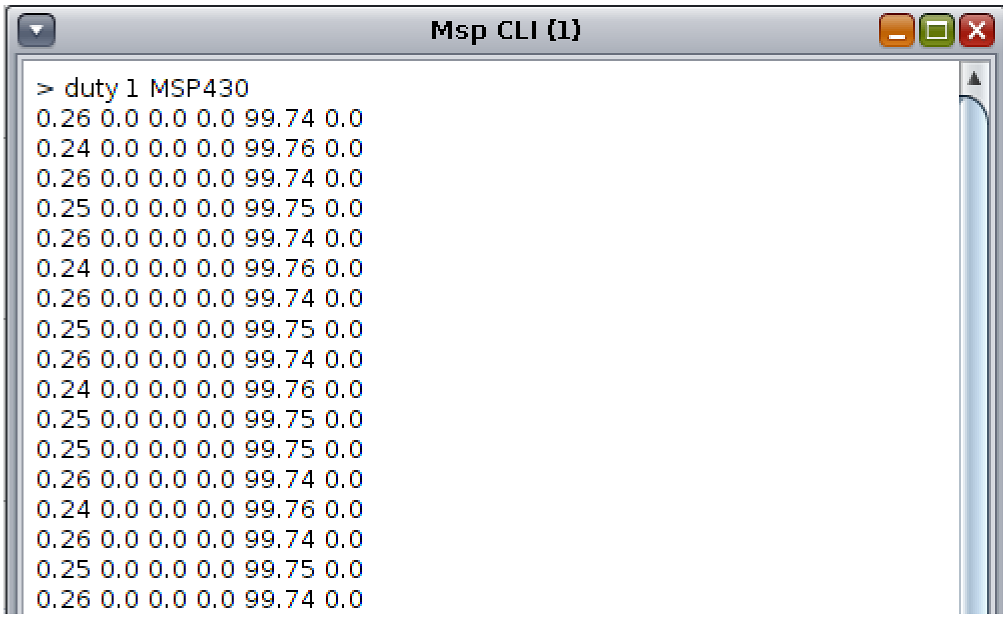
Mote	Radio on (%)	Radio TX (%)	Radio RX (%)
Sky 1	100.00%	0.00%	0.00%
AVERAGE	100.00%	0.00%	0.00%

Figure 4.2.4: Radio duty cycle for the Prototype mote

As far as the MCU is concerned, by right-clicking on the mote and choosing the Msp CLI option we can actually run MSPSim inside Cooja. The command:

```
duty X MSP430
```

prints the duty cycle of all power states of the MSP430 microprocessor X times/sec [1]. Its output looks like this:



```
> duty 1 MSP430
0.26 0.0 0.0 0.0 99.74 0.0
0.24 0.0 0.0 0.0 99.76 0.0
0.26 0.0 0.0 0.0 99.74 0.0
0.25 0.0 0.0 0.0 99.75 0.0
0.26 0.0 0.0 0.0 99.74 0.0
0.24 0.0 0.0 0.0 99.76 0.0
0.26 0.0 0.0 0.0 99.74 0.0
0.25 0.0 0.0 0.0 99.75 0.0
0.26 0.0 0.0 0.0 99.74 0.0
0.24 0.0 0.0 0.0 99.76 0.0
0.25 0.0 0.0 0.0 99.75 0.0
0.25 0.0 0.0 0.0 99.75 0.0
0.26 0.0 0.0 0.0 99.74 0.0
0.24 0.0 0.0 0.0 99.76 0.0
0.26 0.0 0.0 0.0 99.74 0.0
0.25 0.0 0.0 0.0 99.75 0.0
0.26 0.0 0.0 0.0 99.74 0.0
```

Figure 4.2.5: MSP430 duty cycle for the Prototype mote

The first column represents the active state, the second is the power-off state and the next four are the low-power modes, from LPM0 to LPM4 [1]. We see that the MCU spends all of its time either in active mode, or LPM3.

Gathering all the data, we are able to create the table below [4]:

Component	**Current [mA]**	**Duty Cycle [%]**	**Total current [mA]**
MSP430 MCU			
Active	1.8	0.25	0.0045
Sleep	0.0051	99.75	0.00508725
CC2420 Tranceiver			
Receive	18.8	100	18.8
Transmit	17.4	0	0
Sleep	0.001	0	0
SHT11 Temp. Sensor			
Measuring	0.5	0.06	0.0003
Sleep	0.0003	99.94	0.00029982
ST M25P80 Flash			
Active	20	0	0
Sleep	0.001	100	0.001
			18.8112

Table 4.2.3: Duty cycle and current consumption of all components in a Prototype mote

Now to calculate the mote’s expected battery life in relation to its battery capacity, we simply have to divide the battery’s mAh by the total current consumption as calculated above:

Battery Capacity [mAh]	Battery Life [h]	Battery Life [d]	Battery Life [m]
500	26.58	1.11	0.04
1000	53.16	2.21	0.07
1500	79.74	3.32	0.11
2000	106.32	4.43	0.15
2500	132.90	5.54	0.18
3000	159.48	6.64	0.22
3500	186.06	7.75	0.26
4000	212.64	8.86	0.30
4500	239.22	9.97	0.33
5000	265.80	11.07	0.37

Table 4.2.4: Prototype mote's battery life for various battery capacities

The most important part of a mote's power consumption is now obvious. The radio transceiver operation represents approximately 99.94% of the mote's total power consumption. A good quality alkaline battery has an effective capacity of 2000mAh, so connecting two of those in series, gives us about 4000mAh of battery capacity [6]. Under these circumstances, a prototype mote like this would only run for about 9 days before its batteries ran out. This is definitely not a good case when a WSN composed of 50 or more of those motes monitors a forest area. Having to replace the batteries of each mote every 9 days is not a good example of an autonomous WSN. We now see that the first thing a designer has to do, in a non CPU intensive application like this, is to limit the power-on time of the radio chip.

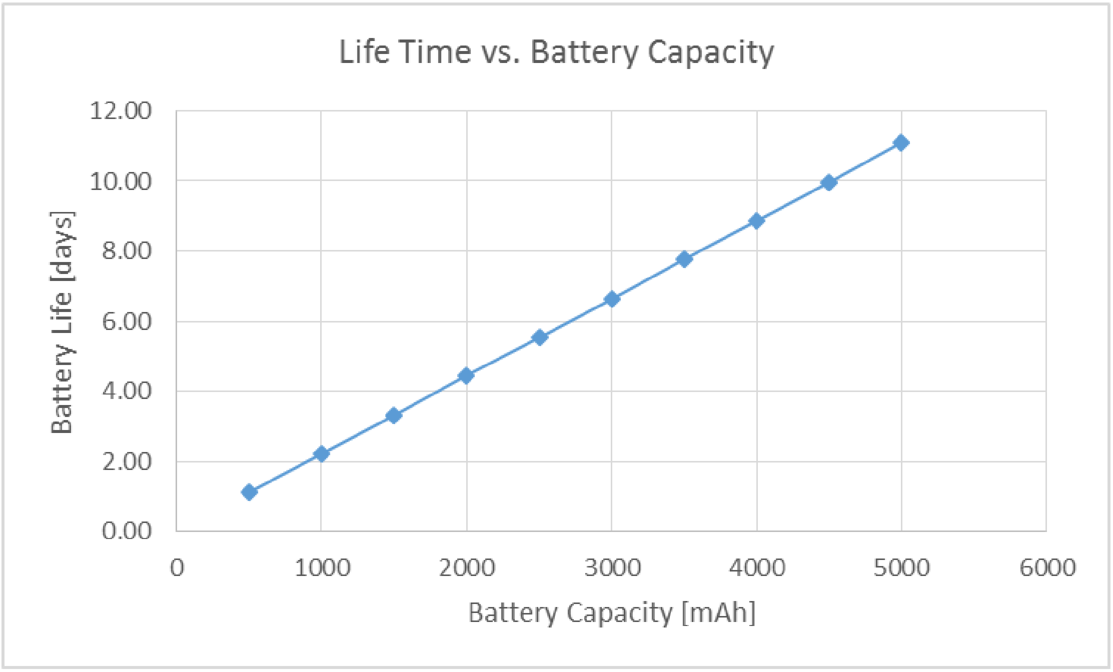


Figure 4.2.6: Mote's battery life vs. battery capacity (Prototype mote)

4.2.3 FireSense Mote

Moving on and based on the results we obtained from the previous section, we will try to extend the prototype mote into one that would perform better in terms of power consumption. We will name this mote FireSense and its simplified state machine is this:

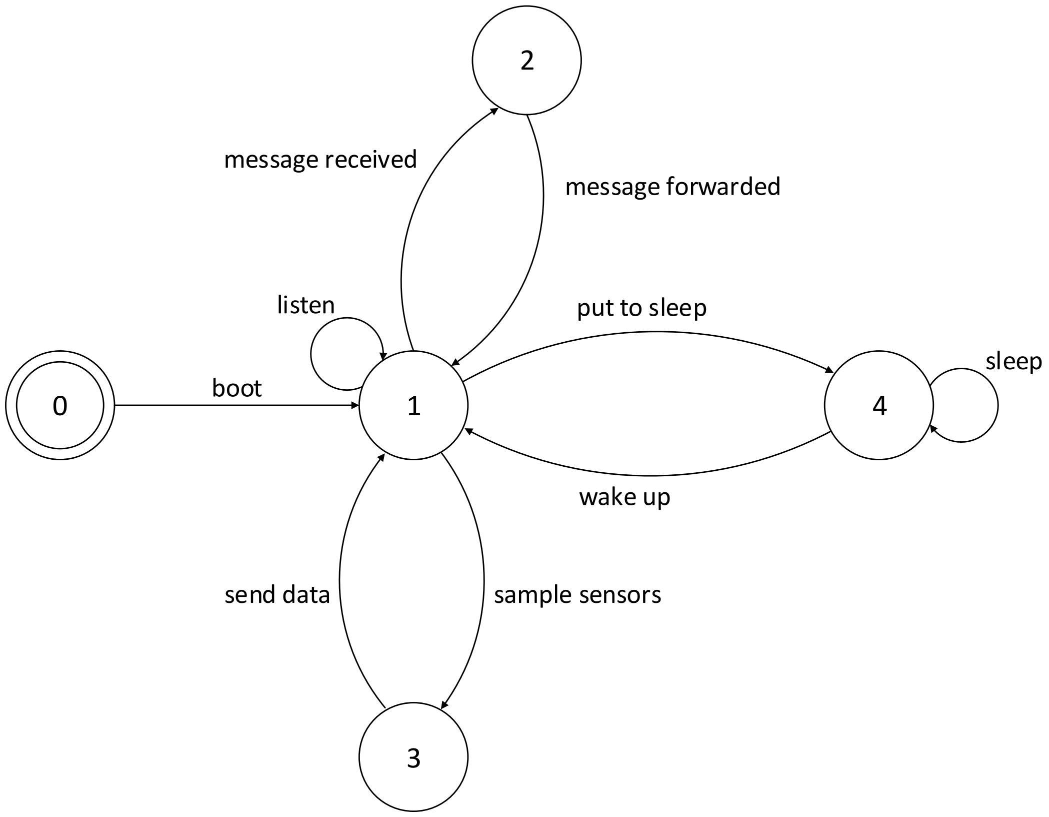


Figure 4.2.7: Simplified state machine of the FireSense mote

What's different than the first mote is that this one can (and will) spend most of its time sleeping. More specifically, we will program this mote to turn on its radio acting as a forwarder for messages coming from other motes, then sample its sensors to send their readings, then act as a forwarder again, and finally sleep for a certain amount of time. What sleep mode does is actually put every component in its lowest power state possible. The second listening period, after the sensor sampling, is necessary because the motes in a WSN do not start operating at exactly the same time, so for instance, when a mote sends its sensor readings over the radio, we have to make sure there is another one listening to forward the message. But even if there were a number of people, each one adjacent to every mote and all of them programmed and started the motes at the same time, this would create very heavy radio traffic (as we will see) that the network could not handle.

We will not present the mote's code in full, as some of its parts have already been examined in detail in Chapter 3. We will just present a few code snippets that show how this mote actually works. At first we have to create a header file that works the same way as a C header file. It defines constants and type structures and provides a comfortable way of quickly modifying applications.

```
#ifndef FIRE_SENSE_H
#define FIRE_SENSE_H

enum {
   MAX_SENSORS = 2, AM_RADIO = 1, LISTEN_DURATION_1 = 1000,
   LISTEN_DURATION_2 = 1000, SLEEP_DURATION = 28000};

typedef nx_struct FireSenseMsg {
   nxuint16_t node_id;
   nxuint16_t temperature;
   nxuint16_t luminance;
} FireSenseMsg_t;

#endif /* FIRE_SENSE_H */
```

In this case, several constants have been defined. The MAX_SENSORS constant determines the number of sensors we are using, in our case 2 (SHT11's temperature and S1087's luminance). The AM_RADIO is an id that will be used to discriminate between different kinds of messages. LISTEN_DURATION_1 and 2 represent the duration that the mote will have its radio on before and after sampling its sensors respectively. SLEEP_DURATION is pretty straightforward. The next lines define our message structure with three data fields storing the mote's ID, temperature and luminance measured, in the payload.

Moving on the module, which we named FireSenseC we have to declare a number of variables first:

```
uint8_t numsensors;
bool radio_busy = FALSE;
bool sensor_data_packet = FALSE;
message_t packet;
FireSenseMsg_t data;
```

The *packet* variable's role has been explained in Chapter 3. *data* is a FireSenseMsg structure used to store our data to be sent. *radio_busy* indicates whether the radio is being used and *sensor_data_packet* indicates whether the message sent is another mote's readings being forwarded or the mote's own sensor readings. We will show the *numsensors* use in a while.

```
event void Boot.booted() {
   post startListening();
}

task void startListening() {
   call RadioControl.start();
   call Timer1.startOneShot(LISTEN_DURATION_1);
}

event message_t * Receive.receive(message_t *msg, void *payload, uint8_t
len) {
   if (len == sizeof(FireSenseMsg_t) && (radio_busy == FALSE)) {
      FireSenseMsg_t * incomingPacket = (FireSenseMsg_t *) payload;
      FireSenseMsg_t * outgoingPacket;
      outgoingPacket = (FireSenseMsg_t *) call Packet.getPayload(&packet,
      sizeof(FireSenseMsg_t));
      outgoingPacket -> node_id     = incomingPacket -> node_id;
      outgoingPacket -> temperature = incomingPacket -> temperature;
      outgoingPacket -> light       = incomingPacket -> light;

      if (call AMSend.send(AM_BROADCAST_ADDR, &packet,
      sizeof(FireSenseMsg_t)) == SUCCESS) {
         radio_busy = TRUE;
         sensor_data_packet = FALSE;
      }
   }
   return msg;
}

event void Timer1.fired() {
   post startSensing();
}
```

When the mote boots, it dispatches the startListening task, which turns on the radio to forward received messages, and starts a timer (not periodic, but a single-shot) to indicate when the mote will dispatch the startSensing task, i.e. sample its sensors. The Receive.receive event, is signaled whenever the mote receives a packet. At first, this code checks if the radio is free and if the packet received is one to be forwarded. The latter is actually a security check and a simple way to prevent interference from unwanted sources with our WSN. It then gets the packet's payload portion and casts it to the incomingPacket pointer, a pointer to the previously declared FireSenseMsg type. It then creates the outgoingPacket pointer for the message to be sent. A simple copy of the incoming packet fields to the outgoing ones is the final step towards creating the outgoing message. The rest is the same as in Chapter 3. When the AMSend.send command is called, radio_busy is set to TRUE and sensor_data_packet is set to FALSE, since it's not the mote's own packet but a forwarded one.

```
task void startSensing() {
   numsensors = 0;
   //call Leds.led0On();
   call TempRead.read();
   call LightRead.read();
}

event void TempRead.readDone(error_t result, uint16_t val) {
   data.temperature = val;
   if (++numsensors == MAX_SENSORS) {
      post sendReadings();
      //call Leds.led0Off();
   }
}

event void LightRead.readDone(error_t result, uint16_t val) {
   data.luminance = val;
   if (++numsensors == MAX_SENSORS)
   {
      post sendReadings();
      //call Leds.led0Off();
   }
}
```

What startSensing does is simply call two read commands, one for the light sensor and one for the temperature sensor. When the sensor value has been read, the corresponding event is being signaled, that puts the values in the appropriate fields in the data variable defined earlier. When both sensors have been sampled, the sendReadings task is being dispatched. Note that we have commented out the commands that toggle LED0. It is something we use to assist us with the simulation and not something a designer should include in their code when programming a physical mote, as LED operation increases power consumption [2][4].

The sendReadings task is similar to the process described in Chapter 3. The only thing worth mentioning is the way data passed to the packet to be sent.

```
aux -> node_id     = TOS_NODE_ID;
aux -> temperature = data.temperature;
aux -> luminance   = data.luminance;
```

TOS_NODE_ID is unique to each mote and represents its ID [10]; it is configured during the programming of the mote, but since we are using Cooja we can easily define it when creating the simulation. *aux* is an auxiliary pointer for sending the packet. When the packet has been sent the AMSend.sendDone event is signaled:

```
event void AMSend.sendDone(message_t *msg, error_t error) {
   radio_busy = FALSE;
   if (sensor_data_packet == TRUE) {
      call Timer2.startOneShot(LISTEN_DURATION_2);
   }
}
```

It sets the radio free and notifies the mote that the packet sent contained its own sensor readings, so the mote continues to act as a forwarder for the duration specified in Timer2. When this timer fires, it’s time to put the mote to sleep.

```
event void Timer2.fired() {
   call RadioControl.stop();
}

event void RadioControl.stopDone(error_t error) {
   call Timer3.startOneShot(SLEEP_DURATION);
}

event void Timer3.fired() {
   post startListening();
}
```

The radio is turned off and when this happens, the mote sleeps for a duration defined in Timer3. When this timer fires, the startListening task is called; the mote wakes up and starts its operation all over again.

4.2.4 BaseStation Mote

A WSN consisting of FireSense motes alone would not be any good if there wasn't a base station where all the information would be gathered. A BaseStation node does exactly that; it is a CM5000 mote, connected via its USB port to a computer where it prints the sensor readings it has received from the FireSense motes. It is not necessary for the user to constantly read the values. If the mote receives an abnormal reading it notifies us by turning on its LEDs. The green LED means that everything is fine, the blue LED signifies a slightly abnormal value and the red LED a dangerously abnormal value. Before printing the values, the BaseStation mote is responsible for converting the raw data values in the received packets, to SI units, using the formulas we introduced in section 4.2.1. We didn't program the FireSense motes to do that, as we wanted to avoid putting even the slightest unnecessary CPU load; battery life is top priority in FireSense motes but it doesn't concern us in BaseStation motes as they are powered through their USB port.

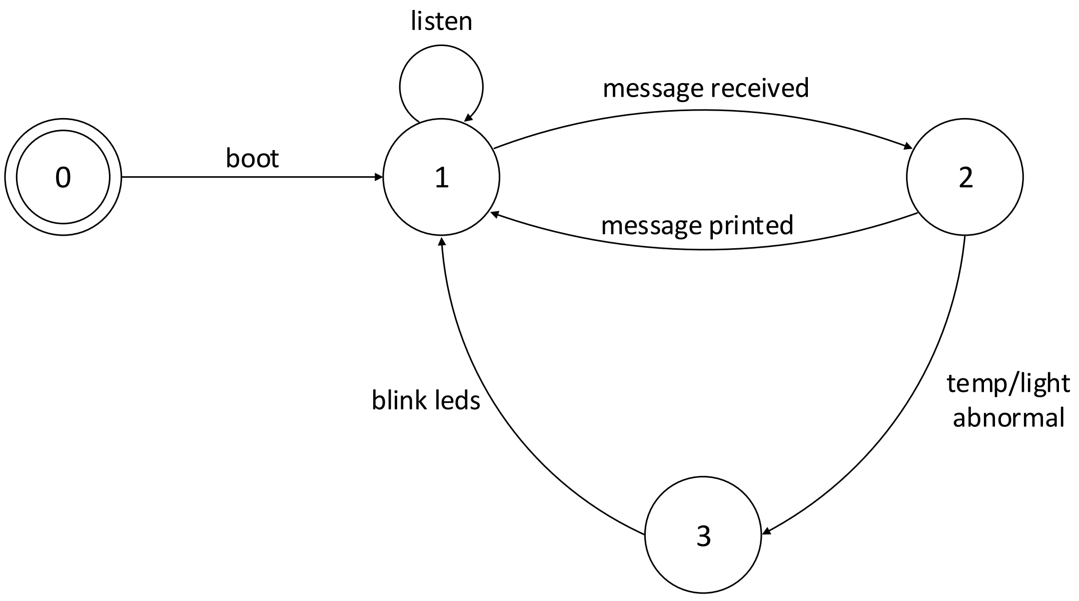


Figure 4.2.8: Simplified state machine of the BaseStation mote

4.3 Testing the WSN

It is time to simulate a WSN consisting of the motes created above. The outdoor range of the CC2420 radio chip, using the embedded antenna, is according to TelosB specs, around 100m, while the indoor range is about 20-30m. A forest area lies somewhere in the middle; it can't be considered indoors but it's not an outdoor area either as there exist trees and other land obstructions that decrease the outdoor range. We will consider a worst-case scenario, because it's preferable that the actual WSN operates better than the simulated one, than the opposite, so the default range will be set equal to 50m which we think it's close to what the real one should be [6]. Changing the default radio range in Cooja is as simple as right-clicking anywhere in the Network Visualizer and changing the value. The wireless channel chosen is UDGM – Distance Loss as it's the most realistic. The interference range is set to 70m. Furthermore, the FireSense motes will be randomly placed as there is no guarantee that a forest area will provide conditions for symmetrical mote placement. The motes vertical position has been randomly selected between 2 and 3 meters.

A large number of motes is not necessary as the delay between the reception of a packet and its forwarding is minimal as we will show. The WSN will consist of 15 FireSense motes and 1 BaseStation mote. The BaseStation mote will just print the packets as they arrive, for the time being. Turning on its LEDs will be better executed when we include a dummy mote. The motes will listen for 1 second before and after sampling their sensors, and they will be put into sleep mode for 58 seconds; thus they will operate periodically with a period of 1min. Moreover, the Mote Delay option in Cooja will be set equal to 1000ms. This means that the time difference between the first and last mote to boot, is 1 second.

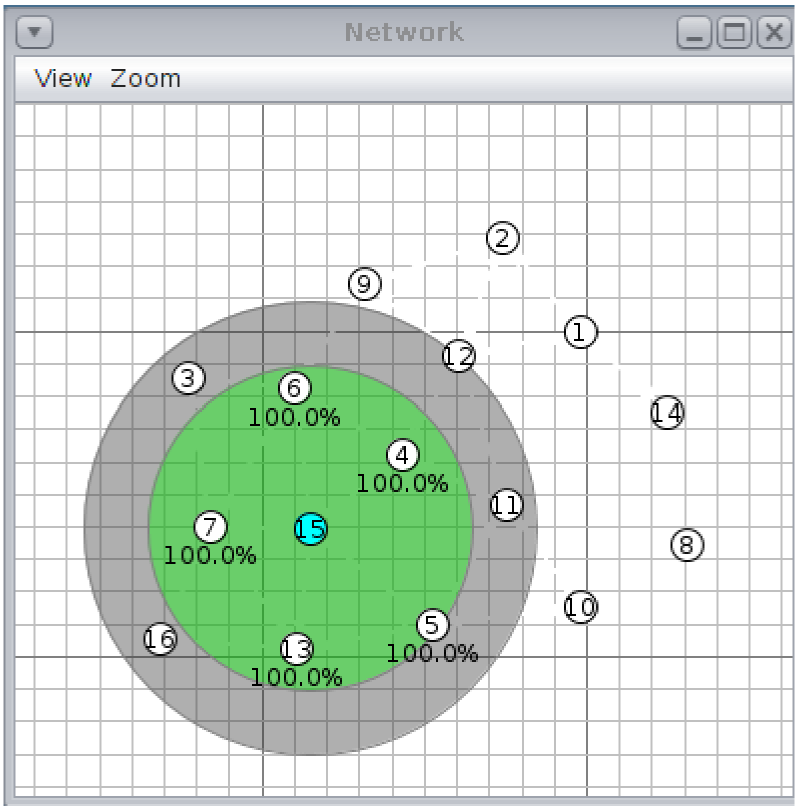
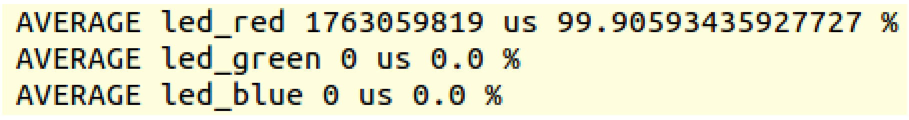


Figure 4.3.1: Network topology depicted

Motes 1-15 are FireSense motes, while mote 16 is the BaseStation. Mote 15's radio and interference range is depicted as an example. The gridlines correspond to a length of 10m, so the WSN covers an area ~160m x 130m = 20800m^2.

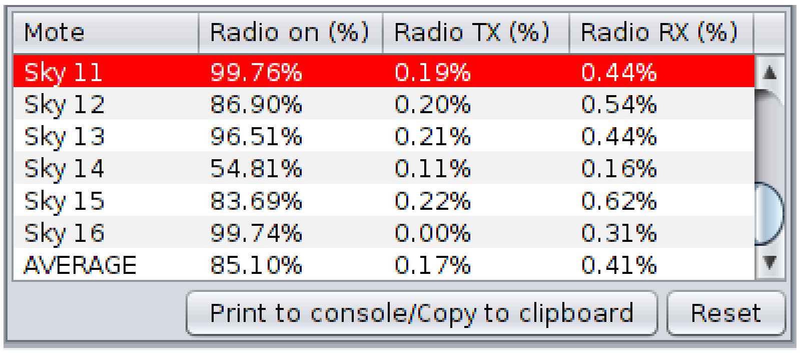
We ran the simulation for 30min to ensure stable duty cycles [2][4]. The results may be disappointing, but quite interesting. The sensors' (i.e. the LED0's) average duty cycle as printed in the command line is this:



```
AVERAGE led_red 1763059819 us 99.90593435927727 %
AVERAGE led_green 0 us 0.0 %
AVERAGE led_blue 0 us 0.0 %
```

Figure 4.3.2: FireSense onboard sensors' duty cycle

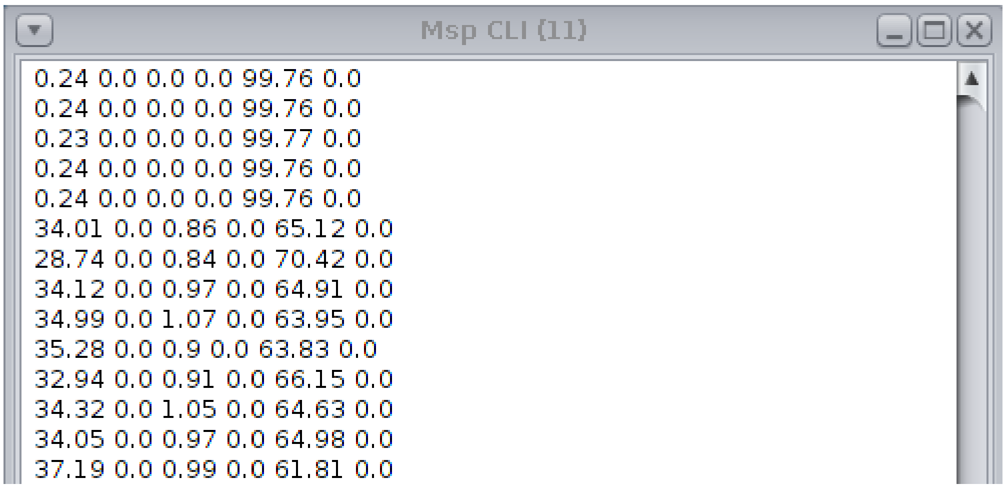
while the PowerTracker bears no good results either:



Mote	Radio on (%)	Radio TX (%)	Radio RX (%)
Sky 11	99.76%	0.19%	0.44%
Sky 12	86.90%	0.20%	0.54%
Sky 13	96.51%	0.21%	0.44%
Sky 14	54.81%	0.11%	0.16%
Sky 15	83.69%	0.22%	0.62%
Sky 16	99.74%	0.00%	0.31%
AVERAGE	85.10%	0.17%	0.41%

Figure 4.3.3: Radio duty cycle for the FireSense motes

Choosing a random mote, e.g. 11, and getting its MCU's duty cycle, we get:



```
0.24 0.0 0.0 0.0 99.76 0.0
0.24 0.0 0.0 0.0 99.76 0.0
0.23 0.0 0.0 0.0 99.77 0.0
0.24 0.0 0.0 0.0 99.76 0.0
0.24 0.0 0.0 0.0 99.76 0.0
34.01 0.0 0.86 0.0 65.12 0.0
28.74 0.0 0.84 0.0 70.42 0.0
34.12 0.0 0.97 0.0 64.91 0.0
34.99 0.0 1.07 0.0 63.95 0.0
35.28 0.0 0.9 0.0 63.83 0.0
32.94 0.0 0.91 0.0 66.15 0.0
34.32 0.0 1.05 0.0 64.63 0.0
34.05 0.0 0.97 0.0 64.98 0.0
37.19 0.0 0.99 0.0 61.81 0.0
```

Figure 4.3.4: MSP430 duty cycle for the FireSense motes

We expected the mote's MCU duty cycle to be higher than that of the Prototype mote's, because it now operates in a network where it has to process and forward several incoming messages but this is definitely not a good sign.

Fortunately, Cooja makes finding the cause of this easy. If we take a look at the Timeline, we see this:

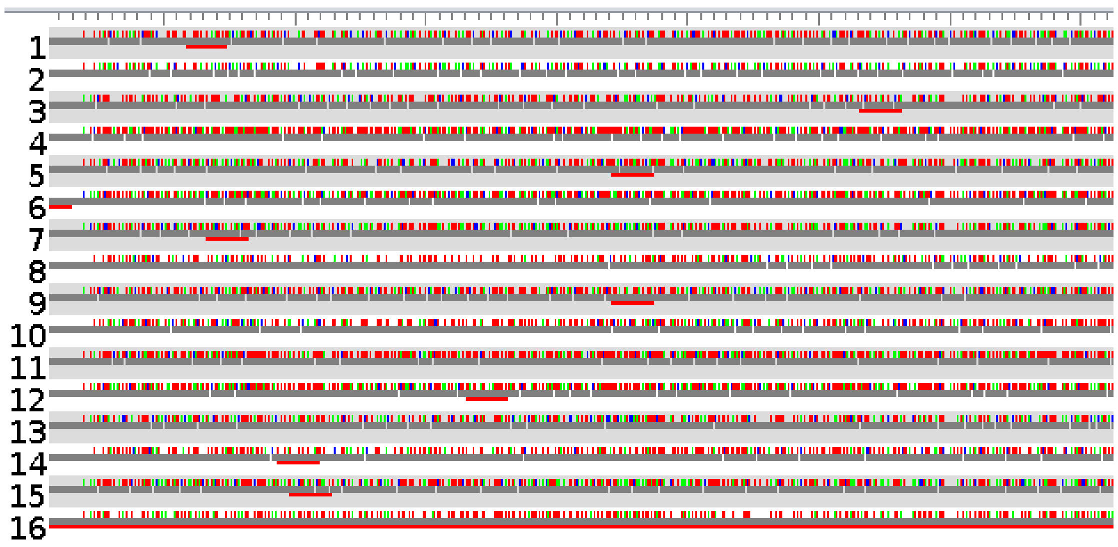


Figure 4.3.5: Heavy radio traffic during FireSense motes' operation

That is an immense amount of radio traffic. It is caused by continuous retransmission of the same packet. Let's take an example. Suppose that Mote 15, transmits its sensor readings. Motes 7 and 13 do a good job of forwarding the packet to the sink node, but they are actually broadcasting the message to all motes within range who are listening. So motes 3, 4, 5 and 6 receive the packet as well, and broadcast it to motes 9, 10, 11 and 12 who in turn, broadcast the packet to motes 1, 2, 8 and 14 who broadcast the message back in the direction of the sink node again. The result is that the first packet to be broadcast, actually chokes the entire network leaving very little available space for other packets to be transmitted, and this is very well depicted in the figure above. The higher duty cycle of the MCU is a direct consequence of that: the constant receiving and transmitting of a packet is indeed, apart from radio intensive, a CPU intensive process as well.

Another interesting deduction can be made by noticing that the average radio duty cycle is 85.10%, even though motes were programmed to sleep for 58 out of 60 seconds. That's a very good way to demonstrate what the split-phase nature of TinyOS can result in, if the code is not written properly. When one of the motes above is listening and it's time to dispatch a task, it is actually so busy receiving and sending packets, that it postpones dispatching the task, to a time when the radio traffic has decreased; the radio traffic is very high all the time though, so the TinyOS scheduler postpones the dispatching of tasks and keeps the mote listening indefinitely [7].

4.4 WSN 1.0

It is obvious that if we want to materialize our WSN, we need to find a way to reduce its radio traffic. One solution would be to reduce each mote's neighboring motes. That creates two other problems though: first, the accuracy of the WSN as a whole is reduced because each mote would be responsible for a larger area, and second: a mote's break down, would limit the alternative routes of a travelling packet. Another solution, would be to manually assign addresses to the motes and program each mote to transmit to certain destinations, but that would make it difficult to modify the network once it has been set up. For example, if we wanted to expand it or add new motes in the same area, we would have to reprogram every mote to include the addresses of the new motes. A good and viable solution is at the same time, the simplest. Every time a mote boots, it will create and initialize to zero a one-dimensional array, the length of which, will be equal to the number of motes in the WSN. The actual size of it will be NUMBER_OF_MOTES + 1, because the first element in an array in nesC (and C) is array[0], and there is no mote with a zero ID. We could also increase the length of the array to make room for additional motes.

The first time, in a listening period, that a mote receives and forwards a packet from mote X, it will set the element array[X] equal to 1. If sometime later in the same period, the mote receives the same packet, recycled in the network, it will check the value of array[X]. If it isn't equal to zero, it will not transmit it. When the mote is put to sleep, the array is set to zero again. The timeline's output looks like this now:

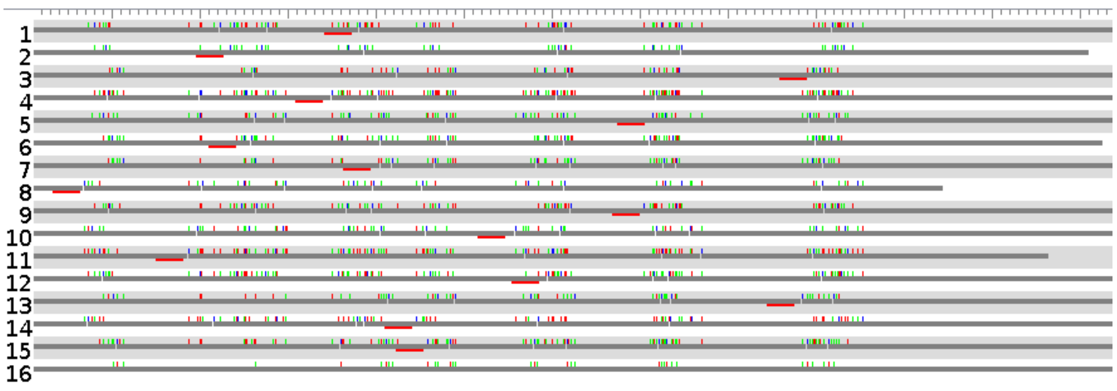
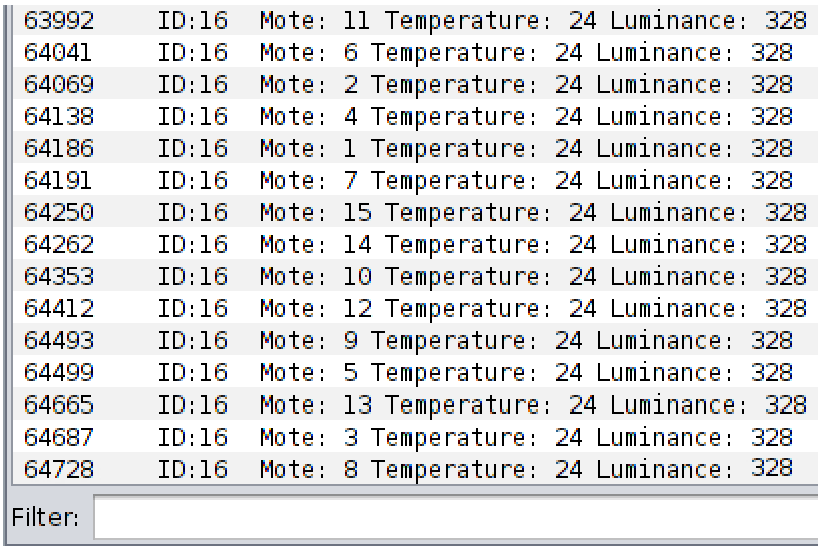


Figure 4.4.1: Normal radio traffic after code improvement

Things are looking good now. Radio traffic is greatly reduced. Five lines of code can make that big a difference. We can clearly distinguish separate radio traffic times in this picture, every time a mote samples its sensors and broadcasts the readings (right after the red LED turns off). There still exist some packets that have been interfered (marked by red dots above the grey line) so in order to determine if we have compromised the WSN's effectiveness in transmitting all packets over the radio we check the serial output of the BaseStation mote:



```
63992    ID:16   Mote: 11 Temperature: 24 Luminance: 328
64041    ID:16   Mote: 6 Temperature: 24 Luminance: 328
64069    ID:16   Mote: 2 Temperature: 24 Luminance: 328
64138    ID:16   Mote: 4 Temperature: 24 Luminance: 328
64186    ID:16   Mote: 1 Temperature: 24 Luminance: 328
64191    ID:16   Mote: 7 Temperature: 24 Luminance: 328
64250    ID:16   Mote: 15 Temperature: 24 Luminance: 328
64262    ID:16   Mote: 14 Temperature: 24 Luminance: 328
64353    ID:16   Mote: 10 Temperature: 24 Luminance: 328
64412    ID:16   Mote: 12 Temperature: 24 Luminance: 328
64493    ID:16   Mote: 9 Temperature: 24 Luminance: 328
64499    ID:16   Mote: 5 Temperature: 24 Luminance: 328
64665    ID:16   Mote: 13 Temperature: 24 Luminance: 328
64687    ID:16   Mote: 3 Temperature: 24 Luminance: 328
64728    ID:16   Mote: 8 Temperature: 24 Luminance: 328
Filter:
```

Figure 4.4.2: BaseStation receives all FireSense motes' packets during a sample period

The columns from left to right stand for: the time the output was printed (in ms), the ID of the mote that gives the output (we have given BaseStation the MoteID: 16) and the mote's printed message. The BaseStation mote prints the readings of all FireSense motes in a sampling period, so the WSN works as expected.

To test the WSN's response time in case of an abnormal reading and to test the LED operation of the BaseStation mote we create a "dummy mote". That is actually a FireSense mote in which we have predefined its sensor readings to be abnormal. The dummy mote has been programmed to send a packet only when we press its button. We are going to measure how much time it takes for a packet containing sensor samples to reach the sink mote. Let's suppose a measured temperature of 50°C. The dummy mote has been given the ID: 16 and the BaseStation is now number 17.

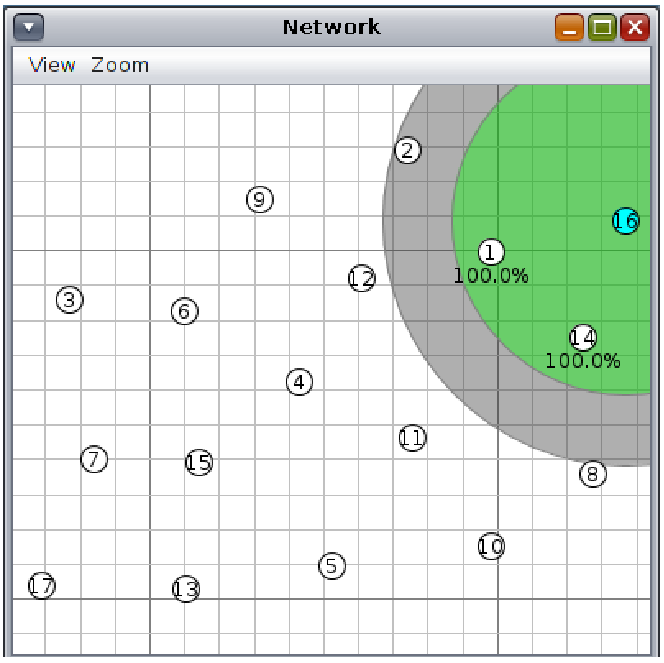


Figure 4.4.3: Dummy mote placement

The BaseStation mote turns its green LED on when it boots, to indicate the user that it's working. Since 50^{o}C is a very high temperature, we expect the red LED to turn on. We press the button on the dummy mote 16 at precisely 34341ms simulation time. The BaseStation mote receives the packet at 34370ms. The radio packet covers a distance of more than 200m in only 29ms. That is practically real time information and the WSN's response time is excellent.

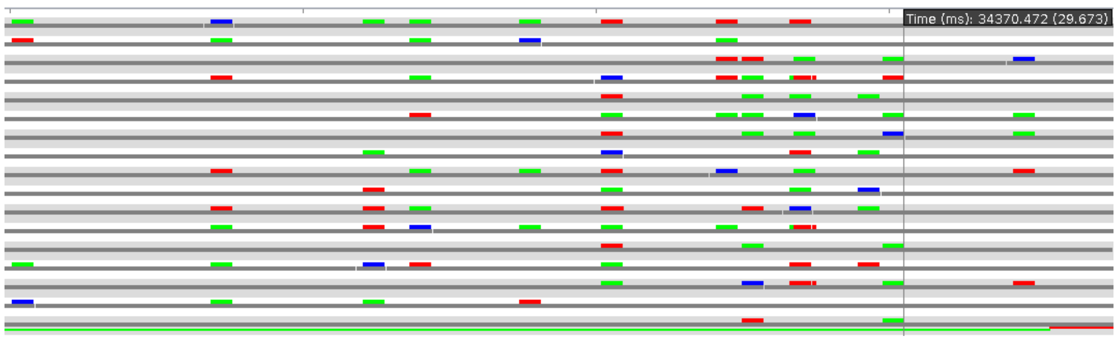
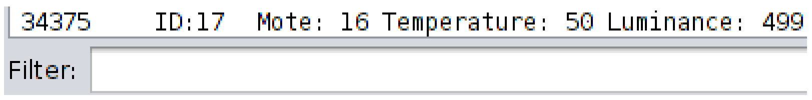


Figure 4.4.4: Packets exchanged while forwarding a dummy mote's packet. Vertical line depicts 29ms

The mote needs 5 more milliseconds to turn on its red LED as seen in the picture above (lower right corner), and forward the packet to its serial port:



```
34375     ID:17   Mote: 16 Temperature: 50 Luminance: 499
Filter:
```

Figure 4.4.5: BaseStation needs 5ms to forward a radio packet to its serial port

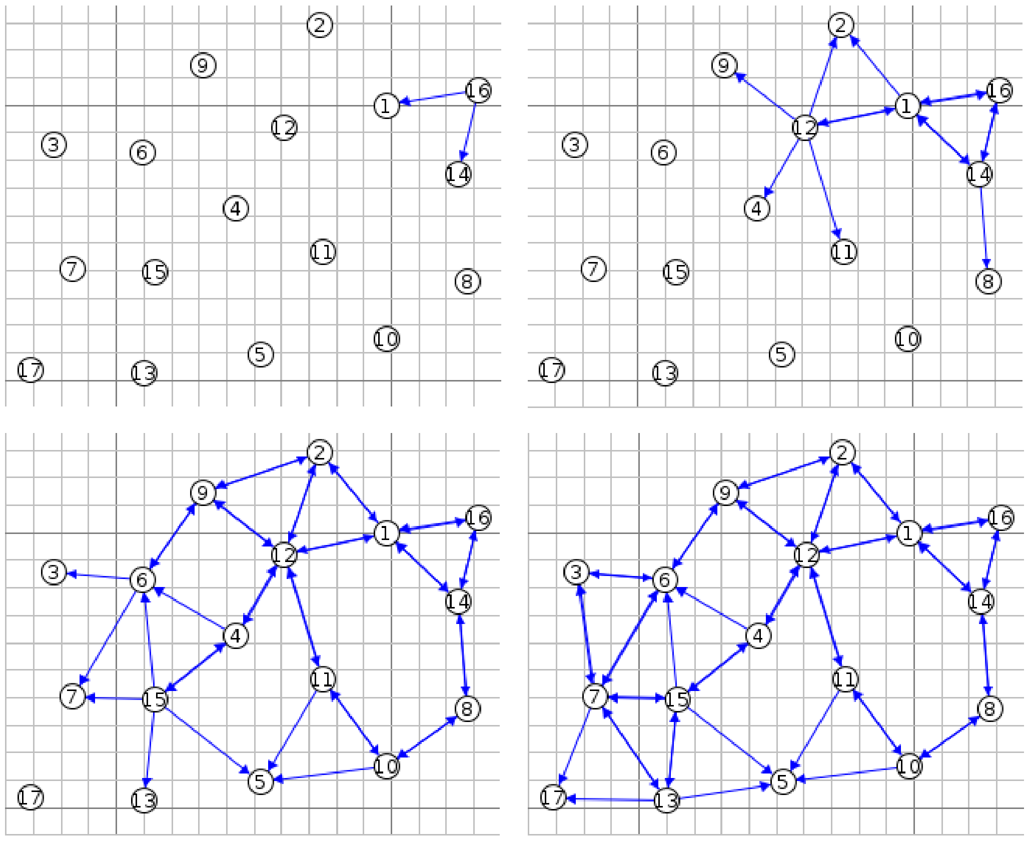


Figure 4.4.6: This is what 29ms look like as the WSN forwards the packet from dummy mote to sink

We consider the WSN’s behavior a success. The source code of the FireSense mote along with the BaseStation mote can be found in the Appendix. It’s now time to examine our motes’ energy characteristics.

4.5 Energy Analysis

We saw from the previous tests that two seconds of listening time (1sec before and 1 sec after sampling) are adequate for forwarding all motes’ packets to the sink mote. We will not try to decrease that time as doing so might increase the chances of lost packets. We will adjust the SLEEP_DURATION constant though, in order to examine the way in which the WSN’s sampling frequency affects the motes’ battery life. The chosen values for the WSN’s sampling period are 30sec, 60sec, and 90sec, so SLEEP_DURATION will be set to 28, 58 and 88 seconds respectively. We are also going to use the same procedure as in section 4.2.2.

- Sampling period: 30sec

Component	Current [mA]	Duty Cycle [%]	Total current [mA]
MSP430 Microprocessor			
Active	1.8	1.18	0.02124
Sleep	0.0051	98.82	0.00503982
CC2420 Tranceiver			
Receive	18.8	7.08	1.33104
Transmit	17.4	0.04	0.00696
Sleep	0.001	92.88	0.0009288
SHT11 Temp. Sensor			
Measuring	0.5	1.69	0.00845
Sleep	0.0003	98.31	0.00029493
ST M25P80 Flash			
Active	20	0	0
Sleep	0.001	100	0.001
			1.3750

Table 4.5.1: Duty cycle and current consumption of all components in a FireSense mote (30sec period)

Battery Capacity [mAh]	Battery Life [h]	Battery Life [d]	Battery Life [m]
500	363.65	15.15	0.51
1000	727.30	30.30	1.01
1500	1090.95	45.46	1.52
2000	1454.59	60.61	2.02
2500	1818.24	75.76	2.53
3000	2181.89	90.91	3.03
3500	2545.54	106.06	3.54
4000	2909.19	121.22	4.04

4500	3272.84	136.37	4.55
5000	3636.49	151.52	5.05

Table 4.5.2: FireSense mote's battery life for various battery capacities (30sec period)

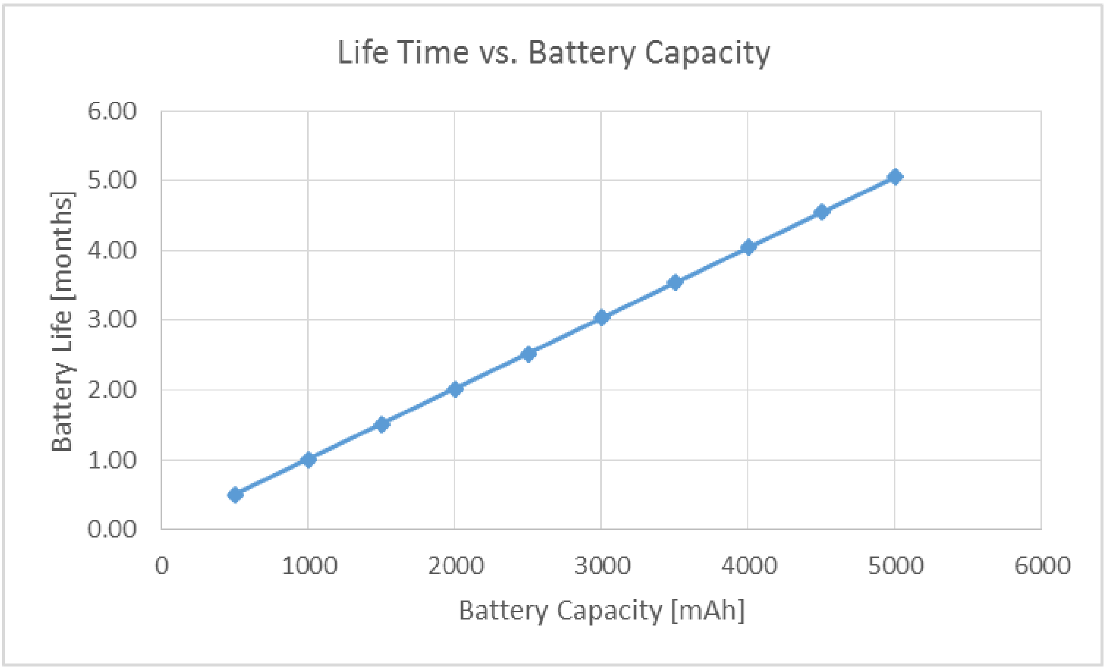


Figure 4.5.1: Mote's battery life vs. battery capacity (FireSense mote, 30sec period)

- Sampling period: 60sec

Component	**Current [mA]**	**Duty Cycle [%]**	**Total current [mA]**
MSP430 Microprocessor			
Active	1.8	0.67	0.01206
Sleep	0.0051	99.33	0.00506583
CC2420 Tranceiver			
Receive	18.8	3.46	0.65048
Transmit	17.4	0.02	0.00348
Sleep	0.001	96.52	0.0009652
SHT11 Temp. Sensor			
Measuring	0.5	0.86	0.0043
Sleep	0.0003	99.14	0.00029742
ST M25P80 Flash			
Active	20	0	0
Sleep	0.001	100	0.001
			0.6776

Table 4.5.3: Duty cycle and current consumption of all components in a FireSense mote (60sec period)

Battery Capacity [mAh]	Battery Life [h]	Battery Life [d]	Battery Life [m]
500	737.85	30.74	1.02
1000	1475.69	61.49	2.05
1500	2213.54	92.23	3.07
2000	2951.38	122.97	4.10
2500	3689.23	153.72	5.12
3000	4427.07	184.46	6.15
3500	5164.92	215.20	7.17
4000	5902.77	245.95	8.20
4500	6640.61	276.69	9.22
5000	7378.46	307.44	10.25

Table 4.5.4: FireSense mote's battery life for various battery capacities (60sec period)

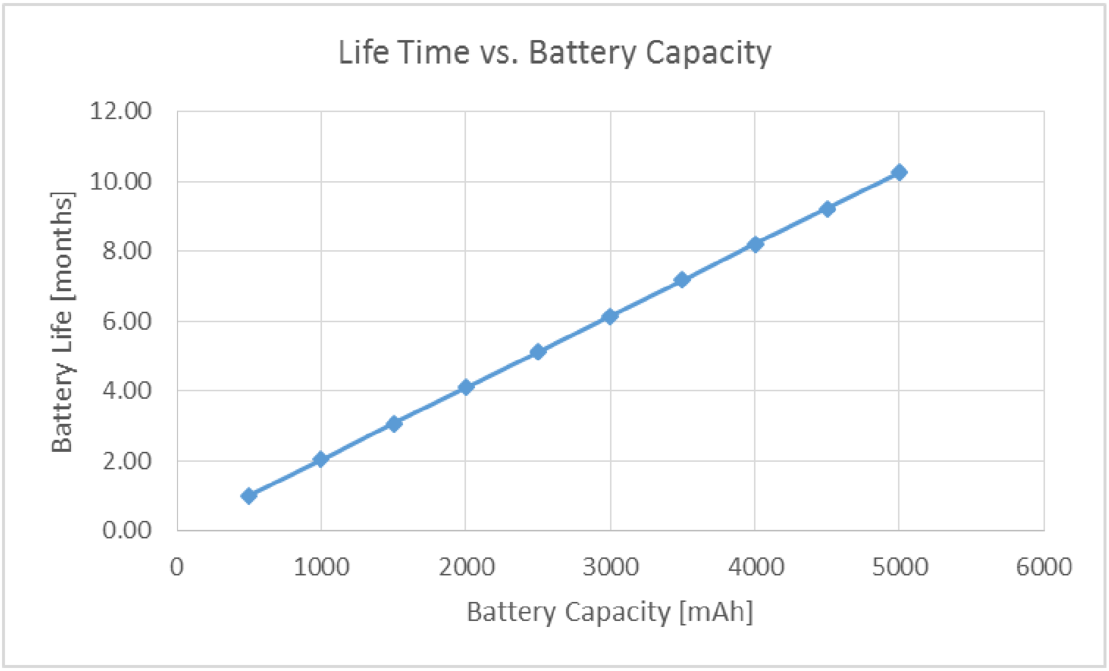


Figure 4.5.2: Mote's battery life vs. battery capacity (FireSense mote, 60sec period)

- Sampling period: 90sec

Component	Current [mA]	Duty Cycle [%]	Total current [mA]
MSP430 Microprocessor			
Active	1.8	0.41	0.00738
Sleep	0.0051	99.59	0.00507909
CC2420 Tranceiver			
Receive	18.8	2.29	0.43052
Transmit	17.4	0.01	0.00174
Sleep	0.001	97.7	0.000977
SHT11 Temp. Sensor			
Measuring	0.5	0.53	0.00265
Sleep	0.0003	99.47	0.00029841

ST M25P80 Flash			
Active	20	0	0
Sleep	0.001	100	0.001
			0.4496

Table 4.5.5: Duty cycle and current consumption of all components in a FireSense mote (90sec period)

Battery Capacity [mAh]	Battery Life [h]	Battery Life [d]	Battery Life [m]
500	1111.99	46.33	1.54
1000	2223.98	92.67	3.09
1500	3335.97	139.00	4.63
2000	4447.96	185.33	6.18
2500	5559.95	231.66	7.72
3000	6671.94	278.00	9.27
3500	7783.93	324.33	10.81
4000	8895.92	370.66	12.36
4500	10007.91	417.00	13.90
5000	11119.90	463.33	15.44

Table 4.5.6: FireSense mote's battery life for various battery capacities (90sec period)

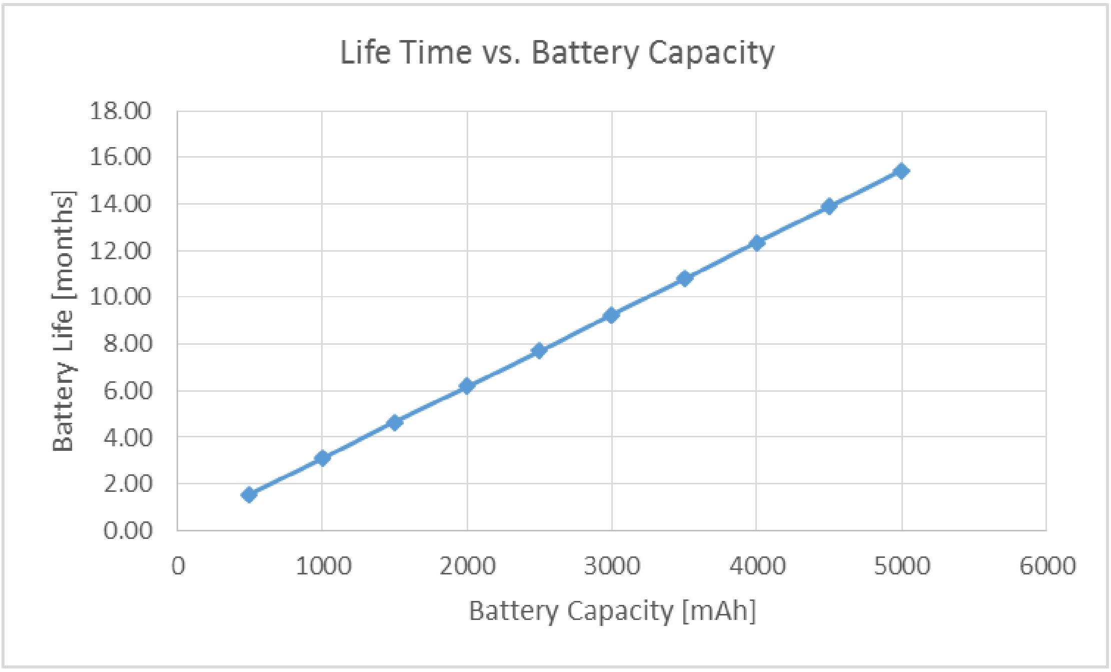


Figure 4.5.3: Mote's battery life vs. battery capacity (FireSense mote, 90sec period)

Chapter 5: WSN 2.0

5.1 Intro

Building upon our experience and gathered data from the previous chapter, we aim to further enhance the designed WSN, increasing its motes' lifetime without sacrificing spatial or temporal accuracy. To achieve this, we are going to examine a different network topology that divides the motes in terms of the operation they execute: fire sensing and packet forwarding. But to do that, we first need to examine the effect the supply current has on radio range.

5.2 Friis Transmission Equation

The Friis transmission equation is used in telecommunications engineering, and gives the power received by one antenna under idealized conditions given another antenna some distance away transmitting a known amount of power. In its simplest form, the Friis transmission equation is as follows. Given two antennas, the ratio of power available at the input of the receiving antenna, Pr, to output power to the transmitting antenna, Pt, is given by [18]:

$$\frac{P_r}{P_t} = G_t \times G_r \times \left(\frac{\lambda}{4\pi R}\right)^2$$

where Gt and Gr are the antenna gains of the transmitting and receiving antennas respectively, λ is the wavelength and R is the distance between the antennas. The equation in this form, is of little use to us. The antenna gains, in our case, are expressed in decibels, so the equation is slightly modified to [18]:

$$P_r = P_t + G_t + G_r + 20\log_{10}\left(\frac{\lambda}{4\pi R}\right)$$

where gain has units in dB, and power has units in dBm.

TelosB motes come equipped with an internal antenna that is an Inverted-F microstrip protruding from the end of the board away from the battery pack. The Inverted-F antenna is a wire monopole where the top section is folded down to be parallel with the ground plane. It has been examined in detail, in Texas Instrument's Design Note 7 [17], where its gain has been measured in the XY, XZ and YZ plane. We won't go into such detail in this thesis, so we will take into consideration only the horizontal XY plane in our calculations. This has been measured to be 1.1dB. We also know from Chapter 2 that the receiving power of TelosB is -90dBm at minimum.

Let’s restructure the equation above so that we can express R in terms of power:

$$20\log_{10}\left(\frac{4\pi R}{c/f}\right) = P_t + G_t + G_r - P_r \Leftrightarrow R = \frac{c \times 10^{(P_t+G_t+G_r-P_r)/20}}{4\pi f}$$

c is the speed of light ($3x10^8$ m/s) and f is the frequency of the signal (2.4 GHz).

As we have mentioned is Chapter 2, the CC2420 has programmable output power. It is adjusted by specifying a parameter called PA_LEVEL in the code’s makefile [7], when programming a mote. In order to gain an understanding of the relation between range and input power, we will consider two motes. One is transmitting and the other is receiving. Both motes’ antenna gain is 1.1dB and the receiver mote’s power is -90dBm [16]. In the table below we present typical PA_LEVEL values and their corresponding current consumption and output power. In the fourth column we have calculated the maximum theoretical distance between a mote transmitting with the corresponding output power and a receiver mote.

PA_LEVEL	Output Power [dBm]	Current Consumption [mA]	Range [m]
31	0	17.4	405.31
27	-1	16.5	361.23
23	-3	15.2	286.93
19	-5	14	227.92
15	-7	12.5	181.04
11	-10	11	128.17
7	-15	9.9	72.07
3	-25	8.5	22.79

Table 5.2.1: PA_LEVEL, output power and current consumption in relation to ideal radio range

We see that that the range at output power equal to 0dBm (the default value) is more than 3 times higher than the one given in the TelosB datasheet. This shouldn’t worry us because the calculations above don’t take into account the dimensional parameters of a mote or interferences in the radio medium, and also assume idealized conditions. A more in-depth analysis has been done in Texas Instruments Design Note 18 [18], in which ground reflections and environmental noise have been accounted for. This is not our purpose though, as through this theoretical calculation we wanted to examine the relation between output power, current consumption and range, in order to adjust the default radio range we assumed in the previous chapter. We notice, that halving the radio chip’s supply current, from 17.4 to 8.5mA brings radio range down to about 1/18th of its value.

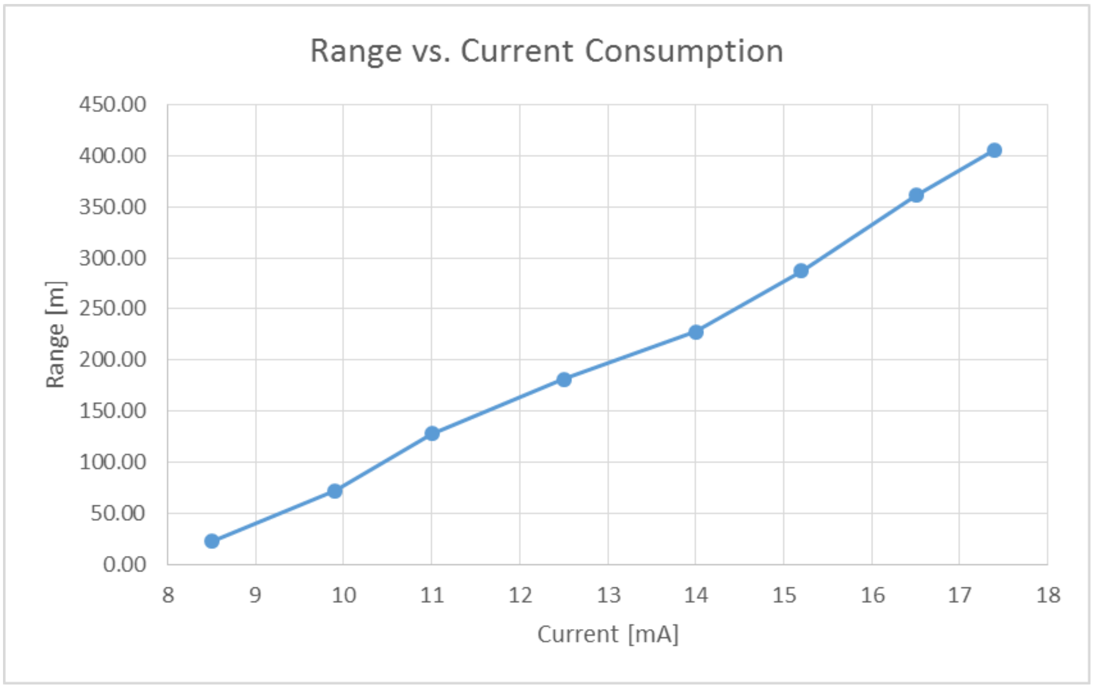
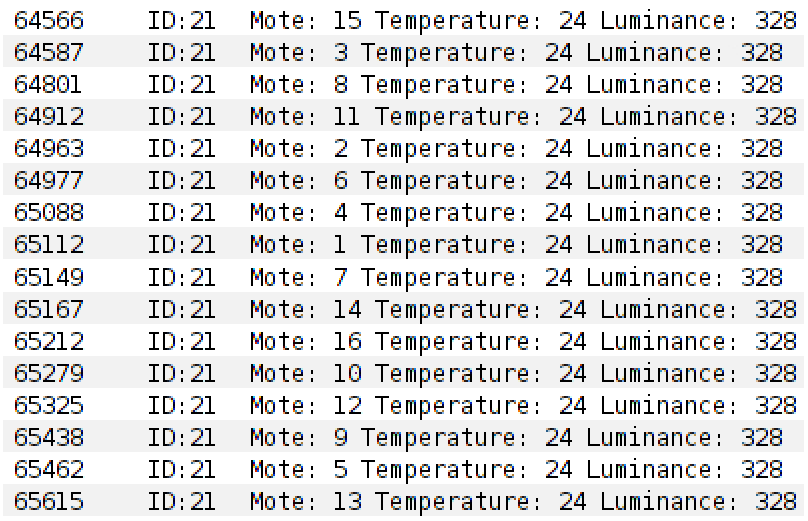


Figure 5.2.1: CC2420 ideal radio range vs. current consumption

5.3 WSN 2.0

Based on the results obtained from the previous section, we aim to reconstruct the topology of our WSN, by dividing each mote in regard to the work it does. The WSN we designed in Chapter 4 was an ad-hoc network. We now want to move towards an "access point" orientation. To achieve this we have to, in a way, split the FireSense mote in half. We will name the first mote FireSense 2.0 and the second one Repeater. FireSense 2.0 will now be responsible only for taking temperature and luminance measurements and transmitting them. It will neither keep its radio on for 2 seconds, nor listen to any other mote's packets. In fact we will entirely strip FireSense 2.0 mote off its Receive interface. The Repeater mote's operation is actually implied by its name: it doesn't use any of its sensors, rather it only turns on to listen to radio traffic and forward it to other Repeater motes. Adding the operations of the two motes together, creates the original FireSense mote.

We will create our WSN in teams of four: one access point (Repeater) assigned to four peers (FireSense 2.0). We have four access points in total, so 16 peers. The access points will communicate with each other at default radio power (0dBm), but the peers will not have to use their full power obviously. We aim to a range of about 15-16m, which is roughly 32% of the full power mode. Going back to Table 5.2.1 we see that, 32% of the original power corresponds to 129.6m. The output power closer to that value is -10dBm which corresponds to a PA_LEVEL value of 11 and a current consumption of 11mA. Putting these parameters into Cooja, we obtain a range of about 17m which is acceptable. The WSN operates properly. Radio traffic is very low, the response time is lower than in that in the last chapter and the BaseStation mote prints all messages in a period.



```
64566    ID:21   Mote: 15 Temperature: 24 Luminance: 328
64587    ID:21   Mote: 3 Temperature: 24 Luminance: 328
64801    ID:21   Mote: 8 Temperature: 24 Luminance: 328
64912    ID:21   Mote: 11 Temperature: 24 Luminance: 328
64963    ID:21   Mote: 2 Temperature: 24 Luminance: 328
64977    ID:21   Mote: 6 Temperature: 24 Luminance: 328
65088    ID:21   Mote: 4 Temperature: 24 Luminance: 328
65112    ID:21   Mote: 1 Temperature: 24 Luminance: 328
65149    ID:21   Mote: 7 Temperature: 24 Luminance: 328
65167    ID:21   Mote: 14 Temperature: 24 Luminance: 328
65212    ID:21   Mote: 16 Temperature: 24 Luminance: 328
65279    ID:21   Mote: 10 Temperature: 24 Luminance: 328
65325    ID:21   Mote: 12 Temperature: 24 Luminance: 328
65438    ID:21   Mote: 9 Temperature: 24 Luminance: 328
65462    ID:21   Mote: 5 Temperature: 24 Luminance: 328
65615    ID:21   Mote: 13 Temperature: 24 Luminance: 328
```

Figure 5.3.1: BaseStation receives all FireSense 2.0 motes' packets during a sample period

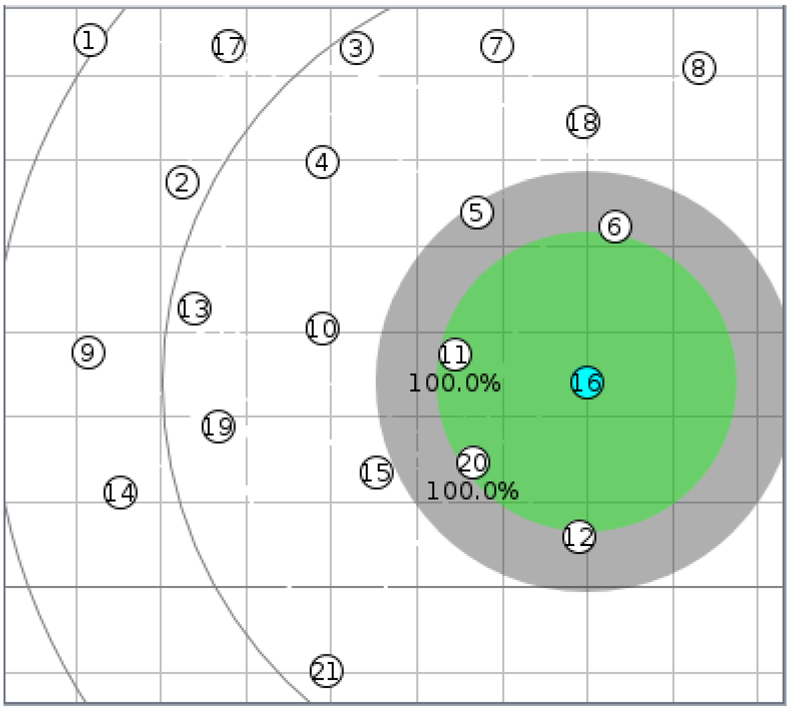


Figure 5.3.2: WSN 2.0 network topology

Nodes 1-16 are FireSense 2.0 motes, 17-20 are Repeater motes and 21 is the BaseStation. The two outer circular lines are the 0dBm transmission and interference ranges respectively. The source code of the FireSense 2.0 and Repeater motes can be found in the Appendix.

5.4 Energy Analysis

5.4.1 FireSense 2.0

Following the same procedure as in Chapter 4, we will consider three sampling periods: 30sec, 60 sec and 90sec.

- Sampling period: 30sec

Component	**Current [mA]**	**Duty Cycle [%]**	**Total current [mA]**
MSP430 Microprocessor			
Active	1.8	0.27	0.00486
Sleep	0.0051	99.73	0.00508623
CC2420 Tranceiver			
Receive	18.8	0.03	0.00564
Transmit	11	0	0
Sleep	0.001	99.97	0.0009997
SHT11 Temp. Sensor			
Measuring	0.5	1.71	0.00855
Sleep	0.0003	98.29	0.00029487
ST M25P80 Flash			
Active	20	0	0
Sleep	0.001	100	0.001
			0.0264

Table 5.4.1: Duty cycle and current consumption of all components in a FireSense 2.0 mote (30sec period)

Battery Capacity [mAh]	**Battery Life [h]**	**Battery Life [d]**	**Battery Life [m]**
500	18917.32	788.22	26.27
1000	37834.65	1576.44	52.55
1500	56751.97	2364.67	78.82
2000	75669.29	3152.89	105.10
2500	94586.62	3941.11	131.37
3000	113503.94	4729.33	157.64
3500	132421.27	5517.55	183.92
4000	151338.59	6305.77	210.19
4500	170255.91	7094.00	236.47
5000	189173.24	7882.22	262.74

Table 5.4.2: FireSense 2.0 mote’s battery life for various battery capacities (30sec period)

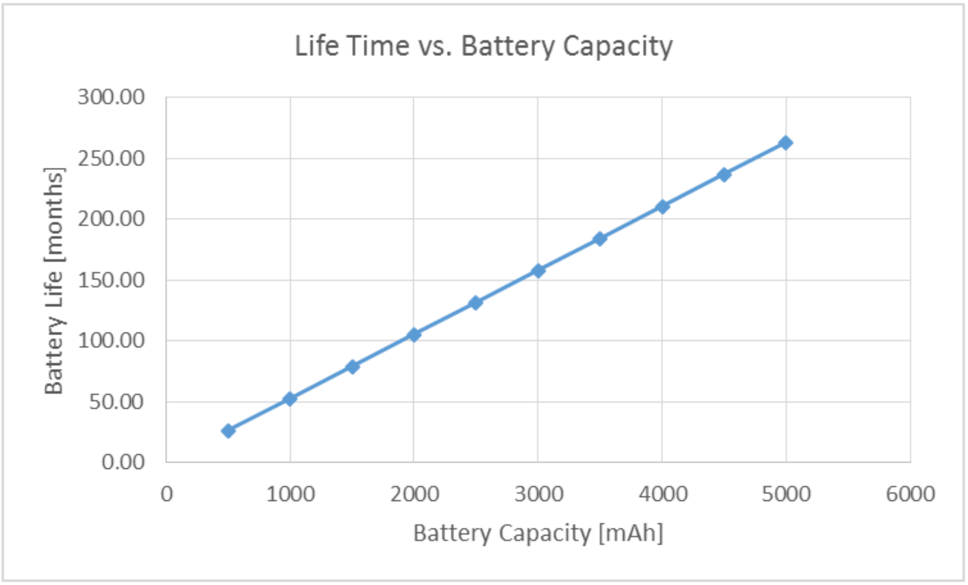


Figure 5.4.1: Mote's battery life vs. battery capacity (FireSense 2.0 mote, 30sec period)

- Sampling period: 60sec

Component	**Current [mA]**	**Duty Cycle [%]**	**Total current [mA]**
MSP430 Microprocessor			
Active	1.8	0.26	0.00468
Sleep	0.0051	99.74	0.00508674
CC2420 Tranceiver			
Receive	18.8	0.01	0.00188
Transmit	11	0	0
Sleep	0.001	99.99	0.0009999
SHT11 Temp. Sensor			
Measuring	0.5	0.88	0.0044
Sleep	0.0003	99.12	0.00029736
ST M25P80 Flash			
Active	20	0	0
Sleep	0.001	100	0.001
			0.0183

Table 5.4.3: Duty cycle and current consumption of all components in a FireSense 2.0 mote (60sec period)

Battery Capacity [mAh]	Battery Life [h]	Battery Life [d]	Battery Life [m]
500	27256.87	1135.70	37.86
1000	54513.74	2271.41	75.71
1500	81770.61	3407.11	113.57
2000	109027.47	4542.81	151.43
2500	136284.34	5678.51	189.28
3000	163541.21	6814.22	227.14
3500	190798.08	7949.92	265.00
4000	218054.95	9085.62	302.85
4500	245311.82	10221.33	340.71
5000	272568.69	11357.03	378.57

Table 5.4.4: FireSense 2.0 mote's battery life for various battery capacities (60sec period)

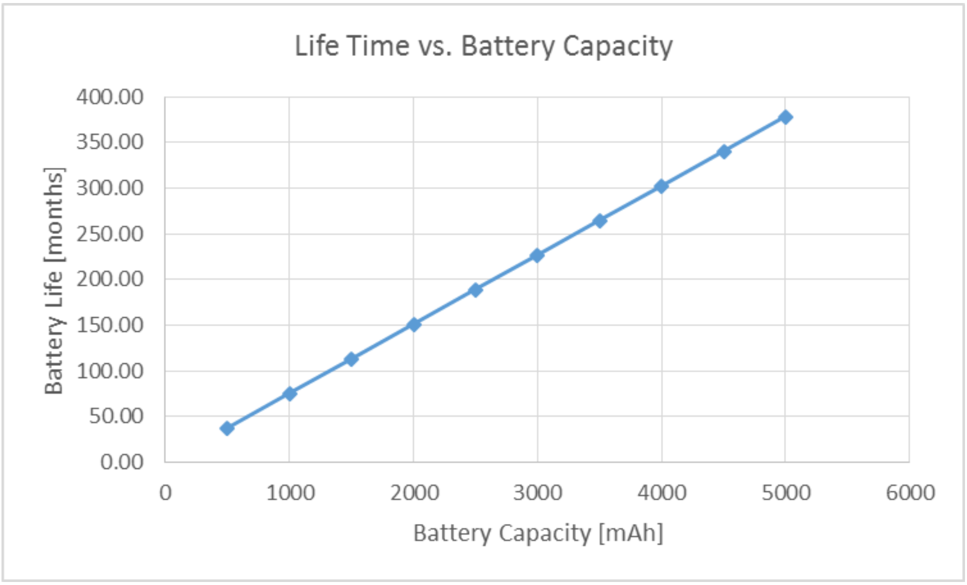


Figure 5.4.2: Mote's battery life vs. battery capacity (FireSense 2.0 mote, 60sec period)

- Sampling period: 90sec

Component	Current [mA]	Duty Cycle [%]	Total current [mA]
MSP430 Microprocessor			
Active	1.8	0.25	0.0045
Sleep	0.0051	99.75	0.00508725
CC2420 Tranceiver			
Receive	18.8	0.01	0.00188
Transmit	11	0	0
Sleep	0.001	99.99	0.0009999

SHT11 Temp. Sensor			
Measuring	0.5	0.55	0.00275
Sleep	0.0003	99.45	0.00029835
ST M25P80 Flash			
Active	20	0	0
Sleep	0.001	100	0.001
			0.0165

Table 5.4.5: Duty cycle and current consumption of all components in a FireSense 2.0 mote (90sec period)

Battery Capacity [mAh]	Battery Life [h]	Battery Life [d]	Battery Life [m]
500	30274.59	1261.44	42.05
1000	60549.18	2522.88	84.10
1500	90823.77	3784.32	126.14
2000	121098.36	5045.77	168.19
2500	151372.95	6307.21	210.24
3000	181647.54	7568.65	252.29
3500	211922.13	8830.09	294.34
4000	242196.72	10091.53	336.38
4500	272471.31	11352.97	378.43
5000	302745.91	12614.41	420.48

Table 5.4.6: FireSense 2.0 mote's battery life for various battery capacities (90sec period)

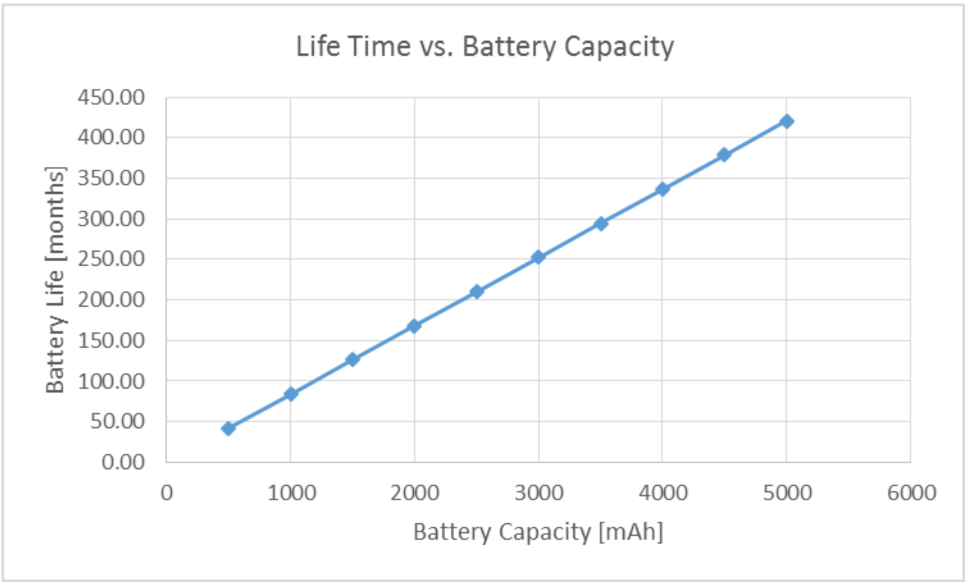


Figure 5.4.3: Mote's battery life vs. battery capacity (FireSense 2.0 mote, 90sec period)

5.4.2 Repeater

- Sampling period: 30sec

Component	**Current [mA]**	**Duty Cycle [%]**	**Total current [mA]**
MSP430 Microprocessor			
Active	1.8	0.44	0.00792
Sleep	0.0051	99.56	0.00507756
CC2420 Tranceiver			
Receive	18.8	6.74	1.26712
Transmit	17.4	0.02	0.00348
Sleep	0.001	93.24	0.0009324
SHT11 Temp. Sensor			
Measuring	0.5	0	0
Sleep	0.0003	100	0.0003
ST M25P80 Flash			
Active	20	0	0
Sleep	0.001	100	0.001
			1.2858

Table 5.4.7: Duty cycle and current consumption of all components in a Repeater mote (30sec period)

Battery Capacity [mAh]	**Battery Life [h]**	**Battery Life [d]**	**Battery Life [m]**
500	388.85	16.20	0.54
1000	777.71	32.40	1.08
1500	1166.56	48.61	1.62
2000	1555.42	64.81	2.16
2500	1944.27	81.01	2.70
3000	2333.12	97.21	3.24
3500	2721.98	113.42	3.78
4000	3110.83	129.62	4.32
4500	3499.69	145.82	4.86
5000	3888.54	162.02	5.40

Table 5.4.8: Repeater mote’s battery life for various battery capacities (30sec period)

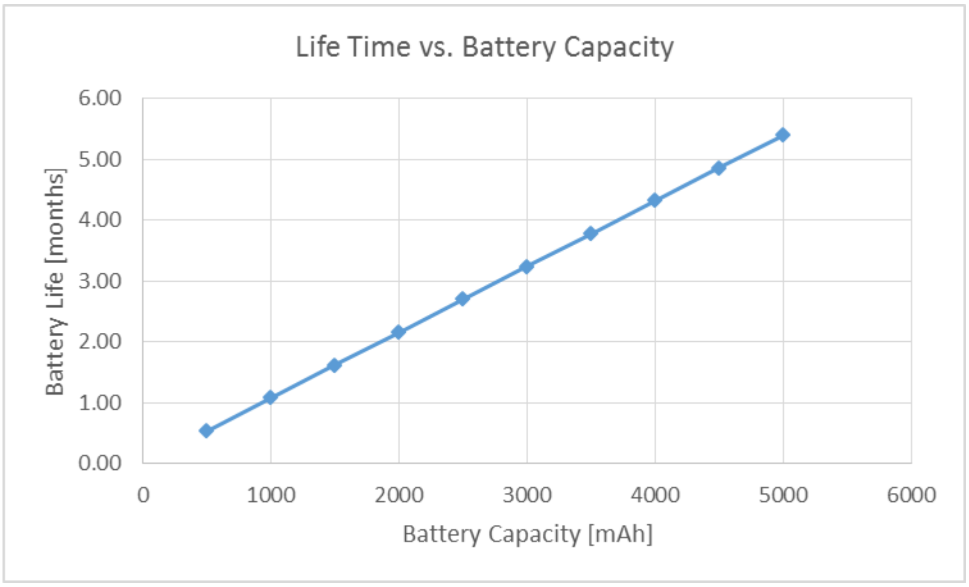


Figure 5.4.4: Mote's battery life vs. battery capacity (Repeater mote, 30sec period)

- Sampling period: 60sec

Component	**Current [mA]**	**Duty Cycle [%]**	**Total current [mA]**
MSP430 Microprocessor			
Active	1.8	0.35	0.0063
Sleep	0.0051	99.65	0.00508215
CC2420 Tranceiver			
Receive	18.8	3.43	0.64484
Transmit	17.4	0.01	0.00174
Sleep	0.001	96.56	0.0009656
SHT11 Temp. Sensor			
Measuring	0.5	0	0
Sleep	0.0003	100	0.0003
ST M25P80 Flash			
Active	20	0	0
Sleep	0.001	100	0.001
			0.6602

Table 5.4.9: Duty cycle and current consumption of all components in a Repeater mote (60sec period)

Battery Capacity [mAh]	**Battery Life [h]**	**Battery Life [d]**	**Battery Life [m]**
500	757.31	31.55	1.05
1000	1514.63	63.11	2.10
1500	2271.94	94.66	3.16
2000	3029.26	126.22	4.21
2500	3786.57	157.77	5.26
3000	4543.89	189.33	6.31
3500	5301.20	220.88	7.36
4000	6058.52	252.44	8.41

4500	6815.83	283.99	9.47
5000	7573.14	315.55	10.52

Table 5.4.10: Repeater mote's battery life for various battery capacities (60sec period)

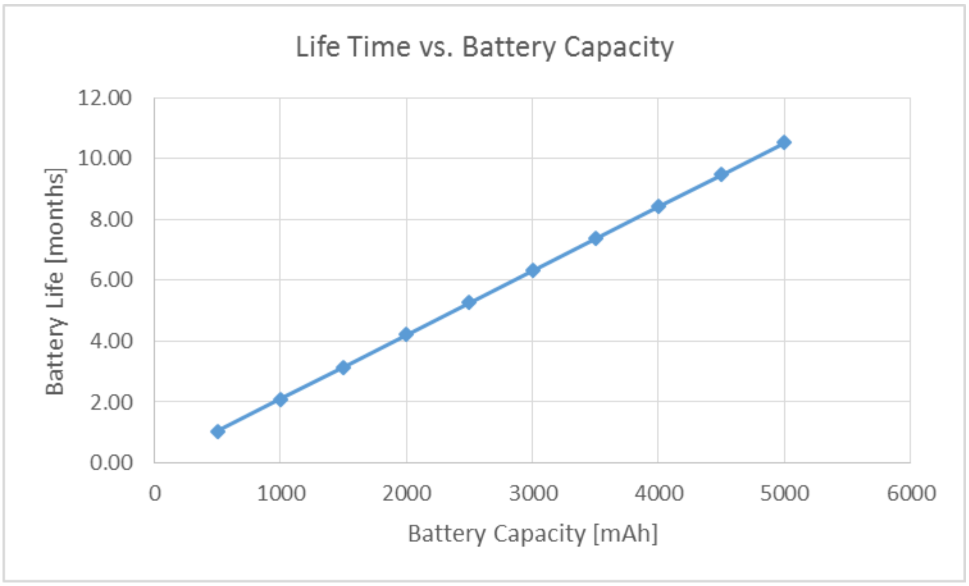


Figure 5.4.5: Mote's battery life vs. battery capacity (Repeater mote, 60sec period)

- Sampling period: 90sec

Component	**Current [mA]**	**Duty Cycle [%]**	**Total current [mA]**
MSP430 Microprocessor			
Active	1.8	0.3	0.0054
Sleep	0.0051	99.7	0.0050847
CC2420 Tranceiver			
Receive	18.8	2.32	0.43616
Transmit	17.4	0	0
Sleep	0.001	97.68	0.0009768
SHT11 Temp. Sensor			
Measuring	0.5	0	0
Sleep	0.0003	100	0.0003
ST M25P80 Flash			
Active	20	0	0
Sleep	0.001	100	0.001
			0.4489

Table 5.4.11: Duty cycle and current consumption of all components in a Repeater mote (90sec period)

Battery Capacity [mAh]	**Battery Life [h]**	**Battery Life [d]**	**Battery Life [m]**

500	1113.78	46.41	1.55
1000	2227.56	92.82	3.09
1500	3341.34	139.22	4.64
2000	4455.12	185.63	6.19
2500	5568.90	232.04	7.73
3000	6682.68	278.45	9.28
3500	7796.46	324.85	10.83
4000	8910.24	371.26	12.38
4500	10024.02	417.67	13.92
5000	11137.80	464.08	15.47

Table 5.4.12: Repeater mote's battery life for various battery capacities (90sec period)

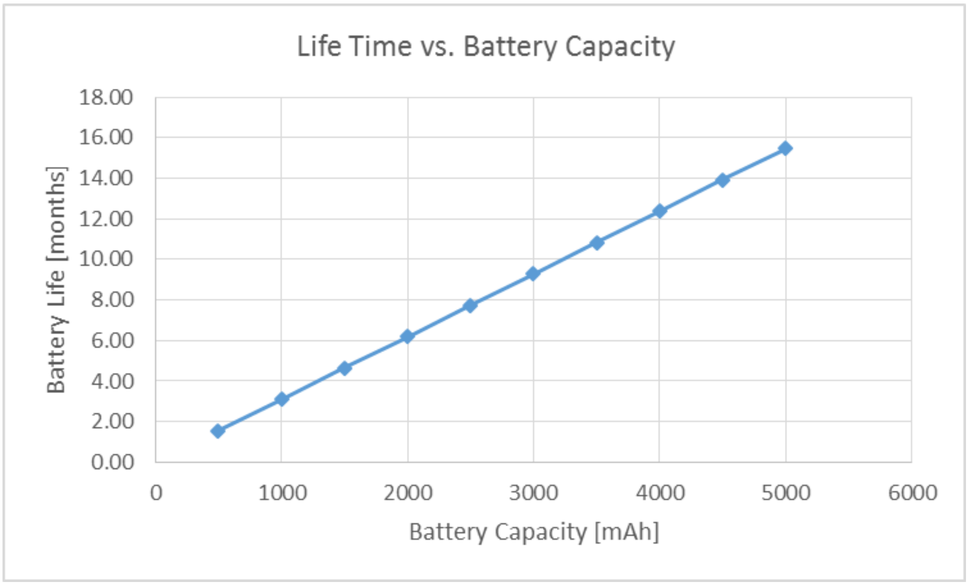


Figure 5.4.6: Mote's battery life vs. battery capacity (Repeater mote, 90sec period)

Chapter 6: Conclusion & Future Work

6.1 Conclusion

In this thesis we presented a cost-effective and autonomous solution for passively monitoring forest areas. The first chapter served as an introduction to Wireless Sensor Networks and their field of applications. In Chapter 2, the reader was presented with the basic building block of a WSN, namely the mote. A few of the most popular mote types were examined, in search of a mote suited to the needs of an application like this. The operating system that made all this possible, TinyOS, along with step-by-step examples and a variety of tools that aided our efforts, are introduced and examined in Chapter 3. Chapters 4 and 5 present what is the main objective of this thesis: the answer to the question "Can an area be efficiently monitored for a time long enough, without the presence of a single person involved in it?"

It is said that a picture is worth a thousand words:

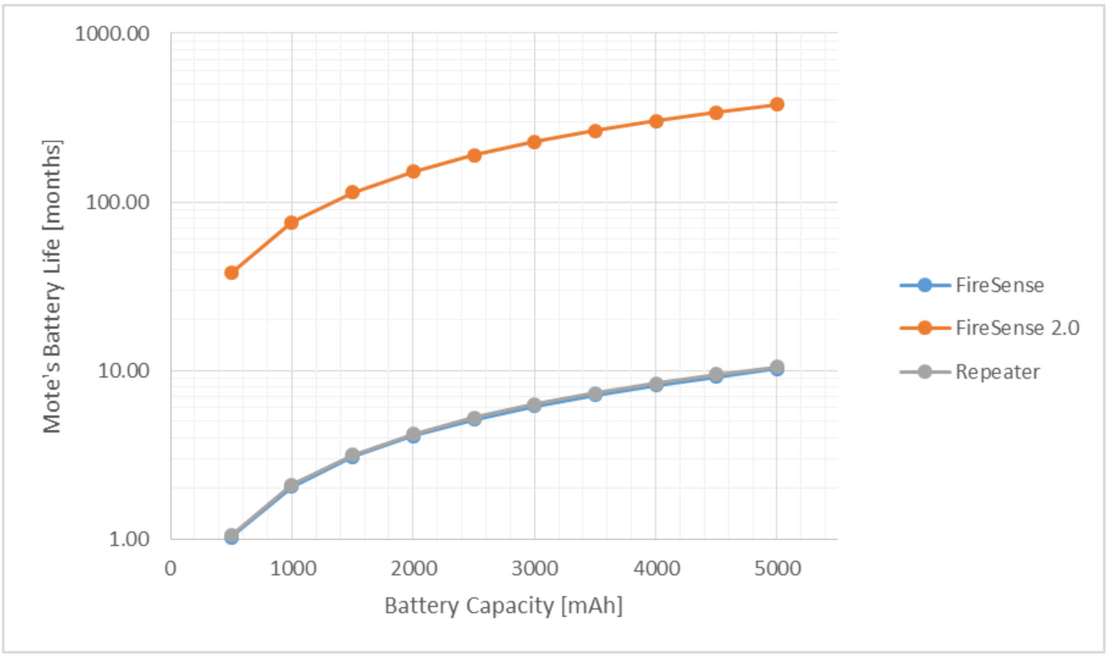


Figure 6.1.1: Logarithmic diagram of all examined motes' lifetime (60sec period)

In short, yes it can. But since we wanted to be more thorough, we studied two different network topologies trying to figure out which one would perform better. An ad-hoc network type and one that uses access points. Looking at the figure above, the answer seems logical. A FireSense 2.0 mote, may run up to 380 months (that's 32½ years), sampling its sensors and sending the readings over the radio every minute, before its batteries run out. That's all in theory of course, since batteries have an expiration date and the mote's actual lifetime would be around 5 or 6 years [4]. But what matters is that we have actually designed a WSN whose fire sensing motes' energy specification outlasts their batteries' lifetime.
That is not the only criterion we should consider though. The ad-hoc WSN is more flexible in its deployment and its "resistance" in case of a mote malfunction or end of life. In the WSN 2.0 however, in case of a Repeater mote failing to operate, the number of peers that

broadcast to it go down as well. What’s more, it is not always certain that the physical characteristics of an area will allow for an effective access point distribution. A FireSense WSN on the other hand, works as a plug-and-play network, because the motes that compose it provide a combination of sensing and packet forwarding.

In the end, it all depends on the needs specific to each case. The first WSN is better suited to easily accessible areas with irregularities in its physical formation, and when performing maintenance every 10 months is not a problem. WSN 2.0 is better for remote and inaccessible areas, in which the motes’ lifetime is the top priority.

6.2 Future Work

The most important enhancement to this thesis would be to test the written source code, on real motes. We would like to see whether our simulated results correspond to real mote operation. Having a physical mote at hand would enable an engineer to perform tasks that are unable to be done in simulation, such as location and time awareness via a GPS expansion card or utilization of additional sensors via external sensor boards. Software wise, it would enable us to utilize several TinyOS components that a simulator couldn’t use, such as the Msp430InternalVoltageC component that gives us the real supply voltage of the mote, or the CC2420Config interface that allows us to change the radio range over the air.

Another proposal would be to examine the behavior and responsiveness of a mote under heavy CPU load. The maximum duty cycle value of the MCU, that we encountered, was about 37% during the first attempt to program the FireSense mote (Section 4.3), but that was not a good example of CPU performance; rather an example of how very heavy radio traffic can take its toll on the CPU. One could create a CPU intensive scenario by e.g. connecting a mote to a camera that takes pictures of an area, compresses them and transmits them.

References

[1] J.Eriksson; F. Osterlind; N. Finne; N. Tsiftes; A.Dunkels; T.Voigt, "COOJA-MSPSim: Interoperability Testing for Wireless_Sensor Networks", Swedish Institute of Computer Science, 2004
[2] O. Landsiedel; K. Wehrle; S. Goetz, "Accurate Prediction of Power Consumption in Sensor Networks", University of Tuebingen, Germany, 2006
[3] M. Johnson; M. Healy; P. vd Ven; M. Hayes; J. Nelson; T. Newe; E. Lewis, "A Comparative Review of Wireless Sensor Network Mote Technologies", IEEE Sensors Conference, 2009
[4] A. Somov; I. Minakov; A. Simalatsar; G. Fontana and R. Passerone, "A Methodology for Power Consumption Evaluation of Wireless Sensor Networks"
[5] J.Eriksson; A. Dunkels; N. Finne; F. Osterlind; T. Voigt, "MSPSim – an Extensible Simulator for MSP430-equipped Sensor Boards", Swedish Institute of Computer Science, 2004
[6] G. Kucuk; B. Kosucu; A. Yavas; S. Baydere, "FireSense: Forest Fire Prediction and Detection System using Wireless Sensor Networks", 2008
[7] P. Levis; D. Gay, "TinyOS Programming", Cambridge University Press, 2009
[8] G. Lorincz and M. Welsh, "Deploying a Wireless Sensor Network on an Active Volcano", IEEE Computer Society, 2006
[9] TinyOS, a free and open source component-based operating system and platform targeting wireless sensor networks, available at www.tinyos.net
[10] C. Merlin, "A tutorial for Programming in TinyOS", University of Rochester, 2009
[11] J.P. Carneiro, "Environmental Monitoring in Wind Farms based on WSANs", Instituto Superior Tecnico, 2009
[12] http://en.wikipedia.org/wiki/Sensor_node
[13] K. Lin; J. Yu; J. Hsu; S. Zahedi; D. Lee, "Heliomote: Enabling Long-Lived Sensor Networks through Solar Energy Harvesting", UCLA, 2005
[14] TelosB datasheet, Available at http://www.memsic.com
[15] TI MSP430F1611 Datasheet, Available at http://www.ti.com
[16] CC24240 Datasheet, Available at http://www.ti.com/lit/ds/swrs041c/swrs041c.pdf
[17] Design Note 007, Available at http://www.ti.com/lit/an/swru120b/swru120b.pdf
[18] Design Note 018, Available at http://www.ti.com/lit/an/swra169a/swra169a.pdf
[19] SHT11 Datasheet, Available at http://www.sensirion.com/en/products/humidity-temperature/humidity-sensor-sht11/
[20] S1087&S108701 Datasheet, Available at http://www.hamamatsu.com/resources/pdf/ssd/s1087_etc_kspd1039e02.pdf
[21] Nano-RK, http://www.nanork.org/projects/nanork
[22] SOS, https://projects.nesl.ucla.edu/public/sos-2x/doc/
[23] Mantis, http://mantisos.org/index/tiki-index.php.html
[24] BTNut, http://www.btnode.ethz.ch/static_docs/doxygen/btnut/
[25] Contiki, http://www.contiki-os.org/
[26] Avrora Simulator, http://compilers.cs.ucla.edu/avrora/

[27] CM5000 specs, http://www.advanticsys.com/shop/mtmcm5000msp-p-14.html
[28] Yeti2 project, http://tos-ide.ethz.ch/wiki/index.php

Appendix

FireSense source code

FireSenseC.nc

```
#include "FireSense.h"

module FireSenseC
{
        uses
        {
                //General Interfaces
                interface Boot;
                interface Timer<TMilli> as Timer1;
                interface Timer<TMilli> as Timer2;
                interface Timer<TMilli> as Timer3;
                interface Leds;

                //Sensor Interfaces
                interface Read<uint16_t> as TempRead;
                interface Read<uint16_t> as LightRead;

                //Radio Interfaces
                interface Packet;
                interface AMSend;
                interface SplitControl as RadioControl;
                interface Receive;
        }
}

implementation
{
        uint8_t numsensors;
        bool radio_busy = FALSE;
        bool sensor_data_packet = FALSE;
        message_t packet;
        FireSenseMsg_t data;
        uint8_t array[NUMBER_OF_MOTES+1] = {0};

        task void startListening();
        task void startSensing();
        task void sendReadings();

        event void Boot.booted()
        {
                post startListening();
        }

        task void startListening()
        {
                call RadioControl.start();
                call Timer1.startOneShot(LISTEN_DURATION_1);
        }

        event void RadioControl.startDone(error_t error)
        {
                if (error == FAIL)
                {
                        call RadioControl.start();
                }
        }
        event void Timer1.fired()
        {
                post startSensing();
        }

        event message_t * Receive.receive(message_t *msg, void *payload, uint8_t len)
        {
                if (len == sizeof(FireSenseMsg_t) && (radio_busy == FALSE))
                {
                        FireSenseMsg_t * incomingPacket = (FireSenseMsg_t *) payload;
                        if (array[incomingPacket -> node_id] == 0)
                        {
                                FireSenseMsg_t * outgoingPacket;
                                outgoingPacket = (FireSenseMsg_t *) call
Packet.getPayload(&packet, sizeof(FireSenseMsg_t));
                                outgoingPacket -> node_id     = incomingPacket -> node_id;
                                outgoingPacket -> temperature = incomingPacket -> temperature;
                                outgoingPacket -> luminance       = incomingPacket ->
luminance;

                                if (call AMSend.send(AM_BROADCAST_ADDR, &packet,
sizeof(FireSenseMsg_t)) == SUCCESS)
                                {
                                        radio_busy = TRUE;
                                        sensor_data_packet = FALSE;
                                        array[outgoingPacket -> node_id] = 1;
                                }
                        }
                }
                return msg;
        }

        event void AMSend.sendDone(message_t *msg, error_t error)
        {
                radio_busy = FALSE;
                if (sensor_data_packet == TRUE)
                {
                        call Timer2.startOneShot(LISTEN_DURATION_2);
                }
        }

        task void startSensing()
        {
                numsensors = 0;
                radio_busy = TRUE;
                call Leds.led0On();
                call TempRead.read();
                call LightRead.read();
        }

        event void TempRead.readDone(error_t result, uint16_t val)
        {
                data.temperature = val;
                if (++numsensors == MAX_SENSORS)
                {
                        post sendReadings();
                        call Leds.led0Off();
                }
        }

        event void LightRead.readDone(error_t result, uint16_t val)
        {
                data.luminance = val;
                if (++numsensors == MAX_SENSORS)
                {
                        post sendReadings();
                        call Leds.led0Off();
                }
        }

        task void sendReadings()
        {
                FireSenseMsg_t * aux;
                aux = (FireSenseMsg_t *) call Packet.getPayload(&packet,
sizeof(FireSenseMsg_t));
                aux -> node_id     = TOS_NODE_ID;
                aux -> temperature = data.temperature;
                aux -> luminance       = data.luminance;
                if (call AMSend.send(AM_BROADCAST_ADDR, &packet, sizeof(FireSenseMsg_t)) ==
SUCCESS)
                {
                        radio_busy = TRUE;
                        sensor_data_packet = TRUE;
                        array[aux -> node_id] = 1;
                }
        }

        event void Timer2.fired()
        {
                call RadioControl.stop();
        }

        event void RadioControl.stopDone(error_t error)
        {
                call Timer3.startOneShot(SLEEP_DURATION);
                memset(array,0,sizeof(array));
        }

        event void Timer3.fired()
        {
                post startListening();
        }
}
```

FireSenseAppC.nc

```
configuration FireSenseAppC{ }

implementation
{
        //General
        components FireSenseC as App; //Main module file
        components MainC; //Boot
        components new TimerMilliC() as TimerA;
        components new TimerMilliC() as TimerB;
        components new TimerMilliC() as TimerC;
        components LedsC;

        App -> MainC.Boot;
        App.Timer1 -> TimerA;
        App.Timer2 -> TimerB;
        App.Timer3 -> TimerC;
        App.Leds -> LedsC;

        //Radio Communication
        components ActiveMessageC;
        components new AMSenderC(AM_RADIO);
        components new AMReceiverC(AM_RADIO);

        App.Packet -> AMSenderC;
        App.AMSend -> AMSenderC;
        App.RadioControl -> ActiveMessageC;
        App.Receive -> AMReceiverC;

        //Temperature components
        components new SensirionSht11C() as TempSensor;
        App.TempRead -> TempSensor.Temperature;

        //Light components
        components new HamamatsuS1087ParC() as LightSensor;
        App.LightRead -> LightSensor;
}
```

FireSense.h

```
#ifndef FIRE_SENSE_H
#define FIRE_SENSE_H

enum
{
        NUMBER_OF_MOTES = 15,
        MAX_SENSORS = 2,
        AM_RADIO = 1,
        LISTEN_DURATION_1 = 1024,
        LISTEN_DURATION_2 = 1024,
        SLEEP_DURATION = 28672
};

typedef nx_struct FireSenseMsg
{
        nx_uint16_t node_id;
        nx_uint16_t temperature;
        nx_uint16_t luminance;
} FireSenseMsg_t;

#endif /* FIRE_SENSE_H */
```

Makefile

```
COMPONENT=FireSenseAppC
include $(MAKERULES)
```

FireSense 2.0 source code

FireSense2C.nc

```
#include "FireSense2.h"

module FireSense2C
{
        uses
        {
                //General Interfaces
                interface Boot;
                interface Timer<TMilli> as Timer;
                interface Leds;

                //Sensor Interfaces
                interface Read<uint16_t> as TempRead;
                interface Read<uint16_t> as LightRead;

                //Radio Interfaces
```

```
		interface Packet;
		interface AMSend;
		interface SplitControl as RadioControl;
	}
}

implementation
{
	uint8_t numsensors;
	message_t packet;
	FireSenseMsg_t data;

	task void startSensing();
	task void sendReadings();

	event void Boot.booted()
	{
		post startSensing();

	}

	task void startSensing()
	{
		numsensors = 0;
		call Leds.led0On();
		call TempRead.read();
		call LightRead.read();
	}

	event void TempRead.readDone(error_t result, uint16_t val)
	{
		data.temperature = val;
		if (++numsensors == MAX_SENSORS)
		{
			post sendReadings();
			call Leds.led0Off();
		}
	}

	event void LightRead.readDone(error_t result, uint16_t val)
	{
		data.luminance = val;
		if (++numsensors == MAX_SENSORS)
		{
			post sendReadings();
			call Leds.led0Off();
		}
	}

	task void sendReadings()
	{
		FireSenseMsg_t * aux;
		aux = (FireSenseMsg_t *) call Packet.getPayload(&packet,
sizeof(FireSenseMsg_t));
		aux -> node_id     = TOS_NODE_ID;
		aux -> temperature = data.temperature;
		aux -> luminance       = data.luminance;
		call RadioControl.start();
	}

	event void RadioControl.startDone(error_t error)
	{
		if (error == FAIL)
		{
			call RadioControl.start();
		}
		else
```

```
				{
					call AMSend.send(AM_BROADCAST_ADDR, &packet, sizeof(FireSenseMsg_t));
				}
		}

		event void AMSend.sendDone(message_t *msg, error_t error)
		{
				call RadioControl.stop();
		}

		event void RadioControl.stopDone(error_t error)
		{
				if (error == FAIL)
				{
					call RadioControl.stop();
				}
				else
				{
					call Timer.startOneShot(PERIOD);
				}
		}

		event void Timer.fired()
		{
				post startSensing();
		}
}
```

FireSense2AppC.nc

```
configuration FireSense2AppC
{

}
implementation
{
		//General
		components FireSense2C as App; //Main module file
		components MainC; //Boot
		components new TimerMilliC() as Timer;
		components LedsC;

		App -> MainC.Boot;
		App.Timer -> Timer;
		App.Leds -> LedsC;

		//Radio Communication
		components ActiveMessageC;
		components new AMSenderC(AM_RADIO);

		App.Packet -> AMSenderC;
		App.AMSend -> AMSenderC;
		App.RadioControl -> ActiveMessageC;

		//Temperature components
		components new SensirionSht11C() as TempSensor;
		App.TempRead -> TempSensor.Temperature;

		//Light components
		components new HamamatsuS1087ParC() as LightSensor;
		App.LightRead -> LightSensor;
}
```

FireSense2.h

```
#ifndef FIRE_SENSE2_H
#define FIRE_SENSE2_H

enum
{
	MAX_SENSORS = 2,
	AM_RADIO = 1,
	PERIOD = 28672
};

typedef nx_struct FireSenseMsg
{
	nx_uint16_t node_id;
	nx_uint16_t temperature;
	nx_uint16_t luminance;
} FireSenseMsg_t;

#endif /* FIRE_SENSE2_H */
```

Makefile

```
COMPONENT=FireSense2AppC
CFLAGS += -DCC2420_DEF_RFPOWER=11
include $(MAKERULES)
```

Repeater source code

RepeaterC.nc

```
#include "Repeater.h"

module RepeaterC
{
	uses
	{
		//General Interfaces
		interface Boot;
		interface Timer<TMilli> as Timer1;
		interface Timer<TMilli> as Timer2;

		//Radio Interfaces
		interface Packet;
		interface AMSend;
		interface SplitControl as RadioControl;
		interface Receive;
	}
}

implementation
{
	bool radio_busy = FALSE;
	message_t packet;
	FireSenseMsg_t data;
	uint8_t array[NUMBER_OF_MOTES+1] = {0};

	task void startListening();

	event void Boot.booted()
```

```
        {
                post startListening();
        }

        task void startListening()
        {
                call RadioControl.start();
                call Timer1.startOneShot(LISTEN_DURATION);
        }

        event void RadioControl.startDone(error_t error)
        {
                if (error == FAIL)
                {
                        call RadioControl.start();
                }
        }

        event message_t * Receive.receive(message_t *msg, void *payload, uint8_t len)
        {
                if (len == sizeof(FireSenseMsg_t) && (radio_busy == FALSE))
                {
                        FireSenseMsg_t * incomingPacket = (FireSenseMsg_t *) payload;
                        if (array[incomingPacket -> node_id] == 0)
                        {
                        FireSenseMsg_t * outgoingPacket;
                        outgoingPacket = (FireSenseMsg_t *) call Packet.getPayload(&packet,
sizeof(FireSenseMsg_t));
                        outgoingPacket -> node_id     = incomingPacket -> node_id;
                        outgoingPacket -> temperature = incomingPacket -> temperature;
                        outgoingPacket -> luminance   = incomingPacket -> luminance;

                        if (call AMSend.send(AM_BROADCAST_ADDR, &packet,
sizeof(FireSenseMsg_t)) == SUCCESS)
                        {
                                radio_busy = TRUE;
                                array[outgoingPacket -> node_id] = 1;
                        }
                        }
                }
                return msg;
        }

        event void AMSend.sendDone(message_t *msg, error_t error)
        {
                radio_busy = FALSE;
        }

        event void Timer1.fired()
        {
                call RadioControl.stop();
        }

        event void RadioControl.stopDone(error_t error)
        {
                call Timer2.startOneShot(SLEEP_DURATION);
                memset(array,0,sizeof(array));
        }

        event void Timer2.fired()
        {
                post startListening();
        }
}
```

RepeaterAppC.nc

```
configuration RepeaterAppC{}

implementation
{
        //General
        components RepeaterC as App; //Main module file
        components MainC; //Boot
        components new TimerMilliC() as TimerA;
        components new TimerMilliC() as TimerB;

        App -> MainC.Boot;
        App.Timer1 -> TimerA;
        App.Timer2 -> TimerB;

        //Radio Communication
        components ActiveMessageC;
        components new AMSenderC(AM_RADIO);
        components new AMReceiverC(AM_RADIO);

        App.Packet -> AMSenderC;
        App.AMSend -> AMSenderC;
        App.RadioControl -> ActiveMessageC;
        App.Receive -> AMReceiverC;
}
```

Repeater.h

```
#ifndef REPEATER_H
#define REPEATER_H

enum
{
        NUMBER_OF_MOTES = 16,
        LISTEN_DURATION = 2048,
        SLEEP_DURATION = 30720,
        AM_RADIO = 1
};

typedef nx_struct FireSenseMsg
{
        nx_uint16_t node_id;
        nx_uint16_t temperature;
        nx_uint16_t luminance;
} FireSenseMsg_t;

#endif /* REPEATER_H */
```

Makefile

```
COMPONENT=RepeaterAppC
include $(MAKERULES)
```

BaseStation source code

BaseStationC.nc

```
#include "BaseStation.h"
#include <stdio.h>
#include <string.h>

module BaseStationC
{
	uses
	{
		//General Interfaces
		interface Boot;
		interface Leds;

		//Radio Interfaces
		interface Packet;
		interface AMSend;
		interface SplitControl as RadioControl;
		interface Receive;

		interface Timer<TMilli> as Timer;
	}
}

implementation
{
	uint8_t array[NUMBER_OF_MOTES+1] = {0};
	uint16_t celsius;
	uint16_t lux;

	event void Boot.booted()
	{
		call Timer.startPeriodic(TIMER_RESET);
		call RadioControl.start();
		call Leds.led1On();
	}

	event void RadioControl.startDone(error_t error)
	{
		if (error == FAIL)
		{
			call RadioControl.start();
		}
	}

	event message_t * Receive.receive(message_t *msg, void *payload, uint8_t len)
	{
		FireSenseMsg_t *incomingPacket = (FireSenseMsg_t *) payload;
		if (array[incomingPacket -> node_id] == 0)
		{
			uint16_t val1 = incomingPacket -> temperature;
			uint16_t val2 = incomingPacket -> luminance;
			uint16_t val3 = incomingPacket -> node_id;
			celsius = -39.6 + 0.01 * val1;
			lux = 3.815 * val2;
			printf("Mote: %d Temperature: %d Luminance: %d \n", val3, celsius, 
lux);
			array[val3] = 1;
			if ( ((celsius >= 40) && (celsius <= 45)) || ((lux >= 600) && (lux <= 
300)) )
			{
				call Leds.led2On();
			}
```

```
                        else if ( (celsius >= 45) || ((lux >= 800) && (lux <= 100)) )
                        {
                                call Leds.led0On();
                        }
                }
                return msg;
        }

        event void Timer.fired()
        {
                memset(array,0,sizeof(array));
                call Leds.led0Off();
                call Leds.led2Off();
        }

        event void RadioControl.stopDone(error_t error)
        {

        }

        event void AMSend.sendDone(message_t *msg, error_t error)
        {

        }
}
```

BaseStationAppC.nc

```
configuration BaseStationAppC
{

}

implementation
{
        components BaseStationC as App;
        components MainC;
        components LedsC;
        components new TimerMilliC();

        App -> MainC.Boot;
        App -> LedsC.Leds;
        App -> TimerMilliC.Timer;

        components ActiveMessageC;
        components new AMSenderC(AM_RADIO);
        components new AMReceiverC(AM_RADIO);

        App.Packet -> AMSenderC;
        App.AMSend -> AMSenderC;
        App.RadioControl -> ActiveMessageC;
        App.Receive -> AMReceiverC;

        components SerialPrintfC;
}
```

BaseStation.h

```
#ifndef BASE_STATION_H
#define BASE_STATION_H

enum
{
        NUMBER_OF_MOTES = 15,
        AM_RADIO = 1,
        TIMER_RESET = 30720
};

typedef nx_struct FireSenseMsg
{
        nx_uint16_t node_id;
        nx_uint16_t temperature;
        nx_uint16_t luminance;
} FireSenseMsg_t;

#endif /* BASE_STATION_H */
```

Makefile

```
COMPONENT = BaseStationAppC
PFLAGS += -I$(TOSDIR)/lib/printf
include $(MAKERULES)
```